ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

***Science School***

***Master Degree in Computer Science***

# *Deep Reinforcement Learning and sub-problem decomposition using Hierarchical Architectures in partially observable environments*

***Supervisor:***
***Prof. Andrea Asperti***

***Presented by:***
***Francesco Sovrano***

***Session II***

***Accademic Year 2017/2018***

*To anyone with a little bit of madness*

# Introduction

Reinforcement Learning (RL) is an area of machine learning which studies how an *agent* should take *actions* in an *environment* (or problem) having the goal of maximizing some *cumulative reward*. One of the main differences between RL and supervised learning is the fact that in RL the actions/decisions taken by the agent may change the environment, while in supervised learning this is not true.
The RL framework is very generic and can be used to tackle different classes of problems. In recent years, there has been a huge amount of work on the application of deep learning techniques in combination with Reinforcement Learning (the so called *Deep Reinforcement Learning*), specially for the development of automatic agents for different kind of games. Game-like environments provide realistic abstractions of real-life situations, creating new and innovative dimensions in the kind of problems that can be addressed by means of neural networks.
Reinforcement Learning is based on the Markov Decision Process (MDP) framework, but not all the problems of interest can be modelled with MDPs because some of them have non-markovian dependencies. To handle them, one of the solutions proposed in literature is Hierarchical Reinforcement Learning (HRL). HRL takes inspiration from hierarchical planning in artificial intelligence literature and it is an emerging sub-discipline for RL, in which RL methods are augmented with some kind of prior knowledge about the high-level structure of behaviour in order to decompose the underlying problem into simpler sub-problems.
The high-level goal of our thesis is to investigate the advantages that a HRL approach may have over a simple RL approach. Thus, we study problems of interest (rarely tackled by mean of RL) like Sentiment Analysis, Rogue and Car Controller, showing how

the ability of RL algorithms to solve them in a partially observable environment is affected by using (or not) generic hierarchical architectures based on RL algorithms of the Actor-Critic family. Remarkably, we claim that especially our work in Sentiment Analysis is very innovative for RL, resulting in state-of-the-art performances; as far as the author knows, Reinforcement Learning (RL) approach is only rarely applied to the domain of computational linguistic and sentiment analysis. Furthermore, our work on the famous video-game Rogue is probably the first example of Deep RL architecture able to explore Rogue dungeons and fight against its monsters achieving a success rate of more than 75% on the first game level. While our work on Car Controller allowed us to make some interesting considerations on the nature of some components of the policy gradient equation (Section 1.3.2.1).
Anyway, in order to give a proper background to the reader, in chapter 1 we provide some basic information on Reinforcement Learning and the underlying Markov Decision Process (MDP) and Partially Observable MDP frameworks, while in chapter 2 we enumerate some of the most recent and important Deep RL algorithms of the Actor-Critic family, showing details of the inner mechanisms of the algorithms subset used in our experiments: A3C [MBM+16a], GAE [SML+15], PPO [SWD+17].
In chapter 3 we present some interesting state-of-the-art techniques to speed up Reinforcement Learning, describing more in detail those we are going to adopt in the experiments, including HRL.
In chapter 4 we introduce the environments used for our experiments, making distinction between those who have a discrete action space and those who have a continuous one. In the same chapter we also briefly describe the adopted software framework.
Finally, in chapter 5 we describe the results of three different experiments, on three different environments and neural networks, using Generic HRL techniques based on simple variations of the Options Framework (Section 3.4.1).
In all these experiments the hierarchical architecture is made by an A3C partitioner able to partition the state space into $n$ different sub-spaces for respectively $n$ different A3C workers.
Our first experiment (Section 5.1) is conducted on Sentiment Analysis using the SENTIPOLC 2016 environment described in Section 4.1.1.2, and we show that in this envi-

ronment the benefits of HRL may be significant. The main challenge of this experiment was to model the underlying problem as MDP, in order to use the RL framework.
Our second experiment (Section 5.2) is conducted on the Rogue environment introduced in Section 4.1.1.1, and we show that in the chosen environmental settings the benefits of HRL do not seem to be significant. The main challenges of this experiment were probably: to choose a good state representation, to tackle the exploration problem mentioned in Chapter 3, and to find a useful generic hierarchical architecture.
Our last experiment (Section 5.3) is conducted on the Car Controller environment described in Section 4.1.2.1, and we show that in this particular environment there are no evident benefits in using HRL. The biggest challenge of this second experiment was probably to understand how to model a policy for continuous action-spaces using a normal distribution.

# Contents

# List of Figures

# Pseudo-codes

# List of Tables

# Chapter 1

# Background Information

Reinforcement Learning (RL) is an area of machine learning which studies how an *agent* should take *actions* in an *environment* (or problem) having the goal of maximizing some *cumulative reward*.
In this chapter we try to give to the reader the necessary background information required for understanding what RL is. For this reason we firstly introduce the Markov Decision Process (MDP) framework at the foundations of RL and then we approach Partially Observable MDPs and the main RL algorithms families.

## 1.1 Markov Decision Process

MDPs provide a mathematical framework for modelling sequential decision-making in situations where the results are partially random and partly under the control of a decision maker/agent.
More in detail, an MDP is a probabilistic control process defined over discrete time intervals. At each time interval $t$, the process is in some state $s_t$ and the agent chooses some action $a_t$. When the agent chooses $a_t$, it modifies the environment by passing from state $s_t$ to the state $s_{t+1}$ and receives a feedback (called reward $r_t$) on the goodness of $a_t$ results. The agent goal is to maximize the total cumulative reward, which affects the actions policy. A summary image of the aforementioned process is shown in Fig.1.1.
An MDP is a Markov model and it is defined by a tuple of 5 values:

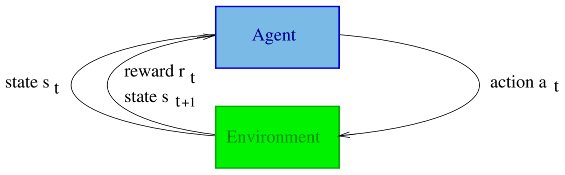


Figure 1.1: **Markov Decision Process**: Basic operations

- $S$: a finite set of states.
- $A$: a finite set of actions.
- $P(s_{t+1}|a_t \wedge s_t)$: a function defining the probability of reaching state $s_{t+1}$ at time $t+1$ starting from state $s_t$ and executing action $a_t$ at time $t$.
- $R(s_t, a_t)$: a function to define the reward $r_t$ obtained by reaching the state $s_{t+1}$ at the time instant $t+1$ starting from state $s_t$ by executing the action $a_t$ at the time instant $t$.
- $\gamma \in [0, 1]$: the discount factor. It represents the difference in importance between present and future rewards. Values close to 0 are awarded when present rewards are more important than future rewards.

In an MDP the state transition probabilities must be determined solely and exclusively by the current state and action, this property is called Markov's property. In other words, a stochastic process has the property of Markov if, given the present, the future does not depend on the past. More formally we have:

$$P(S_t = s_t|S_{t-1} = s_{t-1}, \ldots, S_0 = s_0) = P(S_t = s_t|S_{t-1} = s_{t-1}) \tag{1.1}$$

## 1.2 Partially Observable Markov Decision Process

A Markov decision process is called partially observable (POMDP) when the state $s_t$ is not completely known. A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. A POMPD is a generalization of a MPD. The

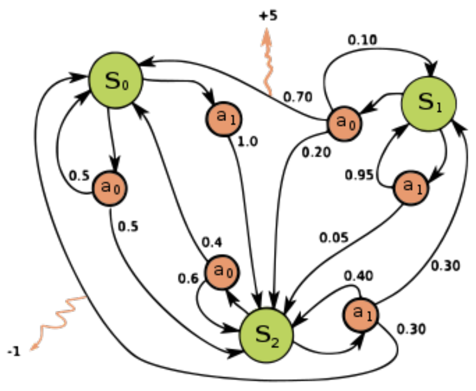


Figure 1.2: **Markov Decision Process**: Example from Wikipedia of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)

POMDP framework is general enough to model a variety of real-world sequential decision processes. Applications include robot navigation problems, machine maintenance, and planning under uncertainty in general.
A discrete-time POMDP models the relationship between an agent and its environment. Formally, a POMDP is a MDP with a set of observations $\Omega$ and a set of conditional observation probabilities $O_{a_t}(s_{t+1}, o_t)$. Thus a POMPD is a 7-tuple ($S$, $A$, $P$, $R$, $\gamma$, $\Omega$, $O$). At time period $t$, the environment is in some state $s_t$, but the agent observes only $o_t$ with probability $O(o_t|s_t)$. The agent takes an action $a_t$, which causes the environment to transition to state $s_{t+1}$ with probability $P(s_{t+1}|a_t \wedge s_t)$. As result the agent receives a new observation $o_{t+1}$ which depends on the new state of the environment. Finally, the agent receives a reward $r = R_{a_t}(s_t, s_{t+1})$. The process is then iterated until a terminal state is reached.

## 1.3 Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning which studies how an *agent* should take *actions* in an *environment* (or problem) having the goal of maximizing some *cumulative reward*. One of the main differences between RL and supervised

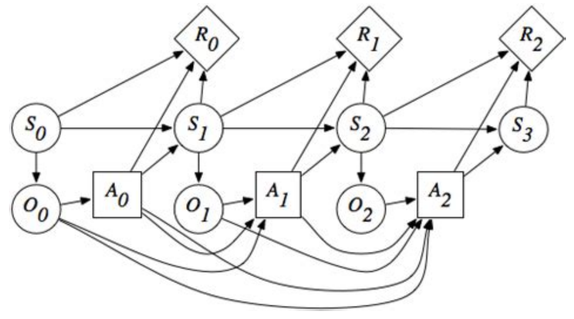


Figure 1.3: **Partially Observable Markov Decision Process**: Example of POMDP

learning is the fact that in RL the actions/decisions taken by the agent may change the environment, while in supervised learning this is not true.
RL can solve MDPs without explicit specification of the transition probabilities. The goal of reinforcement learning is to maximize the expected total reward from all actions performed in the future:

$$\mathbb{E}[R(s_0, a_0) + \gamma R(s_1, a_1) + \cdots + \gamma^n R(s_n, a_n)] = \mathbb{E}[\sum_n \gamma^n R(s_n, a_n)] \tag{1.2}$$

Not every action immediately gives a reward/feedback, sometimes you need long sequences of actions in order to get a reward, but future actions have not been performed yet. This means that the less frequent (the more sparse) are the rewards, the harder is for the algorithm to learn.
In order to solve the expected total reward problem we have to define a policy $\pi : S \to A$. A policy is a function that takes the current state $s_t$ and outputs an optimal action and this policy may be deterministic or stochastic.
For a MDP, an optimal deterministic policy always exists [Bel13]. An Optimal Policy is a policy where you are always choosing the action that maximizes the expected cumulative reward. The optimal policies may be found using the Bellman Optimal equation and Dynamic Programming (DP) techniques [Bel13]:

$$\pi(s_t) = \arg\max_{a_t} \left\{ \sum_{s_{t+1}} P(s_{t+1}|a_t \wedge s_t) \cdot (R(s_t, a_t) + \gamma V(s_{t+1})) \right\} \tag{1.3}$$

$$V(s_t) = \sum_{s_{t+1}} P(s_{\pi(s_t)}|a_t \wedge s_t) \cdot (R(s_t, \pi(s_t)) + \gamma V(s_{t+1})) \tag{1.4}$$

DP approaches update estimates based on other estimates (one step ahead), thus they assume complete knowledge of the environment (the MDP), but in practice this is not possible and even assuming that this is possible, the applicability of dynamic programming to many important problems is limited by the enormous size of the underlying state spaces. This limitation is called *curse of dimensionality* [Bel13]. Neuro-dynamic programming (also known as "Reinforcement Learning" in the Artificial Intelligence literature) uses neural networks and other approximation architectures to overcome the *curse of dimensionality* [Ber08].
The starting points for the RL methodology are two fundamental dynamic programming algorithms: policy iteration and value iteration. In the first approach, we define the parameters of a value function that quantifies the maximum cumulative reward obtainable from a state belonging to the state space. In mathematical terms, the goal is to approximate a solution to the Bellman equation, so that we can build optimal policies. While, in the policy-based approach the policy parameters are tuned in a direction of improvement.
In Fig. 1.4 a simple overview of the RL families is shown.

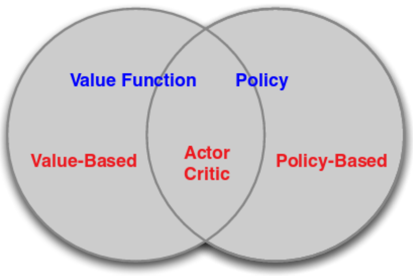


Figure 1.4: **Reinforcement Learning**: Algorithms families

## 1.3.1 Value function approximation

Value-based approaches try to find a policy that maximizes the cumulative return by keeping a set of expected returns estimates for some policy $\pi$. Usually $\pi$ is either the "current" policy or the optimal one. An algorithm that uses the current policy is called

on-policy algorithm, while an algorithm using the optimal one is called off-policy.
In value-based methods, a policy is defined optimal if it achieves the best expected return from any initial state. To define optimality in a formal manner, we have to define the value of a policy $\pi$ by

$$V^{\pi}(s) = \mathbb{E}^{\pi}[R_t | s = s_t] \tag{1.5}$$

where $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total cumulative reward from time starting at $t$. Thus, the optimal value function is $V^*(s) = \max_\pi V^{\pi}(s)$. Although state-values are enough for defining optimality, it is useful to define the action-value function

$$Q^{\pi}(s, a) = \mathbb{E}^{\pi}[R_t | s = s_t, a = a_t] \tag{1.6}$$

$Q^{\pi}(s, a)$ is the expected return for selecting action $a$ in state $s$ and prosecuting with policy $\pi$. Given a state $s$ and an action $a$, the optimal action-value function $Q^*(s, a) = \max_\pi Q^{\pi}(s, a)$ is the best possible action value that any policy may achieve.
There are many methods for value function approximations, among them: Monte Carlo methods, Temporal Difference methods and N-step methods.

#### 1.3.1.1 Monte Carlo methods

Monte Carlo methods (MC) learn directly from the experience gathered by interacting with the environment. An MC method randomly samples entire episodes of experience and updates accordingly the value estimates. MC methods have high variance (due to lots of random decisions within an episode) but are unbiased.
But how do we ensure that we explore all states if we do not know the full environment? The solution to this exploration problem is to use $\epsilon$-greedy policies with $\epsilon > 0$, where $\epsilon$ is the probability of acting randomly.
Problems with MC include:

1. MC procedure may spend too much time evaluating a suboptimal policy.
2. It is sample inefficient.
3. When the returns along the trajectories have high variance, convergence is slow.
4. It works in episodic problems only.

5. It works in small, finite MDPs only.

#### 1.3.1.2 Temporal Difference and N-step methods

Temporal difference (TD) methods are a combination of Monte Carlo and Dynamic Programming ideas. These methods sample from the environment, like Monte Carlo methods, and perform updates based on current estimates, like dynamic programming methods.
While Monte Carlo methods only update their estimates at the end of an episode, TD methods update their estimates to match later, more accurate, predictions about the future, before the final cumulative reward is known. The general update rule for TD learning is:

$$Q[s_t, a_t] \leftarrow (1 - \alpha) \cdot Q[s_t, a_t] + \alpha \cdot \text{td\_target} \tag{1.7}$$

td_target $- Q[s_t, a_t]$ is called TD error, and the value-action table $Q$ is initialized arbitrarily.
SARSA [Sut96] is an example of on-policy TD control, with TD target: $R_t + \gamma * Q[s_{t+1}, a_{t+1}]$, while Q-Learning [WD92] is an off-policy TD control, with TD target: $R_t + \gamma * \max(Q[s_{t+1}])$.
Q-Learning has a positive bias because it uses the maximum of estimated $Q$ values to estimate the best action value, all from the same experience. Double Q-Learning [VHGS16] gets around this by splitting the experience and using different $Q$ functions for maximization and estimation.
N-Step methods unify MC and TD approaches, making updates based on n-steps instead of a single step (TD-0) or a full episode (MC).

### 1.3.2 Policy function approximation

In policy-based RL, instead of parametrizing the value function and doing $\epsilon$-greedy policy improvements to the value function parameters, we parametrize the policy $\pi_\theta(a|s)$ and update the parameters $\theta$ descending gradient into a direction that improves $\pi$, because sometimes the policy is easier to approximate than the value function.
With respect to value-based RL, the advantages of policy-based RL are:

- Better convergence properties.
- Effective in high-dimensional or continuous action spaces.
- Can learn stochastic policies.

While the disadvantages are:

- Typically converge to a local than global optimum.
- Evaluating a policy is typically inefficient and high variance.

Similarly to value-based approaches, also policy-based approaches can be on-policy or off-policy. The difference between on-policy and off-policy policy based algorithms is in the policy gradient formula.

#### 1.3.2.1 On-policy gradient

Policy gradient methods maximize the expected total reward by repeatedly estimating the gradient:

$$\nabla_\theta J^{\mathrm{ON}}(\theta) := \nabla_\theta \mathbb{E}\left[\sum_{t=0}^{\infty} r_t\right] \tag{1.8}$$

There are several different related expressions for the policy gradient, which have the form:

$$\nabla_\theta J^{\mathrm{ON}}(\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \Psi_t \cdot \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{1.9}$$

where $\Psi_t$ may be one of the following:

- The total reward of the trajectory: $\sum_{t=0}^{\infty} r_t$.
- The state-action value function: $Q^\pi(s_t, a_t)$.
- The TD residual: $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$.

#### 1.3.2.2 Off-policy Gradient

An off-policy policy-based method uses past experience to update its parameters. The past experience is sampled by a known and predefined strategy $\beta(a|s)$. This experience replay technique may improve the algorithms performances giving better sample efficiency (that in many cases also brings better exploration).
The off-policy policy gradient is slightly different from the on-policy one, in fact eq.(1.9) has to be rewritten in order to consider also the importance of the sampled experience. Thus we have:

$$\nabla_\theta J^{\mathrm{OFF}}(\theta) = \mathbb{E}_\beta\left[\frac{\pi_\theta(a|s)}{\beta(a|s)} \cdot \Psi \cdot \nabla_\theta \log \pi_\theta(a|s)\right] \tag{1.10}$$

where $\frac{\pi_\theta(a|s)}{\beta(a|s)}$ is the importance weight.

### 1.3.3 Actor-Critic

As shown in Fig.1.4, Actor-Critic methods are in the middle between policy-based and N-step value-based RL.
In order to properly perform the bootstrapping operation required by N-step, a Critic that approximates the value function is used while an Actor decides the actions accordingly. Thus, in Actor-Critic we have two function approximators: one for the policy (the Actor), one for the value function (the Critic). This is basically N-step, but for Policy Gradients.
More in details, the idea behind Actor-Critic is that it is possible to reduce the variance of the policy (eq.1.9) while keeping it unbiased by subtracting a learned state-value function $V(s)$ known as baseline. $R - V(s)$ is called the advantage function $A(s)$. The advantage is used to measure how much better than "average" it is to take an action given a state. Thus, the Actor updates its parameters $\theta_1$ in the direction of:

$$\mathbb{E}[A(s) \cdot \nabla_{\theta_1} \log \pi_{\theta_1}(a|s)] \tag{1.11}$$

While the Critic updates its parameters $\theta_2$ in the direction of:

$$c_1 \nabla_{\theta_2} \Omega\left(R - V^{\theta_2}(s)\right) \tag{1.12}$$

Where $\Omega$ usually is the L2 loss [Tena], and $c_1$ is a regularization constant used to keep the Critic learning rate lower than the Actor. Commonly $c_1 < 1$ (eg: $c_1 = \frac{1}{2}$).

# Chapter 2

# State-of-the-art for Actor-Critic

The Actor-Critic family can count many different algorithms including:

- A3C: asynchronous actor-critic [MBM+16a]
- A2C: synchronous actor-critic [SWD+17]
- DPG: deterministic policy gradient [SLH+14]
- DDPG: deep deterministic policy gradient [LHP+15]
- D4PG: distributed distributional DDPG [BMHB+18]
- MADDPG: multi-agent DDPG [LWT+17]
- TD3: Twin Delayed Deep Deterministic [FvHM18]
- TRPO: trust region policy optimization [SLA+15]
- PPO: proximal policy optimization [SWD+17]
- GAE: generalized advantage estimator [SML+15]
- ACER: actor-critic with experience replay [WBH+16]
- ACTKR: actor-critic using Kronecker-factored trust region [WMG+17]

- Soft Actor-Critic: incorporates the entropy measure of the policy into the reward to encourage exploration [HZAL18]
- IMPALA: Importance Weighted Actor-Learner Architecture for scalable Distributed Deep-RL and incredibly effective on multi-task reinforcement learning [ESM+18]

In the next sections we are going to summarize those actor-critic algorithms used in our experiments, but for more details about the other algorithms and their code implementations please refer to the cited papers or to [Wen18].

## 2.1 A3C

The Asynchronous Advantage Actor-Critic (A3C) algorithm [MBM+16a] is an on-policy technique based on Actor-Critic. A3C, instead of using an experience replay buffer, uses multiple agents on different threads to explore the state spaces and makes decorrelated updates to the Actor and the Critic. Hence, A3C is designed to work well for parallel training as shown in Fig.2.1.

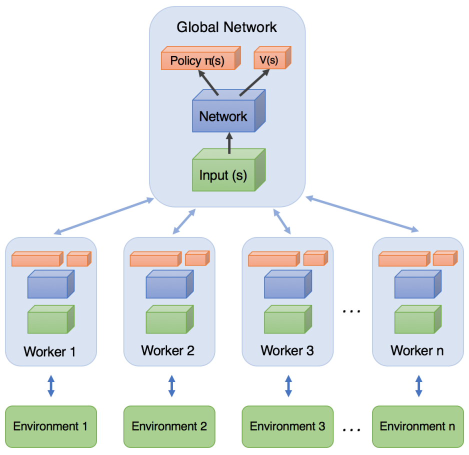


Figure 2.1: **State-of-the-art for Actor-Critic**: A3C overview

It is important to mention that A3C uses a different version of the policy gradient formula in order to better tackle the exploration problem (see Chapter 3 for more details on the exploration problem). Thus, in A3C, the Actor objective function is changed to:

$$J^{\mathrm{ON}}(\theta_1) = \mathbb{E}\left[A(s) \cdot \log \pi_{\theta_1}(a|s) - \beta S\left(\pi_{\theta_1}|s\right)\right] \tag{2.1}$$

Where $S\left(\pi_\theta|s\right)$ is the entropy of policy $\pi_\theta$ given state $s$, and $\beta$ is an entropy regularization constant.
A slightly different variation of A3C is A2C. In A3C each agent talks to the global parameters independently, so it is possible sometimes the parallel agents would be playing with different policies and therefore the aggregated update would not be optimal. The aim of A2C is to resolve this inconsistency. As shown in Fig.2.2, A2C uses a coordinator that waits for all the parallel actors to finish before updating the global parameters, for this reason A2C is said to be the synchronous version of A3C.
A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C. [SA18]

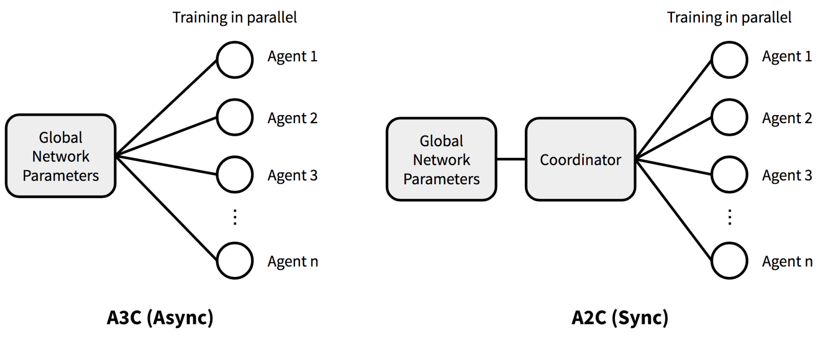


Figure 2.2: **State-of-the-art for Actor-Critic**: A3C vs A2C

## 2.2 GAE

The Generalized Advantage Estimation (GAE) is a method for computing the actor-critic advantage, reducing the variance but introducing some extra bias.
GAE is designed to achieve constant policy improvement despite the non-stationary incoming data. GAE uses value functions to reduce the variance of policy gradient estimates at the cost of some errors. GAE allows better control in the presence of spaces

of high dimensional actions. [SML+15]
In GAE, the advantage estimation is computed using a the k-step discounted sum of TD residuals $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$:

$$\hat{A}_t^{\text{GAE}(\lambda,\gamma)} = \sum_{l=0}^{\inf} (\lambda\gamma)^l \delta_{t+l}^V \tag{2.2}$$

This generalized estimator of the advantage function allows to tune bias and variance using the parameter $0 \leq \lambda \leq 1$ instead of parameter $\gamma$. For $\lambda = 0$ the problem reduces to the (unbiased) TD(0) function, while increasing $\lambda$ towards 1 the variance of the estimator reduces increasing the bias.

## 2.3 PPO

In order to improve training stability, we should avoid parameter updates that change the policy too much at one step. Trust region policy optimization (TRPO) [SLA+15] carries out this idea by enforcing a Kullback–Leibler divergence [KL51] constraint on the size of policy update at each iteration. While, Proximal Policy Optimization (PPO) [SWD+17] is a simpler variation of TRPO that prevents too big changes to the policy parameters by clipping them in a predefined range. PPO is much simpler to implement than TRPO, more general, and according to [SWD+17] it has empirically better sample complexity.
In order to understand PPO, first of all lets denote the probability ratio between old and new policies as:

$$r(\theta_1) = \frac{\pi_{\theta_1}(a|s)}{\pi_{\theta_1^{\text{old}}}(a|s)} \tag{2.3}$$

Then, the new Actor's objective function for PPO is:

$$\mathbb{E}\left[A(s) \cdot r(\theta_1) - \beta S(\pi_{\theta_1}|s)\right] \tag{2.4}$$

Let $\hat{r}(\theta_1) = \text{clip}\,(r(\theta_1), 1 - \epsilon, 1 + \epsilon)$, then the Actor objective function of PPO is:

$$J^{\text{PPO}}(\theta_1) = \mathbb{E}\left[\min\left(r(\theta_1) \cdot A(s), \hat{r}(\theta_1) \cdot A(s)\right) - \beta S(\pi_{\theta_1}|s)\right] \tag{2.5}$$

where $\epsilon$ is the clipping range hyper-parameter.
In PPO the Critic uses the same clipping technique used by the Actor, but instead of

keeping the minimum between the clipped and the non-clipped objective, it keeps the maximum. Let $\hat{V}^{\theta_2}(s) = V^{\theta_2{}^{\text{old}}}(s) + \text{clip}\left(V^{\theta_2}(s) - V^{\theta_2{}^{\text{old}}}(s), -\epsilon, \epsilon\right)$, the objective function of the Critic is:

$$J^{\text{PVO}}(\theta_2) = c_1 \max\left(\Omega\left(R - V^{\theta_2}(s)\right), \Omega\left(R - \hat{V}^{\theta_2}(s)\right)\right) \tag{2.6}$$

with $c_1 = 0.5$.

# Chapter 3

# Interesting methods to speed up Reinforcement Learning

Analysing the algorithms mentioned in Chapter 2 we can list a few building blocks or principles that seem to be common among them:

- Try to reduce the variance and keep the bias unchanged.
- Use Experience Replay (training data sampled from a replay memory buffer) for improving Sample Efficiency.
- Critic network is either frozen periodically or updated slower than the actively learned policy network.
- Entropy maximization of the policy helps encourage exploration.
- The critic and actor can share lower layer parameters of the network.
- Put constraint on the divergence between policy updates.

Some of the main bottlenecks, for RL, encountered in literature are:

1. Temporal resolution: expanding the unit of learning from the smallest possible step in the task.
2. Division-and-conquest: finding smaller sub-tasks that are easier to solve.

3. Exploration: exploring the whole state space.

4. Sample efficiency: use efficiently the already explored state space in order to reduce the need of re-exploring it again.

5. Structural generalization: find the most general solution.

All these bottlenecks are obviously related. For instance, altering the temporal resolution can have a dramatic effect on exploration. [DH93]
But how can we tackle the aforementioned problems? Some ideas from literature are:

- "Temporal resolution" and "division-and-conquest" may be tackled with hierarchical architectures (Section 3.4).

- "Exploration" may be improved using intrinsic rewards (Section 3.2)

- "Sample efficiency" may be obtained using experience replay (Section 3.1)

- "Structural generalization" may be probably tackled with unsupervised auxiliary tasks (Section 3.3) or more generally multi-tasking [ESM+18]

## 3.1 Experience Replay

Experience Replay [LM92] is actually a valuable and common tool for RL that has gained popularity thanks to Deep Q-learning [MKS+15].
The benefits coming from experience replay are:

- More efficient use of previous experience.

- Less sample correlation giving better convergence behaviour when training a function approximator.

Furthermore to improve learning performances it possible to prioritize experience, instead of sampling it uniformly, in order to replay important (eg: with more informative content) experience more frequently [SQAS15].

## 3.2 Intrinsic Rewards for Exploration

Improving exploration may be a complex task and can be achieved in several ways. Some of them requires changing the gradient formula in order to maximize policy entropy, but usually entropy regularization is not enough. Another interesting approach consists in giving intrinsic rewards in order to motivate exploration/curiosity. Intrinsic rewards are rewards that do not involve receiving feedback from the environment (the outside), in fact they are a feedback from the agent itself (the inside) and for this reason they completely differs from usual rewards in RL (the extrinsic rewards from outside). Motivation and curiosity were used both to explain the need to explore the environment and discover new goals, but also as a way to learn new skills that may be useful in the future. Most formulations of intrinsic reward for exploration can be grouped into two broad classes [PAED17]:

- Count-based: encourage the agent to move in unseen states.
- Curiosity driven: encourage the agent to perform actions that reduce the error/uncertainty in its ability to predict the consequence of its own actions.

#### 3.2.0.1 Count-based exploration

A typical count-based exploration reward function counts the occurrences of every state visited by the agent, and gives a reward proportional to the number of occurrences $\Gamma(s)$ of the current state of the agent. According to the classic count-based exploration theory, a typical count-based exploration reward function would be in the form:

$$R(s) = \frac{\theta}{\sqrt{\Gamma(s)}} \tag{3.1}$$

where $\theta$ is called bonus coefficient.
A simple generalization presented in [THF$^+$17] can reach near state-of-the-art performance on various high-dimensional and/or continuous deep RL benchmarks, simply mapping states to locality-sensitive hash (LSH) codes or domain-dependent learned hash codes. A good hash function for count-based exploration should:

1. Have appropriate granularity.

2. Encode information relevant for solving the MDP.

#### 3.2.0.2 Curiosity driven exploration

In curiosity driven exploration, intrinsic rewards are a feedback on how hard it is for the agent to predict the consequences of its own actions, or in other words to predict the next state given the current state and the executed action. An interesting example of curiosity driven exploration is described in [PAED17]. Differently from existing literature, [PAED17] proposes to only predict those changes in the environment that could possibly be due to the actions of the agent or affect the agent, and ignore the rest.

## 3.3 Unsupervised auxiliary tasks

Unsupervised Auxiliary Tasks (UAT) for improving RL have been probably explicitly introduced for the first time by [JMC+16a] and they consist in augmenting the standard deep reinforcement learning methods with additional tasks to perform during training. A very interesting architecture using UAT is [JCD+18], getting human-level performance in first-person multi-player games with population-based deep reinforcement learning.
The additional tasks described in [JMC+16a] are:

- Reward Prediction: the agent is trained to predict immediate rewards from a brief historical context, in which rewarding and non-rewarding histories are presented in equal proportions to better address the sparse reward scenarios.

- Pixel Control: designed to teach the agent to understand how actions affect changes in the pixels on the screen. This task seems closely related to the ideas described in Section 3.2.0.2.

But it is possible to define much more auxiliary tasks.

## 3.4 Hierarchical Deep Reinforcement Learning

Reinforcement Learning algorithms are based on the MDP framework, and they work well under the assumption that the Markov property is valid (read sec.1.1). But not all problems of practical interest respect Markov's property.
Hierarchical Reinforcement Learning (HRL) purpose is to overcome the RL limitation given by Markov property, for example by segmenting the action sequences in order to reduce non-markovian dependencies and look for appropriate configurations of long and short-term dependencies. [SS00, Bot12]
HRL is related to the idea of hierarchical planning in artificial intelligence [DFJ+81, Cor79]. In hierarchical planning, an abstract and generic plan (or solution) is proposed, which is then recursively reduced to simpler plans before being executed. This idea is also known in Computer Science as sub-problem decomposition.
Consider an example of an agent learning to go from "Venice" to "New York" using the primitive actions $[N, E, S, W]$. The agent is going to take a lot of time to converge, if at all it converges. But decomposing the problem into a more simple one using an abstract action set like:

- get enough money
- go to airport
- buy a ticket
- take a plane

the planning is going to become simpler. The abstract action set is different from the set of primitive actions, because they are compounded and are an abstraction of long sequences of primitive actions. This way of abstracting the action space should reduce the time taken for an agent to plan.
The most prominent framework to hierarchical reinforcement learning is the *options framework* [SPS99, SP02, BHP17]: a high level policy learns to choose between a fixed set of lower level policies, executing the chosen policy for several time steps. The lower policies a usually trained on simpler sub-tasks of the general problem. Alternative approaches include *feudal reinforcement learning* [DH93, VOS+17] where higher levels

reward lower levels, and *modulated controllers* [HWT+16] where higher levels provide context information to lower levels.
A *naive approach* to build hierarchies would be to derive their structure using an a priori knowledge of the domain. However, the problem with this approach is its lack of generality which impacts on the cost (because it is expensive to obtain an a priori knowledge of the domain), and makes such hierarchies inflexible (because the characteristics of the domain may change over time). [SS00] But in practice naive hierarchies may be very effective and very performing, sometimes resulting in the fastest solution; in our last conference paper [ACS18] we show an interesting example of Naive HRL based on a simple variation of the options framework.
Instead, a more *generic approach* than the aforementioned one would be to let the algorithm to autonomously learn an optimal hierarchy (of actions) from the problem itself, using a priori knowledge only in the form of some generic structure (eg: a fixed number of hierarchical levels). [SS00]
In Chapter 5 we show examples of Generic HRL based on the options framework and applied to video-games, robotic controllers and sentiment analysis.

### 3.4.1 Options framework

Options (or abilities or macro-actions) [SPS99, SP02] formalize the idea of actions occurring in extended time intervals, equipping agents with the ability to plan at different levels of temporal abstraction.
Much of the existing work on options is focusing on finding useful sub-goals that can be reached by the agent, and then learning policies to achieve them. [BHP17]
A recent work of Bacon et al. [BHP17] shows the possibility of learning options jointly with a policy-over-options in an end-to-end fashion by extending the policy gradient theorem to options. The architecture proposed in [BHP17] is shown in Fig.3.1 and it uses an Actor-Critic framework with an Actor for each option, an Actor for the option picker (or manager) and a single shared Critic network.

Usually, when options are learnt end-to-end, they tend to degenerate to one of two trivial solutions:

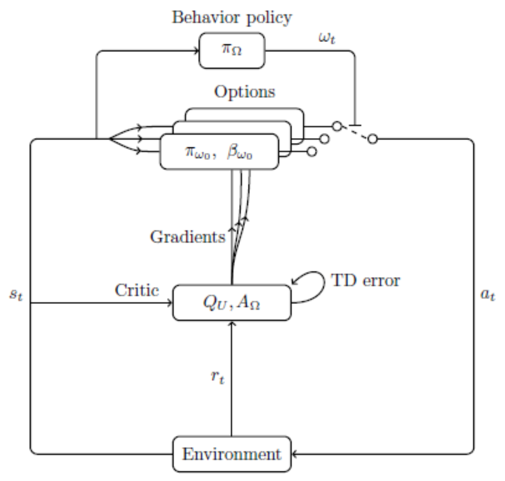


Figure 3.1: **Hierarchical Deep RL**: Option critic architecture

- only one active option that solves the whole task;
- a policy-over-options that changes options at every step

Consequently, regularization mechanisms [BHP17, VMO+16] are usually used to force the solution to have more options of extended length, and this is believed to help generalisation.

# Chapter 4

# Software framework used for experiments

Reinforcement Learning (RL) is an area of machine learning which studies how an *agent* should take *actions* in an *environment* (or problem) having the goal of maximizing some notion of *cumulative reward*.
Part of this thesis work is to experiment reinforcement learning techniques for automatic decision-making. In order to make these experiments we need to set up one or more training environments representing some interesting underlying problem that we can try to solve with our RL algorithms. For our experiments we want to tackle three different problems: Rogue, Sentiment Analysis and Car Controller.
In literature there are a lot of different RL algorithms, each of them with its own strengths and drawbacks, and in Chapter 2 we have seen some of them.
This thesis work is focused primarily on generic hierarchical architectures based on the actor-critic paradigm, but we also want to make some experiments on Experience Replay (Section 3.1) and Intrinsic Rewards for Exploration (Section 3.2). Thus, we need a software framework we may use for experimenting the algorithms of interest, in all the required environments.
In Section 4.1 we describe the chosen environments, while in Section 4.2 we briefly introduce the adopted software framework.

## 4.1 Environments

We can say there are two main classes of RL problems, those who have a *discrete action space* and those who have a *continuous action space*. We are interested in both of them.
The discrete action-space environments we have chosen are:

- Rogue, a video-game
- SENTIPOLC 2016, a dataset for sentiment analysis

while the continuous action-space environment is:

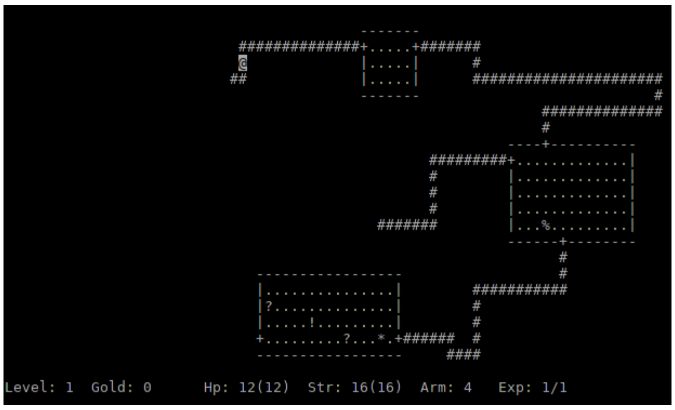
- Car Controller, a simulator for autonomous driving

### 4.1.1 Discrete action space

In discrete action space the agent policy is usually modelled with a categorical distribution, for example using a softmax function [Tenb].

#### 4.1.1.1 Rogue

Rogue is a complex dungeon-crawling video-game of the eighties, the first of its gender. The player (the “rogue”) is supposed to roam inside a dungeon structured over many different levels, with the final aim to retrieve the amulet of Yendor, coming back to the surface with it. If the player dies, the game restarts from scratch (permanent death); moreover, all levels are randomly generated and different from each other.
During his quest, the player must perform many different activities: explore the dungeon (partially visible), when he enters a new level; defend himself from enemies, using the items scattered through the dungeon; avoid traps; prevent starvation, finding and eating food inside the dungeon.
In its standard configuration, the dungeon consists of 26 floors and each floor consists of up to 9 rooms of varying size and location, randomly connected through non-linear corridors, and small mazes. To reach a new floor the rogue must find and go down the stairs; the position of the stairs is different at each floor, likely located in a yet unexplored room, and hence hidden from sight at the beginning of exploration.



```
Level: 1  Gold: 0      Hp: 12(12)  Str: 16(16)  Arm: 4   Exp: 1/1
```

Figure 4.1: **Software framework**: The two-dimensional ASCII-based graphical interface of Rogue

#### 4.1.1.2 SENTIPOLC 2016

The environment SENTIPOLC 2016 is a dataset made for sentiment analysis classification and used in the EVALITA 2016 competition. [BBC+16] This environment is completely different from the previous one, mainly because it is not intuitive how to model the sentiment polarity classification as a Markov Decision Process.
Anyway, Sentiment Analysis (SA) is the application of analytical techniques to extract subjective information from documents written in natural language. Some of the main tasks in Sentiment Analysis are:

- *Polarity Classification*: classification of the polarity (positivity, negativity, neutrality) of a text at document granularity.
- *Subjectivity Classification*
- *Irony Detection*

More details about these tasks may be found in [BBN+14].
SENTIPOLC 2016 is composed by approximatively 9000 **Italian tweets** from Twitter, annotated with a tuple of boolean values representing subjectivity, irony and polarity (positive and/or negative) of the document. The training set is made of about 7000 tweets, while the test set is composed by 2000 tweets.

### 4.1.2 Continuous action space

Learning in real-world domains often requires to deal with continuous state and action spaces. [LHP+15] In continuous action space the policy is usually modelled with a normal distribution.
Continuous action space problems are usually addressed using policy-based or actor-critic algorithms, because finding the greedy policy in continuous spaces with value-based algorithms requires an optimization of the action $a_t$ at each time-step $t$, and this optimization on continuous spaces may become too expensive from a computational point of view. [LHP+15]

#### 4.1.2.1 Car Controller

Car Controller is a new environment we made for training Deep RL networks to autonomously drive a car on a road with obstacles.
We decided to represent roads using the same data structure adopted by OpenDrive format [e.a]. OpenDRIVE is an open file format for the logical description of road networks. It was developed and is being maintained by a team of simulation professionals with large support from the simulation industry.
In OpenDrive format and in this environment, roads are efficiently represented using cubic splines [Wol]. A cubic spline is a third-order polynomial $Y(x) = a + bx + cx^2 + dx^3$ with $x \in [0, 1]$. Cubic splines are used to generate control points. If we want to generate $N$ consecutive and different control points, than we can simply sample $N$ different and consecutive $x_n$ for each integer $0 \leq n < N$.
More in detail, every road is composed by one or more spline sets (pieces of road). Every spline set is made of exactly two cubic spline functions: one for generating the $x$ coordinates of the control points and the other one for generating the $y$ coordinates.
One of the main advantages of representing roads with cubic splines is that, in this way, it is very simple to generate random paths. In fact, for a more robust training we need to train the agent on random roads, and representing roads with cubic splines make us super easy to generate random paths, because we just need to stochastically generate the 8 parameters of the spline sets with very few constraints, instead of generating directly the $n$ control points.

The Car Controller environment is a simplistic model at its first stage of development. For now and for this reason we decided to represent obstacles with the following limitations:

- Obstacles cannot move.
- Obstacles are circles.
- Obstacles are spawned only in the middle of the road.
- Only a few number of different obstacles is simultaneously spawned.

We also represented the car with the following limitations:

- The car is a point.
- The car has an heading vector and can control only its acceleration and steering angle during every step of the simulation.
- The car cannot move in backward direction (it has to perform a U-turn instead).
- The car is able to see (roads, obstacles) only up to 1 meter in forward direction.

The Car Controller environment randomly generates:

- Sequences of cubic splines for representing roads.
- A fixed random number of immobile obstacles on the road represented by circles with random radius.

A simulation step is taken in average every $\mu = 0.1$ seconds, where $\mu$ is the mean of an exponential distribution. At every step, a random noise is added to the chosen car action in order to realistically simulate actuation errors.
The car goal is to follow the road, avoid obstacles, and not exceed a given maximum speed limit.
In Fig.4.2 we show an example of car overtaking random obstacles on a random road in the Car Controller environment.

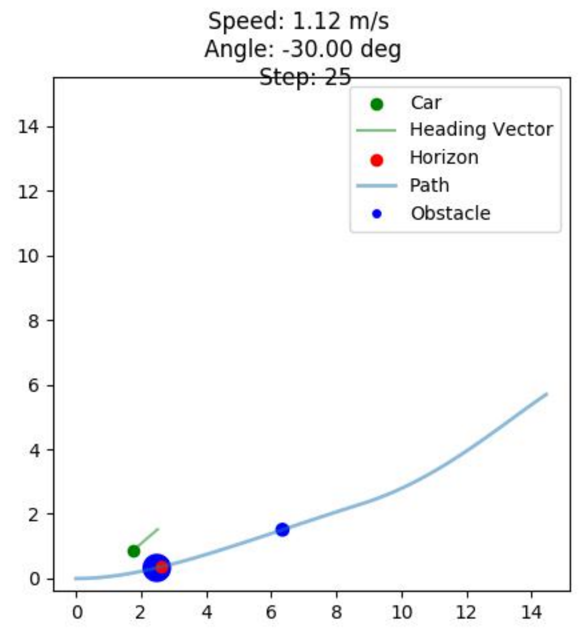


Figure 4.2: **Software framework**: Example of car overtaking random obstacles on a random road in the Car Controller environment

## 4.2 Framework

As we have seen in Chapter 2, Actor-Critic is a big family of RL algorithms. In this work we want to focus primarily on:

- Actor-Critic paradigm
- Hierarchical networks
- Experience Replay
- Exploration intrinsic rewards

In order to do so, we need a framework for experimenting RL algorithms, with ease, on different types of problem.
In May 2017, OpenAI has published an interesting repository of RL baselines [DHK$^+$17], and it is still maintaining it with continuous improvements and updates. The aforementioned repository is probably the best choice for testing the performances of already

existing RL algorithms requiring very minimal changes, but it is hard to read and modify (at least for the author).
Thus we decided to use as code-base for our experiments the open-source A3C algorithm, built on Tensorflow 1.10.1 [ABC+16], that comes with our last conference paper [ACS18], mainly because we already had experience with it and we know the details of its inner mechanisms. But even the chosen code-base is not generic and abstract enough for our goals, for example that code-base is made for Rogue only, thus we had to make some changes to it:

1. We created a unique configuration file in the Framework root directory, for configuring and combining with ease (in a single point) all the Framework features, algorithms, methods, environments, etc.. (included those that are going to be mentioned in the following points of this enumeration)

2. Added support for all the Atari games available in the OpenAI Gym repository [BCP+16].

3. Created a new environment for Sentiment Analysis (Section 4.1.1.2).

4. Created a new environment for Car Controller (Section 4.1.2.1).

5. Added support for A2C (Section 2.1).

6. Added Experience Replay and Prioritized Experience Replay (Section 4.2.1).

7. Added Count-Based Exploration (Section 4.2.2).

8. Added PPO and PVO (Section 2.3).

9. In many OpenAI baselines the vanilla policy and value gradient (see eq.1.9) has been slightly modified in order to perform a reduce mean instead of a reduce sum, because this way it is possible to reduce numerical errors when training with huge batches. Thus, it has been added support for both mean-based and sum-based losses.

10. Added GAE (Section 2.2).

11. Added support for all the gradient optimizers supported by Tensorflow 1.10.1: Adam, Adagrad, RMSProp, ProximalAdagrad, etc..
12. Added support for global gradient norm clipping and learning rate decay using some of the decay functions supported by Tensorflow 1.10.1: exponential decay, inverse time decay, natural exp decay.
13. Added different generic hierarchical structures (Section 5.2.5 and 5.1.10) based on the Options Framework (Section 3.4.1) for partitioning the state space using:
    - K-Means clustering [HW79]
    - Reinforcement Learning
14. Made possible to create and use new neural network architectures, simply extending the base one. The base neural network, by default, allows to share network layers between the elements of the hierarchy: parent, siblings.
15. In order to simplify experiments analysis, it is required a mechanism for an intuitive graphic visualization. We implemented an automatic system for generating GIFs of all episodes observations, and an automatic plotting system (Section 4.2.3) for showing statistics of training and testing. The plotting system can also be used to easily compare different experiments (every experiment is colored differently in the plot).
16. Added support for continuous control (Section 5.3.3).
17. Added support for multi-action control (this the same policy network can learn to perform simultaneously more than one action).

The source-code of the aforementioned software framework is available at [Sova].

### 4.2.1 Experience Replay

Experience Replay (ER) is a technique described in 3.1 used to turn an on-policy algorithm to off-policy, improving sample efficiency. We have seen in Section 1.3.2.2 that the off-policy gradient works using importance weights. In our naive implementation of ER we are going to consider only importance weights of 1, thus practically using the on-policy gradient instead of the off-policy one.

ER works using a circular buffer called Experience Buffer. This buffer has a fixed size and it is fed with batches until it is completely full. When the buffer is full, the old batches are replaced using a FIFO (First In First Out) policy.

In this particular implementation of ER it is possible to group batches using a key specified during batch insertion into Experience Buffer. During sampling operation, a group $g$ is chosen from the groups set $G$ with probability $\frac{1}{|G|}$ and a batch $b \in g$ is sampled with probability $\frac{1}{|g|}$. In all our experiments we usually separate batches into two groups using the following disequation as guard $R > 0$ where $R$ is the batch cumulative non-intrinsic reward.

Furthermore, it has been added an ad-hoc mechanism for filtering the batches to add in the experience buffer in order to sample only those considered the most important. When this mechanism is activated, only those batches with $R \neq 0$ are added to the Experience Buffer.

The replay frequency is defined by a Poisson distribution with mean $\mu$ (the replay ratio constant). At every step, $k$ replay batches are randomly sampled from the experience buffer, where $k$ is sampled from the aforementioned Poisson distribution.

In our software framework we also added support for Prioritized ER. The idea behind Prioritized ER is that using a prioritized buffer it is possible to specify a sampling priority. In this particular implementation of Prioritized ER, to each buffer batch is added a numerical key (the priority) and the whole buffer is kept ordered with respect to that key. During sampling operation, we sample a random number $z$ lower than the sum of all the priorities, then the first batch having prefix sum greater or equal than $z$ is taken.

### 4.2.2 Count-Based Exploration

In this particular implementation of Count-Based Exploration (CBE), the states are mapped to hash codes $h$ using Locality-Sensitive Hashes (LSH) which allows to count their occurrences with a hash table $T$. The LSH algorithm is implemented using a Sparse Random Projection (SRP) [LHC06, Ach01] of $k$ components, where $k$ is exactly the size

of the resulting hash. An efficient implementation of SRP is available in Scikit-Learn library [PVG+11].
The SRP is trained every $n$ steps using the states seen during these steps. This means that the projection may change every $n$ steps producing different hash representations for the same state $s$ (eg: at step $n$ and at step $2n$).
Assuming that within an episode the projection cannot be trained, the hash table $T$ is reset every episode.
Let $r = \frac{2}{\sqrt{T[h]}} - 1$ be the adopted count function, then the final CBE reward is computed according to the following equations:

- if $r > 0$, then CBE reward is $r \cdot p$
- if $r <= 0$, then CBE reward is $r \cdot n$

Where $p$ and $n$ are respectively the positive and negative exploration coefficients.

### 4.2.3 Statistics for evaluation metrics

The plotting system, briefly introduced a while ago in Section 4.2, is generic enough to plot any desired statistic just adding it in a statistics dictionary. There are some statistics that may be common to all experiments, and in this section we are going to describe them.
In all the experiments of this thesis we are going to use some variants of A3C trained on $t$ threads (usually 4 or 8), thus the common statistics may be:

- "avg_reward": average extrinsic cumulative reward
- "loss_actor_avg": average actor loss
- "loss_critic_avg": average critic loss
- "loss_actor_clipping_frequency_avg": the average clipping frequency of the actor, when using PPO loss
- "loss_actor_entropy_contribution_avg": the average entropy contribution to the total loss

- “loss_actor_kl_divergence_avg”: the average Kullback-Leibler divergence of the actor policy

All the aforementioned averages are the average of the average of $n$ episodes statistics of $t$ threads. For example, let $s_i^j$ be the episode statistic of the last i-th episode of thread $j$, then the final plotted average for $s$ is:

$$\frac{\sum_{j=0}^{t} \frac{\sum_{i=0}^{n} s_i^j}{n}}{t} \tag{4.1}$$

In all the experiments we set $n = 200$.
In the case of hierarchical architectures if the architecture has more actors and critics, then the related actor and critic statistics are computed separately for each actor and critic. For example, if the hierarchy is composed by two A3C networks then we will have “loss_actor0_avg”, “loss_actor1_avg”, etc..

# Chapter 5

# Experiments on Generic Hierarchical Reinforcement Learning

As the name may suggest, Generic HRL is the application of generic hierarchies in reinforcement learning. Differently from *naive* hierarchies, *generic* hierarchies do not use a priori knowledge on the underlying problem and for this reason they are considered much more interesting and powerful, but more difficult to properly design.
In this chapter we are going to describe three different experiments, on three different environments and neural networks, using Generic HRL techniques based on simple variations of the Options Framework. Our goal is to investigate the benefits of HRL over simple RL.
In all the experiments the hierarchical architecture is made by an A3C partitioner able to partition the state space into *n* different sub-spaces for respectively *n* different A3C workers.
Our first experiment (Section 5.1) is conducted on Sentiment Analysis using the SENTIPOLC 2016 environment described in Section 4.1.1.2, and we show that in this environment the benefits of HRL may be significant. The main challenge of this experiment was to model the underlying problem as MDP, in order to use the RL framework.
Our second experiment (Section 5.2) is conducted on the Rogue environment introduced in Section 4.1.1.1, and we show that in the chosen environmental settings the benefits of HRL do not seem to be significant. The main challenges of this experiment were prob-

ably to choose a good state representation, to tackle the exploration problem mentioned in Chapter 3, and to find a useful generic hierarchical architecture.
Our last experiment (Section 5.3) is conducted on the Car Controller environment described in Section 4.1.2.1, and we show that in this particular environment there are no evident benefits in using HRL. The biggest challenge of this second experiment was probably to understand how to model a policy for continuous action-spaces using a normal distribution, and in order to do that we had to investigate different interpretations (from math and information theory) on the nature of some components of the policy gradient equation.

## 5.1 Experiment 1 - Sentiment Analysis

Sentiment analysis is the application of analytical techniques to extract subjective information from documents written in natural language.
Given its complexity, historically sentiment analysis has been a difficult task to automate. In fact, the accuracy of a sentiment analysis system is, in principle, how well it agrees with human judgements. However, according to research, human critics typically only agree about 80% of the time. [KNP13]
Recently, the use of deep learning techniques has allowed the creation of state-of-the-art classifiers for the sentiment polarity classification and other sentiment analysis tasks [NRR+16, RFN17, BBC+16].
In this experiment we deal with the problem of sentiment analysis using an approach that involves deep reinforcement learning. The idea is to model sentiment analysis in the form of a Markov Decision Process through the representation of an emotional state associated with the reading (even partial) of a document. Conceptually, the process adopted is inspired by the natural way a person performs sentiment analysis.
Human-like learning seems intriguing and, in addition, reinforcement learning can easily process texts of arbitrary length in a word-by-word manner. The episode-based and flexible structure of this method can handle highly complex sentences and thus appears to be particularly well-suited for sentiment analysis. [PFN16] Thanks to reinforcement learning it has been possible to teach the agent to properly stochastically understand

how sequences of words in a document affect the perception of sentiment expressed by the document itself, thus allowing an analysis much finer and more detailed than the applications discussed in [BBC+16, NRR+16].
The source code used for this experiment is publicly available at [Sovb].

### 5.1.1 Related Work

As far as the author knows, Reinforcement Learning (RL) approach is only rarely applied to the domain of computational linguistic.
The most related work seems to be [PFN16]. In this research about sentiment analysis, the goal is to detect negation scopes in financial news in order to enhance the measured accurateness of sentiment. To do so, they used an *off-policy RL algorithm* as Q-learning to invert (or not) the *precomputed sentiment* polarity of a word.
Some of the state-of-the-art applications for sentiment analysis are resumed in [BBC+16, NRR+16]. What is in common between them is:

- The adoption of deep *supervised* learning techniques.
- The geometrical representation of words/sentences through Word Embeddings, mostly derived by using the Word2Vec algorithm [MSC+13] or similar methods.
- The use of additional word features derived from Distributional Polarity Lexicons.

### 5.1.2 Sentiment Analysis as Markov Decision Process

In order to make use of Reinforcement Learning, it is required to model the Sentiment Analysis problem in the form of a Markov Decision Process (MDP).
The idea is to model sentiment analysis in the form of a MDP through the representation of an emotional state associated with the reading (even partial) of a document. Conceptually, the process adopted is inspired by the natural way a person performs sentiment analysis: the *reader* reads the document (eg.: from left to right) and for each observed and contextualized *token* it changes (or not) its emotional state according to the most probable contribution of that token to the overall perceived sentiment for the document.
We call *sentidoc* a vector representing the agent emotional state related to the ordered

sequence of tokens composing a document. The 1st token of the sequence is always the 1st token of the document, the 2nd token of the sequence is the 2nd token of the document, and so on.
Let $k_t$ be the token currently reading at time step $t$, and let $\sigma_t$ be the estimated partial sentidoc up to $k$, we have that:

- A *state* at $t$ is defined by $k_t$ and by $\sigma_t$. This way, the sentidoc is part of the environment.

- An *action* at $t$ is defined as a decision to change (or not) $\sigma_t$. For this reason, the environment is affected by agent actions as required by RL.

- A *reward* at $t$ is given by a function of the similarity between $\sigma_t$ and the overall sentiment annotation (eg: if the sentidoc is different from the one noted, then the agent receives a negative reward)

In this setting, during the learning phase, the overall perceived sentiment for the documents in the training set should be known a priori. Thus, an annotated dataset is required for training and testing, for this reason we are going to use the dataset of SENTIPOLC 2016 by Evalita [BBC+16] described in Section 4.1.1.2.
Anyway, let's go back for a moment to the contextualization problem. The dependency between a token $k_t$ and all the other tokens $k_{t'}$ with $t' < t$ is clearly a non-markovian temporal dependency. For this reason we decided to approach the Sentiment Analysis problem with Hierarchical Reinforcement Learning instead of simple Reinforcement Learning.
Until now we have defined the state space of the Sentiment Analysis MDP, but an MDP needs also a reward function. The reward function indicates how well the learner is behaving while not explicitly detailing how to improve its behaviour. Thus, we may model the reward function as a function of the similarity between $\sigma_t$ and the overall sentiment annotation, for example we may use the F1 score [Wik18c] or the Matthews Correlation Coefficient (MCC) [Wik18d] or the cosine similarity [Wik18a], etc..

### 5.1.3 Architecture

Sentiment analysis goal is to extract subjective information from documents written in natural language, this implies that the reinforcement learner (the actor) has to learn a stochastic policy that might find in agreement the most of the human critics. It is interesting to notice the analogy between the aforementioned goal description and the Actor-Critic framework.
In this section we describe a novel hierarchical architecture made by an A3C partitioner able to partition the state space into $n$ different sub-spaces for respectively $n$ different A3C multi-task learners. All the $n$ A3C multi-task learners contributes to build a common cumulative reward without sharing any other information, and for this reason they are said to be highly independent. Each agent employs the same architecture, state representation and reward function.
This architecture is shown to be able to give better results, in Polarity Classification and Subjectivity Detection for the Italian language on the SENTIPOLC 2016 dataset, than those described in [BBC+16].
In the following sections we are going to discuss: the document preprocessing (Sec. 5.1.4), the state representation (Sec. 5.1.5), multi-task learning (Sec. 5.1.6), the neural network (Sec. 5.1.7), how we shaped the reward function (Sec. 5.1.8), how we tackled the overfitting problem deriving from the small size of the dataset (Sec. 5.1.9), the hierarchical architecture (Sec. 5.1.10) and the adopted hyper-parameters (Sec. 5.1.11).

### 5.1.4 Document Preprocessing

The adopted deep learning method requires to model the documents through a geometric representation (eg: vectors).
The document preprocessing can be decomposed in the following steps:

1. Tokenization.

2. Part-of-Speech Tagging (and lemmatization).

3. Features extraction from Distributional Polarity Lexicons.

4. Word embedding.

At step 1 the document is decomposed into a list of semantic units called tokens (emojis, verbs, adjectives, etc..). At step 2, to each token is associated the corresponding part-of-speech tag (postag) and lemma. At step 3, both the lemmas and the postags are used with a bunch of lexicons to retrieve:

- The lemma *polarity*: the frequency of occurrences of the lemma inside positive/negative sentences.
- Whether the lemma is a *negator*: a token that negates the meaning of other tokens.
- Whether the lemma is a *intensifier*: a token that intensifies the meaning of other tokens.
- Whether the lemma is a *stop word*: a very commonly used token such as “a”, “and”, “in”, etc..

Finally, at step 4 the lemmas are mapped to vectors using a technique called “word embedding” [GRMJ15]. Word embedding is a type of mapping that allows words with similar meaning to have similar representation. The basic idea behind word embedding (and distributional semantics) can be summed up in the so-called distributional hypothesis [Sah08]: linguistic items with similar distributions have similar meanings; words that are used and occur in the same contexts tend to purport similar meanings.
At step 1, a combination of the Tweet Tokenizer and the Moses Tokenizer has been used. The Tweet Tokenizer [NLT] is not designed for Italian thus it is used only to identify emojis, hashtags, URLs and e-mails, while the Italian Moses Tokenizer [Mos] is used to identify the remaining tokens.
At step 2 the Italian TreeTagger [Sch95] has been used as PoS-tagger and lemmatizer.
Step 3 is probably the most complicated. This step can be decomposed into 4 different sub-steps:

1. All the emojis are converted into their unicode representation. The unicode representation is used to search the distributional polarity inside [NSSM15]. If the aforementioned search fails, then the polarity of the English emoji shortcode tokens (retrieved from SentiWordNet [BES10]) is used.

2. The NLTK library [BL04] is used to know whether the lemma is a stop word.

3. The lemmas of the tokens that are not emojis, hashtags, URLs, e-mails and stop words are automatically translated into English using the Google Translator. The resulting translation is used to search the distributional polarity in SentiWordNet [BES10].

4. The OpeNER Italian lexicon [Izq] is used to know whether the lemma is a negator or an intensifier.

At step 4, fastText [GBG+18] has been used as word embedder because it provides a good-enough model pre-trained on the Italian Wikipedia and Common Crawl [Com]. This pre-trained model represents a word embedding with a vector of 300 real numbers. Empirically, it has been observed that multiplying the vectors by a scale factor of 100 allows the network to better distinguish features, thus leading to better results.

### 5.1.5 State representation

In this setting a document is an episode and it is represented by its ordered list of tokens plus a special token at the end of the list to represent the entire document. The lemma embedding of the special token (the last one) is the average of the lemma embeddings of each other token in the document. Each token is represented using the concatenation of its lemma embedding and the following features extracted during document preprocessing:

- negativity: real number in $[0, 1]$
- positivity: real number in $[0, 1]$
- is negator: integer representation of a boolean, multiplied by 2
- is intensifier: integer representation of a boolean, multiplied by 4
- is emoji: integer representation of a boolean, multiplied by 8
- is last token: integer representation of a boolean, multiplied by 16

Thus, the overall size of a each token representation is 300 + 6 = 306.
In Sec. 5.1.2 we gave the definition of *state*. A state at step $t$ is defined by the token representation $k_t$ and by the sentidoc $\sigma_t$. We assume that stop words, e-mails and URIs do not influence the overall sentiment of a document, thus only the remaining tokens are taken under consideration.
The sentidoc is a vector of size $p$, where $p$ is the number of different features composing the emotional state. In other terms, $p$ is the size of the vectorial representation of the sentiment annotation of the document.
In Sec. 5.1.6 more details are given about the sentidoc representation.

### 5.1.6 Multi-Task Training

An annotation from the Evalita dataset for SENTIPOLC 2016 is a vector made of 6 boolean values representing:

- subjectivity
- irony
- overall positivity
- overall negativity
- literal positivity
- literal negativity

Each boolean value can be associated to a different task of binary classification. Many participants to the SENTIPOLC competition [BBC+16] have trained a separate binary classifier for each task. But in our setting this approach seems to lead to poor results.
We used Multi-task Learning over all the 6 tasks represented by the boolean values of a document annotation. Thus, the chosen representation for sentidoc is a vector of 6 boolean, one for each task.
Multi-task Learning is an approach to inductive transfer that improves generalization by using the domain information contained in the training signals of related tasks as an

inductive bias. It does this by learning tasks in parallel while using a shared representation; what is learned for each task can help other tasks be learned better. [Car98].

### 5.1.7 The neural network

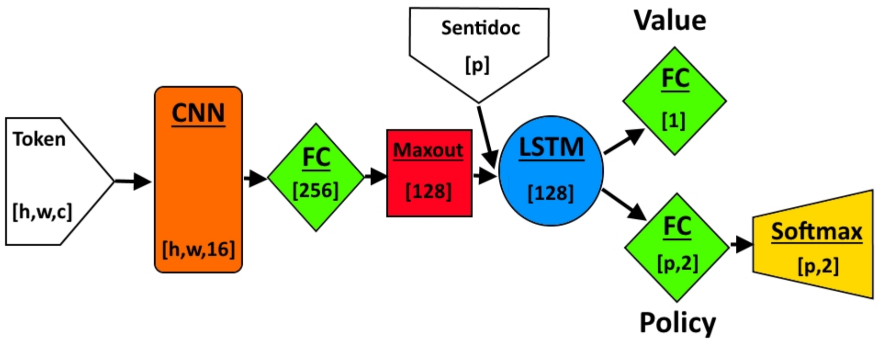


Figure 5.1: **Experiment 1 - Sentiment Analysis**: Neural networks architecture

The neural network architecture we used is shown in Fig. 5.1. This network consists of a convolutional layer followed by a dense layer to process spatial dependencies and a LSTM layer to process temporal dependencies, and finally, value and policy output layers. The CNN has a RELU, a $1 \times 3$ kernel with unitary stride and 16 filters. The CNN output is flattened and it is the input of a FC without activation function and 256 units followed by a Maxout layer [GWFM+13] with 128 units. We call this structure: tower.

At step $t$, the tower input is the the token representation $k_t$ described in 5.1.5 and its output is concatenated with the sentidoc $\sigma_t$. This concatenation is fed into an LSTM composed of 128 units.

The output of the LSTM is then the input for the value and policy layers.

A network with the aforementioned structure implements an agent for each situation described in 5.1.9. The loss is computed separately for each network, and corresponds to the A3C loss computed in [JMC+16a].

The output size of each layer is shown between brackets in figure **??**. More in detail, each $k_t$ has height $h = 1$, width $w = 306$ and $c = 1$ channels. Because we are performing multi-task learning on binary classes, the output size of the Actor is $[p, 2]$, where $p = 6$.

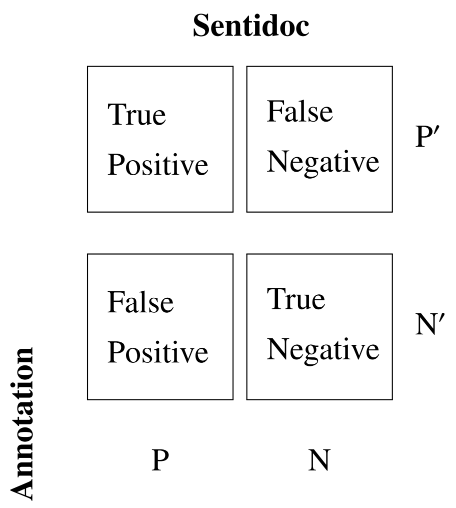


Figure 5.2: **Experiment 1**: Example of confusion matrix; in predictive analytics, a table of confusion (sometimes also called a confusion matrix), is a table with two rows and two columns that reports the number of false positives, false negatives, true positives, and true negatives.

### 5.1.8 Reward shaping

In Sec. 5.1.2 we gave the following guidelines on how to choose a good reward function for this environment: a *reward* at step $t$ should be given by a function of the similarity between $\sigma_t$ and the overall sentiment annotation.
Both the sentiment annotation and the sentidoc are vectors of booleans. Each boolean represents a binary class for a particular task. The similarity between a sentidoc and an annotation may be calculated using a confusion matrix for each task. An example of confusion matrix is shown in Fig. 5.2.

In order to avoid sparse rewards (and all the related problems mentioned in the previous chapters), we shaped the reward function not to give rewards only when a final state is reached (when the episode ends), but to give rewards also in intermediate states. We designed and tested many different reward functions. Experimentally we found that the reward function that seems to give better results pushes the agent to prefer a false positive to a false negative. A plausible reason why this approach works in practice is that in most classes the amount of positives is significantly less than the amount of

negatives. In fact, by giving equal rewards for both true positives and true negatives, the agent would learn to underestimate the positives because in the dataset they are much less frequent than the negatives. For more details on the distribution of positives in the various classes of SENTIPOLC 2016, please refer to [BBN+14].
Anyway, until now the best reward function found for the neural network described section 5.1.7 is:

- $\sum_1^p \left(0.05 \cdot tp_p + -0.05 \cdot fn_p\right)$ if the state is *not final*
- $\sum_1^p \left(tp_p + 0.1 \cdot tn_p - fp_p - 0.1 \cdot fn_p\right)$ if the state is *final*

Where $tp, tn, fp, fn$ are integer representations of boolean values respectively representing the presence of a true positive, true negative, false positive, false negative.

## 5.1.9 Over-fitting mitigation

Over-fitting occurs when the agent learns to solve the problem only for the data on which it has been trained and not also for the other possible data. If the training dataset is not big enough, then over-fitting may be a big problem. In SENTIPOLC 2016 the training dataset is made only of approximatively 7000 tweets, thus to mitigate the over-fitting problem we:

- Try to keep the neural network size relatively small.
- Perform multi-task training.
- Use a Maxout layer.
- Process the training set by epochs (partitioning it between the A3C threads).
- Randomly shuffle the training set at each epoch,

## 5.1.10 Hierarchical Structure

As mentioned in Section 5.1.2 and 5.1.3, for approaching Sentiment Analysis with Reinforcement Learning we use a novel hierarchical architecture based on the Options

Framework (see Section 3.4.1) and made by an A3C partitioner for optimally partitioning the state space into $n$ different sub-spaces for respectively $n$ different A3C multi-task situational agents.
Every situational agent has the same reward function described in Section 5.1.8, and contributes to build a common cumulative reward without sharing any other information with other agents except the value estimation used for bootstrapping.
The partitioner is an unsupervised classifier able to decide at run-time which A3C agent should process a state, and its goal is to maximize the overall cumulative reward, distributing the states between the available agents thus specializing themselves to solve sub-problems (said situations) of the original problem.
For each state, exactly two A3C networks can be simultaneously active: the partitioner and the agent chosen by the partitioner. Each A3C network is trained only on the states in which it has been activated.
Let $v_P$ be the value estimate of the Partitioner $P$ and $v_A$ be the value estimate of the Agent $A$ chosen by the partitioner for a particular state $s$. When building the training batch, the value that is more distant from 0 between $v_A$ and $v_P$ is used in order to produce the same bootstrapped cumulative reward and advantage for both $A$ and $P$. This way, better convergence properties have been empirically observed.
This work on unsupervised partitioning of the state space may be seen as the generic version for Sentiment Analysis of the A3C naive hierarchical architecture described in [ACS18].

### 5.1.11 Hyper-parameters tuning

Let $n$ be the number of partitions in which the state space is divided by the partitioner, then the number of A3C models composing the overall network is $n + 1$ (1 manager + n workers). The best value for the hyper-parameter $n$ in this particular environment has been empirically found to be 3.
At the very start of the training phase the situational agents workload is balanced. During the training, the workload changes very frequently and (usually) at the end of the training one of the situational agents is used around 70% of the times, another one is used 30%, and the last one is left unused, as shown in Fig. 5.3.

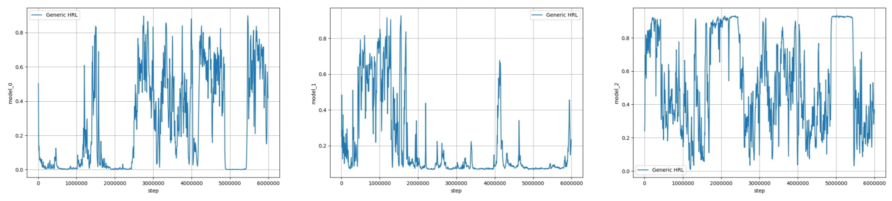


Figure 5.3: **Experiment 1 - Sentiment Analysis**: Hierarchical Structure workload

All the adopted hyper-parameters came from [Miy, ACS18], except the following:

- discount factor $\gamma$: 0.99
- batch size $t_{max}$: 5
- entropy $\beta$: 0.001 for the partitioner and $0.001 \cdot i$ for the i-th situational agent

An important fact to highlight is that every situational agent has a different entropy regularization constant $\beta$ because it has been empirically observed that this significantly improves the overall stability of the network. Also, in this experiment we are not going to use Experience Replay, nor count-based exploration rewards.
We employed the same Tensorflow's RMSprop optimizer available in [Miy, ACS18], with parameters:

| | |
|---|---|
| **decay** | 0.99 |
| **momentum** | 0 |
| **epsilon** | 0.1 |
| **clip norm** | 40 |

The initial learning rate is approximatively $\eta = 0.0007$. The learning rate is annealed over time according to the following equation:

$$\alpha = \phi_m \eta \frac{T_{max} - \chi_m}{T_{max}} \tag{5.1}$$

where $T_{max} = 3 \cdot 10^6$ is the maximum global step, $\chi_a$ is the number of times an A3C model $m$ is used, and $\phi_m$ is the regularization factor for $m$. The factor $\phi$ for the partitioner is 1 and for the other models is $\frac{1}{n}$. This way the partitioner learning rate is always the greatest, and empirically this seems to lead to better performances for the overall network.

### 5.1.12 Experiment Results and Evaluation

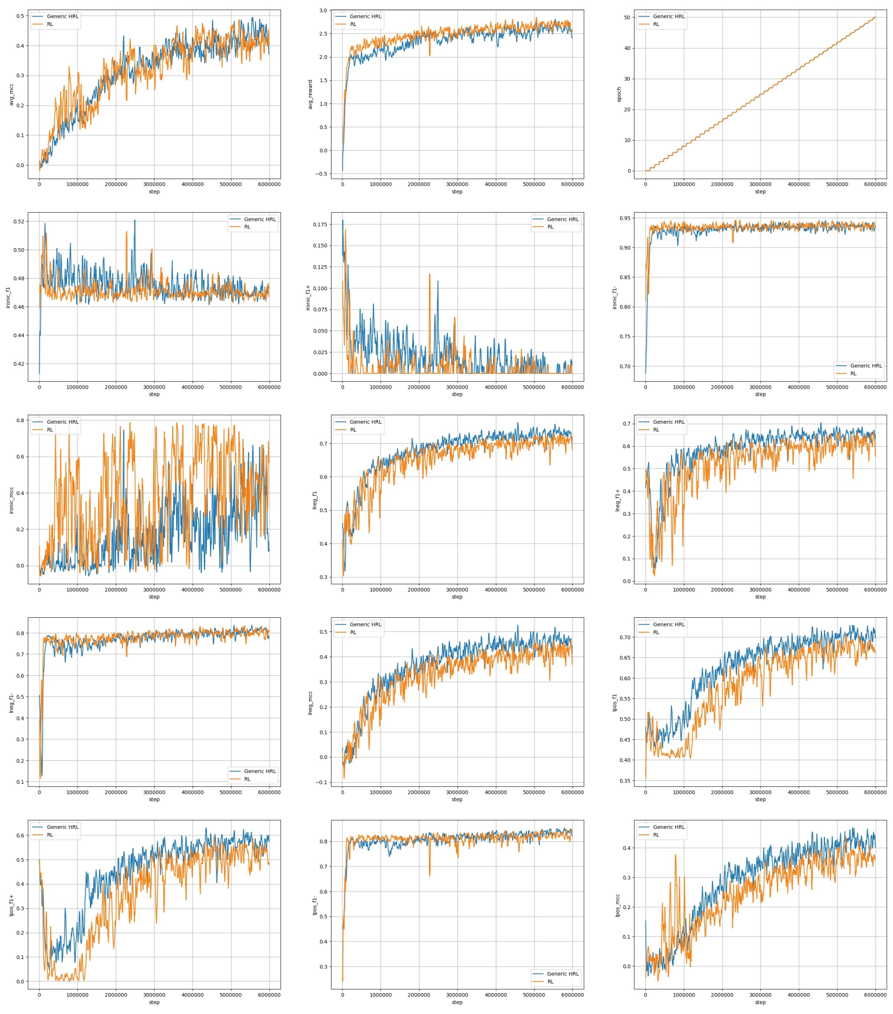


Figure 5.4: **Experiment 1 - Sentiment Analysis**: Training statistics; Part 1

The proposed architecture has been trained and tested on the Evalita SENTIPOLC 2016 dataset that is supplied with a evaluation script which score is based on F1 score. Obviously, during the training phase the test set has never been used.

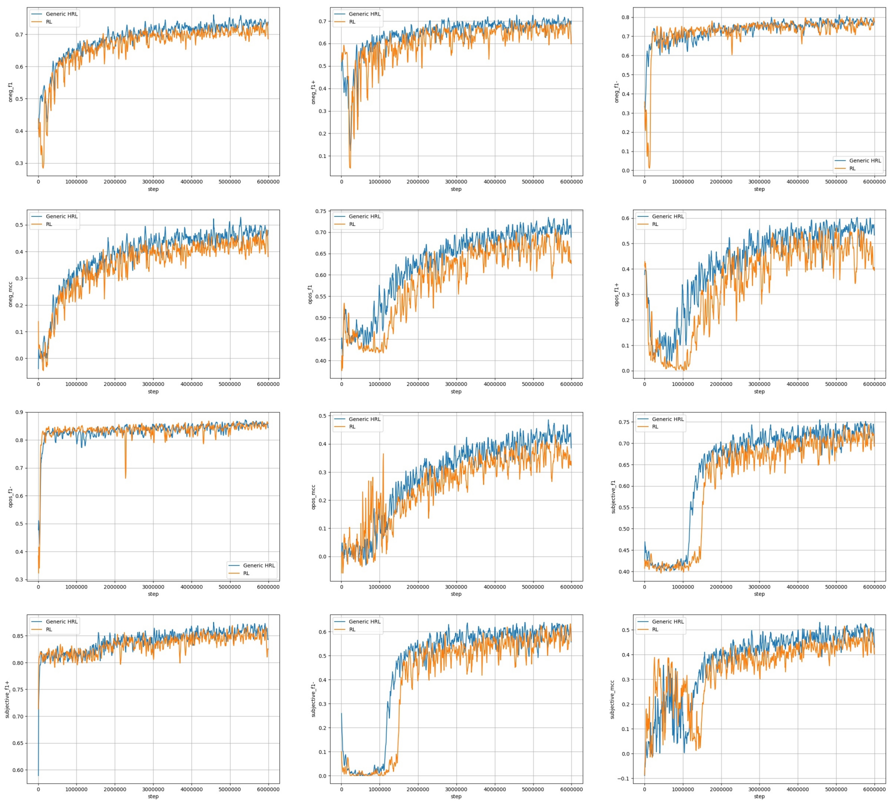


Figure 5.5: **Experiment 1 - Sentiment Analysis**: Training statistics; Part 2

The obtained results are *constrained* to the Evalita SENTIPOLC 2016 test dataset, thus they do not depend on additional Twitter annotated data.
The statistics (average positive and negative F1 scores, MCC, and accuracy for all the annotations described in Section 5.1.6) obtained during the training phase are shown in Fig. 5.4 and Fig. 5.5 together with a comparison of the performances of the Generic HRL approach and the RL approach. While the statistics obtained during the test phase are shown in tables 5.1 and 5.2.
It is interesting to notice that the Generic HRL method produces lower average cumulative rewards than the RL approach, but it gives significantly better results on all the F1 scores. The comparison results are not unexpected, in fact HRL is theoretically better than RL in dealing with non-markovian dependencies.

On the test set, using Generic HRL, we obtained a combined F1 score for positivity and negativity (called polarity) of 70.13.

Table 5.1: **Experiment 1 - Sentiment Analysis**: Test set results: Precision and recall, scaled by 100

| | $Pr._0$ | $Re._0$ | $Pr._1$ | $Re._1$ |
|---|---|---|---|---|
| **Subj.** | 70.38 | 66.33 | 82.60 | 85.13 |
| **Pos.** | 87.74 | 92.48 | 52.85 | 39.49 |
| **Neg.** | 81.08 | 74.23 | 63.73 | 72.34 |

Table 5.2: **Experiment 1 - Sentiment Analysis**: Test set results: F1 scores, scaled by 100

| | $F1_0$ | $F1_1$ | **F1** |
|---|---|---|---|
| **Subj.** | 68.30 | 83.85 | 76.07 |
| **Pos.** | 90.04 | 45.20 | 67.62 |
| **Neg.** | 77.50 | 67.76 | 72.63 |

Please, note that in [BBC+16] the best reported result for the polarity task is a F1 score (trained with *constrained* runs) of 68.28 by *SwissCheese*, while for the subjectivity task the best reported F1 score (trained with *unconstrained* runs) is 74.44 by *UniTor*.

## 5.2 Experiment 2 - Rogue

This second experiment is conducted on the Rogue environment introduced in Section 4.1.1.1, and we show that in the chosen environmental settings the benefits of HRL do not seem to be significant. The main challenges of this experiment were probably to choose a good state representation, to tackle the exploration problem mentioned in Chapter 3, and to find a useful generic hierarchical architecture.
This experiment differs significantly from the work in [ACS18], and the main differences shown in table 5.3 are on the following dimensions: hierarchical structure, game version, reward function, neural network, learning algorithm.

Table 5.3: **Experiment 2 - Rogue**: Main differences between [ACS18] and Experiment 2

| | **Experiment [ACS18]** | **This experiment** |
|---|---|---|
| **Hierarchy** | hand-crafted/naive | generic |
| **Rogue version** | no monsters | monsters and items |
| **Reward function** | non-sparse | sparse |
| **Neural network size** | $\approx 3 \cdot 10^6$ | $\approx 2 \cdot 10^5$ |
| **Learning algorithm** | A3C [MBM+16b] | PPO [SWD+17] |

We can say that the experiment described in [ACS18] was conducted in a somewhat

unsystematic way, preventing to clearly compare different approaches; it exploited predefined situations; it had no monsters; it was based on a ad-hoc rewarding mechanism made for having frequent rewards.
Thus, in this experiment we make use of generic HRL in a version of Rogue with monsters, items and sparse rewards.
This new environmental setting is much more challenging than [ACS18], especially because of:

- The presence of monsters adds a lot of complexity to the game.
- Sparse rewards makes the learning process harder.

Like in [ACS18], in this experiment we use a cropped view state representation and organize the training process on the base of a single level, terminating the episode as soon as the rogue takes the stairs.

### 5.2.1 State Representation

Let the game screen be a $24 \times 80$ matrix as shown in Fig. 4.1.
The adopted state is a $17 \times 17 \times 6$ matrix corresponding to a cropped view of the screen centred on the rogue (i.e. the rogue position is always on the centre of the matrix) having width and height equal to 17, and 6 different channels for: doors, walls, monsters, items, stairs, passages. The aforementioned matrix is filled with the following values:

**1** for stairs

**1** for walls

**1** for doors

**1** for passages

**i** for monsters where i is the index (starting from 1) of the monster in the monsters set

**i** for items where i is the index (starting from 1) of the item in the items set

**0** everywhere else

where the number of different monsters is 26, while the number of different items is 11. This state representation has the advantage of being sufficiently small to be fed to dense layers (possibly after convolutions), and can be intended as a form of attention. Moreover, in principle it could be used for 2-dimensional mazes that are arbitrarily larger than Rogue and it doesn't require to represent the rogue into the map.
On the other hand, the disadvantage of this representation is that it requires some form of memory, such as a recurrent unit or another more explicit and hand-made history of past actions or states.
Furthermore, we have experimentally observed that the performance of the chosen RL architecture drastically changes in worst when using a full view representation (with an extra channel for representing the rogue with a 1) instead of the aforementioned cropped view representation.
We suppose that the full view is more challenging because:

1. The neural network has to understand where the agent is on the map.
2. There is more information to process and interpret: in the screen view more parts of the map are visible.
3. The state representation is not normalized and the different semantic elements in the map are not clearly separated into different channels.

We leave the full view representation for future experiments.

### 5.2.2 Reward Function

In this experiment we use the simplest possible reward function: a positive reward (+10) for descending the stairs and zero for everything else. This choice mainly follows two considerations:

1. We want to assess the performance of agent when just awarded for the accomplishment of its task: finding and descending the stairs.
2. We do not want to introduce any bias in the agent behaviour.

For the task at hand this reward proves to be adequate, as can be seen from the results discussed in section 5.2.6.

### 5.2.3 Action Space

In our experiment, the agent can take only one action per step taken from the following set of actions:

- go left
- go right
- go up
- go down
- descend stairs

This means that the agent cannot open the inventory, use scrolls, etc..

### 5.2.4 Neural Network

The neural network adopted in this experiment is shown in Fig. 5.6 and it consists of two convolutional layers (CNNs) followed by a fully connected layer (to process spatial dependencies) and a stateful LSTM layer (to process temporal dependencies) followed by a Dropout and finally, value and policy output layers.
The CNNs have a RELU, a $3 \times 3$ kernel with unitary stride and respectively 16 and 32 filters. The CNNs output is flattened and it is the input for the fully connected layer (FC) with RELU activation functions and 64 units. We call this structure: shared layers. The shared layers input is the state representation described in Section 5.2.1 and its output is concatenated with a numerical "one hot" representation of the action taken in the previous state and the obtained reward. This concatenation is fed into an LSTM composed of 64 units and followed by a dropout with keep probability 0.5 as in [BBU+18]. The idea of concatenating previous actions and rewards to the LSTM input comes from [JMC+16b].

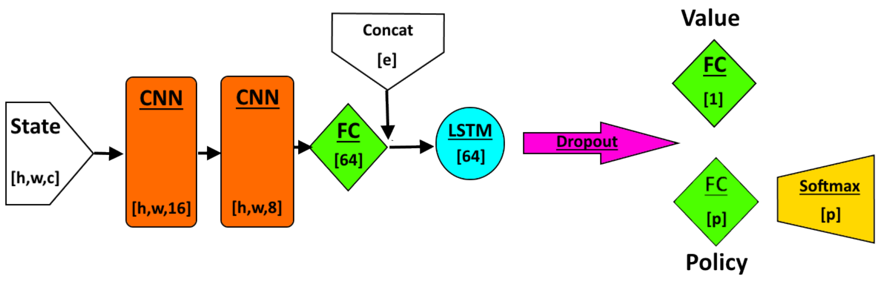


Figure 5.6: **Experiment 2 - Rogue**: Neural network. The output size of each layer is shown between brackets.

The output of the dropout is then the input for the value and policy layers. In Fig. 5.6, we have that $[h \times w \times c]$ is the size of the state representation, $e$ is the size of the *concat* vector, and $p$ is the size of the actions set.

### 5.2.5 Hierarchical Structure

Like in the experiment described in Chapter 5.1, the adopted hierarchical structure is composed by one RL partitioner and $n$ situational agents (or workers). But, differently from Chapter 5.1, parts of the neural network are shared among all the models (partitioner and workers).

More in detail, all the models share the inital CNN and FC layers. The stateful LSTM layer is shared only by workers, while the partitioner has its own separated stateful LSTM layer. An example of the aforementioned architecture is shown in Fig. 5.7

Every worker has the same reward function described in Section 5.2.2, and contributes to build a common cumulative reward.

The partitioner is an unsupervised classifier able to decide at run-time which worker should process a set of $p$ consecutive states, and its goal is to maximize the overall cumulative reward, distributing the states between the available agents. $p$ is called “partitioner granularity” an it can be tuned in order to speed up the algorithm. In fact, when $p = 1$ the partitioner is called at every step and the granularity is the finest possible, but when $p > 1$ the partitioner is called every $p$ steps thus reducing the computational complexity of the partitioning process.

Similarly to Chapter 5.1, the partitioner is trained with an higher learning rate than the

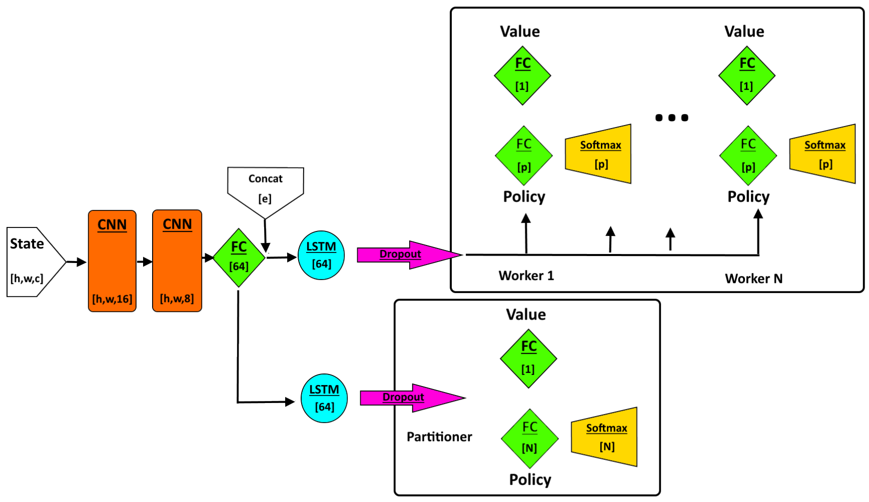


Figure 5.7: **Experiment 2 - Rogue**: Hierarchical structure

workers, and every worker has a different entropy regularization constant $\beta$ because it has been empirically observed that this is beneficial for the network stability. But differently from Chapter 5.1 the partitioner and the workers do not share the bootstrapped value estimation.

### 5.2.6 Experiment Results

In this section we show the experimental results of different tests with respect to a baseline with the following hyper-parameters:

a) **Gradient optimizer:** Adam

b) **Policy loss:** PPO

c) **Value loss:** PVO

d) **Max. batch size:** 8

e) **Workers number:** 4

f) **Entropy $\beta$:** 0.001 for the partitioner and $0.001 \cdot i$ for the i-th situational agent

g) **$\gamma$:** 0.99

h) **Use GAE:** True

i) **GAE $\lambda$:** 0.95

j) **Replay ratio:** 1

k) **Prioritized replay:** False

l) **Batch in buffer before starting replay:** 1

m) **Save only batches with reward:** True

n) **$\alpha$:** 3.5e-4

o) **$\alpha$ decay:** Exponential Decay

p) **Clip range:** 0.2

q) **Clip range decay:** Exponential Decay

r) **Only non negative entropy:** True

s) **Partitioner type:** ReinforcementLearning

t) **Partitioner optimizer:** Proximal Adagrad

u) **Partitioner $\alpha$:** 7e-4

v) **Partitioner $\beta$:** 0.001

w) **Partitioner $\gamma$:** 0.99

x) **Use count-based exploration reward:** True

y) **Positive exploration coefficient:** 0.1

z) **Negative exploration coefficient:** 0.001

In this experiment we organized the training process on the base of a single level, terminating the episode as soon as the rogue takes the stairs. In the rest of this chapter, unless stated otherwise, when we talk about the *performance* of an agent, we refer to the average percentage of episodes terminated with the rogue finding and taking the stairs within a maximum of 500 moves. This average is calculated using the formula described in Section 4.2.3.
The baseline maximum performance is around 76%.

#### 5.2.6.1 Test 1 - HRL benefits

Differently from the experiment in Chapter 5.1, in this experiment the beneficial effects of generic HRL are less evident and it is not clear whether there are positive effects at all.
In our first test we compare three different hierarchical structures: the one described in Section 5.2.5 (the baseline), a variation of the baseline with 3 partitions instead of 5 and a new architecture based on the K-Means clustering algorithm [HW79].

In the K-Means based hierarchical architecture the clustering algorithm is trained on an initial set of states, generated during training, with the goal of partitioning the whole state space in *n* different clusters. Please note that K-Means partitions the state space using the Euclidean distance of their vectorial representation, and this partition may not be optimal, especially if the goal of the agent is to maximize the cumulative reward of the agent.
The difference in performance between the aforementioned hierarchical structures and an identical architecture without any hierarchy is shown in Fig. 5.8. We can see that the baseline gives the best performance, but the margin is very small (around 5%) and the situational agents workload is too much unbalanced. But an interesting fact to highlight is that the workload of the K-Means based hierarchy is unbalanced too.
We have empirically found out that removing the dropout layer and bootstrapping the manager with the worker's value estimation (instead of the manager's one) produces more balanced workloads, but apparently without sensibly improving the overall performance of the hierarchy. In this same test we have also tried to tune the partitioner granularity hyper-parameter described in Section 5.2.5, in order to understand its role in the algorithm. As shown in Fig. 5.9 granularities of 1 and 4 result in worst performances (losing more than 20%) than granularity 8. We suspect that this is due to the fact that the batch size is 8 too, thus granularities lower than 8 produces more "non-bootstrapped" mini-batches. The results of another interesting test on HRL are shown in Fig. 5.10. In this test we compare the hierarchical structure of the experiment in Chapter 5.1 (that does not share layers among models) with the baseline of this experiment. Here we can see that sharing layers in the hierarchical structure may be slightly beneficial for Rogue (giving a performance bonus of +5 %).

#### 5.2.6.2 Test 2 - Generic vs Naive HRL

In Chapter 3.4 we have briefly written about the difference between *naive* and *generic* HRL. At the begin of Section 5.2, we made a comparison between our previous work on Rogue [ACS18] and this new work, showing the differences between the two approaches. In this second test we want to compare the generic HRL architecture of the baseline (Section 5.2.6) with a variation of the baseline using the naive HRL architec-

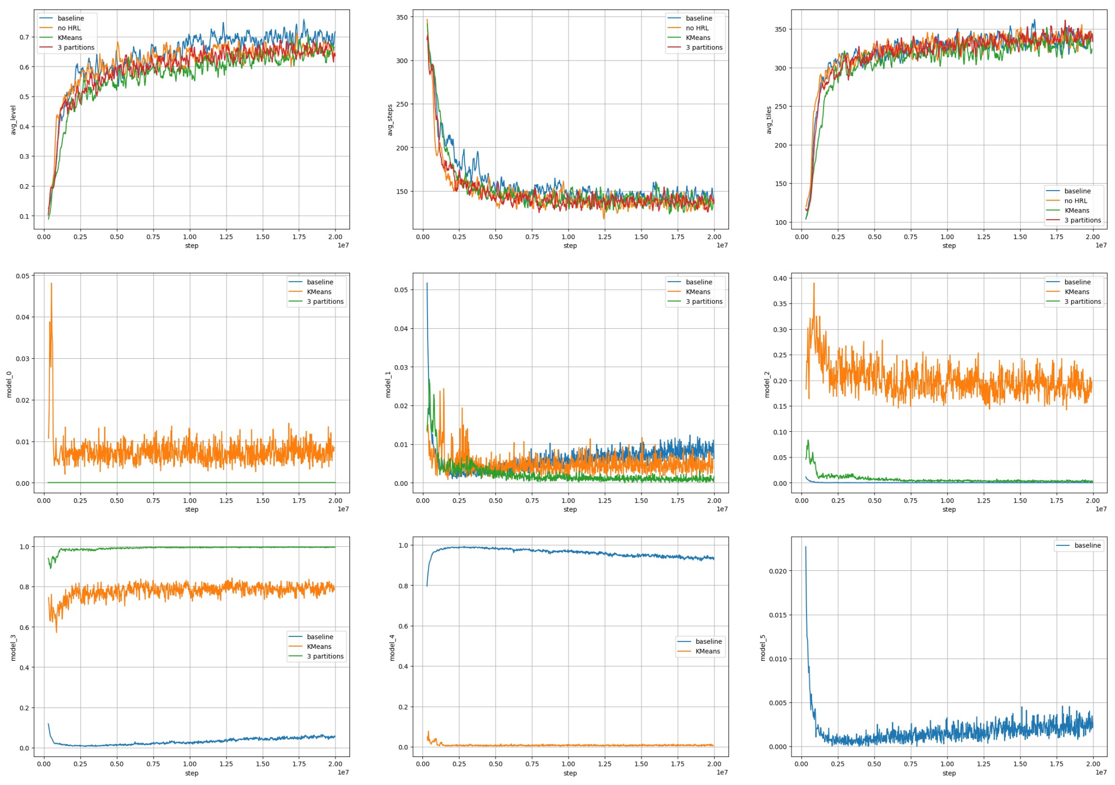


Figure 5.8: **Experiment 2 - Rogue**: Training statistics of the test conducted to understand the *effects of HRL*

ture adopted in [ACS18].

The aforementioned naive architecture is made of two separated situational agents, invoked respectively in two different situations: "*stairs are visible on the screen*", "*stairs are not visible on the screen*". The situations are determined programmatically and are not learned.

The comparison in Fig.5.11 shows that the two architectures achieve almost identical performance in this environment settings, but the generic one explores more tiles while the naive one takes less steps to descend the stairs.

#### 5.2.6.3 Test 3 - Proximal Optimization

Proximal optimization is an interesting technique described in 2.3. In Fig.5.12 the beneficial effects of Proximal Optimization are quite evident, both PPO and PVO signif-

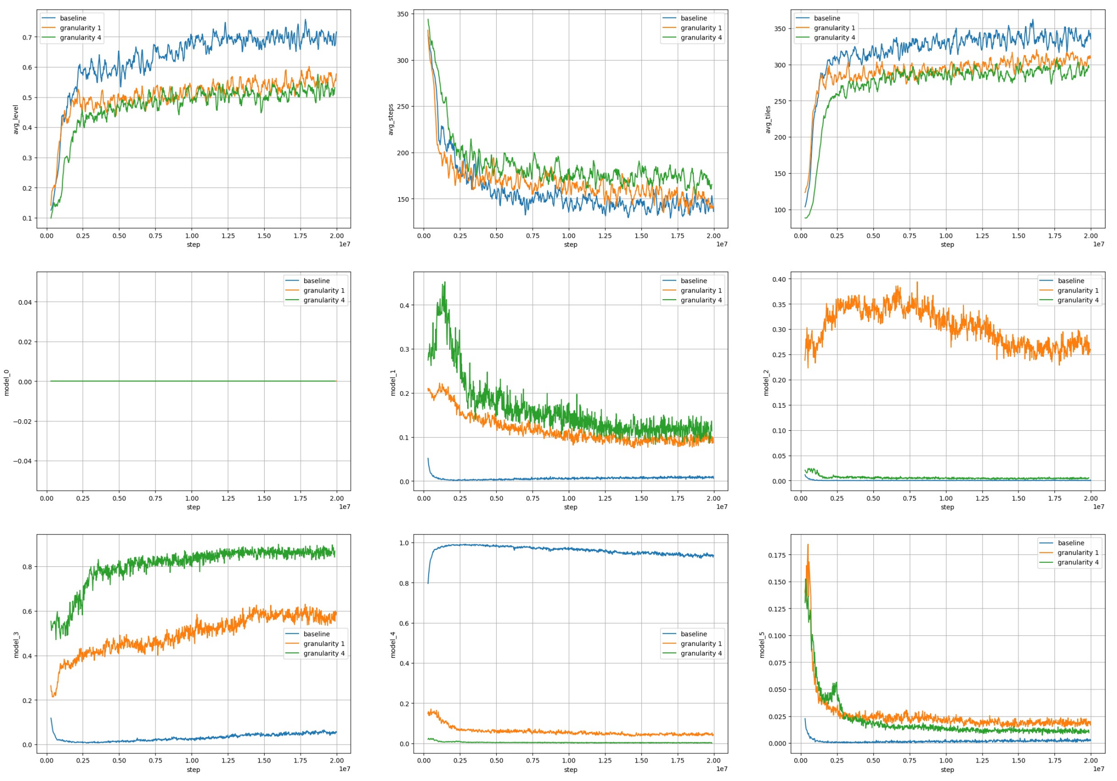


Figure 5.9: **Experiment 2 - Rogue**: Training statistics of the test conducted to understand the *effects of the partitioner granularity hyper-parameter*

icantly improve the overall algorithm performance respectively by more than 70% and 40%.

### 5.2.6.4 Test 4 - Count-based exploration and Experience Replay

Both count-based exploration and experience replay (Chapter 3) are interesting techniques that may be used to improve a RL algorithm performance. An implementation of the aforementioned methods has been already described in Section 4.2.2 and 4.2.1. Count-based exploration is made for improving *exploration*, while experience replay is made for improving *sample efficiency*. In Fig. 5.13 it is possible to see the beneficial effects of these techniques when used on Rogue environment (with monsters). As you can see, the effects of count-based exploration are much more significant (increasing performance by more than 30%), while experience replay plays only a marginal role on

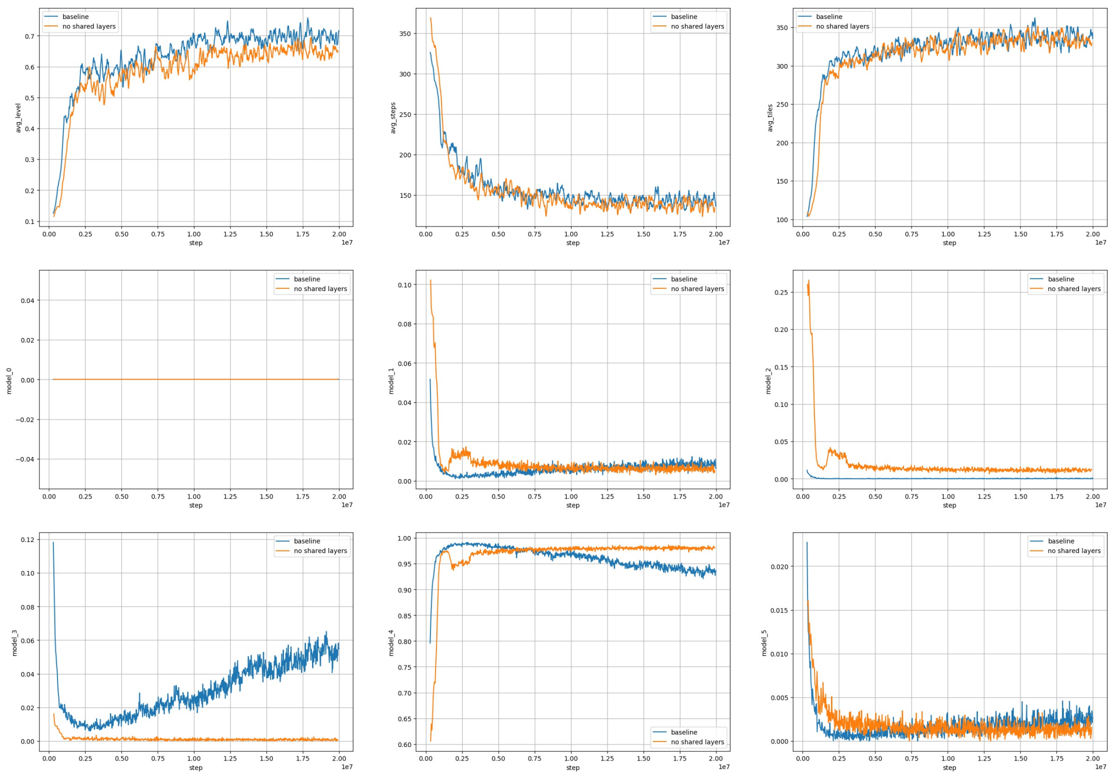


Figure 5.10: **Experiment 2 - Rogue**: Training statistics of the test conducted to *understand the effects of sharing layers in the hierarchical structure*

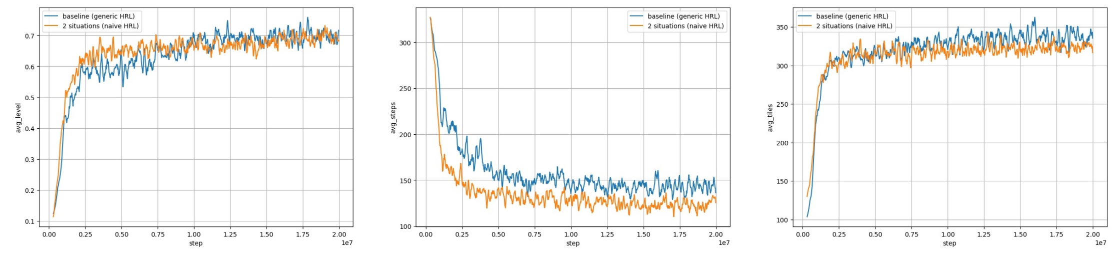


Figure 5.11: **Experiment 2 - Rogue**: Training statistics of the test conducted to *compare naive and generic HRL*

the overall performances (increasing performance by less than 10%).
An interesting fact to mention is that the intrinsic reward, given by the adopted count-based exploration function, shapes the overall reward function similarly to the reward function adopted in [ACS18].

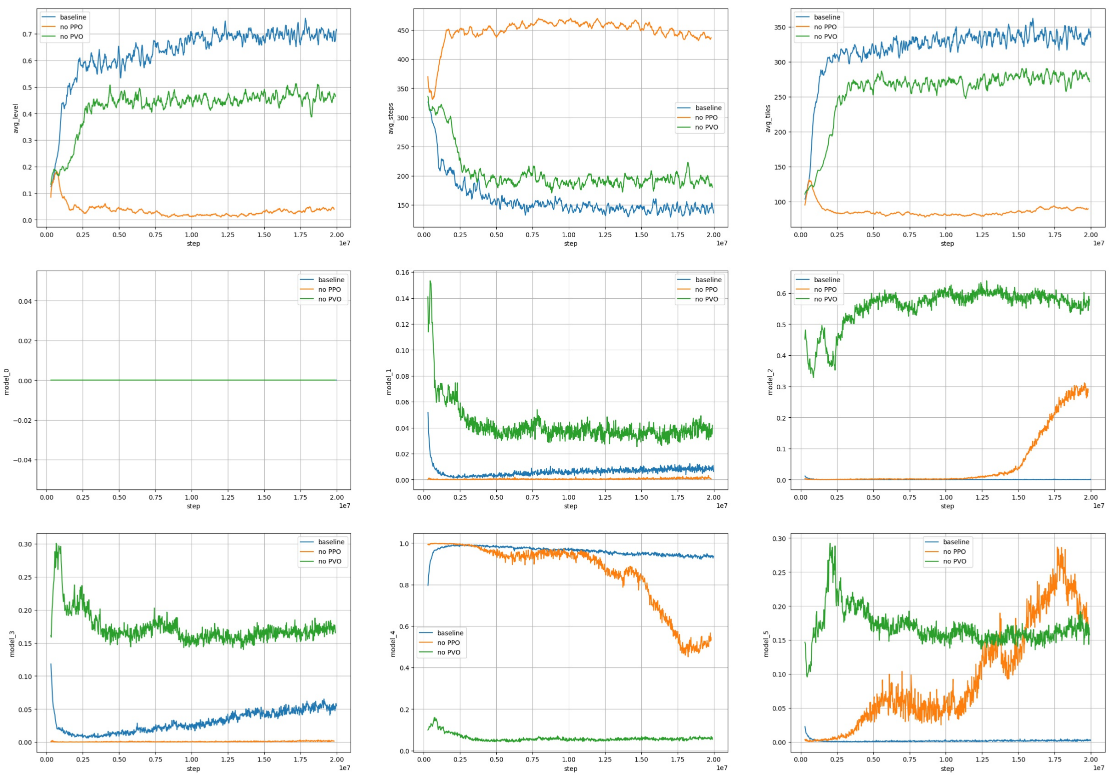


Figure 5.12: **Experiment 2 - Rogue**: Training statistics of the test conducted to understand the *effects of proximal optimization*

## 5.3 Experiment 3 - Car Controller

Car Controller is a new environment, described in section 4.1.2.1, that we made for training Deep RL networks to autonomously drive a car on a road with obstacles.
In this environment:

- roads are represented with splines and splines are used to generate 2D control points
- obstacles are represented with circles and thus are defined by a 2D point (the centre) and a radius
- a car is a 2D point with speed and steering angle

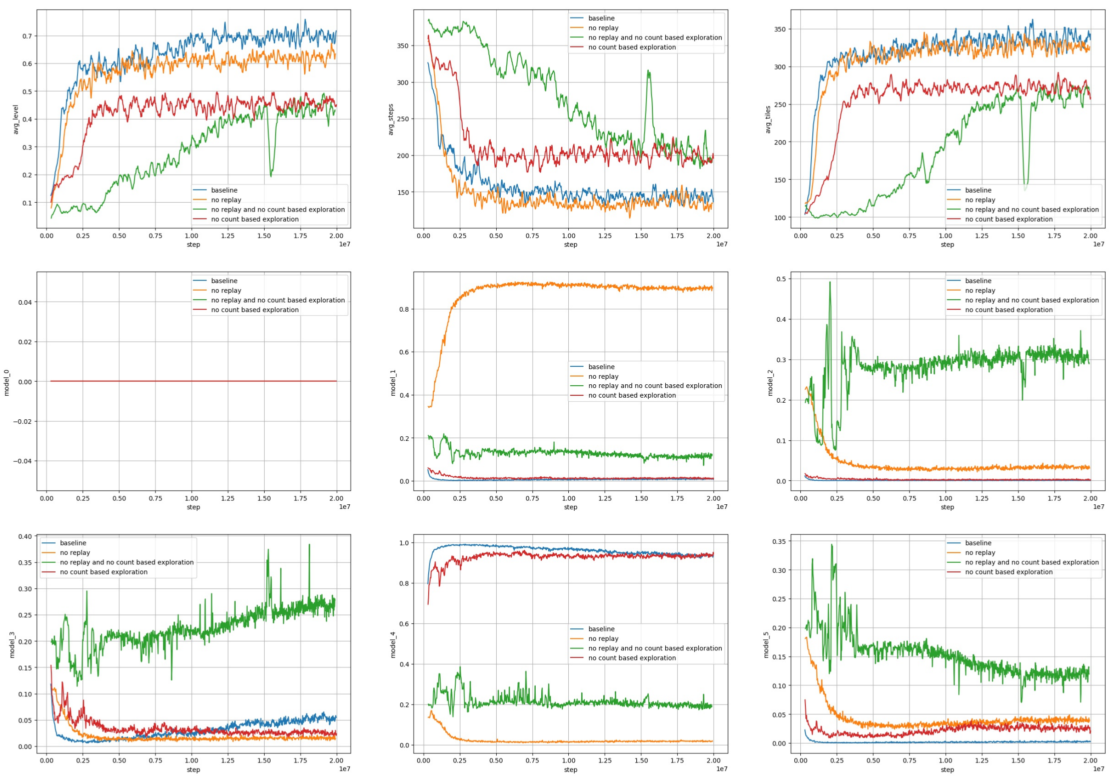


Figure 5.13: **Experiment 2 - Rogue**: Training statistics of the test conducted to understand the *effects of count-based exploration*

The car can see only up to $1m$ in forward direction and has no knowledge of what it is outside its field of view. The controller is called in average every $\mu_c = 0.1$ seconds (where $\mu_c$ is the mean of an exponential distribution) and at every call/step the car can change its speed and steering angle. Furthermore the car has the following limitations:

- The maximum acceleration is $0.7\frac{m}{s^2}$.
- The steering angle is in $[-30, 30]$ deg.
- The speed is in $[0.1, 1.4]\frac{m}{s}$.

At every car action, some random noise is added to the actuators inputs in order to simulate real environment scenarios:

- Speed noise in $[-0.25, 0.25]\frac{m}{s}$.

- Steering angle noise in $[-2, 2]$ deg.

Obstacles spawning is defined by the following constants:

- Maximum obstacle count: 3.
- Random obstacle radius in $[0.15, 0.45]m$.

At every episode:

- 2 random splines are generated and concatenated in order to form a road.
- An upper speed limit is randomly taken from $[0.7, 1.4]\frac{m}{s}$.

An episode may end: or when the number of steps is greater than 100, or when the car has reached the end of the second spline, or when the car has hit an obstacle.
The goal of the car is to follow the path given by control points, to avoid obstacles and to not exceed the speed limit.

### 5.3.1 State Representation

The main state is composed by two $5 \times 3$ matrices:

- The obstacles matrix: has to be a vector of 5 circles. As mentioned before, each circle is made of 3 floats representing a 2D point for the centre and the radius of the circle. Only the visible obstacles are fed in this matrix.
- The control points matrix: has to be a vector of 5 *equidistant* control points represented with circles having radius 0. Only the visible control points are fed in this matrix.

All the coordinates defined by the 2D points are expressed with respect to the car point of view. This means that the car is always in the origin.
The control points are represented with circles just to have the same representation for obstacles and control points, this way we can put together both obstacles and control points in a single $2 \times 5 \times 3$ matrix.

In the state representation we decided to represent the road using the control points instead of the spline parameters, because using the spline requires to know the exact car position within a very very small error range (eg: less than 5 cm) in order to properly avoid collisions, and achieving highly precise positioning may be computationally expensive due to numerical errors in odometric computations. In fact control points may be generated using splines but they can also be generated using line detection systems based on view sensors (eg: video-camera), thus they do not necessary require to know the exact car position in the world.
Furthermore, there is also a secondary state called "concat" and represented by a 1D vector with length 4 containing the following information:

- current car steering angle in radians: a float
- current car speed: a float
- seconds passed from last step: a float
- speed upper limit: a float

### 5.3.2 Reward Function

The goal of the car is to follow the path given by control points, to avoid obstacles and to not exceed the speed limit. Furthermore, the car should not exceed a maximum distance $\Delta_{\max} = 0.1m$ from path, unless some obstacle with radius $\rho$ is on the road. We shaped the reward function in order to have frequent rewards, as follows:

1. If the car hits an obstacle, then the return is $-1$.
2. If the car is not moving toward the end of the path, then the return is $-0.1$.
3. If the euclidean distance $\delta$ from path and car is lower than $\Delta_{\max} + \rho$, then the car get a bonus proportional to the space crossed in the step. But if the car exceeds the upper speed limit, then it gets a malus proportional to the space crossed.

A python pseudo-code of the aforementioned reward function is:

**Algorithm 5.1 : Experiment 3 - Car Controller: Reward function Python code**

```
max_distance_to_path = 0.1
seconds_per_step = 0.1

def get_reward(speed, point, progress, position,
        closest_visible_obstacle, speed_limit):

  projection_point = get_point_from_spline_x(position)
  if closest_visible_obstacle is not None:
    ob_point, ob_radius = closest_visible_obstacle
    # check collision
    if euclidean_distance(ob_point,point) <= ob_radius:
      return -1 # terminate episode
    if euclidean_distance(ob_point, projection_point) <= ob_radius:
      # could collide obstacle
      max_distance_to_path += ob_radius
  if position > progress:
    # is moving toward next position
    distance = euclidean_distance(point, projection_point)
    distance_ratio = np.clip(distance/max_distance_to_path, 0,1) # always
        in [0,1]
    inverse_distance_ratio = 1 - distance_ratio
    # the more speed > speed_limit, the bigger the malus
    malus = speed_limit*max(0,speed/speed_limit-1)*seconds_per_step
    # smaller distances to path give higher rewards
    bonus = min(speed,speed_limit)*seconds_per_step*
        inverse_distance_ratio
    return bonus-malus # do not terminate episode
  # else is NOT moving toward next position
  return -0.1
```

### 5.3.3 Action Space

Car Controller is an environment for continuous control. In this environment the car has to control its speed and its steering angle, and both speed and steering angle are continuous variables.
In all previous experiments we have worked on discrete action spaces modelling the policy with a categorical distribution. But categorical distributions cannot be used for continuous actions, because they involve discretization. In a continuous action space we may model the policy with a normal distribution, and this is what we do in this experiment.
Let's go back for a moment to equation 1.11, the formula of $log\pi_{\theta_1}(a|s)$ depends to the chosen distribution for modelling the policy. Thus, if we are going to model policy as a normal distribution, we have to make the following changes to the network used for discrete controlling:

- The policy layer has to find the mean $\mu$ and the standard deviation $\sigma$ of a normal distribution $N(\mu, \sigma)$, instead of the logits of a softmax.
- $-log\pi_{\theta_1}(a|s)$ is now the negative log probability density/mass function of $N(\mu, \sigma)$ given $a$ and $s$, instead of the softmax cross-entropy.

The numerical values of $\mu$ and $\sigma$ depends on the actions the agent has to perform. For example, if the agent has to control steering angle in range $[-r, r]$, then a reasonable $\mu$ is in $[-r, r]$ and a reasonable $\sigma$ is in $]0, \sqrt{2r}]$. What does it imply? At early stages of the training process the agent has to explore values in those numerical ranges, but the bigger is $r$ and the more it will be difficult to the agent to casually explore all the possible numerical values in the range because the numerical changes in the policy parameters has to be bigger. For this reason we prefer to keep $\mu \in [-1, 1]$ and $\sigma \in ]0, 1]$ and then multiply by $r$ the sample taken from $N(\mu, \sigma)$.
Let $\mu_l, \sigma_l$ be the output of the policy layer, then we have:

- $\mu = \text{clip}(\mu_l, -1, 1)$
- $\sigma = \text{clip}(|\sigma_l|, \epsilon, 1)$, with $\epsilon$ close enough to 0 (eg: $\epsilon = 0.0001$)

Please, note that $|\sigma_l|$ is used instead of $\sigma_l$ because this way we halve the probability of clipping $\sigma_l$.
Let $-1 \leq a \leq 1$ and $-1 \leq b \leq 1$ be respectively the acceleration and steering values sampled by the car controller (the agent) using two different normal distribution following the aforementioned rules, then at every step the new speed will be:

$$\text{new\_speed} = \text{old\_speed} + \mu_c \cdot a \cdot \text{maximum\_acceleration} \tag{5.2}$$

while the new steering angle will be:

$$\text{new\_steering\_angle} = b \cdot \text{maximum\_steering\_angle} \tag{5.3}$$

#### 5.3.3.1 An interesting hypothesis on $\psi$

At this point of the experiment it is interesting to investigate the meaning of $\psi = -log\pi_{\theta_1}(a|s)$.
When talking about categorical distributions we have that $\psi$ is equivalent to the softmax cross-entropy. In information theory, cross-entropy can be interpreted as the expected message-length per datum when a wrong distribution $Q$ is assumed while the data actually follows a distribution $P$ [Wik18b].
While, when talking about normal distributions we have that $\psi$ is the negative logarithm of the probability density function (PDF) of $N(\mu, \sigma)$. In probability theory, PDF is a function that estimates a relative likelihood that the value of the random variable $N(\mu, \sigma)$ would equal the sample [Wik18e].
Cross-entropy and negative log PDF can both be negative, and in the current experiment scenario it happens very frequently that negative log PDF is lower than 0. But, if we assume true the information theory point of view, hypothesizing that $\psi$ is an expected message-length, then we can surely say that $\psi \geq 0$. In Section 5.3.5.1 we show some experimental results that gives some credit to this hypothesis.

### 5.3.4 Neural Network

The Actor-Critic network for the Car Controller is shown in fig. 5.14. This network consists of two convolutional layers followed by a dense layer to process spatial dependencies and finally, value and policy output layers.

The CNNs have a RELU, a $3 \times 3$ kernel with unitary stride and respectively 16 and 32 filters. The CNNs output is flattened and it is the input for a FC with RELU and 64 units. We call this structure: shared layers.
The shared layers input is the state representation described in Section 5.3.1.
The output of the shared layers is then concatenated with the "concat" vector (described in Section 5.3.1) and fed as input for the value and policy layers.

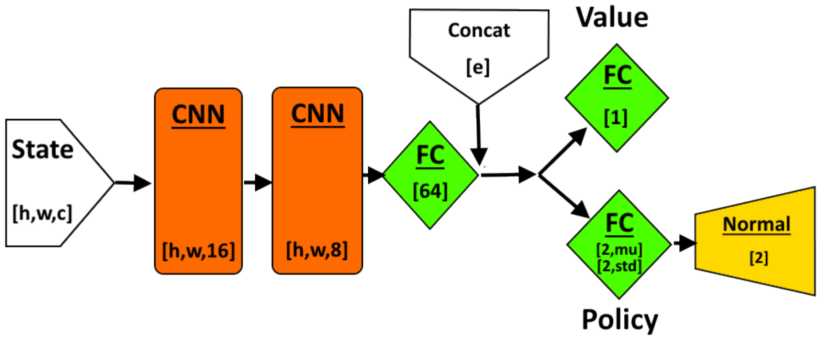


Figure 5.14: **Experiment 3 - Car Controller**: Neural network architecture. The output size of each layer is shown between brackets, and $h = 2$, $w = 5$, $c = 2$, $e = 4$.

### 5.3.5 Experiment Results

In this section we show the experimental results of different tests with respect to a baseline with the following hyper-parameters:

a) **Gradient optimizer:** Adam

b) **Policy loss:** PPO

c) **Value loss:** Vanilla

d) **Max. batch size:** 8

e) **Workers number:** 4

f) **Entropy $\beta$:** 0.001

g) **$\gamma$:** 0.99

h) **Use GAE:** True

i) **GAE $\lambda$:** 0.95

j) **Replay ratio:** 1

k) **Prioritized replay:** True

l) **Batch in buffer before starting replay:** 1

m) **Save only batches with reward:** True
n) **Predict reward:** False
o) $\alpha$**:** 3.5e-4
p) $\alpha$ **decay:** Exponential Decay
q) **Clip range:** 0.2
r) **Clip range decay:** Exponential Decay
s) **Partitioner type:** None
t) **Only non negative entropy:** True
u) **Use count-based exploration reward:** False

In the rest of this chapter, unless stated otherwise, when we talk about the *performance* of an agent, we refer to the average extrinsic cumulative reward obtained by the agent during an episode. This average is calculated using the formula described in Section 4.2.3.
The baseline maximum performance is around 6.
Furthermore, in the following tests we will consider some new statistics specific to the car controller environment:

- "avg_hit": the average number of collisions with obstacles. Please note that during simulations some obstacles can spawn too close to the car and cannot be avoided.
- "avg_completed": the percentage of episodes terminated when the car has reached the end of the second spline.

#### 5.3.5.1 Test 1 - Non-negative $\psi$

The goal of this first test is to verify the hypothesis described in Section 5.3.3.1. We want to understand whether $\psi$ is the expected message-length or not. In order to do it, we set up two identical training environments but in one of them (the baseline) we keep always $\psi \geq 0$ and entropy $\geq 0$, where entropy is the function $S(\pi_\theta|s_t)$ used in eq.2.1. The test results are shown in fig.5.15 and they experimentally give some credit to the hypothesis that $\psi$ is an expected message-length. As we can see, the baseline significantly outperforms (by more than 5 performance points) the performance of the architecture without constraints on $\psi$.

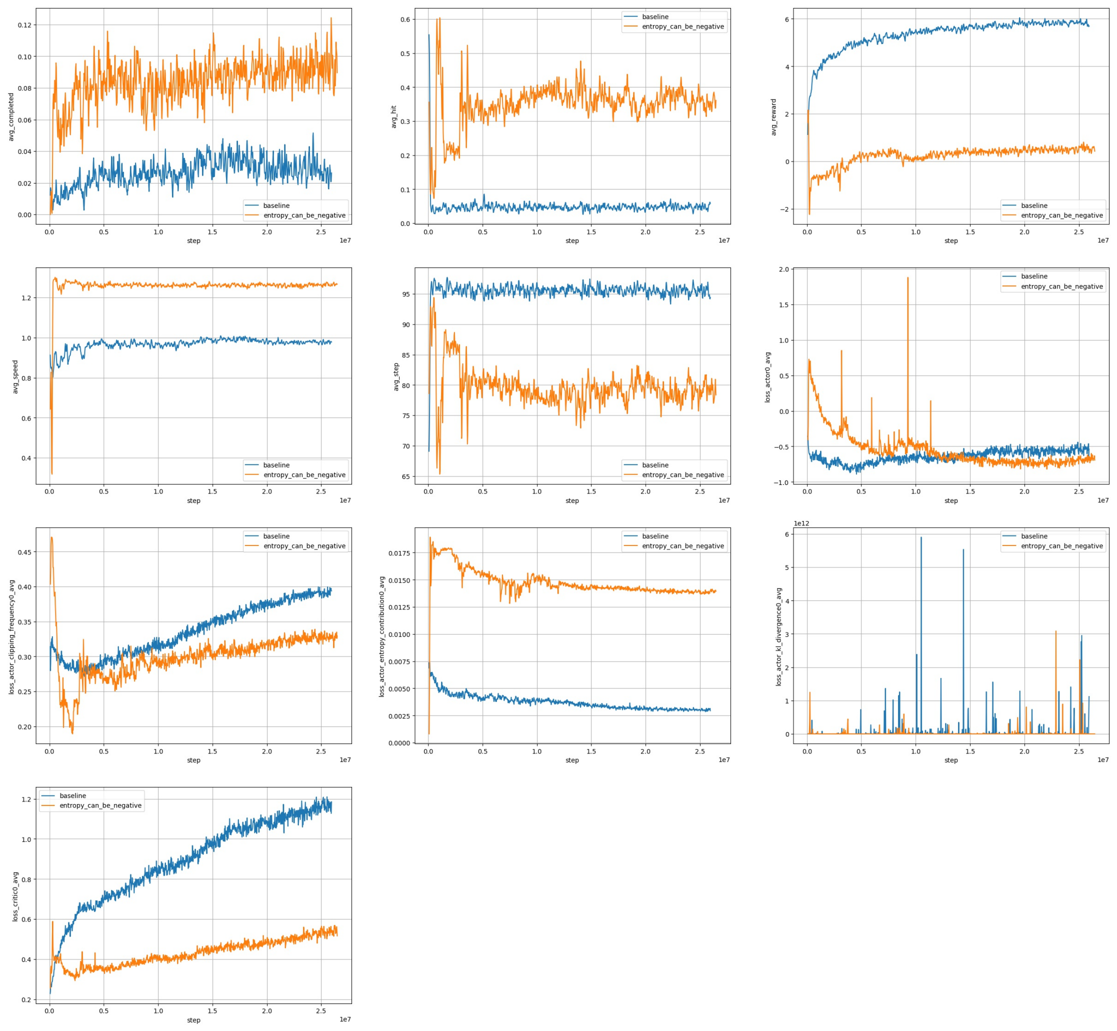


Figure 5.15: **Experiment 3 - Car Controller**: Training statistics of the test conducted to *investigate the information theory interpretation of entropy*

#### 5.3.5.2 Test 2 - HRL and non-markovian temporal dependencies

With this second test we try to understand the role of Hierarchical Reinforcement Learning and non-markovian temporal dependencies in the Car Controller problem. Recurrent Neural Networks (RNN) such as the Long-Short Term Memory (LSTM) may be used for handling temporal dependencies, and in the experiment shown in Fig.5.16 we compare the baseline performance with the performance of a variation of the baseline having an extra stateful LSTM (of 64 units) followed by a Dropout (with keep probability 0.5) between the concatenation layer and the policy and value layers. In Fig.5.16 we can see that we get more stable results without the aforementioned LSTM and Dropout layers.

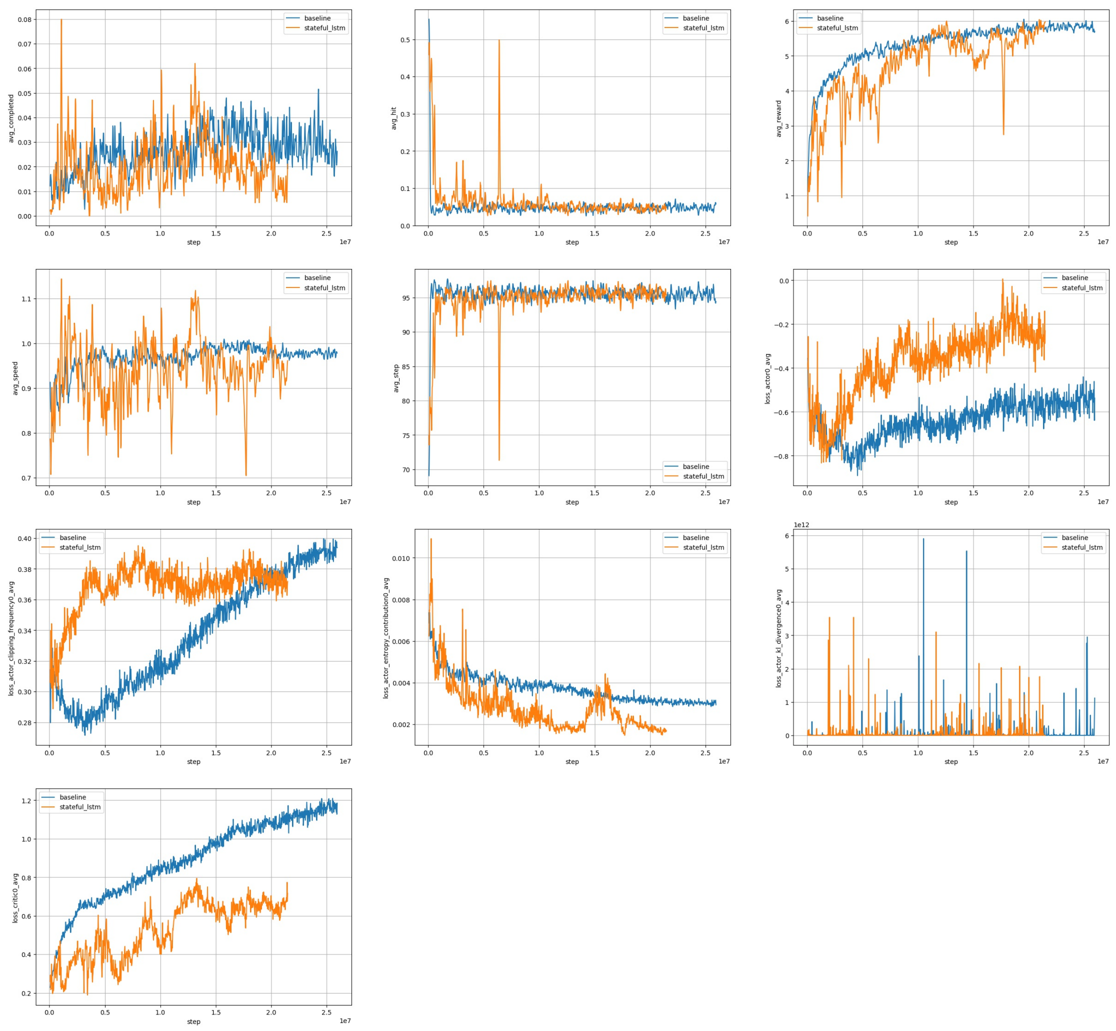


Figure 5.16: **Experiment 3 - Car Controller**: Training statistics of the test conducted to *understand the role of temporal dependencies*

Furthermore, instead of adding an extra LSTM layer we may want to use a hierarchical architecture for handling non-markovian dependencies. Thus, in the experiment shown in fig.5.17 we compare the baseline with a variation of the baseline using the same hierarchical structure described in Section 5.2.5, but with no shared layers among models like in the hierarchical structure described in Section 5.1.10. The results achieved with this new hierarchical architecture are definitely more stable than those achieved using a stateful LSTM, but the performance is worse by 0.5 points than the baseline.

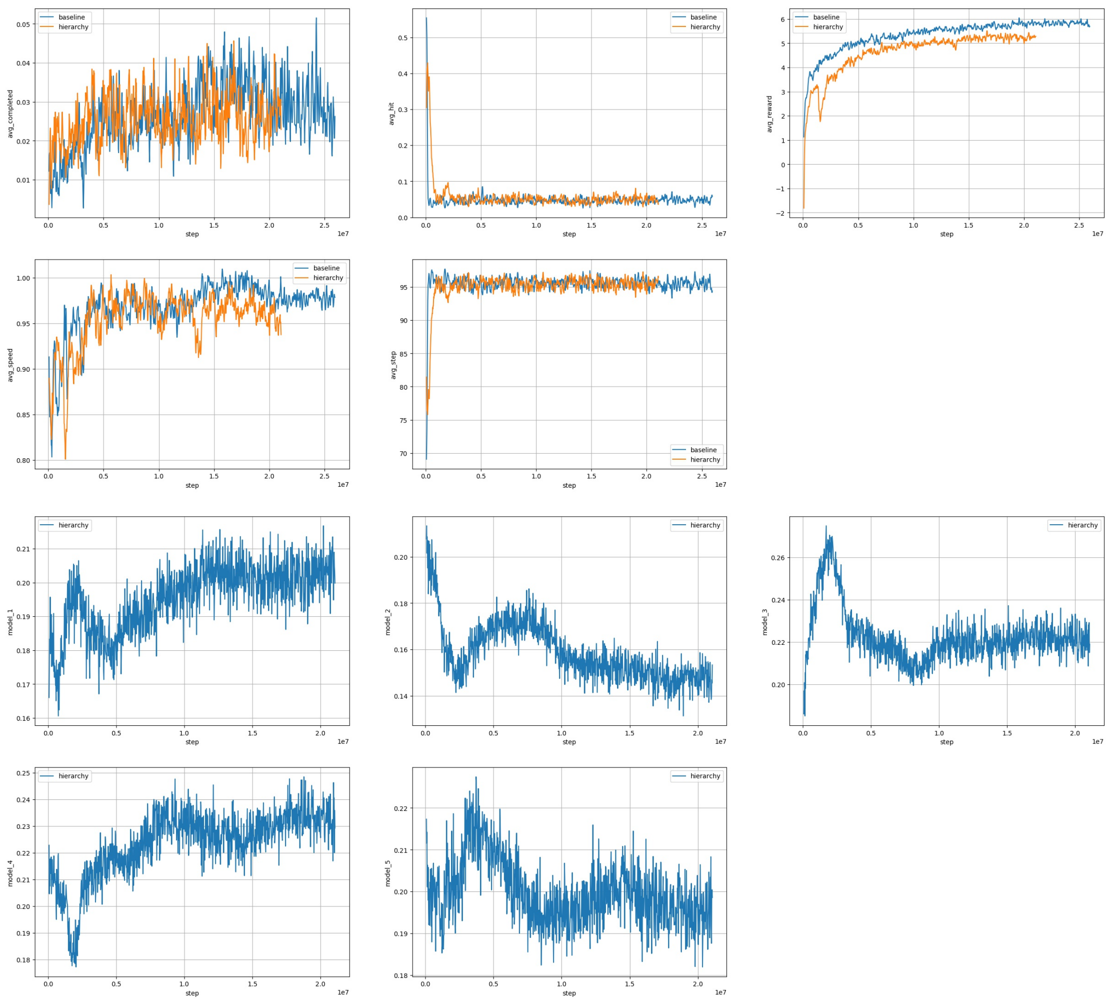


Figure 5.17: **Experiment 3 - Car Controller**: Training statistics of the test conducted to *understand the role of HRL*

#### 5.3.5.3 Test 3 - Experience replay

With this third test we try to understand the role of experience replay in the car controller problem. As shown in fig.5.18, experience replay plays an important role for achieving best results (increasing performance by around 1 point) and the experiment using prioritized experience replay (the baseline) gives the most stable results.

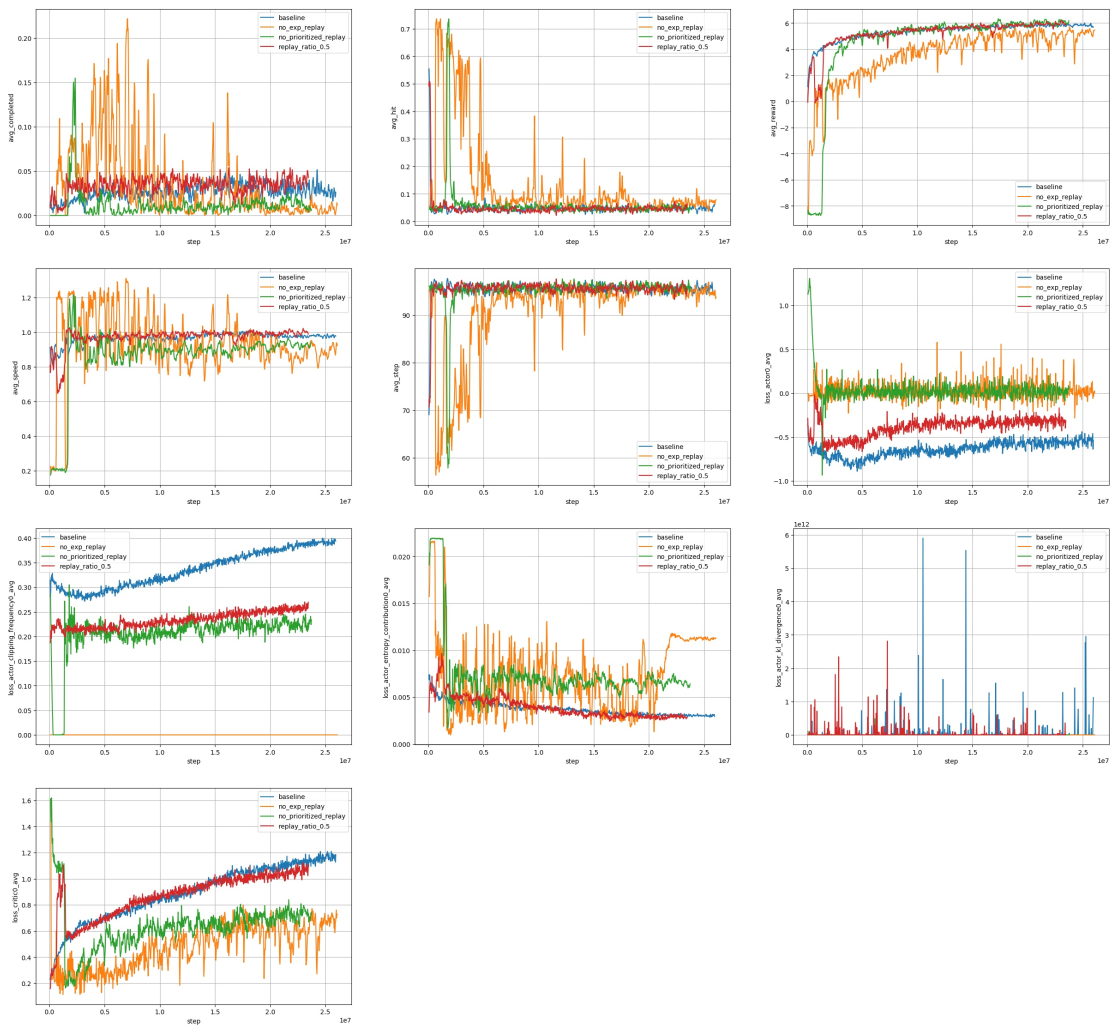


Figure 5.18: **Experiment 3 - Car Controller**: Training statistics of the test conducted to *understand the role of experience replay*

#### 5.3.5.4 Test 4 - Proximal Optimization

In the test described in Section 5.2.6.4 we have seen that PVO plays an important role in the performance of the experiments conducted on Rogue (Section 5.2). With this fourth and last test we try to understand the role of Proximal Value Optimization (PVO) in the car controller problem.
In this test we compare the baseline with a baseline variation that uses PVO. As shown in fig.5.17, using PVO in the car controller problem seems to be not helpful, worsening the performance by about 0.5 points.

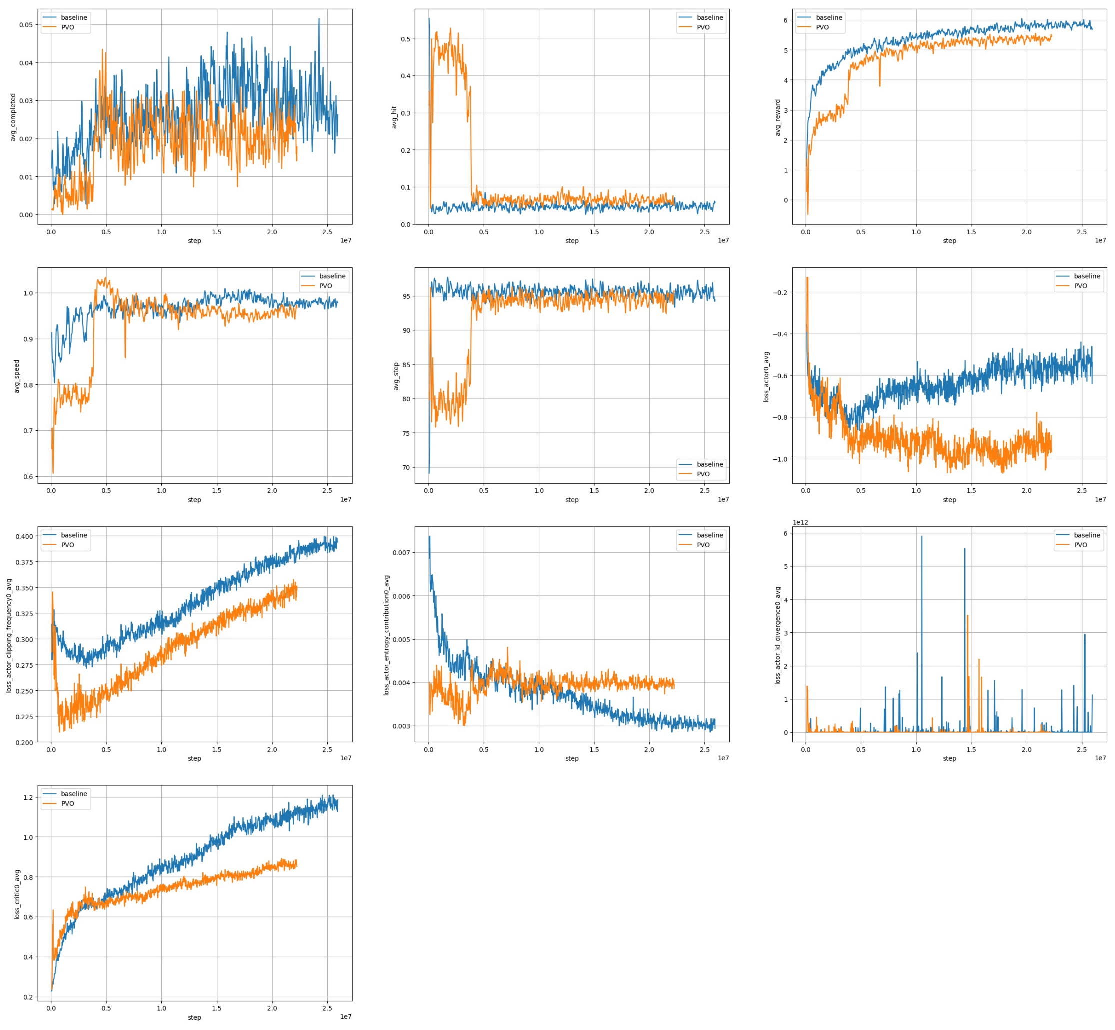


Figure 5.19: **Experiment 3 - Car Controller**: Training statistics of the test conducted to *understand the role of PVO*

# Chapter 6

# Conclusions

In this thesis we have investigated the advantages that a HRL approach may have over a simple RL approach, showing examples in which HRL brings concrete advantages (Section 5.1), or examples in where the usefulness of HRL may be uncertain or none (Sections 5.3, 5.2). But we have also investigated the effect of other interesting techniques like Experience Replay, Count-Based Exploration and Proximal Optimization, using them to solve new and completely different problems/environments like Sentiment Analysis (Section 5.1), Rogue with monsters (Section 5.2) and Car Controller (Section 5.3).
Remarkably, we claim that especially our work in Sentiment Analysis is very innovative for RL, resulting in state-of-the-art performances; as far as the author knows, Reinforcement Learning (RL) approach is only rarely applied to the domain of computational linguistic and sentiment analysis. Furthermore, our work on the famous video-game Rogue is probably the first example of Deep RL architecture able to explore Rogue dungeons and fight against its monsters achieving a success rate of more than 75% on the first game level. While our work on Car Controller allowed us to make some interesting considerations on the nature of some components of the policy gradient equation.
But what is the point? The ability to understand and analyse natural language and sentiments, and the ability to play games or drive a car are all human skills. Thus, the real questions now are:

1. Are there commonalities between RL algorithms and human brain? [BBU+18,

DD08, LG09]

2. Can we describe human intelligence with an algorithm?

# Appendices

# Appendix A

# Legend

Some words used in this thesis have been abbreviated. Below here you will find the list of abbreviations with their equivalent meanings.

- eq. ≡ equation
- neq. ≡ disequation
- pg. ≡ page
- fig. ≡ figure
- eg. ≡ example
- sec. ≡ section
- ch. ≡ chapter

# Bibliography

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[Ach01] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 274–281. ACM, 2001.

[ACS18] Andrea Asperti, Daniele Cortesi, and Francesco Sovrano. Crawling in rogue's dungeons with (partitioned) a3c. *arXiv preprint arXiv:1804.08685*, 2018.

[BBC+16] Francesco Barbieri, Valerio Basile, Danilo Croce, Malvina Nissim, Nicole Novielli, and Viviana Patti. Overview of the evalita 2016 sentiment polarity classification task. In *Proceedings of Third Italian Conference on Computational Linguistics (CLiC-it 2016) & Fifth Evaluation Campaign of Natural Language Processing and Speech Tools for Italian. Final Workshop (EVALITA 2016)*, 2016.

[BBN+14] Valerio Basile, Andrea Bolioli, Malvina Nissim, Viviana Patti, and Paolo Rosso. Evalita 2014 sentipolc task: Task guidelines. Technical report, Technical report, 2014.

[BBU+18] Andrea Banino, Caswell Barry, Benigno Uria, Charles Blundell, Timothy Lillicrap, Piotr Mirowski, Alexander Pritzel, Martin J Chadwick, Thomas

Degris, Joseph Modayil, et al. Vector-based navigation using grid-like representations in artificial agents. *Nature*, 557(7705):429, 2018.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

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

[Ber08] Dimitri P Bertsekas. Neuro-dynamic programming. In *Encyclopedia of optimization*, pages 2555–2560. Springer, 2008.

[BES10] Stefano Baccianella, Andrea Esuli, and Fabrizio Sebastiani. Sentiwordnet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining. In *Lrec*, volume 10, pages 2200–2204, 2010.

[BHP17] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *AAAI*, pages 1726–1734, 2017.

[BL04] Steven Bird and Edward Loper. Nltk: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, page 31. Association for Computational Linguistics, 2004.

[BMHB+18] Gabriel Barth-Maron, Matthew W Hoffman, David Budden, Will Dabney, Dan Horgan, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[Bot12] Matthew Michael Botvinick. Hierarchical reinforcement learning and decision making. *Current opinion in neurobiology*, 22(6):956–962, 2012.

[Car98] Rich Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

[Com] CommonCrawl. Commoncrawl. `https://commoncrawl.org/`.

[Cor79] Daniel D Corkill. Hierarchical planning in a distributed environment. In *IJCAI*, volume 79, pages 168–175, 1979.

[DD08] Peter Dayan and Nathaniel D Daw. Decision theory, reinforcement learning, and the brain. *Cognitive, Affective, & Behavioral Neuroscience*, 8(4):429–453, 2008.

[DFJ+81] Michael Alan Howarth Dempster, ML Fisher, L Jansen, BJ Lageweg, Jan Karel Lenstra, and AHG Rinnooy Kan. Analytical evaluation of hierarchical planning systems. *Operations Research*, 29(4):707–716, 1981.

[DH93] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in neural information processing systems*, pages 271–278, 1993.

[DHK+17] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. *GitHub, GitHub repository*, 2017.

[e.a] Marius Dupuis e.a. Opendrive format specification, rev. 1.4. `http://opendrive.org/docs/OpenDRIVEFormatSpecRev1.4H.pdf`.

[ESM+18] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018.

[FvHM18] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[GBG+18] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. Learning word vectors for 157 languages. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[GRMJ15] Debasis Ganguly, Dwaipayan Roy, Mandar Mitra, and Gareth JF Jones. Word embedding based generalized language model for information retrieval. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 795–798. ACM, 2015.

[GWFM$^+$13] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[HW79] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[HWT$^+$16] Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[Izq] Ruben Izquierdo. Opener project - sentiment lexicons. `https://github.com/opener-project/sentiment-lexicons`.

[JCD$^+$18] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *arXiv preprint arXiv:1807.01281*, 2018.

[JMC$^+$16a] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[JMC+16b] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *CoRR*, abs/1611.05397, 2016.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[KNP13] Tomás Kincl, Michal Novák, and Jirí Pribil. Getting inside the minds of the customers: automated sentiment analysis. In *European Conference on Management, Leadership & Governance*, page 122. Academic Conferences International Limited, 2013.

[LG09] Chi-Tat Law and Joshua I Gold. Reinforcement learning can account for associative and perceptual learning on a visual-decision task. *Nature neuroscience*, 12(5):655, 2009.

[LHC06] Ping Li, Trevor J Hastie, and Kenneth W Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 287–296. ACM, 2006.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[LM92] Long-Ji Lin and Tom M Mitchell. *Memory approaches to reinforcement learning in non-Markovian domains*. Citeseer, 1992.

[LWT+17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[MBM+16a] 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.

[MBM+16b] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[Miy] Kosuke Miyoshi. Unreal implementation. `https://github.com/miyosuda/unreal`.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Mos] Moses. Moses. `http://www.statmt.org/moses/?n=Moses.Baseline`.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[NLT] NLTK. Tokenize package. `http://www.nltk.org/api/nltk.tokenize.html`.

[NRR+16] Preslav Nakov, Alan Ritter, Sara Rosenthal, Fabrizio Sebastiani, and Veselin Stoyanov. Semeval-2016 task 4: Sentiment analysis in twitter.

In *Proceedings of the 10th international workshop on semantic evaluation (semeval-2016)*, pages 1–18, 2016.

[NSSM15] Petra Kralj Novak, Jasmina Smailović, Borut Sluban, and Igor Mozetič. Sentiment of emojis. *PloS one*, 10(12):e0144296, 2015.

[PAED17] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, volume 2017, 2017.

[PFN16] Nicholas Prllochs, Stefan Feuerriegel, and Dirk Neumann. Detecting negation scopes for financial news sentiment using reinforcement learning. In *System Sciences (HICSS), 2016 49th Hawaii International Conference on*, pages 1164–1173. IEEE, 2016.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[RFN17] Sara Rosenthal, Noura Farra, and Preslav Nakov. Semeval-2017 task 4: Sentiment analysis in twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, 2017.

[SA18] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[Sah08] Magnus Sahlgren. The distributional hypothesis. *Italian Journal of Disability Studies*, 20:33–53, 2008.

[Sch95] Helmut Schmid. Treetagger| a language independent part-of-speech tagger. *Institut für Maschinelle Sprachverarbeitung, Universität Stuttgart*, 43:28, 1995.

[SLA$^+$15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[SLH$^+$14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[SML$^+$15] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[Sova] Francesco Sovrano. Framework for actor-critic deep reinforcement learning algorithms. `https://github.com/Francesco-Sovrano/Framework-for-Actor-Critic-deep-reinforcement-learning-algorithms`.

[Sovb] Francesco Sovrano. Generic hierarchical deep reinforcement learning for sentiment analysis. `https://github.com/Francesco-Sovrano/Generic-Hierarchical-Deep-Reinforcement-Learning-for-Sentiment-Analys`

[SP02] Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on abstraction, reformulation, and approximation*, pages 212–223. Springer, 2002.

[SPS99] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[SS00] Ron Sun and Chad Sessions. Self-segmentation of sequences: automatic formation of hierarchies of sequential behaviors. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 30(3):403–418, 2000.

[Sut96] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in neural information processing systems*, pages 1038–1044, 1996.

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

[Tena] Tensorflow. Tensorflow's l2 loss. `https://www.tensorflow.org/api_docs/python/tf/nn/l2_loss`.

[Tenb] Tensorflow. Tensorflow's softmax. `https://www.tensorflow.org/api_docs/python/tf/nn/softmax`.

[THF+17] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2753–2762, 2017.

[VHGS16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[VMO+16] Alexander Vezhnevets, Volodymyr Mnih, Simon Osindero, Alex Graves, Oriol Vinyals, John Agapiou, et al. Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*, pages 3486–3494, 2016.

[VOS+17] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1703.01161*, 2017.

[WBH$^+$16] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

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

[Wen18] Lilian Weng. Policy gradient algorithms. `https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#ddpg`, 2018.

[Wik18a] Wikipedia contributors. Cosine similarity — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=861130874`, 2018. [Online; accessed 29-September-2018].

[Wik18b] Wikipedia contributors. Cross entropy — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Cross_entropy&oldid=860088796`, 2018. [Online; accessed 28-September-2018].

[Wik18c] Wikipedia contributors. F1 score — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=F1_score&oldid=861659585`, 2018. [Online; accessed 29-September-2018].

[Wik18d] Wikipedia contributors. Matthews correlation coefficient — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Matthews_correlation_coefficient&oldid=854490628`, 2018. [Online; accessed 29-September-2018].

[Wik18e] Wikipedia contributors. Probability density function — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Probability_density_function&oldid=860853926`, 2018. [Online; accessed 28-September-2018].

[WMG+17] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288, 2017.

[Wol] WolframMathWorld. Cubic spline. `http://mathworld.wolfram.com/CubicSpline.html`.