Radboud Universiteit Nijmegen

Master Thesis

Hierarchical Reinforcement Learning in Space Fortress

Internal Supervisors:
Marcel van Gerven
Tom Heskes

External Supervisors:
Joost Van Oijen
Gerald Poppinga

Author:
Víctor García Cazorla

July, 2018

Abstract

Space Fortress (SF) is a research tool originally designed to train humans in complex tasks which involve the concurrent and coordinate use of perception and motor skills, conceptual and strategic knowledge, at the service of multiple goals [Mané and Donchin, 1989]. Having a Machine Learning (ML) model of SF could be helpful for designing optimal plans for training and selection of personnel, which can be very resource consuming. SF has recently sparked the interest of the Reinforcement Learning (RL) community, because it requires the agent to learn abrupt context-dependent shifts in strategy and temporal sensitivity [Agarwal et al., 2018]. This master thesis presents the first RL agent to learn SF from a sparse reward signal effectively and reach human performance. This agent builds upon *hDQN*, an idea of [Kulkarni et al., 2016]. It is an implementation of Hierarchical Reinforcement Learning where the agent utilizes temporal abstraction over actions in the form of goals in order to make an efficient exploration of its environment. After several experiments conducted with different reward functions and various levels of abstraction for *hDQN* it is demonstrated the potential of Hierarchical Reinforcement Learning in scenarios with scarce positive feedback coming from the environment and the importance of designing reward functions that indicate *what* we want our agents to learn, not *how* we want them to do it.

Les dedico este trabajo de final de máster a mis padres, por todo el apoyo y amor recibido durante estos años. Mi carrera como estudiante no hubiese podido llegar hasta aquí sin vosotros.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Previous approaches . . . 6
- 1.3 Research questions . . . 7

2 Methods **9**
- 2.1 Classic Reinforcement Learning . . . 9
  - 2.1.1 Policies and Value functions . . . 12
  - 2.1.2 Q-learning . . . 13
- 2.2 Deep Reinforcement Learning . . . 14
  - 2.2.1 Multilayer Perceptron . . . 15
  - 2.2.2 Deep Q learning . . . 17
- 2.3 Extensions to Deep Q learning . . . 17
  - 2.3.1 Target Network . . . 17
  - 2.3.2 Double Q learning . . . 17
  - 2.3.3 Dueling architecture . . . 18
  - 2.3.4 Experience Replay . . . 19
  - 2.3.5 Prioritized Experience Replay . . . 20
- 2.4 Hierarchical Reinforcement Learning . . . 20
  - 2.4.1 Temporal Abstraction via Options . . . 21
  - 2.4.2 Related Work . . . 21
  - 2.4.3 h-DQN . . . 22

3 Experiments **26**
- 3.1 Toy Problem: Key MDP . . . 26
- 3.2 The Space Fortress environment . . . 29
- 3.3 Intrinsic motivation . . . 33
- 3.4 Optimal level of abstraction . . . 34
  - 3.4.1 Dense rewards . . . 34
  - 3.4.2 Sparse rewards . . . 37
- 3.5 Ablation study on hDQN extensions . . . 41
- 3.6 Human vs Machine superficial comparison . . . 43

4 Closing **44**
4.1 Conclusions 44
4.2 Discussion and future work 45

5 Bibliography **48**

Appendices **53**

Chapter 1

Introduction

This is the report of my Master's thesis project, which completes my Master's in Computing Science: Data Science at Radboud University. This research project was conducted between February and July of 2018 in collaboration with the *Nederlands Lucht- en Ruimtevaartcentrum* (Netherlands Aerospace Centre, referred to as NLR) in the Aerospace Operations Training & Simulation department (AOTS). In this chapter I will give arguments and explain how three distinct motivations were aligned to make this research project possible and meaningful. A general description of the problem and the proposed solution will be explained as well. At the end of the chapter, a set of research questions will be formulated.

1.1 Motivation

Training of personnel is made up of diverse stages and can be heavy resource-consuming. The basic idea behind this project is that if Machine Learning (ML) can be used to construct a representative model for human learning, that model could then be applied for the purpose of personnel training, personnel selection and task design. A practical example would be to use such a model to determine optimal transfer-of-training between different training environments [Roessingh et al., 2002], or to predict usability and trainability in the case of changes in the working environment. The availability of a human learning model for (part) tasks could help to optimize curricula. Furthermore, this human learning model could be used as a Partner Agent, which would improve the asymptotic performance of trainees who utilize it over those who use standard training protocols [Ioerger et al., 2003] [Whetzel, 2005]. Overall, a pilot's training is not cheap and a deeper understanding of the associated human learning curve is needed in order to optimize the process and design more efficient training plans.

The Space Fortress (SF) original video-game [Mané and Donchin, 1989] is a research tool developed at the University of Illinois and funded by the Defense Advanced Research Projects Agency (DARPA) of the United States. Its purpose was to create an environment to train humans in complex tasks which involve the concurrent and coordinate use of perceptual and motor skills, conceptual and strategic knowledge, at the service of multiple goals. In the game, the subject controls a spaceship in a friction-less environment and the objective is to destroy a fortress by shooting missiles at it, while having to defend itself from the shells shot by the fortress and the mines that are randomly spawn across the arena (more details in 3.2). The authors claim to

have designed it in such a way that it is not possible to play it in an optimal way ignoring any of its cognitive elements. SF is still of big interest in the study of complex skill acquisition (see [Towne et al., 2016, Frederiksen and White, 1989, Destefano and Gray, 2016, Lee et al., 2015] for examples) and it has been used for training of personnel on flight tasks [Gopher et al., 1994]. As an aerospace research institute, this game is highly relevant for NLR, which conducted one experiment with this tool as well. Performance metrics about several subjects being trained on SF were collected in various sessions and part of the motivation of this project is to build an Artificial Intelligent (AI) agent that learns to play SF as a cost-effective method to gain more insight about human learning and set the basis for human-machine comparisons.

Reinforcement Learning (RL) is a well known family of Machine Learning algorithms that has seen important advances in the recent years. An RL problem can be considered as a generalization of an optimal control problem [Sutton et al., 1992]. RL is heavily inspired by psychological theories of human learning such as Pavlovian and instrumental conditioning [Sutton and Barto, 1999]. These aspects make RL a promising candidate for the purpose of this project. SF has mainly three properties that make it a challenging environment for RL. The first difficulty arises when the RL agent tries to learn how to play taking game image pixels as input, having to learn to detect the angles and positions of the relevant elements of the game automatically. In this thesis this perceptual task was abstracted away from the agent and features like position and angles were provided explicitly to the agent (see discussion in section 4.2 for the full argument). The second challenge is related to a rule of the game by which the optimal firing strategy of the player has to reverse depending on the current situation. And third, the reward signal of the game is very sparse in its original form, meaning that the agent will only get useful feedback from the environment in rare occasions. To the best of my knowledge, the work of this thesis presents the first RL agent to successfully solve the second and third challenges together. This has been possible with *Hierarchical* Reinforcement Learning, and more concretely to *h-DQN*, the approach proposed in [Kulkarni et al., 2016]. The *h-DQN* agent operates a two different time-scales. At the low level it uses intrinsic motivation to choose the correct atomic actions so as to accomplish a particular *goal*. At the high level, the agent learns to optimally decide which goals – from a predefined set – should be followed. This basic idea simplifies the exploration problem dramatically and the resulting agent performs at the human expert level.

1.2 Previous approaches

Some approaches to this problem leverage the design of the game to create computational models (such as the ACT-R cognitive architecture [1]) based on the *Decomposition Hypothesis*, which states that the execution of a complex task can be decomposed into a set of information-processing components and that these components combine unchanged in different task conditions [Anderson et al., 2011, Moon et al., 2011, Moon and Anderson, 2012]. In [Whetzel, 2005] an agent was developed that could learn from an expert by observing how an experienced player handled each task within the game and it was shown that such agent better reflected the behavior of a human expert than a known expert-level agent developed using traditional knowledge elicitation techniques. NLR interest in building an AI able to learn SF dates back to the end of the 20th century. It has conducted various experiments during the last years in this direction. Inspired by task decomposition, NLR tried Dynamic Scripting (DS) [Spronck et al., 2006], a type of RL algorithm in which the agent,

[1] ACT-R cognitive architecture: `https://en.wikipedia.org/wiki/ACT-R`

instead of being given a set of atomic actions from which to choose from, is provided with a set of 'scripts'. In DS, each script consists of a computer program hard-coded by the user that resembles a particular behaviour –such as circumnavigating the fortress– and has an associated probability of being chosen at a particular time. By interacting with the environment, the DS agent learns to adjust these probabilities so that each script is triggered at the correct moment. This approach had the potential to be effective but came along with a lack of flexibility due to the hard-coded nature of the model. Seeking for a more general approach NLR moved to Deep Reinforcement Learning (DRL) and used the Deep Q Network (DQN) [Mnih et al., 2013] to try to learn the game in an end-to-end fashion from pixels. The resulting behaviour of the DRL agent was suboptimal and then the full game was divided into simpler tasks that focused on one single element of the game such as aiming or navigating with the idea of tackling them individually [2]. Certain high-order control components of the game were relaxed, such as the friction-less property of the space. Low-order behaviour did emerge in some of the mini-games but the results were still not satisfactory [3] [4]. In another attempt an implementation of the successful A3C algorithm [Mnih et al., 2016] (which outperformed DQN according to the paper) was used to learn SF. However, the agent was still not able to perform high-order controls in most of the mini-games [van der Linden et al., 2017]. Later on, [Aliko, 2017] investigated the effect of temporal extended actions in several experimental environments – including SF– by incorporating a set of scripts (inspired by DS) to the set of actions of the A3C agent. Additionally, a surprise-based intrinsic motivation technique was used to aid the exploration of the agent [Oh et al., 2015]. In the conducted experiments some visual artifacts of the game (mines randomly appearing in the screen, as will be explained in 3.2) increased the instability of the agent and the learning was unsuccessful. In the work of [Agarwal and Sycara, 2018] –which came out while I was in the middle of my experiments–a RL agent with a recurrent unit was devised and effectively learned a time-sensitivity strategy for SF. However, the game physics of the game were modified so as to make it easier for the spaceship to navigate and they had to design a dense reward function to overcome the exploration problem.

1.3 Research questions

The objective of this project is not only to build an AI to play SF but rather to acquire insight about RL algorithms, in particular *Hierarchical* RL. Taking this into account we can formulate the research questions that this thesis will attempt to answer like this:

Question I *Is it possible to learn SF with Reinforcement Learning at all?* After several attempts of trying it with state of the art RL methods it was not clear if RL was the correct answer for building an Artificial Intelligence system that learns how to play Space Fortress. In particular, from previous experiments it was concluded that the friction-less property of the environment was one of the main challenges, because some of the positive results only came out after deactivating it. Learning the required strategy to actually win the game was also thought hard but it had not been tackled yet because the implemented RL agents were still being tested in highly simplified versions of the game.

[2] Here is the repository of that work `https://github.com/rien333/SpaceFortressDQN`

[3] Sample video of the *Control Task*: `https://www.youtube.com/watch?v=D0JC1cPBpPc`

[4] Sample video of the *SF Task*, a simplified version of the full game: `https://www.youtube.com/watch?v=kiTY0FlnrW4`

Question II *Under which circumstances is it recommendable to extend RL with Hierarchical RL? Is Space Fortress the case*? It is likely that the extra features that Hierarchical RL gives are more useful in some situations than others. We would like to derive a set of rules that tells us when is it worth to use Hierarchical Reinforcement Learning instead of Reinforcement Learning. For this, a deep understanding of the limitation of both algorithms must be derived. Of course, this findings should apply outside the Space Fortress environment as well.

Question III *Do Double Q learning, Dueling architecture or Prioritized Replay Memory improve the Hierarchical Reinforcement Learning algorithm?* According to the literature, these extensions to the Deep Q learning algorithm have been proven to improve performance in different ways. They have been tested across various experimental environments but we do not know the extent of such positive impact when the set of actions of the agent is of a higher level of abstraction. This is exactly what will be tried to be answered here, although focused on the Space Fortress Environment.

Chapter 2

Methods

The purpose of this chapter is to expose the existing knowledge on which this thesis is built upon to the reader. First I will explain the basic concepts of Reinforcement Learning and the Q-learning algorithm, which will play a key role in all the experiments run in the project. Then I will give the basics of Neural Networks to better understand why it makes sense to join them with RL to produce Deep Reinforcement Learning (DRL). After that I will list several extensions to DRL that have been researched recently. Finally I will explain Hierarchical Deep Reinforcement Learning, which adds on top of the aforementioned ingredients a hierarchical feature that will have the potential to help the AI agent to solve Space Fortress.

2.1 Classic Reinforcement Learning

One of the special things about RL is that it sits at the intersection of many fields of science because it tries to solve a generic and fundamental problem: how to take optimal decisions (see figure 2.1).

An agent that uses RL learns by trial and error interactions with an environment, as it is shown in diagram 2.2. The agent receives the system's state S_t and a reward R_t associated with the last transition from S_{t-1} to S_t. Then it chooses an action A_t and performs it on the environment, which makes a transition to a new state S_{t+1} and sends it back to the agent together with a new reward R_{t+1}. This cycle is repeated and the problem for the agent is to learn the optimal decision making strategy so as to collect as much reward as possible.

[2] `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`

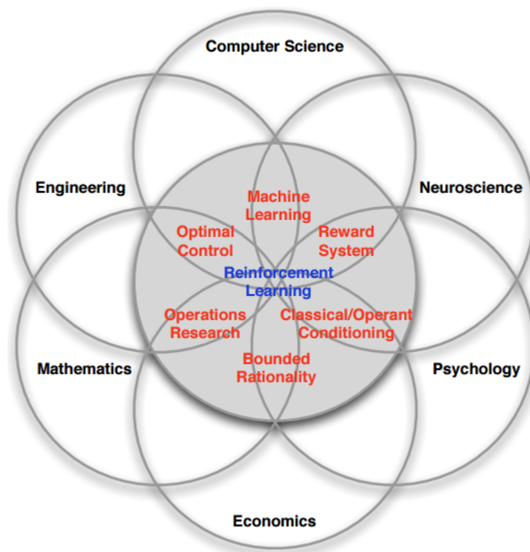

Figure 2.1: RL is of great interest because of the large number of practical applications that it can be used to address, ranging from problems in artificial intelligence to operations research or control engineering. Some examples are learning how to make fly stunt manoeuvres in a helicopter; manage an investment portfolio; control a power station; make a humanoid robot walk and outperform humans at playing (video) games (taken from the amazing David Silver's course on RL [2]).

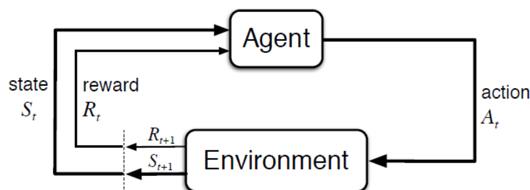

Figure 2.2: The typical setting in which a RL agent operates with an environment $\mathcal{E}$ (taken from [Sutton and Barto, 1999]).

Most of the times, the agent's observation of the environment is partial and does not describe its internal state completely, which would fully determine its dynamics and future states. This property make RL problems suitable to be described within the Markov Decision Processes (MDPs) framework. Formally, an MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where $\mathcal{S}$ is a set of states, all of which satisfy the Markov Property[3]; $\mathcal{A}$ is a set of actions; $\mathcal{P}$ is a state transition probability matrix such that $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$; $\mathcal{R}$ is a reward function such that $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$; and $\gamma \in [0, 1]$ is a discount factor that allows for the computation of the return G_t as the total discounted reward from time-step t,

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

Thus, one important difference between RL and other ML paradigms is that the 'data points' from which the agent learns are not generated i.i.d (independent and identically distributed) but rather according to the sequence of actions performed. There is no 'supervision' but only a reward signal $R_t \in \mathbb{R}$ that indicates a good or bad the agent is doing. This idea is formalized by [Sutton and Barto, 1999] with the *reward hypothesis*:

> All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal.

This concept can seem limiting at first glance but it has proved to be flexible and widely applicable. If we want our agents to do something for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. In many real world problems 'what we want our agent to learn' can only be achieved after a long sequence of very specific actions and the exploration phase of the training becomes harder. We then say that $\mathcal{R}$ is sparse. In scenarios like this it is difficult for the agent to get the first positive reward from its environment either through random, curiosity or surprise driven exploration. We may be tempted then to modify $\mathcal{R}$ so as to give the agent a prior knowledge about how to achieve what we want it to do. Nonetheless, the reward signal is our way of communicating to the agent *what* we want it to achieve, not *how* we want it achieved [Sutton and Barto, 1999]. To better illustrate why this is important we can picture a football player who is training by playing several matches with the hope of discovering a new strategy that will help his team win. $\mathcal{R}$ in this case could be 1 when the match is won and 0 in any other case. If this player could play an infinite amount of matches we could be sure –at

[3] In other words, 'the future is independent of the past given the present'. This is the Markov Property, that states that S_t is *Markov* if and only if $\mathbb{P}[S_{t+1} | S_t] = \mathbb{P}[S_{t+1} | S_1, ..., S_t]$

least asymptotically– that even if he explores by doing random actions he will eventually make his team win once and then he could reinforce its behaviour accordingly. In the real world time and resources are limited so that is not possible and in many cases we find that performing random actions is not enough to find that first positive feedback signal. We may try then to shape $\mathcal{R}$ to reflect our prior knowledge about *how* to win a match. For example, if we see that our player is not even passing the ball we could give him a positive reward for passing it to his teammates and negative for passing it to the adversary. Most likely we would then find that the agent gets more reward by passing the ball around than winning a match. One famous example from OpenAI[4] that illustrates this is a RL agent that learns how to play a boat racing video-game. To encourage the agent to drive faster towards the end of the race the researchers designed a reward function by which the agent –apart from getting a big reward in case of victory– could get small positive rewards if it acquired certain game bonuses such as turbo boosts. As this video shows[5] the agent was not even finishing the race and instead kept collecting speed bonuses, because that was the best way to maximize the total reward accumulated (see [Irpan, 2018] for more examples). Of course, there are smarter ways of shaping $\mathcal{R}$ but there is always a risk that the behaviour of the agent diverges from the goals of the designer. In practice, designing $\mathcal{R}$ is often left to an informal trial-and-error search but there are other possibilities. For instance, if we see $\mathcal{R}$ just as one more hyperparameter we could define a space of feasible candidates and then find the best one by optimization methods, such as evolutionary algorithms or online gradient ascent [Singh et al., 2010] [Singh et al., 2009]. Another approach –the one taken in this thesis– is to make the agent learn skills that can be useful across many different problems that it is likely to face in the future. These skills or *goals* will be pursued by the agent using *intrinsic* motivation –doing it for its own sake– as opposed to *extrinsic* motivation – doing it to maximize an external outcome –.

2.1.1 Policies and Value functions

The standard approach to 'solve' MDPs is to use Dynamic Programming (DP). However, DP becomes unfeasible whenever the MDP doesn't have a small number of states. Actually, RL can be thought of as a way of turning the unfeasible DP methods into practical algorithms so that they can be applied to large-scale problems [Szepesvari, 2010].

In RL, the behaviour of the agent is fully determined by a policy π, which is a distribution over actions given states,

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

In order to find an optimal policy, most of the RL algorithms need to be able to measure how good a particular state is. With this purpose, a *state-value* function $v_\pi(s)$ is defined as the expected return starting from state s, and then following a policy π,

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

Not only we want to know how good it is to be in a particular state but also how good it is to take a particular action from that state. For that matter, we define the *action-value* function $q_\pi(s, a)$ as the expected return starting from state s, taking action a, and then following policy π,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

[4] OpenAI blog post `https://blog.openai.com/faulty-reward-functions/`

[5] Boat racing environment video: `https://www.youtube.com/watch?v=tlOIHko8ySg`

To get the optimal performance in the MDP and hence solve it, we have to find the *optimal state-value function*,

$$V_*(s) = \max_\pi V_\pi(s) \tag{2.2}$$

and also the *optimal action-value function*,

$$Q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.3}$$

which ultimately indicates what is the best action to take (always choose the action for which Q_* is higher) and therefore would solve the problem. In order find V_* and Q_* some RL algorithms make use of the *Bellman Equation*[6]. In the context of value functions, the *Bellman Equation* consists in leveraging the fact that V_π obeys a recurrent decomposition into immediate reward plus discounted value of the next state,

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})|S_t = s]$$

and similarly,

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

In practice, solving the *Bellman Equation* is not straightforward. The max operator in equations 2.2 and 2.3 makes the optimization problem non-linear and in general it isn't possible to find a closed form solution. The most common strategy in this scenario is to resort to iterative methods. When the MDP is known and $\mathcal{S}$ is relatively small using DP algorithms such as *Value Iteration* or *Policy Iteration* works well. However, when the MDP is unknown, which is the most common case, RL methods are the best answer so far.

2.1.2 Q-learning

Temporal Difference learning (TD) is a central idea to RL [Sutton and Barto, 1999]. TD methods can learn directly from raw experience without a model of the environment's dynamics (they are *model-free*). They can also learn a value function directly from experience where predictions are used as targets during the course of learning (they *bootstrap*). And they do it in an online and fully incremental way, so there is not need to wait until the end of a trajectory to learn. One of the most successful TD algorithms is Q-learning [Watkins and Dayan, 1992] which is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha TD \tag{2.4}$$

Where α indicates the learning rate and TD is the *temporal difference* error,

$$TD = y_{true} - y_{pred} \tag{2.5}$$

$$y_{true} = R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \tag{2.6}$$

$$y_{pred} = Q(S_t, A_t) \tag{2.7}$$

We see then that 1) Q-learning bootstraps with only one step into the future and 2) it refines the policy greedily with respect to action values by the max operator. This rule makes Q-learning an *off-policy* algorithm which means that Q_* is approximated by Q independently of the policy being

[6]Many others RL algorithms do not use the *Bellman Equation*. For instance, in contrast to value-based methods, policy-based methods such as the popular REINFORCE [Williams, 1992] don't make use of it and optimize π directly.

followed during training (see algorithm 1). This simplifies the analysis of the algorithm and enables early convergence proofs [Sutton and Barto, 1999]. Indeed, a major attraction of Q-learning is its simplicity and that it allows to use an arbitrary sampling strategy to generate the training data provided that in the limit, all state-action pairs are updated infinitely often [Szepesvari, 2010]. One of the most commonly used is the ϵ-greedy strategy, which consists in choosing actions randomly with probability ϵ and otherwise choose the action $a = \text{argmax}_a Q(s, a)$. The idea is to set $\epsilon = 1$ at the beginning of the training (where the agent *explores* the environment) and to and anneal it towards $\epsilon \to 0^+$ at the end (when the agent *exploits* the environment).

Algorithm 1 Q-learning

```
initialize Q arbitrarily, e.g. to 0 for all states, set action value for terminal states as 0
for each episode do
  initialize state s
  for each step of episode, state s is not terminal do
    a ← action for s derived by Q, e.g., ϵ-greedy
    take action a, observe r, s′
    Q(s,a) ← Q(s,a) + α[r + γ max_a′ Q(s′,a′) − Q(s,a)]
    s ← s′
  end for
end for
```

As [Niv, 2009] pointed out, there is some evidence suggesting that dopaminergic neurons in the brain may be responsible for the generation of a TD error based on state-action values like those used in Q learning [Morris et al., 2006].

2.2 Deep Reinforcement Learning

In many of the tasks to which we would apply RL the state space $\mathcal{S}$ can be enormous because the number of combinations is too high or simply because its domain is continuous. In such cases we cannot expect to find neither q_* or v_* even in the limit of infinite time and data; our goal instead is to find a good approximate solution using limited computational resources. In many real world problems each new state will almost always have not been seen before. Therefore, we need a *function approximation* that *generalizes* well [Sutton and Barto, 1999]. For this matter we resort to methods used for pattern recognition and statistical curve fitting. The fact that any of this algorithms works well in classic supervised learning problems does not translate directly to RL, since RL encounters new issues related with nonstationarity, bootstrapping and delayed targets. Linear approximation used to be a popular choice, partially because of its nice theoretical properties and also it has potential for explainability. However, non-linear approximators have captured most of the attention in the last decade. The usage of Artificial Neural Networks (ANNs) in combination with RL was first introduced in [Bertsekas and Tsitsiklis, 1995] but it was not until the work of [Mnih et al., 2013] that it became the norm among RL researchers. This combination also is used in this work.

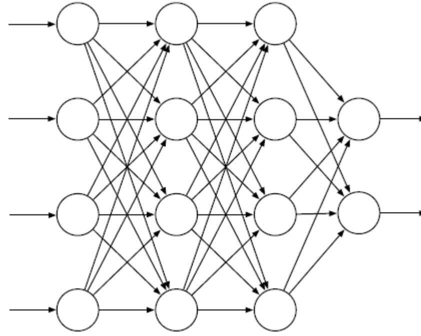

Figure 2.3: Simplified scheme of a a four layered Multilayer Perceptron, from right to left: one input layer of 4 neurons, two hidden layers of four neurons each and one output layer of two neurons.

2.2.1 Multilayer Perceptron

Perceptrons are the simplest feed-forward ANNs and were first invented in 1957 by Frank Rosenblatt [Rosenblatt, 1957], who also provided in 1962 the learning perceptron's rule with proven convergence [Rosenblatt, 1962]. These can be grouped in layers and then stacked together to form a Multilayer Perceptron or MLP (see figure 2.3). An MLP will be used to build the $Q(s,a)$ function in our agent that will approximate q_*. Each neuron in a MLP has a bias and weighted connections with every neuron of the next layer. These weights and biases are the model's parameters and will be referred to as θ. The activations of each layer are transformed using a differentiable, nonlinear function to provide an output to the next layer, which makes them universal approximators. In this work ReLU's are used as activation functions ($f(x) = \max(0,x)$). After each forward pass, these supervised learning models output a 'prediction' which error can be measured in terms of θ via the $L(\theta)$ loss function. In supervised learning a popular choice for the loss function is the mean squared or absolute error. However, when learning q_* the 'true labels' are built upon a potentially sparse scalar signal $\mathcal{R}$ that comes from an stochastic environment. This means that often our agent will encounter outlier experiences that will lead to large updates in θ and make convergence harder. To aid this problem we can clip the produced Bellman gradient. In this thesis though, a more robust function called the Huber loss [Huber, 1964] will be utilized,

$$L_\delta(\theta) = \begin{cases} \frac{1}{2}TD^2 & \text{for} \quad |TD| \leq \delta, \\ \delta(|TD| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{2.8}$$

that is quadratic for small values of TD and linear for large values, with equal values and slopes of the different sections at the two points where $|TD| = \delta$ (see figure 2.4). In this thesis δ is set to 1.

The idea is then to solve $\nabla L(\theta) = 0$ so that we can know what is the θ that minimizes the error loss. There is no method to do this analytically hence we have to resort to iterative numerical procedures that update the parameters on each time step t following:

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

[8]Huber Loss Wikipedia definition `https://en.wikipedia.org/wiki/Huber_loss`

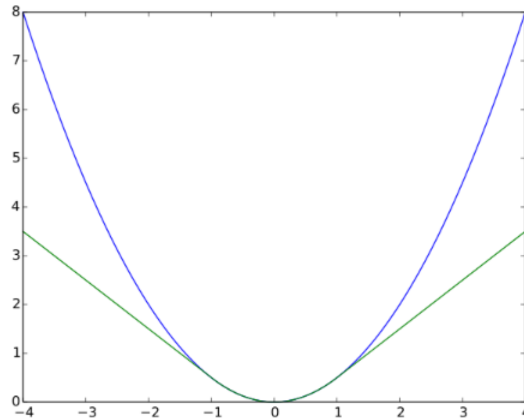

Figure 2.4: Huber loss (green, $\delta = 1$) and squared error loss (blue) as function of $y - f(x)$. Taken from Wikipedia [8].

Most of the successful techniques of this optimization problem rely on backpropagation, which is a procedure that provides gradient information in the form of $\partial L(\theta_i)/\partial\theta_i$ where $i = 0, ..., P$ and P is the number of parameters in θ. There are different ways of introducing these gradients into the weight vector update term $\Delta\theta_t$ of equation 2.2.1 in order to find a θ for which $L(\theta) \approx 0$. The most basic make use of a learning rate α to update the parameters at each time step t like this:

$$\theta_{t+1} = \theta_t - \alpha\nabla L(\theta_t) \tag{2.9}$$

The idea is that at each step θ is moved in the direction of the greatest rate of decrease of $L(\theta)$. In the *batch* gradient descent variant the whole training set is used to compute $L(\theta)$. On the other hand there is the *stochastic* or online version, in which only a different small portion is used to compute $L(\theta_t)$ at each t. In a RL context, where the training set is comprised of experiences that are generated while learning is happening the *stochastic* variant is more suitable.

In this thesis I used a more advanced gradient descent optimization algorithm called Root Mean Square Propagation (RMSprop) [Tieleman and Hinton, 2012]. It provides a speed-up over the vanilla gradient descent in equation 2.9 by keeping one learning rate α_i for each network parameter θ_i. The idea is to divide α_i for a weight by a running average of the magnitudes of recent gradients for that weight. Furthermore, the gradients ∇L computed by deriving the loss function are clipped to 1 before making the update of equation 2.9. This can make stochastic gradient descent behave better in the vicinity of steep cliffs [Goodfellow et al., 2016]. This is a common practice in Deep RL to tackle the usual instability of the algorithms [Wang et al., 2015] [Mnih et al., 2016] (in [Mnih et al., 2015] the *rewards* are clipped instead of the *gradients*).

An MLP that has one hidden layer containing a large enough finite number of non-linear units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy [Cybenko, 1989]. This gives them the name of 'universal approximators' and is a nice theoretical result but actually some architectures are much easier to learn than others. For instance, empirical results show that adding more hidden layers and hence forcing the network to learn features of higher level of abstraction can help learning substantially [Bengio et al., 2009].

2.2.2 Deep Q learning

In section 2.1.1 the advantages of *off-policy* methods such as Q-learning were introduced. Unfortunately, when using them in conjunction with non linear function approximation, instability increases and divergence may occur [Tsitsiklis and Van Roy, 1997].Why this happens and how to optimally handle it is still a cutting-edge research area [Li, 2017].

Few years ago [Mnih et al., 2013] revolutionized the field of RL with the Deep Q Network (DQN), a model that was able to learn how to play Atari games end-to-end. The DQN agent took raw pixels as input so they used Convolutional Neural Networks (CNNs) as function approximator of $\mathcal{S}$. CNNs are a type of ANN widely used for extracting features from images. In this thesis, how it will be explained later, $\mathcal{S}$ will be a higher level abstraction of the pixel space, so an MLP will be used instead of a CNN. Despite this difference, the combination of MLP and Q-learning will be referred to as DQN throughout this report.

2.3 Extensions to Deep Q learning

Apart from showing how RL can be combined with image input, the original DQN of [Mnih et al., 2015] introduced a couple of techniques to help stabilizing the aforementioned instability issues, namely *Experience Replay* and a *Target Network*. These two features among three other extensions to the original Q learning posterior to DQN (*Double Q*, *Dueling* architecture and *Prioritized Experience Replay*) were implemented in this project and I will explain briefly in what they consist.

2.3.1 Target Network

In equations 2.5, 2.6 and 2.7 we see that at each training step, the same version of $Q_\theta(S, A)$ is used (*online network*), which intuitively makes sense. However, this means that we are generating our y_{true} values from a function that is changing. In other words, the y_{true} generated values are *inconsistent* and this is an important source of instability. To tackle this problem [Mnih et al., 2013] proposed to maintain a copy of the *online network*, which we will call the *target* or *offline network*, with parameters θ^- and referred to as $Q_{\theta^-}(S, A)$ from now on. The idea is to train our *online network* via RMSprop as we would normally do with the only difference that the y_{true} target samples will be generated from the *target network*, whose weights θ^- will be copied from θ from time to time. This simple trick makes the target values y_{true} to be ‘outdated’ but much more *consistent* at the same time, so $Q_\theta(S, A)$ will struggle less to converge.

2.3.2 Double Q learning

The classic Q learning algorithm (see pseudocode 1) is known to produce overestimated action-value pairs due to the max operator when computing the targets y_{true}. This is not necessarily an issue. For instance, it wouldn’t be problem if all the action-value pairs are uniformally overestimated. Furthermore, [Kaelbling et al., 1996] suggested that optimism is a good exploration technique in the face of uncertainty. However, if the overestimations are not uniform and not concentrated at states about which we wish to learn more, they can lead to suboptimal policies. [Van Hasselt et al., 2016] gave empirical evidence that overestimations have a negative performance when using the DQN algorithm. They did it comparing the baseline DQN with and extended version of it that they called Double DQN, which yields better estimates of y_{true} and results into better policies.

Double DQN is inspired by [Hasselt, 2010], where the original Double Q learning algorithm was proposed. To better understand how the overestimation is solved it must be noted that in the regular update rule $y_{true} = R_{t+1} + \gamma \max_a Q_\theta(S_{t+1}, a)$ the same network (the *online network*, Q_θ) is being used for both selecting the action and also for evaluating its value. This is easily seen if we rewrite it like

$$y_{true} = R_{t+1} + \gamma Q_\theta(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_\theta(s_{t+1}, a))$$

What Double Q learning proposes is to decouple selection from evaluation by learning two value functions Q_θ and $Q_{\theta'}$ whose weights are learning by assigning each experience randomly to either of the networks. This effectively allows for decoupling. More precisely, we can now select the action using Q_θ and evaluate it with $Q_{\theta'}$. To implement this novel idea, Double DQN leverages the fact that a second network is already being used, namely the *target network*. Periodically copying $\theta^- \leftarrow \theta$ doesn't make Q_{θ^-} fully decoupled from Q_θ as $Q_{\theta'}$ is from Q_θ but as the author say in the paper:

> This version of Double DQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most of the benefit of Double Q-learning, while keeping the rest of the DQN algorithm intact for a fair comparison, and with minimal computational overhead.

Therefore, the formula for the target values in the update rule of the Double DQN algorithm require no additional networks and is formalized as

$$y_{true} = R_{t+1} + \gamma Q_{\theta^-}(s_{t+1}, \underset{a}{\operatorname{argmax}}\, Q_\theta(s_{t+1}, a)) \tag{2.10}$$

It should be noted that introducing this extension in the algorithm implies having to do two forward passes when computing (first one with Q_θ and one with Q_{θ^-} afterwards) while in the normal DQN only one with Q_{θ^-} is enough.

2.3.3 Dueling architecture

[Wang et al., 2015] proposed a novel ANN architecture called *dueling* that could be easily combined with *model free* RL algorithms, such as DQN. The architecture (see figure 2.5) is designed in such a way that when the Q_θ of the agent takes a state s as input and before producing any output it must build internally a representation of how good it is to be in s, $V_\theta(s)$; and a relative measure of importance of each possible action. This last quantity is called the *advantage function*:

$$A_\theta(s, a) = Q_\theta(s, a) - V_\theta \tag{2.11}$$

It must be clarified that even though Q, V and A share the subscript θ in equation 2.11, V and A actually refer to different modules of the network hence they have different parameters. This is omitted for simplification. By making the separation explicit we achieved that for many states, it is unnecessary to estimate the value of each action choice. The assumption is that while in some states it is of highly importance to know which action to take, in many others the choice is irrelevant.

The way the outputs of A_θ and V_θ are gathered together to form the final output of Q_θ is not as straightforward as it may seem. Given the relation in equation 2.11 we could be tempted to aggregate them together implementing $Q_\theta(s, a) = V_\theta(s) + A_\theta(s, a)$ but this equation is unidentifiable in the sense that given Q recovering V and A uniquely is not possible, which causes poor

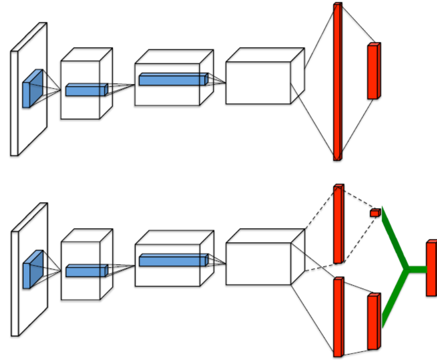

Figure 2.5: Figure taken from the original paper [Wang et al., 2015]. At the top there is the common single stream network and at the bottom the *dueling* architecture. The lower layers of both architectures are CNNs, which are not used in this thesis (MLP will be utilized instead). What is important to note is that the *dueling* architecture has two streams to separately estimate $V_\theta(s)$ (scalar value) and the different advantages $A_\theta(s,a)$, which are later aggregated together according to equation 2.12 (in green).

practical learning performance when used directly. The authors of [Wang et al., 2015] argue that a workaround with desirable algorithmic properties consists into forcing the A estimator to have zero advantage at the chosen action and they explain that it can be achieved implementing

$$Q_\theta(s,a) = V_\theta(s) + \Big(A_\theta(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A_\theta(s,a')\Big) \tag{2.12}$$

where as clarification it must be noted that $\mathcal{A}$ is the set of possible actions, different from the *advantage function*, A.

2.3.4 Experience Replay

We can define each experience of the agent as a tuple $e_t = (s_t, a_t, r_t, s_{t+1})$. This is read as 'from state s_t action a was used to go to a new state s_{t+1} and generate a reward r_t'. In most of the cases, as the agent interacts with the environment we expect the experiences of the sequence e_0, e_1, e_2... etc. to be highly correlated with each other. Due to this, if the agent processed those experiences and update θ in the same order as they were generated its learning would be inefficient [Riedmiller, 2005]. [Mnih et al., 2013] proposed to leverage the fact that Q learning is an *off policy* algorithm by randomizing the order in which the generated experiences are actually used to update Q_θ. In other words, since in Q learning the policy used to interact with the environment is already different from the policy being learn, there is no need to learn from experiences as they are generated, so we can get rid of the correlations between experiences by randomizing them.

This is the goal of using Experience Replay, which was first proposed in [Lin, 1993]. The idea is to store each e_t in a N-size dataset $\mathcal{D} = e_0, ..., e_N$ which is called the *replay memory*. The agent follows a particular exploration strategy, such as ϵ-greedy to generate e_t and then storing it in $\mathcal{D}$.

Then, in order to update Q_θ it periodically samples a batch of experiences from the *replay memory* $e \sim \mathcal{D}$.

2.3.5 Prioritized Experience Replay

Prioritized Experience Replay is a work from [Schaul et al., 2015b] that also builds on top of DQN. It consists in having a *replay memory* $\mathcal{D} = e_0, ..., e_N$ as explained in 2.3.4 but changing the way experiences are sampled from it when updating Q_θ. The idea is to leverage the fact that some experiences are more significant than others. More significant in this context means that such experience has information that the agent does not know yet. Therefore, if instead of drawing experiences uniformally random from $\mathcal{D}$ we could bias our sampling strategy towards those experiences that are more significant then the learning process would be more efficient.

A good proxy for the significance of e_t is to measure the *temporal difference* error (TD, equation 2.5) that was produced by the agent when it experienced e_t. TD error can be understood as indicating the level of surprise or how unexpected a particular transition was. This approximation is also inspired by the results of the neuroscience studies of [Singer and Frank, 2009] and [McNamara et al., 2014], which suggest that experiences with high TD error appear to be replayed more often in humans and animals.

In the original paper two variants of implementing this idea are proposed: proportional priorization and rank-based priorization. In this work the former is used. The probability of sampling transition i is

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.13}$$

where $p_i = |TD| + \epsilon$ is the priority of transition i. ϵ is a small positive constant that ensures that a every transition has some probability of being sampled even if it has TD=0 and $0 \leq \alpha \leq 1$ determines how much priorization is used ($\alpha = 0$ being the uniform case). As the training goes the priority of the stored experiences should change as well because the level of surprise of some experiences – even if they have not been sampled yet – changes every time Q_θ is updated. It would be very inefficient to recompute all the priorities after each update and have them in $\mathcal{D}$ as a sorted array. To make an effective implementation feasible for large $\mathcal{D}$ the authors proposed to use a 'sum-tree' data structure for $\mathcal{D}$, where the parent's value is the sum of its children (and the experiences are in the leaf nodes). Also, the values p_i in $\mathcal{D}$ are only updated for those experiences that belong sampled batches for each W_θ update. This setting introduces a bias that can harm the convergence of the algorithm. This bias is corrected by computing Importance Sampling weights like

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta \tag{2.14}$$

and using $w_i TD$ instead of just TD in the update rule of equation 2.4. As the authors argue on the paper, this importance of having unbiased updates is bigger at the end of the training than at the beginning. Therefore the hyperparameter β is linearly annealed from 0 to 1 during the process.

2.4 Hierarchical Reinforcement Learning

One of the biggest problem in RL and also DRL is how to generate effective exploration strategies to deal with environments that have sparse rewards. This situation occurs often in problems for which

RL has lots of potential. As it will exposed later, Space Fortress can be one of those situations. To deal with it I propose to use hDQN, an implementation of Hierarchical Reinforcement Learning combined with ANNs that utilizes temporal abstraction over low level actions conceptualized as goals to help the agent explore its environment in an explicit manner.

2.4.1 Temporal Abstraction via Options

Classic exploration strategies such as ϵ-greedy are based in behaving randomly at the beginning of the training session with the hope that at some point a positive reward will be encountered. Then, the agent will reinforce the actions that led to that particular situation to make it happen again and then – if $\mathcal{R}$ is well specified – we could say that the 'hard part' of the problem is solved. Exploration strategies are at the core of RL research and there are other more powerful alternatives to ϵ-greedy such as Thomson sampling or Boltzmann exploration [Stadie et al., 2015] [Osband et al., 2016], but they all operate at the level of primitive actions. This becomes problematic in environments where $\mathcal{R}$ is highly sparse, because then encountering a positive reward by trying out different combinations of low primitive actions is almost impossible. Another sometimes valid approach was proposed in [Nair et al., 2017], who recorded human experiences used as demonstrations to aid the agent explore the environment more effectively.

Sparse rewards scenarios could benefit from the fact that an agent, besides having its original set of primitive actions $\mathcal{A}$ had another type of actions that were temporally abstracted from $\mathcal{A}$. For instance, when humans are learning a task their $\mathcal{A}$ is composed by actions at different temporal scales. Lets recall the example of our football player from section 2.1. At the beginning he only knows the very basics so he has to *explore* to find a good policy. Lets assume for simplicity that he follows something close to an ϵ-greedy strategy. He will rarely think of trying out random sequences of muscles contractions and twitches, which are very low level actions until his team wins. Instead, he will try random sequences of actions at a higher level of abstraction. For instance, he may try to pass at a random player each time or shoot the ball at different heights. Of course, this exploration technique is naive compared to a normal human but the chances that at some point he will make his team score a goal by following it instead of thinking only at the muscular level are much higher.

These actions of higher level of abstraction are known as *options* in RL [Sutton and Barto, 1999]. Options are understood as policies over actions and even over other options, so that under this new setting the mathematical framework is formalized as a *Semi Markov Decision Process* (SMDP). If the set of actions from which an agent can choose from is composed by actions with different levels of abstraction we say that the agent is able to 'think' at different time scales. We can easily see that human learning is not only full of abstractions but also that we have the ability to choose the appropriate level of abstraction at any time. Implementing an agent that combine low-level primitive actions and options is a central topic of this thesis.

2.4.2 Related Work

In order to learn options in real-time, [Szepesvari et al., 2014] proposes the *Universal Option Model* to construct option models independently from the reward function and they experiment on strategic games and article recommendations. In [Sorg and Singh, 2010] a knowledge construct is developed, the *Linear Option* model, which is capable of modeling temporally abstract dynamics in continuous state spaces with a linear expectation model and give some theoretical convergence guarantees of the TD error. In [Vezhnevets et al., 2016] the Strategic Attentive Writer (STRAW)

was introduced: and end-to-end implementation of an agent that learns options using a deep recurrent neural network. STRAW was able to automatically discover multi-step plan and the authors validated its work on next character prediction in text, navigation of a 2D maze and some Atari games. [Florensa et al., 2017] developed a framework that uses a Stochastic Neural Network to learn a span of skills in an unsupervised manner with proxy reward signals. After, this is leveraged for aiding exploration by training a hierarchical structure on top of it.

Another approach is Hierarchical Reinforcement Learning in which the separation between the execution of actions and options is often made explicit in the architecture of the learning model. With this idea [Vezhnevets et al., 2017] created the FeUdal network, where a part of the architecture called the *manager* produces and sets goals which are then accomplished by another part called *worker* operating at a lower time-scale and utilizes low-level actions to realize those goals. [Schaul et al., 2015a] introduced the *Universal Value Function Approximator* to learn value functions $V_\theta(s, g)$ that are able to generalize over combinations of states $s \in \mathcal{S}$ and goals $g \in \mathcal{G}$. θ are the parameters of the model, which can be an ANN if we are using function approximation. This idea inspired [Kulkarni et al., 2016], which developed the *Hierarchical Deep Q Network* (h-DQN), a framework to integrate hierarchical value functions, operating at different temporal scales, with intrinsically motivated DRL.

2.4.3 h-DQN

This is the chosen learning model for this project. h-DQN learns $V_\theta(s, g)$ taking $\mathcal{G}$ as a hand-crafted predefined set with all the goals that can be accomplished by the agent, leaving the agent with the task of learning how to achieve them and when to use them. The lack of an automated goal discovery mechanism has been the most criticized aspect of the work of [Kulkarni et al., 2016], specially from those sectors in the RL community that focus on general end-to-end to learning systems. As usual, there exists a trade-off between the generality of a solution and its efficacy. General RL stills suffers from important problems (see ‘Deep Reinforcement Learning Doesn’t Work Yet’ [Irpan, 2018]), so in projects that focus on solving one particular problem - like this thesis tries to solve Space Fortress - one must take into consideration where to position in the aforementioned trade-off. Due to the struggle of previous attempts at learning Space Fortress I considered that exchanging some generality for effectiveness was a good decision. h-DQN allows for a very flexible definition of ‘goals’ and its architecture is simpler than most of the approaches discussed in section 2.4.2, which makes it easier to implement and debug. Furthermore, the explicit definition of goals allows to easily monitor what the agent is doing at a particular time in a meaningful way. This is something valuable in cases where explainability matters (see motivation in 1.1).

The architecture of the h-DQN learning model and its description can be seen in figure 2.6. This architecture implies that two objectives are being optimized by the agent, which are formalized as follows: we can rewrite equation 2.1 as G_{MC_t}, the total *extrinsic* reward discounted from time-step t

$$G_{MC_t} = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.15}$$

which the *meta-controller* tries to maximize. And G_{C_t}, the total *intrinsic* reward discounted from time-step t

$$G_{C_t}(g) = I_{t+1} + \gamma I_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k I_{t+k+1}(g) \tag{2.16}$$

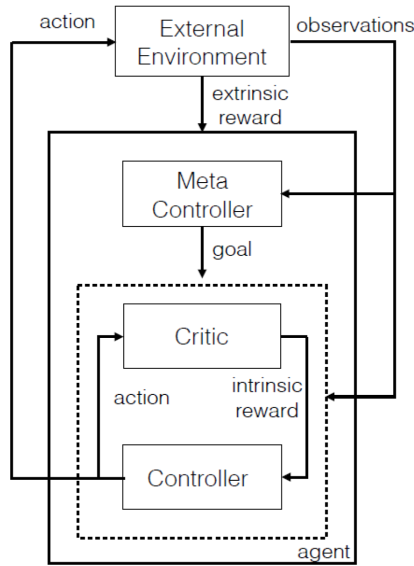

Figure 2.6: Diagram taken from [Kulkarni et al., 2016]. The h-DQN agent has two principal modules: the *meta-controller* (MC) and the *controller* (C). The *meta-controller* reads an observation s_t from the environment and uses $Q_{\theta_{MC}}(s_t, g_t)$ to set goal g_t. Then the *controller* comes into place. It takes both s_t and g_t as input and uses its Q function, $Q_{\theta_C}(s_t, g_t, a_t)$ to choose an action a_t that will be performed on the environment. The idea is that the *controller* will interact with the environment until its internal *critic* tells it if it has either failed or succeeded at reaching goal g_t. The *critic* has another important function: it also provides an *intrinsic* reward signal $I_t(g) \in \mathbb{R}$ to guide the *controller*'s learning towards accomplishing g_t. This signal is independent from the *extrinsic* reward R_t that comes directly from the environment, depends on each particular problem and it is only seen by the *meta-controller* each time the *controller* fails/succeeds at reaching g_t. The *meta-controller* then receives the total *extrinsic* reward accumulated during the N step (see diagram 2.7).

which the *controller* tries to maximize and where $I(g)$ varies depending on the goal g that the agent is following at that moment. It should be noted also that even though the same index t is used as time-step, the MC and C never run in parallel (unless the trivial case of $\mathcal{G} = \mathcal{A}$). They operate at different time scales. The execution flow of the h-DQN agent is illustrated in figure 2.7. In principle we could use any function approximator to estimate the optimal Q_* and in this case, as it was introduced in section 2.2 we will use Neural Networks. It is actually possible to understand $Q_{\theta_{MC}}$ and Q_{θ_C} as two separate DQN's, each one having its own Huber loss function (equation 2.8) L_{MC} and L_C and its own replay memory D_{MC} and D_C. An important remark is that the *controller* receives as input a concatenation of the state and the current goal, which is efficient for several reasons. First of all the *controller* is forced to find one single set of parameters θ_C that has the capability of achieving every goal in $\mathcal{G}$. Some goals in $\mathcal{G}$ may share certain characteristics and this design choice makes it possible for the *controller* to reuse behaviour patterns among them. Furthermore, the simplistic one-hot-encoding of goals allows the model to - in principle - scale up well when $\mathcal{G}$ is big.

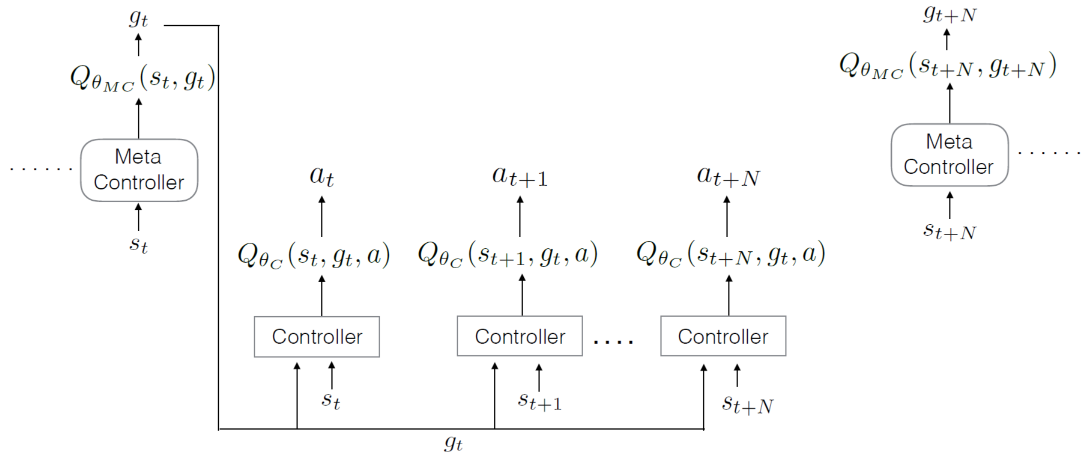

Figure 2.7: Flowchart adapted from [Kulkarni et al., 2016]. *Controller* and *meta-controller* operate at different time-scales. The *controller* sees all the states but the *meta-controller* only sees and acts upon those states occurring after the *controller* has failed/succeeded at achieving the current goal.

The exploration strategy for the *controller* and the *meta-controller* is based on ϵ-greedy but differs in how ϵ – the probability of behaving randomly – is set. In the case of the *controller* we keep an ϵ_g $\forall g \in \mathcal{G}$ and its value is set to to $1/sr_g$, where sr_g is the success rate of goal g. This quantity is computed as dividing the number of successes by the number of attempts at achieving g within the last K steps, which is an hyperparameter. If g_i has been set by the *meta-controller*, then ϵ_i is used in the *controller*. As for the *meta-controller* the classic ϵ-greedy strategy is used. It starts at $\epsilon = 1$ and it is linearly annealed towards $\epsilon = 0.05$. However, the annealing of ϵ only starts when the *controller* has learned how to achieve every $g \in \mathcal{G}$. This condition will only suffice when $sr_g > threshold$ $\forall g \in \mathcal{G}$.

The *meta-controller* takes in control whenever the *controller* succeeds/fails at achieving a goal. It is thus necessary to formalize a definition of goal achievement and failure. Each g should be

manually defined as something specific enough so that it is easier for the critic to check for its achievement at any point of the interaction with $\mathcal{S}$. Usually the achievement of g is done by checking if s_t belongs to a specific sub-region of $\mathcal{S}$ but the current action a_t could also be included in the condition. For instance, one of the goals that will be defined for learning Space Fortress is to shoot twice within a fixed (small) number of steps m. As it will be explained, a feature that will be included in s_t is the number of steps since the last shot. Therefore, the condition for checking if the goal 'double shot' is met will consist in checking if $a_t = \ Shoot$ and that the number of steps since the last shot is less than m. On the other hand, knowing when the agent has failed at achieving g is not as straightforward. Acknowledging failure is important because if the *controller* is doing very bad at pursuing a specific goal maybe the optimal thing to do for the *meta-controller* would be to set a new goal and move on. A goal can only be learned if it is achieved at some point because only then the *controller* will see $I_t > 0$. Achieving some goals may take a big number of steps the first time so one has to be very careful when setting a 'maximum number of steps per goal attempt' or something of the sort. As in the original paper, here 'failure' at achieving a specific goal is understood as an episode terminating before g is achieved.

In [Kulkarni et al., 2016] both $Q_{\theta_{MC}}$ and Q_{θ_C} are CNNs because $\mathcal{S}$ was a two-dimensional pixel space. The 'deepness' in Deep Reinforcement Learning often translates into using Neural Networks to capture complex patterns in $\mathcal{S}$, which is indeed the case when $\mathcal{S}$ consists in images. In the h-DQN version implemented in this project $\mathcal{S}$ is not a pixel-space but rather a richer set of features (positions, angles etc.) arranged in a one dimensional vector, so a Convolutional Neural Network was not needed. This allowed to use more simple functions to approximate Q. Due to time constraints, experimenting with other simpler and more explainable models was left to future work (section 4.2) and Multilayer Perceptrons were used for both the *controller* and *meta-controller*.

Chapter 3

Experiments

This chapter describes the experiments that I designed with the purpose of answering the research questions of section 1.3. Initially, I used a toy example to illustrate the main difference between h-DQN and DQN and to show a situation where DQN fails due to its lack of temporal abstraction. The main part of this chapter is about experiments executed on the Space Fortress environment, which will be fully described along with some implementation concerns. The first SF experiment consists in testing a modification that I propose on the way the *controller* is intrinsically motivated to achieve goals. What follows is the the main experiment of the project, which consists in analyzing how the incorporation of options via Hierarchical Reinforcement Learning can help the agent when the reward signal is dense and when it is sparse. Then I will describe and ablation study for testing how the DQN extensions *Dueling*, *Double Q learning* and *Prioritized Experience Replay* affect the performance of the *controller* and *meta-controller* separately. Finally, I will show a visual informal comparison between the learning curves of the different agents and that of a human learning to play SF.

The project is hosted publicly[1]. Some remarks about the experiments: several 'line plots' are included in the figures' experiments. Due to the instability of RL, the same experiment was run multiple times and the reported values consist in the mean of the value that is being measured ± one sample standard deviation. Also, since the curves were usually too peaked they were smoothed before plotting using a Savitzky–Golay filter (see `plots.ipynb` in the code)

3.1 Toy Problem: Key MDP

In this experiment the agent has to solve a deterministic *Markov Decision Process* where $\mathcal{S} = \{S_1, ...S_9\}$. $\mathcal{S}$ is arranged in a 3×3 grid and the agent can navigate it both horizontally and vertically so that $\mathcal{A} = \{up,\ right,\ down,\ left\}$. The reward function is the following:

$$R_t(S) = \begin{cases} 1 & \text{if} \quad S = S_1 \text{ and } S_9 \text{ was visited} \\ 0.1 & \text{if} \quad S = S_1 \text{ and } S_9 \text{ was not visited} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

[1] Project's repository: `https://github.com/hipoglucido/Hierarchical-DRL-Tensorflow`

The only terminal state is S_1 and the idea is to test if the agent can learn that the optimal behaviour is to first visit S_9 to 'pick up the key' instead of going directly to S_1. The agent observation from the environment is a flattened vector of the grid and it always starts at S_5. From this setting we define the optimal behaviour as going from S_5 to S_9 and then to S_1 and the suboptimal strategy as directly going to S_1 from S_5. In figure 3.1 an example of these two trajectories is shown.

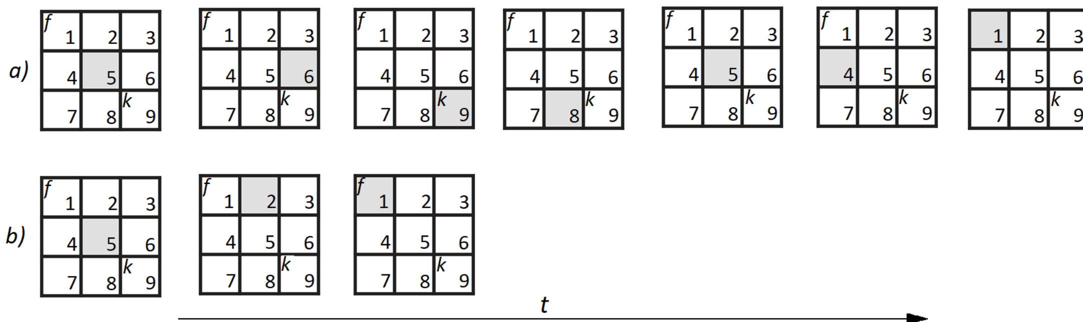

Figure 3.1: Two different trajectories for the Key MDP after training is complete. a) corresponds to hDQN and the total episode reward is 1, and b) to DQN, where the reward is 0.1.

We compare the DQN algorithm versus hDQN. With the aim of investigating only the key differences between the two we perform the experiment without extensions. In the case of hDQN $\mathcal{G} = \{G_1, ...G_9\}$, where each goal matches with reaching a particular state. For example, if the *meta controller* sets G_9 as the current goal then the critic will reward the *controller* when its actions lead the agent to reach S_9, and then the *meta controller* will set the next goal. In the left side of figure 3.2 we see the main result of this comparison in terms of reward accumulated per episode. In figure 3.1 the strategies learned by both agents are illustrated. Another interesting way of verifying that hDQN solves the problem effectively is by keeping track of the relative frequencies of the goals set by the meta-controller. In figure 3.3 it is shown that at the end of training G_9 and G_1 are the only goals used. On the other hand, DQN is not able to learn the optimal strategy because it lacks any kind of memory, which is needed in order to solve this MDP. For example, when the agent is in S_2 it cannot know if it has been previously in S_9 or not, since it bases its decisions in its observation which only tells it where it is at the current moment. By ways of contrast, in hDQN G_9 is set since the beginning and once achieved the *meta controller* chooses G_1. The current goal is part of the input to the *controller* which provides a kind of memory to know that the optimal action when the agent is in S_2 is $left$.

Another issue with the vanilla DQN was found. It was not even able to converge to the sub-optimal policy exemplified in figure 3.1 b). When ϵ becomes closer to zero the agent starts acting greedily with respect of the estimated Q values. This manifested that the agent had learned feed-back loops, causing it to perform ineffective moves such as repeatedly choosing *right* from S_3 or just moving between two adjacent states. It is believed that, since this MDP is not solvable by DQN with the current setup, finding sometimes that $R(S_t) = 1$ is a source of instability for the agent. Interestingly enough, it was found that getting rid of the overestimations of Q learning by extending the vanilla DQN algorithm with *Double Q learning* (explained in section 2.3.2) made the agent more robust to this source of instability and it helped it to converge to the suboptimal policy.

This is illustrated at the right side of figure 3.2.

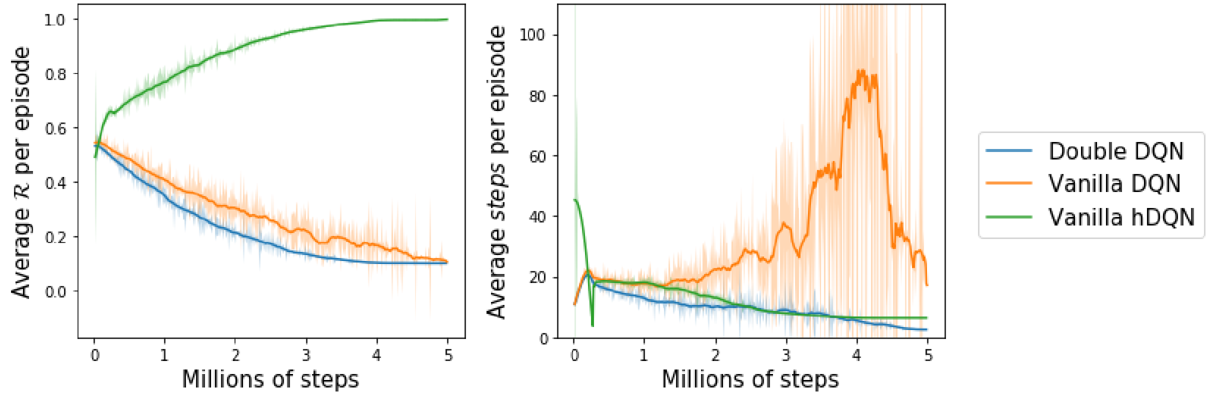

Figure 3.2: DQN versus Double DQN versus hDQN in the Key Grid MDP. Each configuration was run 5 times with different seeds. On the left side is is shown that hDQN is the only algorithm that learns the optimal policy. At the right side we see the average number of steps that each algorithm takes. hDQN converges to 6 and Double DQN to 2. We can appreciate that Vanilla DQN takes a lot of steps in order to complete an episode. This is because the learned Q values contain feedback loops.

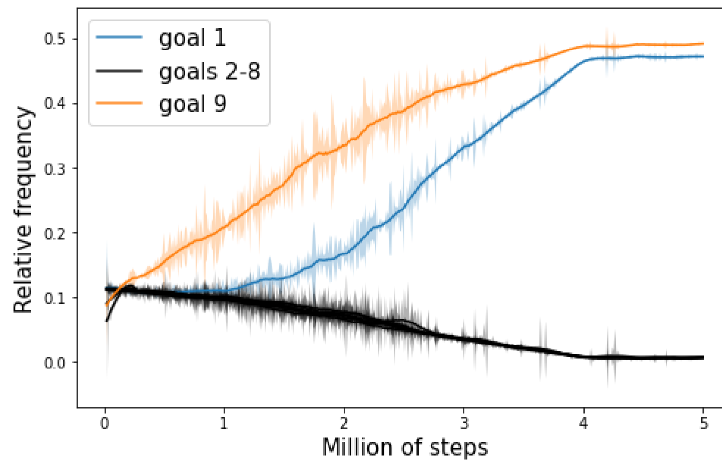

Figure 3.3: Goal relative frequencies along training in hDQN. We see that at the end only G_1 and G_9 are used by the meta controller, which resembles the optimal policy.

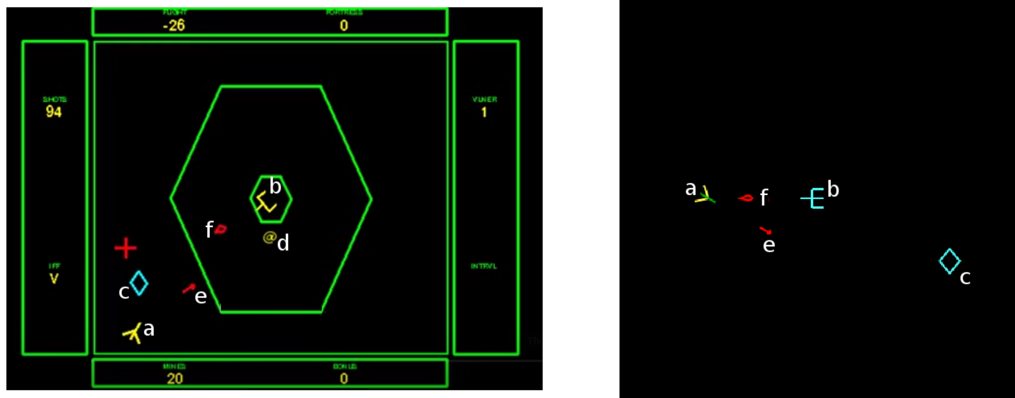

Figure 3.4: Left) a screen shot of the original Space Fortress and right) a screen shot of the version used in this project. The most important component are shared between them: a) the space ship, b) the fortress, c) a mine, e) a missile shot by the space ship and f) a shell shot by the fortress. The component d) is an instance of a bonus and it is not included in these experiments.

3.2 The Space Fortress environment

A screen shot of the original game is displayed at the left of figure 3.4 and a link to a video game-play of an experienced human can be found here[2] an another one of an amateur player here [3].

The original Space Fortress game was ported to PC in [Gopher et al., 1994] but further progress in hardware and software rendered it unusable. In [Shebilske et al., 2005] a revised version was implemented. In this project the Space Fortress code from this public repository[4] was used. It is based on a C++ port made in NLR from a MS-DOS version of Space Fortress. In order to perform the desired experiments the well known gym interface [Brockman et al., 2016] was used with the idea that the resulting environment can be used by everyone for testing any other learning algorithms. At the very end of this thesis project [Agarwal et al., 2018] released their SF environment also implementing the gym interface[5].

Rules Some adjustments to the code needed to be performed as part of this thesis in order to resemble the version of the game used by NLR in its experiments with personnel as good as possible. Although in the original version from 1989 a global score is computed taking into account several factors of the human performance such as navigational patterns, in the NLR human experiments – and hence this project– the focus is on learning the optimal strategy to destroy the fortress as many times as possible throughout a fixed amount of sessions. The specific rules of the NLR game follow what was introduced in section 1.1 but they miss some of the components of the 1989 original game. For instance, in the full version there were two types of mines (friend and foe) which determined the correct action to perform. In order to identify them the subject had to remember three characters

[2]Video of experienced human player on SF: `https://www.youtube.com/watch?v=FWuTP_JgZwo`
[3]Video of amateur human player on SF: `https://www.youtube.com/watch?v=tmRITTg70VE`
[4]SF code used in this project: `https://github.com/DerkBarten/SpaceFortress`
[5]Another SF gym: `https://github.com/agakshat/spacefortress`

that were shown at the beginning of each episode. NLR experimented with a version that had only one type of mines and another version without any mines at all. Part of this project consisted in setting up both versions although the one with mines was chosen for the RL experiments since it was more challenging.

The most important rules from the Space Fortress version used in this thesis were implemented to replicate the NLR version of the game and will be now specified. Some of the rules are actually time constraints whose values were chosen taking into account human limitations. The NLR game runs at 50Hz, which can be translated as the relation 50 milliseconds = 1 game step.

- The player's $\mathcal{A}$ is composed by 5 actions: (apply) *thrust*, (turn) *right*, (turn) *left*, *shoot* and *wait*.
- The space is *friction less*. This means that the spaceship can only modify its momentum if it applies thrust in a particular direction.
- There are no spatial limits at the borders of the screen. If the spaceship hits the left (down) side of the screen it will appear in right (up) side of the screen. This is referred to as *wrapping*.
- The fortress has 10 lives. When the fortress looses them all the player *wins* and the game restarts.
- The player –spaceship– has 3 lives. When it looses all of them the game restarts.
- The fortress is always aiming at the spaceship and shoots shells at it, taking one life away from it with each hit.
- Each episode begins each time the game restarts and finishes when either the fortress or the space ship have no lives left [6].
- Each episode begins with the space ship in the same position (same Y coordinate as the fortress, in the middle of the left side of the screen) aiming upwards and still.
- When a mine is present it always chase the space ship at constant speed. Mines can disappear if 1) they hit it the spaceship (space ship looses 1 life), 2) if the spaceship hits them with a missile or 3) 10 seconds pass without 1) or 2) happening. A new mine will be spawned then after 4 seconds at a random location.
- When the fortress has only 1 life (it is *vulnerable*) left it will only loose it if the spaceship *double shoots* it. This means at least two consecutive missile hits with a maximum time-span of 250 milliseconds (5 game steps) between them.
- When the fortress has between 1 and 10 lives and the space ship hits it with a missile it will loose 1 life as long as 1) the last fortress hit was more than 250 milliseconds (5 game steps) ago, 2) There is no mine present (or there is but it was spawned less than 2 seconds = 40 game steps ago).
- If the spaceship *double shoots* the fortress when this one has more than 1 life the fortress will recover its 10 lives.

[6] When training RL agents thousands of episodes were played. In $\approx 4\%$ of them a bug occurred causing the values returned by the game to be corrupted hence harming the training of the agents. Corrupted episodes kept running until the game was reloaded and a normal episode could begin. Fortunately reloading added no noticeable overhead so the solution was to reload the game after each episode. However, to ensure that corrupted episodes finished a maximum number of 3000 steps per episodes was set.

State space At a particular time step t, the RL agents trained on this environment act upon their observed state of the environment, s_t, which comes from the environment space $\mathcal{S}$. Despite the common approach in RL to learn video-games from image pixels in this case I decided to provide the agent with a richer set of features. This one-dimensional array was designed in such a way that it is possible to reconstruct all the meaningful elements of the game play given a single s_t so that it is enough to know which is the next optimal action (the game is *markovian*). Each cell of s_t can be understood as belonging to one of two groups of features. The first group contains basic perceptual information: spaceship's position, angle and velocity; position of the shells fired by the fortress and position of the mines. Note that providing the spaceship's angle as a single scalar is not optimal because it is a cyclic feature. To ensure that the difference between $s_{t|angle=359}$ and $s_{t|angle=1}$ is correct we encode it as two separate values in s_t: $\sin(angle)$ and $\cos(angle)$[7]. Because of the wrapping, the same applies to the spaceship's position. The second group of features is provided to make the game learnable without the need of adding extra complexity to the machine learning model architecture such as feeding a history of frames at each t or using recurrent neural networks. This second group is comprised of the number of time-steps since the last time the spaceship fired a missile, the number of time-steps since the last mine appeared and the fortress remaining lives. s_t was scaled to range between 0 and 1. In features with known maximum value the scaling is trivial. In features without a maximum the hyperbolic tangent was used. The most direct benefit when training on a $\mathcal{S}$ designed like this instead of pixel-based is that the computational costs are reduced dramatically. Abstracting away the perceptual component from the learning process has a price too, specially in the generality of the solution. In the discussion of section 4.2 more comments that are relevant to this decision are given.

Reward function The last piece of our environment is to specify the reward function $\mathcal{R}$ that the agent will try to maximize (equations 2.1 and 2.15). The end objective of this experiment is to be able to make human-machine comparisons so both players should follow the same goals. When humans learned this game, they were told to destroy the fortress as many times as possible while maintaining the spaceship maneuver under control (wrapping was penalized). For simplicity, we will assume that goal of the game then is defined as follows: *destroy the fortress as many times as possible in a fixed period of time without wrapping.* According to the reward design principle described in section 2.1 our reward function should express exactly *what* we want and nothing else. This could be formulated as

$$R^s(s_t) = \begin{cases} 10 & \text{if} \quad \text{fortress destroyed} \\ -1 & \text{if} \quad \text{wrapping} \\ -0.01 & \text{otherwise} \end{cases} \tag{3.2}$$

Where s stands for *sparse*[8]. A time penalty is added to encourage the agent destroying the fortress quickly. Note that this time penalty imposes a requirement of *what* we want the agent to learn and does not tell the agent *how* to destroy the fortress. The problem with R^s is that its sparsity on the positive side makes it very hard to learn from. An agent that operates with a low level action set

[7]The marginal effect on performance of this preprocessing technique was not thoroughly tested. Due to their vast amount of parameters, Neural Networks excel at memorizing entire datasets. They are possibly among those Machine Learning models that are less sensitive to this type of optimization. It would be more important when using simpler function approximators, which is included as future work.

[8]Named like this for simplicity. To be more precise it should be called *positively* sparse, because it is only the positive side of $\mathcal{R}$ which is sparse

according to an ϵ-greedy exploration strategy will have a hard time destroying the fortress for the first time. As a workaround to the problem I included a second reward function in the experiments,

$$R^d(s_t) = \begin{cases} 5 & \text{if} \quad \text{fortress destroyed} \\ 1 & \text{if} \quad \text{hit fortress } normally \\ -5 & \text{if} \quad \text{hit fortress } too\ fast \\ -1 & \text{if} \quad \text{hit by fortress} \\ -1 & \text{if} \quad \text{hit by mine} \\ -1 & \text{if} \quad \text{wrapping} \\ -0.01 & \text{otherwise} \end{cases} \tag{3.3}$$

Where d stands for *dense*. *normally* means one effective fortress hit when it is not vulnerable yet. Two or more shots within less than 250ms when the fortress is not vulnerable is considered *too fast* and then the fortress restores all its lives. As it is seen in 3.3, R^d contains many elements that are easy to *experience* by the agent in its initial steps and that will guide it towards the goal of destroying the fortress.

Temporal level of abstraction The purpose of this experiment is to investigate what is the optimal level of temporal abstraction when operating on the SF environment. We therefore define three different sets of goals $\mathcal{G}$ that will be used to identify the type of agent that uses them:

- **Low level of abstraction**, $\mathcal{G}_0$. Here, there is a one to one mapping between goals and actions so that $\mathcal{G}_0 = \mathcal{A}$. In this trivial case the h-DQN would resemble the DQN so the DQN archtiecture will be used instead for simplicity and efficiency.
- **Medium level of abstraction**, $\mathcal{G}_1$. Here the h-DQN agent will be used with 4 different goals. $\mathcal{G}_1 = \{Aim\ at\ fortress,\ Aim\ at\ mine,\ Single\ shoot,\ Double\ shoot\}$. It is worth to mention that the critic (recall from figure 2.6 that the critic is the part of agent which judges if a goal has been achieved or not) will understand that the goal *Single shoot* is accomplished only if the agent shoots and then stays without shooting for 5 game steps. In the same fashion, *Double shoots* requires the agent to shoot while the last shot happened less than 5 steps ago. Note also that in neither case the critic takes into account if the missiles shot by the agent hit something or not.
- **High level of abstraction**, $\mathcal{G}_2$. Here the h-DQN agent will be used with only 2 different goals. $\mathcal{G}_2 = \{Hit\ fortress\ once,\ Hit\ fortress\ twice\}$. The accomplishment of these goals is verified similarly as in *Single shoot* and *Double shoot* from $\mathcal{G}_1$ but in this case instead of checking if at a particular time-step the agent has shot or not the critic will check whether the fortress has been hit. It will also look at the number of steps since the fortress was hit.

Having $\mathcal{G}_1$ and $\mathcal{G}_2$ defined like this makes it possible for the agent to learn the game (example[9]). However, it was found that adding the motion low level actions $\{Thrust,\ Right,\ Left,\ Wait\}$ as *goals* to $\mathcal{G}_1$ and $\mathcal{G}_2$ made the agent's moves more fluent and flexible and are used for all the experiments reported here. Of course, these new goals are trivial to accomplish. For example, when the MC sets *Thrust* as the current goal, all that C has to do is to choose the action *Thrust*.

[9] https://www.youtube.com/watch?v=lpBQyqopmWQ

Other hyperparameters The tested learning models contain many hyperparameters. Some of them were fine tuned via grid search. The default values can be found in `configuration.py`[10].

3.3 Intrinsic motivation

It would not make sense to start training the meta-controller if the controller does not know how to achieve $\mathcal{G}$. That is why, as explained in section 2.4.3, the meta controller chooses random goals without updating $Q_{\theta_{MC}}$ until the controller performs good enough, which we formalized as $sr_g > threshold \; \forall g \in \mathcal{G}$, sr_g being the success rate of g. In the experiments *threshold* was set to 0.95 and such condition was usually sufficed quite early in the training. This is good but one has to take into account that the only situation where we can say that the controller has failed at achieving g is when the episode terminates. In other words, even if g is simple, it may happen that the controller 'succeeds' at achieving it only after a big amount of steps, which means that the controller still has a lot to learn. In SF we want to destroy the fortress as many time as possible, so we would like out agent to learn how to achieve every $g \in \mathcal{G}$ quick. In this experiment I investigated if adding a time penalty to $I_t(g)$ would help with that. Recall from equation 2.16 that the controller tries to maximize the total intrinsic reward which depends on the goal that is currently being followed. In the original h-DQN described in [Kulkarni et al., 2016] the intrinsic motivation mechanism provided by the critic implements the following internal reward function:

$$I_t(g) = \begin{cases} 1 & \text{if} \quad g \text{ is achieved at } t \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

And now we define another version of it in which a time penalty term is added so that

$$I'_t(g) = \begin{cases} 1 & \text{if} \quad g \text{ is achieved at } t \\ -0.01 & \text{otherwise} \end{cases} \tag{3.5}$$

We compare $I_t(g)$ versus $I'_t(g)$ by running various h-DQN agents equipped with either $\mathcal{G}_1$ or $\mathcal{G}_2$ on the SF environment and kept track of the average number of steps required to carry out each goal. The result of the experiment is shown in figure 3.5. We clearly see that $I'_t(g)$ helps the controller achieving the goals faster. When $g \in \mathcal{G}_1$ (upper row) the time penalty makes the controller learn to achieve the goals optimally quicker, but we see that $I_t(g)$ and $I'_t(g)$ actually converge at some point. On the other hand, the biggest impact occurs when $g \in \mathcal{G}_2$ (bottom row), because those goals are of a higher level of abstraction and the controller takes longer to achieve them. In this case, the time penalty not only makes the controller learn to achieve the goals optimally quicker, but we see that $I_t(g)$ and $I'_t(g)$ converge much slower. The design of $I_t(g)$ is theoretically correct because it follows the principle of using the reward function to express *what* we want – achieve g–, and both $I_t(g)$ and $I'_t(g)$ make Q_{θ_C} converge to the optimal $Q^*_{\theta_C}$, but $I'_t(g)$ does so faster, specially when g has a high level of abstraction.

[10] `https://github.com/hipoglucido/Hierarchical-DRL-Tensorflow/blob/master/src/configuration.py`

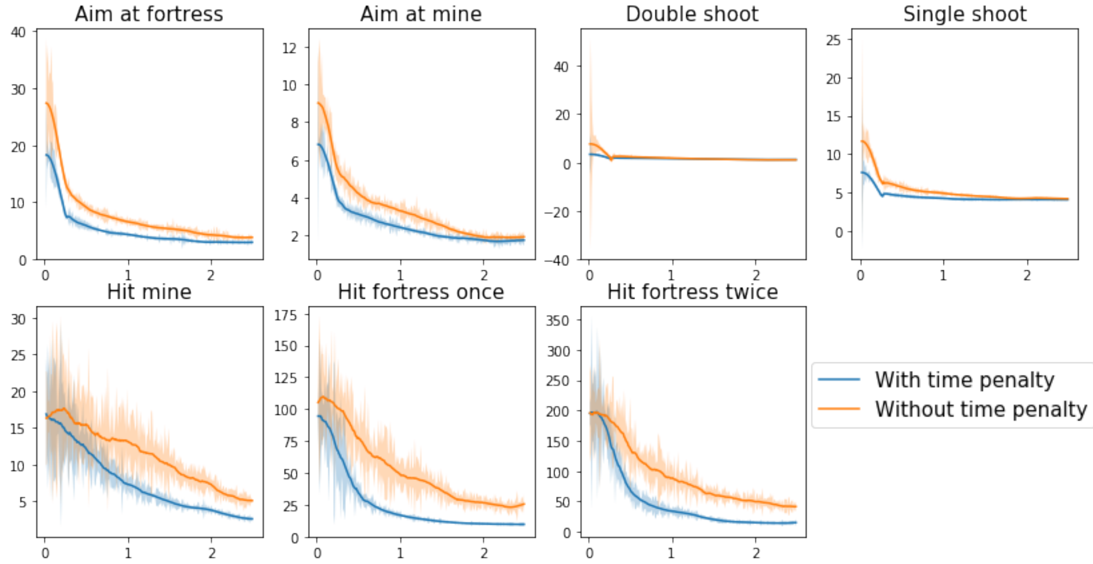

Figure 3.5: For each goal we have the evolution of the average number of steps (Y axes) needed to achieve it as the training goes (X axes, in millions of steps). Averaged over three runs

3.4 Optimal level of abstraction

In this section the main experiment is explained. The idea is to see if having options as actions in a Hierarchical RL setting adds any value, for the different levels of abstraction already defined. First this is done when the reward signal is dense (easy case) and then when the reward signal is sparse (hard).

3.4.1 Dense rewards

We first compare the performance of our three agents (with $\mathcal{G}_0$, $\mathcal{G}_1$ and $\mathcal{G}_2$) when learning how to solve SF by trying to maximize $\mathcal{R}^d$. Here is a video[11] of the three agents after 25 million steps of training (around 100 thousand episodes). The vanilla version of DQN is used as the core of the algorithms, so no Dueling, Prioritized Replay nor Double Q learning is used. Performance is first measured as the total extrinsic reward accumulated per epoch[12]. In order to verify that the RL task of maximizing rewards aligns with the objective of destroying the fortress as many times as possible during a fixed time interval, the number of wins per epoch is also recorded. For both metrics (displayed in figure 3.6) the agent with medium level of abstraction ($\mathcal{G}_1$) outperforms DQN ($\mathcal{G}_0$), which outperforms hDQN with high level of abstraction ($\mathcal{G}_2$). hDQN's first win occurs at the very beginning in both $\mathcal{G}_1$ and $\mathcal{G}_2$ while DQN's does not happen until 10 million of steps, so it

[11] Dense experiment video: `https://www.youtube.com/watch?v=OXT7jyvMJ2g`

[12] In all the SF experiments one epoch consists of 10000 steps, regardless of how many games/episodes/rounds are included in such interval

clearly seems that having goals helps the agent to learn the task earlier in time[13].

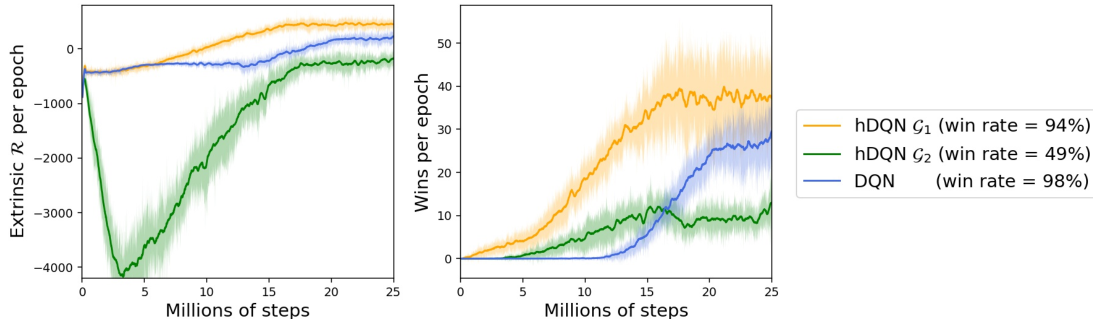

Figure 3.6: (Left) Total extrinsic reward accumulated per epoch and (right) total amount of fortress destroyed per epoch. In both cases the three agents are tested in SF with a dense reward signal $\mathcal{R}^d$ and the average win rate is measured by taking the last 50 epochs into account.

However, the precise impact of the level of abstraction of agent's actions on the final performance is not so straightforward to interpret. We first compare DQN and hDQN $\mathcal{G}_1$. They both manage to solve the game effectively, winning 98% and 94% of the times. Despite having a slightly lower winning rate, hDQN $\mathcal{G}_1$ destroys more fortresses over time. The reason is that when the fortress is vulnerable DQN sometimes hesitates (specially when there is a mine on the screen), whereas the *meta controller* of the hDQN $\mathcal{G}_1$ immediately switches to *Double shoot* or *Aim at mine*, guiding the low level actions of the agent to specific behaviours in a more direct manner.

By ways of contrast, for all the (five) seeds in this experiment it seems that having $\mathcal{G}_2$ instead of $\mathcal{G}_0$ makes things worse. Thanks to the big amount of indicators that were recorded during training (see `src/metrics.py`) we can analyze why this happens a posteriori. The policy found by hDQN $\mathcal{G}_2$ at the end of the training does not make use of the goal *Hit fortress twice*. We can see how this anomaly manifests by keeping track of the relative frequencies of the goals set by the *meta controller* $\forall g \in \mathcal{G}_2$, which are shown in figure 3.7. Interestingly enough, the frequency of use of *Hit fortress twice* decreases at the same pace as the probability with which the *meta controller* sets a goal randomly, ϵ_{MC} (displayed in figure 3.8).

[13]Note that in Q-learning even if the agent has already learned something its behaviour will not resemble it fully until ϵ – which decays deterministically– reaches a low value. In this implementation ϵ reaches its lowest value 0.05 when training is 65% complete.

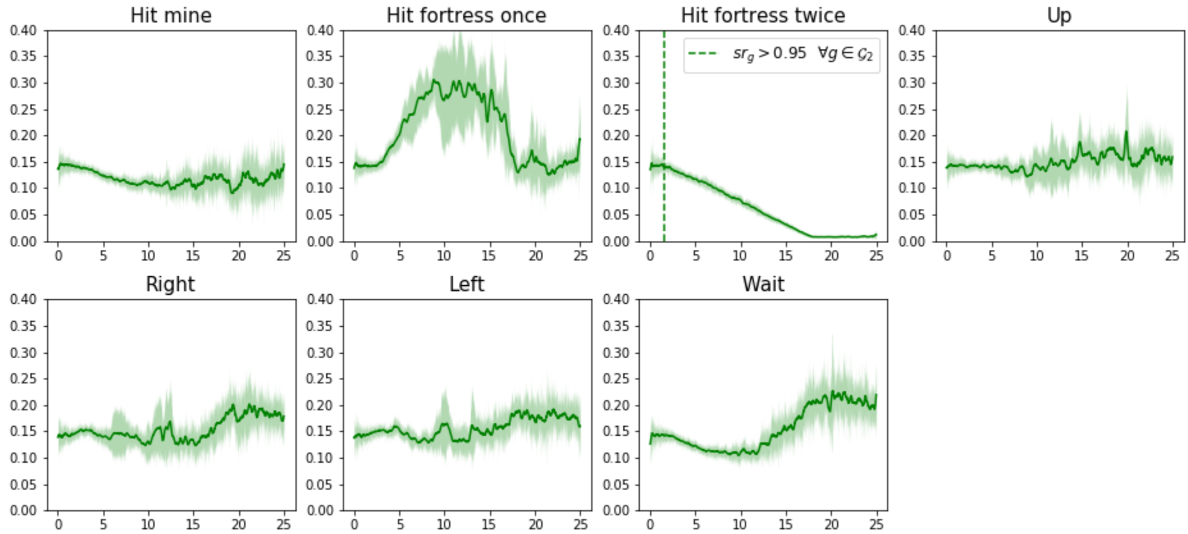

Figure 3.7: Relative frequencies per epoch of each goal that belongs to $\mathcal{G}_2$ during training on SF with $\mathcal{R}^d$ (Y-axis is the normalized frequency $\in (0,\ 1)$ and X-axis is million of steps). Each point is calculated as the division of the number of times that the MC set the goal in one epoch by the total amount of goals set by the MC in that epoch. The summation of all the frequencies must add up to one for all the epochs.

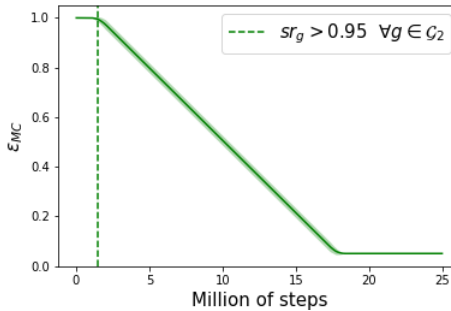

Figure 3.8: Annealing of ϵ_{MC} during the training of hDQN $\mathcal{G}_2$ on SF with $\mathcal{R}^d$. Following the exploration strategy explained in section 2.4.3, $\epsilon_{MC} = 1$ until the *controller* learns how to achieve all the goals, which in this case happens for the first time after 2 million steps of training (represented with a dashed line).

Specially at the beginning, it happens quite often that the MC sets and achieves *Hit fortress twice* without the fortress being vulnerable yet, resulting in a negative reward of -5 (recall $\mathcal{R}^d$

from equation 3.3) that is stored in the replay memory D_{MC}. After ∼2 million steps C learns to accomplish all the goals and the agent starts to draw experiences from D_{MC} to update θ_{MC}, rapidly establishing a negative value to $Q_{\theta_{MC}}(s, g = Hit\ fortress\ twice)$ for almost any $s \in \mathcal{S}$ that is plausible to be observed by the agent. Therefore, whenever the agent acts greedily –which happens more often as ϵ_{MC} decays– it will choose another goal but that one. This explain the decrease in the frequency of use. On the other hand, if the exploration leads the agent to choose *Hit fortress twice* by chance the fortress won't be vulnerable in most of those cases so the negative connection will be reinforced; and if it happens to be then the agent will get a +5 reward but this will not compensate for the past bad experiences and *Hit fortress twice* will remain as a bad choice. Another way of visualizing this effect is by monitoring the amount of penalties that the agent incurs due to fast-hitting the fortress when it is not vulnerable yet (right plot in figure 3.9). Despite *Hit fortress twice* being almost dropped from $\mathcal{G}_2$, the agent will still try to maximize G_{MC} from equation 2.15 and the MC will find that when the fortress is vulnerable the only goal it can use to destroy it is *Hit mine*. To understand how this is possible it is required to take into account how this hDQN implementation addresses the question: what happens when the MC sets *Hit mine* and there is no mine on the screen? The answer given is to set it as accomplished no matter what action the *controller* chooses afterwards. hDQN with $\mathcal{G}_2$ leverages this in order to be able to shoot fast and destroy the fortress without using *Hit fortress twice*. Of course, this makes the optimization of $Q_{\theta_{MC}}$ complicated. There are several ways to fix this issue: one option would be to include the action *Shoot* as a goal in the same way we did with the motion-related actions. Another alternative would be to rethink the implementation, for example by disabling *Hit mine* from $\mathcal{G}_2$ if there is no mine on the screen. Finally, we could redesign $\mathcal{R}^d$ diminishing or removing the penalization for hitting the fortress too fast when it is not vulnerable. It is maybe surprising that the resulting sub-optimal policy actually works and the agent manages to destroy the fortress 49% of the times. In any case, it is important to realize that the problem with $\mathcal{G}_2$ and $\mathcal{R}^d$ is a clear example of the negatives consequences of including information related with *how* to solve the problem in $\mathcal{R}$.

From this experiment we conclude that having goals as actions gives and advantage to hDQN over DQN but we are still unable to determine which is the optimal level of abstraction ($\mathcal{G}_1$ or $\mathcal{G}_2$) because by trying to make the problem easier when introducing information relative to *how* to solve the problem in $\mathcal{R}$ we made the learning difficult for one of the hDQN agents.

3.4.2 Sparse rewards

The goal of this experiment is to find the optimal level of abstraction of $\mathcal{G}$ on SF. To do it $\mathcal{G}_0$, $\mathcal{G}_1$ and $\mathcal{G}_2$ will be tested again on SF but this time we will be using a reward signal that only tells the agent *what* to do. This is the case of $\mathcal{R}^s$ (defined in 3.2). As in section 3.4.1, the extrinsic reward and wins accumulated over training are plotted (figure 3.10). In this experiment five seeds are also used per agent configuration. Here is a video[14] of the three agents (the best seed is selected in each case). First thing to notice is that DQN fails in all of them. The only thing it learns is to avoid wrapping and the resulting policy consists in executing the action *Wait* at mostly every step, thus always getting -0.01 from $\mathcal{R}^s$ ($-0.01 \times 10000 \frac{steps}{epoch} = -100$ reward accumulated per epoch as left hand side of figure 3.10 shows) and never destroying the fortress.

[14] Sparse experiment video `https://www.youtube.com/watch?v=ZJGslxgm2Uw`

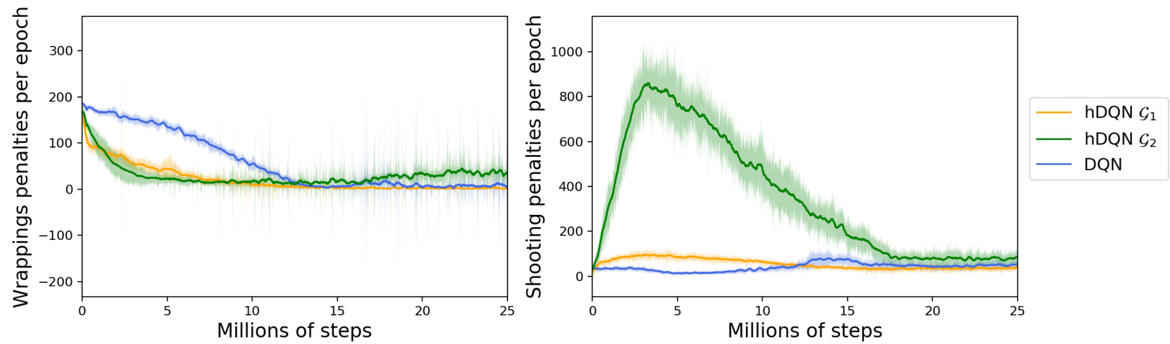

Figure 3.9: (Left) times that the agent is penalized for wrapping around the screen and (right) times that the agent hits the fortress too fast (two hits between less than 250 milliseconds) making the fortress recover its 10 lives.

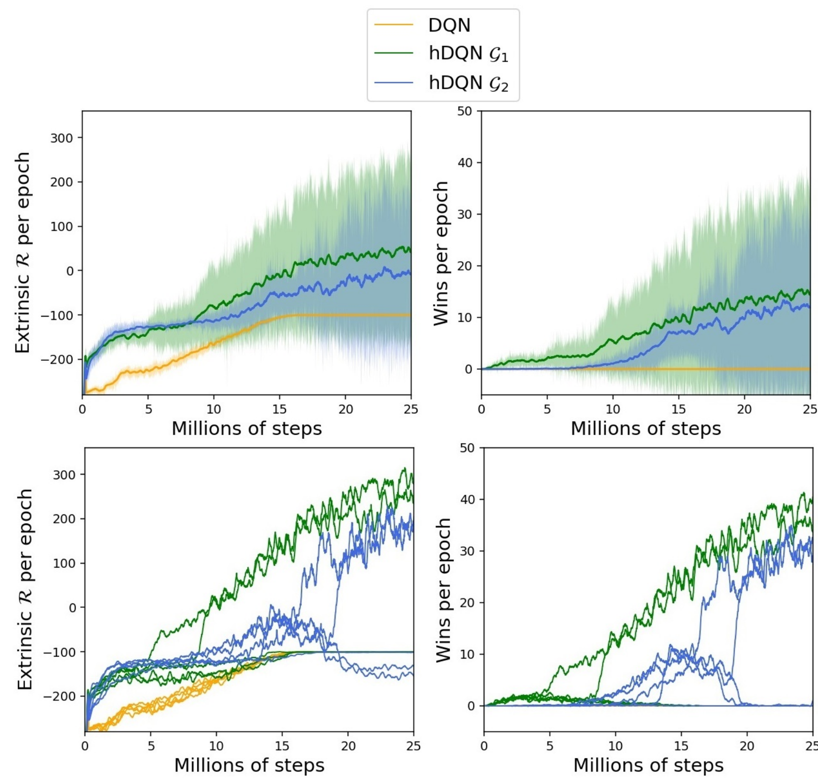

Figure 3.10: (Left) Total extrinsic reward accumulated per epoch and (right) total amount of fortress destroyed per epoch. (Upper) means and standard deviations among the 5 seeds are plotted and (bottom) each run is plotted separately. In all the cases the three agents are tested in SF with a sparse reward signal $\mathcal{R}^s$ and the average win rate is measured by taking the last 50 epochs into account.

As the upper part of figure 3.10 shows, there is a huge variation across runs within the same configuration. Much more than it was when learning from a dense reward. For this reason, each of the five runs for each configuration is plotted separately at the bottom of the figure. In both hDQN cases two runs were successful and three failed. In an attempt to gain insight about how this instability manifests through training, several scalar indicators were monitored and compared among successful and unsuccessful rollouts. These can be found in appendix 4. It is hard to draw useful conclusions from that data yet it is an interesting exercise to hypothesize about how is it that some runs failed and others did not. For example, by looking at the five runs of the hDQN $\mathcal{G}_2$ agent (figure 3.11) we see that two of the failed runs actually managed to destroy the fortress several times in the middle of training but then they just stopped doing it. It is hard to say why (since the accumulated $\mathcal{R}$ also decreased, as shown in 1) but it seems that at some point in training they started decreasing the use of the goal *Hit fortress once* and started overusing *Hit fortress twice*, incurring in shooting penalties (that had no negative effect in terms of rewards under $\mathcal{R}^s$).

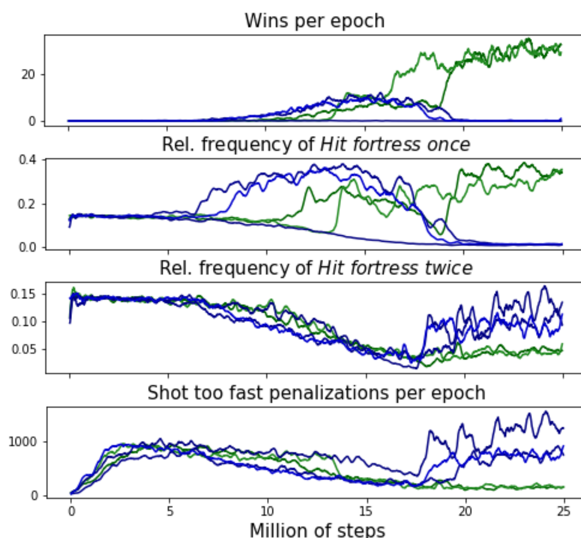

Figure 3.11: Scalar indicators monitored during the training of five hDQN $\mathcal{G}_2$ agents with different seeds each. Successful runs are colored with green tones and the unsuccessful ones with blue tones.

The agents do not circumnavigate the fortress as they did with $\mathcal{R}^d$. As [Frederiksen and White, 1989] found, that is the optimal strategy to elude fortress's shells. In what follows I explain one hypothesis about why this happens. $\mathcal{R}^s$ does not provide a negative feedback when the spaceship is hit by the fortress, so the only incentive that the agent has to dodge the shells is that at the third hit the game will restart and the fortress lives will reset to 10, so it will take extra steps to get the next positive reward. Figure 3.12 compares several SF indicators while the hDQN $\mathcal{G}_1$ learns from either $\mathcal{R}^d$ or $\mathcal{R}^s$ for 25 million steps. If we look at the amount of wins per epoch we can see that the agent that learns from $\mathcal{R}^s$ was still improving at the end of the training. This supports the idea that in the long run the agent should be able to fully optimize the MDP and learn to avoid the fortress's shells. However, this does not necessarily mean that it would outperform the learning from $\mathcal{R}^d$.

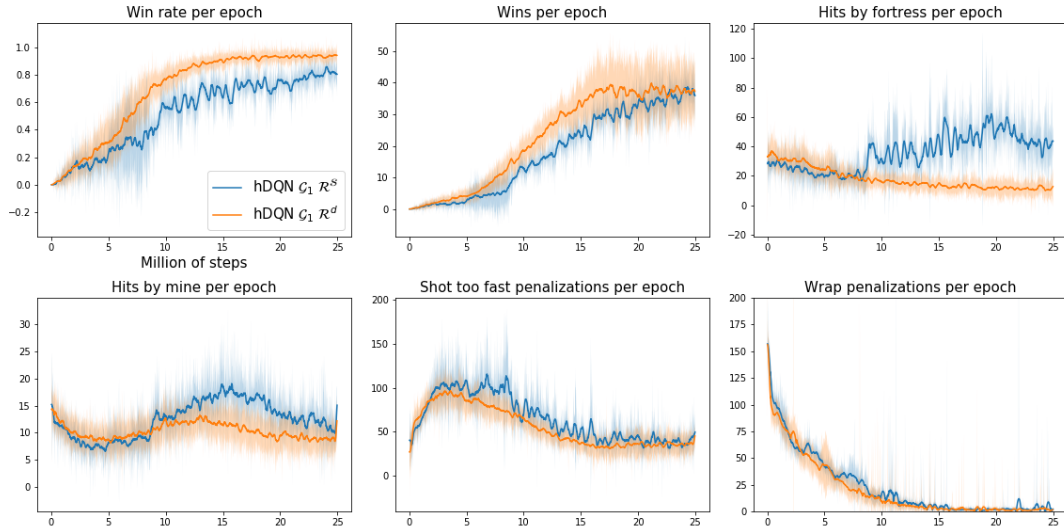

Figure 3.12: Several SF indicators during training coming from the case in which the hDQN $\mathcal{G}_1$ agent learns from $\mathcal{R}^d$ and when it learns from $\mathcal{R}^s$. Five different seeds were used in the former case and *only two* in the latter because the rest failed.

One possible thing to do in order to accelerate the MDP optimization would be to include in $\mathcal{S}$ a feature related with the remaining lives of the spaceship. I claim that it should work in $\mathcal{R}^s$ because first) the agent would be able to understand how $\mathcal{S}$ changes if the spaceship is hit by a shell or mine and second) with the only goal of maximizing rewards the agent will try to reach those states closer to destroying the fortress, which usually are those where the spaceship has many lives. From an MDP perspective, this extra feature will discretize $\mathcal{S}$ in such way that the connection between the states of having less lives and having less reward over time is more easily recognizable for the agent.

The main finding of this experiment is that in situations with reward sparsity having options as actions in a hierarchical architecture can make the difference from a solution that fails completely to solving the problem achieving a good performance. Another phenomenon was observed, that sparse rewards – besides being harder to learn from – make the RL algorithms behave with more instability than under dense rewards. Also, some odd behaviours were found in the resulting policies of the hierarchical agents, which were analyzed and an argumented solution to the anomalies produced was proposed. Regarding what level of abstraction (between $\mathcal{G}_1$ and $\mathcal{G}_2$) is better in Space Fortress, it seems that $\mathcal{G}_1$ achieves better results. However, a solid conclusion about $\mathcal{G}_1$ versus $\mathcal{G}_2$ can not be drawn because of two reasons. First, because the variance between runs with distinct seeds is too high. And secondly because at the end of the training the slope of the learning curve of hDQN $\mathcal{G}_2$ was bigger than that one of hDQN $\mathcal{G}_1$. For more precise results I suggest to include the spaceship lives in $\mathcal{S}$ and to run the agents with more seeds and for longer time.

3.5 Ablation study on hDQN extensions

Failed attempts at solving SF with RL previous to this thesis made me decide to incorporate extensions to the core algorithm - DQN - to make the agents more capable of solving the task. According to the literature, Double Q learning, Dueling architecture and Prioritized Experience Replay were extensions to DQN that improved performance across several environments. Therefore I decided to spend a considerable amount of effort in adapting those extensions to the DQN and hDQN architectures[15]. My implementation takes the incorporation of each extension to either the *meta-controller* or the *controller* as one binary hyperparameter more so that it was easy to experiment with them. Actually, I made the code highly modular so that hDQN and DQN execute the same lines of code but in different ways.

First I found that the vanilla DQN agent was already able to solve the game with $\mathcal{R}^d$. Then, as I continued doing experiments with both DQN and hDQN the results I was getting were always telling me that the extensions did not make a noticeable impact on the performance in any case, but training times were considerably higher. In Double Q learning, the decoupling of selection and evaluation when computing the target (equation 2.10) requires one extra forward pass. The model proposed in the Dueling Architecture adds up on network complexity requiring more calculations for each forward pass. Finally, even though the data structure used in the Prioritized Experience Replay scheme allows for an efficient implementation of the idea of prioritizing the most useful experiences it is still expensive to keep the priorities of the experiences in memory updated, specially in comparison with uniform sampling. All these computational drawbacks were exacerbated in the *Prioritized Dueling Double Q* hDQN agent (PDD-hDQN), where each extension was added to the *controller* and *meta-controller*, taking around 40% more time to train than vanilla hDQN. As a result of this I dropped the DQN extensions and focus on hDQN versus DQN comparisons having vanilla DQN at the core.

Nonetheless, I tried to illustrate the effect of the extensions with two ablation studies (shown in figure 3.13). At the left, the total amount of *intrinsic* reward per epoch accumulated by the *controller* is measured in five different configurations: The Vanilla agent (hDQN without any extension whatsoever), the PDD-hDQN agent (fully extended hDQN) and the PDD-hDQN with each of the extensions ablated from the *controller* only (e.g. 'Dueling $\sim\in\ \mathcal{C}$' means that the Dueling extension of the *controller* was ablated). At the right, the same idea but the Y-axis being *extrinsic* reward and the ablations being done on the *meta-controller*. $\mathcal{G}_1$ and $\mathcal{R}^s$ were used. We knew already about the variance of hDQN in SF with R^s so it is difficult to draw conclusions from the right plot [16]. For the left experiment, the results are all about how good is the *controller* learning $\mathcal{G}$ because here the reward plotted in the Y-axis is provided intrinsically by the agent independently of the extrinsic reward, $\mathcal{R}^s$. In this case the averages look more alike, suggesting that learning medium level of abstraction goals ($\mathcal{G}_1$) can be easy enough so that adding extensions to the Vanilla *controller* is not worth the computational overhead. Indeed, the task of the *controller* is usually easy compared with that one of the *meta-controller*.

[15] I adapted the code from this repository: `https://github.com/cmusjtuliuyuan/RainBow`

[16] If we only looked at the average between runs we could conclude that the extension of Double Q learning in the *meta-controller* harms performance but this would be naive because there is too much variance.

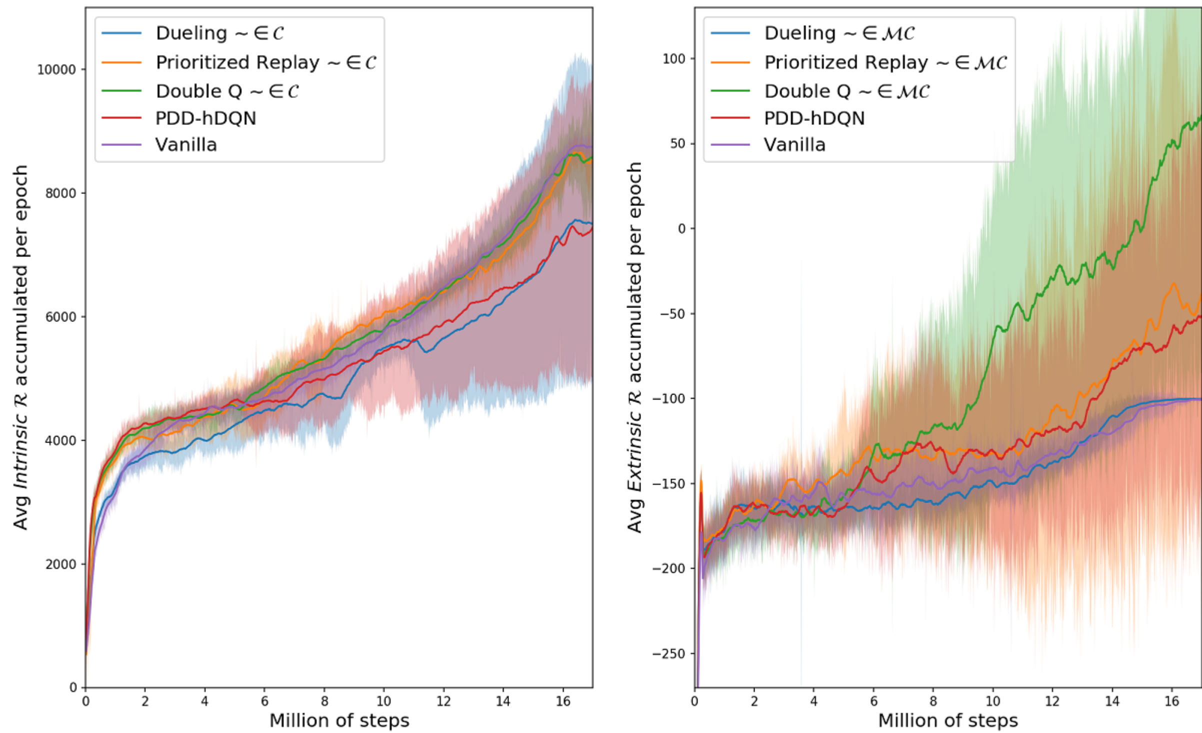

Figure 3.13: Left) Ablation study of DQN extensions on *controller* in terms of intrinsic reward and right) ablation study of DQN extensions in the *meta-controller* in terms of extrinsic reward. PDD-hDQN is the fully extended agent and 'Dueling $\sim\in\ \mathcal{C}$' means that the Dueling extension of the *controller* was ablated.

3.6 Human vs Machine superficial comparison

One point of interest since the beginning of the project was that if an ML model can be constructed to learn SF, how does it compare to human performance. This experiment is meant to make a visual comparison between the learning curves of the human and the machine, to give an intuition of the results that could be expected if after this project a thorough comparison is carried out, or at least to tell us if it would be worth to conduct it. [J.J.M. Roessingh and Kappers, 2003] is the report of an experiment conducted at NLR where human learning curves of trainees playing SF were recorded. The authors proposed the progressive average function as an alternative to regular learning curve models in order to better describe the experimental data. The resulting model was able to predict stability of performance with practice, long-lasting individual differences and specificity of skill-transfer. The upper left plot in figure 3.14 was taken from that report and measures how does the time required to destroy the fortress changes throughout the 16 hours of training. Next to it are the learning curves of the $\mathcal{G}_0$, $\mathcal{G}_1$ and $\mathcal{G}_2$ agents after 25 million of steps (347 hours) of training on SF with $\mathcal{R}^d$, plotted in a similar fashion. The only conclusion of this section that among all of them, hDQN $\mathcal{G}_1$'s curve is the most similar to that of the human[17] reaching the same performance in term of fortress destructions per time.

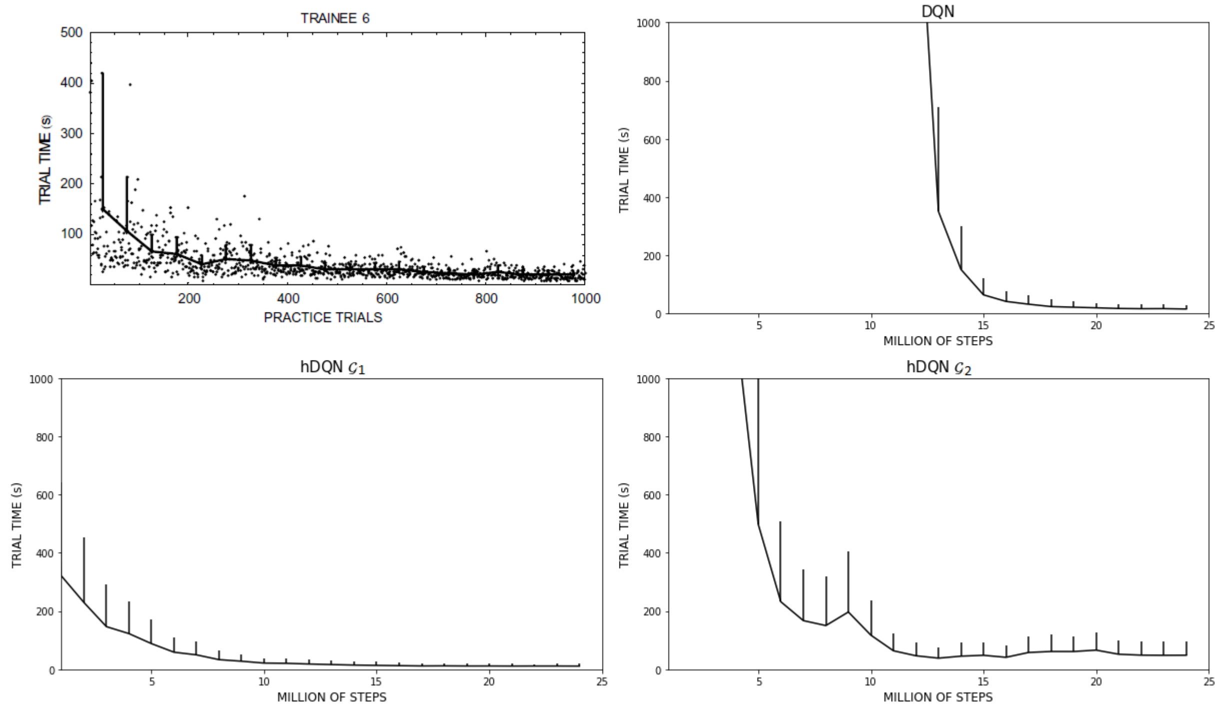

Figure 3.14: Human trainee learning curve on SF plotted along that one of the agents DQN hDQN $\mathcal{G}_1$ and $\mathcal{G}_2$. Time required to destroy the fortress is measured and its variance through training.

[17] Making solid claims about how is human learning similar to machine learning is beyond the scope of this project

Chapter 4

Closing

In this last chapter the research questions asked in section 1.3 will be answered. After that a conclusion with the most relevant findings of the project is given and finally a section with some interesting points and future lines work will be discussed.

4.1 Conclusions

Question I *Is it possible to learn SF with Reinforcement Learning at all?* Yes, SF is learnable with a simple implementation of DQN and it is possible to achieve a good performance at it. It is important to note that the representation of the environment - which in this project consists of a high level feature 1D vector instead of pixels - and the reward function used are likely to play a role in the difficulty of the task. For instance, it is possible to learn to play Space Fortress from a sparse reward function with Hierarchical Reinforcement Learning but not with simple DQN.

Question II *Under which circumstances is it recommendable to extend RL with Hierarchical RL? Is Space Fortress the case*? In general, I would recommend to use Hierarchical RL in those scenarios where we want to make our agents more effective at exploring their environment, which can be tremendously useful when the reward signal is sparse. In the type of Hierarchical RL researched in this thesis the notion of goals arises. Different ways of how these goals get inside the agent are still under research in the RL community. In the method used here – hDQN – the goals are given to the agent as predefined set. If the RL practitioner has access to the environment's entities and dynamics this allows for a richer definition of goals which allows to turn it into an injection of expert knowledge. This boosts performance but it makes the solution ad hoc to the problem. Therefore, it may be an issue in scenarios were the model needs to be directly applicable to other problems. If generality is important a more simplistic design of the goals can be utilized (as the authors of hDQN did in [Kulkarni et al., 2016], defining every goal as *entity A reaches entity B*).

Question III *Do Double Q learning, Dueling architecture or Prioritized Replay Memory improve the Hierarchical Reinforcement Learning algorithm?* After two ablation experiments, no solid evidence was found for the fact that these extensions add any value to hDQN when training on SF. It should be noted that the experiments from which this conclusion was drawn consisted in training

with a sparse reward which originated a big amount of instability on the performance. This was an issue specially when doing the ablation study focused on the agent's module in charge of choosing the goals that should be followed at each time (*meta controller*). In the case of the module that accomplishes those goals by choosing the right low level actions (*controller*) the results were more stable because that part of the agent is agnostic to the sparsity of the extrinsic reward signal and the goals were relatively easy to achieve. In that case there was small difference in performance across ablations. This suggests that *controller* can be successfully implemented with a simpler RL algorithm than the *meta controller*.

Final conclusion This master thesis presents the first Reinforcement Learning agent to learn to play Space Fortress at human level. This means being able to use a sparse reward signal to learn abrupt context-dependent shifts in strategy and temporal sensitivity. This was done thanks to *hDQN*, an implementation of Hierarchical Reinforcement Learning. With a toy example it was shown that the higher level module of the *hDQN* agent can be utilized to act as a long term memory and solve problems where the baseline DQN fails. An environment to learn Space Fortress using high level features was implemented and used in this project. No relevant improvements were found by extending the *hDQN* agent with *Double Q learning*, *Dueling architecture* or *Prioritized Replay Memory* so they were discarded for the rest of the experiments. It was found that a shaped dense reward for learning Space Fortress with DQN can be engineered but it was shown that it could also be a source of problems. So the outcome is to try to avoid designing reward functions that have information about *how* to solve the problem as much as possible and restrict ourselves to only include *what* we want. This turns the reward function of Space Fortress to be sparse, and the problem becomes way more hard because there is a lack of feedback in the exploratory phase. It was shown that in this case having temporally extended actions is of great help, making the agent able to successfully solve the game and reach human performance.

4.2 Discussion and future work

Learning from pixels While in this thesis the state space is formed by high level features – lets call it $\mathcal{S}_H$ –, in all the previous experiments of learning SF with RL conducted at NLR the state space consisted in image pixels, $\mathcal{S}_P$. The reason why $\mathcal{S}_P$ was not used in this project was not to avoid a drop in *effectiveness* but in *efficiency*, due to the fact that using $\mathcal{S}_P$ would have required much more computational power. Moreover, I argue that having our agent learn how to recognize edges, positions, missiles etc. from pixels is not relevant enough to this research in order to justify the extra computational costs, let alone the complication for the RL task. Lets take the case of the simplified version of SF in which there is no time sensitivity constrains, it is all about hitting the fortress and avoid being shot. In that case, as for what the results of this thesis are, both $\mathcal{S}_H$ and $\mathcal{S}_P$ contain the required elements to learn the game but the other projects that used $\mathcal{S}_P$ did not succeed. A very interesting research project would be to quantify the difference in *efficacy* of the solution when using $\mathcal{S}_H$ versus $\mathcal{S}_P$ as observation from the environment.

Learning the time-sensitivity strategy with more generality As explained in 3.2, among the features included in s_t there are the number of steps since the last mine appeared and the number of steps since the last spaceship shot. Some may argue that including these temporal features is a workaround to the time sensitivity learning problem. Without these features, a possible

design choice for the agent to be aware about the pace of its shooting could be to use a recurrent architecture, as it was done in [Agarwal and Sycara, 2018]. In my opinion, this added a complexity to the agent that was not worth the computation overhead and extra model parameters. Actually, I think that feeding the number of steps since the last shot is valid because it is something that depends entirely on the agent, so there is not any loss of generality in the solution. On the other hand, the number of steps since the last mine appeared is a feature that depends exclusively on $\mathcal{S}$, which makes the approach less general. However, this feature turned out to be of little use and it could have been removed. Its only purpose was to make the agent realize that it is actually possible to hit the fortress in the first 2 seconds after a mine is spawned, but without it it would have learned to just shoot it down whenever it is spawned. In a similar way, knowing the remaining lives of the fortress makes it possible for the agent to understand when does it have to reverse the firing strategy. In theory, it should be possible to learn this behaviour without this feature by using a recurrent architecture also, but this time it would be way more challenging because of the dependence over previous actions would be very long, having the agent to learn something similar as to how many times has it hit the fortress. [Agarwal and Sycara, 2018] claims that this fact renders the game non-Markovian, justifying its decision of using a recurrent architecture. I believe that this is not true because the vulnerability of the fortress is displayed on the screen, and the agent of that paper learns from pixels. Therefore, I think that reason why the game is non-Markovian is not this but only the fact that the optimal decision at a given time depends on how fast the agent is shooting and it has no direct to such information access in s_t [1]. Therefore, with the chosen representation of s_t the game came be effectively converted into a Markov Decision Process and a solution without a recurrent architecture is possible. A possible next step could be to drop the steps since last shot feature and include a recurrent architecture. This extra module would not need to be complex because it should only be able to handle short dependencies – the shooting pace –. Nonetheless, dropping the feature of fortress vulnerability (or hiding it from the screen if we were learning from pixels) would require more research effort and in my opinion would be a bit far fetched because I consider it a fair assumption that the agent or player always knows how many lives the fortress has.

Explainable AI Many industries can not leverage the potential that AI has to offer yet because often AI uses a black box model under the hood and the decisions that need to be automated are too important for taking an output as such, regardless of how accurate the model is [Gunning, 2017] [Samek et al., 2017]. While recent advances in interpreting the decisions of supervised learning black box models has been getting very interesting results [Ribeiro et al., 2016], there has not been that much research trying to explain Deep Reinforcement Learning models. In this thesis we have used hDQN, an implementation in which goals are followed explicitly, which provides some insight about the agent's decision making. However, we still can not explain why the agent chooses a specific goal. For tackling this we could again leverage the fact that the observations from the state are an abstracted vector of features by replacing the MLP for a linear model or a decision tree. Since these models have certain degree of explainability but are not good feature extractors, some extra feature engineering could help them. Another possible research direction would be to add more level to the hierarchy of hDQN, being those at the top more abstract than the ones at the bottom.

[1] Actually, when learning from pixels the agent can see how many missiles are in the screen and use it as a proxy of its current pace of shooting. This is something that alleviates the non-Markovian behaviour of the environment.

Optimal transfer of training Having an ML model for SF opens the possibility of various research directions. One that was of particular interest of NLR was to use it to determine optimal transfer-of-training between different training environments [Roessingh et al., 2002]. Years ago an experiment was conducted with SF trainees where researchers tested an hypothesis. This stated that if we had X amount of hours of training available, the final performance achieved by the trainees will be higher if he/she trained the first $X/2$ hours with a simplified version of SF and the last $X/2$ with the full version than if he/she tried the full version for all the X hours. The result was that starting with a simpler task and by gradually increasing it resulted in better performance. Now this hypothesis can also be tested for the ML model. If the results are similar then we would be closer to having a model that could be used to predict human change in performance after changes in the environment occur.

Chapter 5

Bibliography

[Agarwal et al., 2018] Agarwal, A., Hope, R., and Sycara, K. (2018). Challenges of context and time in reinforcement learning: Introducing space fortress as a benchmark. *arXiv preprint arXiv:1809.02206*.

[Agarwal and Sycara, 2018] Agarwal, A. and Sycara, K. (2018). Learning time-sensitive strategies in space fortress. *arXiv preprint arXiv:1805.06824*.

[Aliko, 2017] Aliko, A. (2017). Temporally extended and intrinsically motivated deep reinforcement learning. master thesis. *University of Amsterdam*.

[Anderson et al., 2011] Anderson, J. R., Bothell, D., Fincham, J. M., Anderson, A. R., Poole, B., and Qin, Y. (2011). Brain regions engaged by part-and whole-task performance in a video game: a model-based test of the decomposition hypothesis. *Journal of cognitive neuroscience*, 23(12):3983–3997.

[Bengio et al., 2009] Bengio, Y. et al. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127.

[Bertsekas and Tsitsiklis, 1995] Bertsekas, D. P. and Tsitsiklis, J. N. (1995). Neuro-dynamic programming: an overview. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[Cybenko, 1989] Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

[Destefano and Gray, 2016] Destefano, M. and Gray, W. D. (2016). Where should researchers look for strategy discoveries during the acquisition of complex task performance? the case of space fortress. In *Proceedings of the 38th Annual Conference of the Cognitive Science Society*, pages 668–673.

[Florensa et al., 2017] Florensa, C., Duan, Y., and Abbeel, P. (2017). Stochastic neural networks for hierarchical reinforcement learning. *CoRR*, abs/1704.03012.

[Frederiksen and White, 1989] Frederiksen, J. R. and White, B. Y. (1989). An approach to training based upon principled task decomposition. *Acta Psychologica*, 71(1-3):89–146.

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

[Gopher et al., 1994] Gopher, D., Well, M., and Bareket, T. (1994). Transfer of skill from a computer game trainer to flight. *Human Factors*, 36(3):387–405.

[Gunning, 2017] Gunning, D. (2017). Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web.*

[Hasselt, 2010] Hasselt, H. V. (2010). Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613–2621.

[Huber, 1964] Huber, P. J. (1964). Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35(1):73–101.

[Ioerger et al., 2003] Ioerger, T. R., Sims, J., Volz, R. A., Workman, J., and Shebilske, W. L. (2003). On the use of intelligent agents as partners in training systems for complex tasks1. In *Proceedings of the Annual Meeting of the Cognitive Science Society*.

[Irpan, 2018] Irpan, A. (2018). Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

[J.J.M. Roessingh and Kappers, 2003] J.J.M. Roessingh, J. K. and Kappers, A. (2003). The acquisition of complex skills and the linear rate model. *National Aerospace Laboratory NLR,*.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285.

[Kulkarni et al., 2016] Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. B. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *CoRR*, abs/1604.06057.

[Lee et al., 2015] Lee, H., Boot, W. R., Baniqued, P. L., Voss, M. W., Prakash, R. S., Basak, C., and Kramer, A. F. (2015). The relationship between intelligence and training gains is moderated by training strategy. *PloS one*, 10(4):e0123259.

[Li, 2017] Li, Y. (2017). Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274.

[Lin, 1993] Lin, L.-J. (1993). Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

[Mané and Donchin, 1989] Mané, A. and Donchin, E. (1989). The space fortress game. *Acta psychologica*, 71(1-3):17–22.

[McNamara et al., 2014] McNamara, C. G., Tejero-Cantero, Á., Trouche, S., Campo-Urriza, N., and Dupret, D. (2014). Dopaminergic neurons promote hippocampal reactivation and spatial memory persistence. *Nature neuroscience*, 17(12):1658.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

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

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[Moon and Anderson, 2012] Moon, J. and Anderson, J. (2012). Modeling millisecond time interval estimation in space fortress game. In *Proceedings of the Annual Meeting of the Cognitive Science Society*.

[Moon et al., 2011] Moon, J., Bothell, D., and Anderson, J. (2011). Using a cognitive model to provide instruction for a dynamic task. In *Proceedings of the Annual Meeting of the Cognitive Science Society*.

[Morris et al., 2006] Morris, G., Nevet, A., Arkadir, D., Vaadia, E., and Bergman, H. (2006). Midbrain dopamine neurons encode decisions for future action. *Nature neuroscience*, 9(8):1057.

[Nair et al., 2017] Nair, A., McGrew, B., Andrychowicz, M., Zaremba, W., and Abbeel, P. (2017). Overcoming exploration in reinforcement learning with demonstrations. *arXiv preprint arXiv:1709.10089*.

[Niv, 2009] Niv, Y. (2009). Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154.

[Oh et al., 2015] Oh, J., Guo, X., Lee, H., Lewis, R. L., and Singh, S. (2015). Action-conditional video prediction using deep networks in atari games. In *Advances in neural information processing systems*, pages 2863–2871.

[Osband et al., 2016] Osband, I., Blundell, C., Pritzel, A., and Roy, B. V. (2016). Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621.

[Ribeiro et al., 2016] Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM.

[Riedmiller, 2005] Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer.

[Roessingh et al., 2002] Roessingh, J., Kappers, A., and Koenderink, J. (2002). Transfer between training of part-tasks in complex skill training. *NLR*.

[Rosenblatt, 1957] Rosenblatt, F. (1957). The perceptron: A perceiving and recognizing automaton. *Cornell Aeronautical Laboratory*.

[Rosenblatt, 1962] Rosenblatt, F. (1962). Principles of neurodynamics; perceptrons and the theory of brain mechanisms. *Washington: Spartan Books.*

[Samek et al., 2017] Samek, W., Wiegand, T., and Müller, K.-R. (2017). Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296*.

[Schaul et al., 2015a] Schaul, T., Horgan, D., Gregor, K., and Silver, D. (2015a). Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320.

[Schaul et al., 2015b] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015b). Prioritized experience replay. *CoRR*, abs/1511.05952.

[Shebilske et al., 2005] Shebilske, W. L., Volz, R. A., Gildea, K. M., Workman, J. W., Nanjanath, M., Cao, S., and Whetzel, J. (2005). Revised space fortress: a validation study. *Behavior Research Methods*, 37(4):591–601.

[Singer and Frank, 2009] Singer, A. C. and Frank, L. M. (2009). Rewarded outcomes enhance reactivation of experience in the hippocampus. *Neuron*, 64(6):910–921.

[Singh et al., 2009] Singh, S., Lewis, R. L., and Barto, A. G. (2009). Where do rewards come from. In *Proceedings of the annual conference of the cognitive science society*, pages 2601–2606.

[Singh et al., 2010] Singh, S., Lewis, R. L., Barto, A. G., and Sorg, J. (2010). Intrinsically motivated reinforcement learning: An evolutionary perspective. *IEEE Transactions on Autonomous Mental Development*, 2(2):70–82.

[Sorg and Singh, 2010] Sorg, J. and Singh, S. (2010). Linear options. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 31–38. International Foundation for Autonomous Agents and Multiagent Systems.

[Spronck et al., 2006] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., and Postma, E. (2006). Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248.

[Stadie et al., 2015] Stadie, B. C., Levine, S., and Abbeel, P. (2015). Incentivizing exploration in reinforcement learning with deep predictive models. *CoRR*, abs/1507.00814.

[Sutton and Barto, 1999] Sutton, R. and Barto, A. (1999). Reinforcement learning: An introduction. *Trends in Cognitive Sciences*, 3(9):360.

[Sutton et al., 1992] Sutton, R. S., Barto, A. G., and Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12(2):19–22.

[Szepesvari, 2010] Szepesvari, C. (2010). *Algorithms for Reinforcement Learning.* Morgan and Claypool Publishers.

[Szepesvari et al., 2014] Szepesvari, C., Sutton, R. S., Modayil, J., Bhatnagar, S., et al. (2014). Universal option models. In *Advances in Neural Information Processing Systems*, pages 990–998.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

[Towne et al., 2016] Towne, T. J., Boot, W. R., and Ericsson, K. A. (2016). Understanding the structure of skill through a detailed analysis of individuals' performance on the space fortress game. *Acta psychologica*, 169:27–37.

[Tsitsiklis and Van Roy, 1997] Tsitsiklis, J. N. and Van Roy, B. (1997). Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081.

[van der Linden et al., 2017] van der Linden, P. A., Intelligentie, B. O. K., Poppinga, G., Roessingh, J., and van Splunter, S. (2017). On higher-order control tasks: The application of a3c on space fortress. *University of Amsterdam. Bachelor thesis.*

[Van Hasselt et al., 2016] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100.

[Vezhnevets et al., 2016] Vezhnevets, A., Mnih, V., Osindero, S., Graves, A., Vinyals, O., Agapiou, J., et al. (2016). Strategic attentive writer for learning macro-actions. In *Advances in neural information processing systems*, pages 3486–3494.

[Vezhnevets et al., 2017] Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. *CoRR*, abs/1703.01161.

[Wang et al., 2015] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N. (2015). Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.

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

[Whetzel, 2005] Whetzel, J. H. (2005). *Developing intelligent agents for training systems that learn their strategies from expert players*. PhD thesis, Texas A&M University.

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

Appendices

Instability on sparse rewards

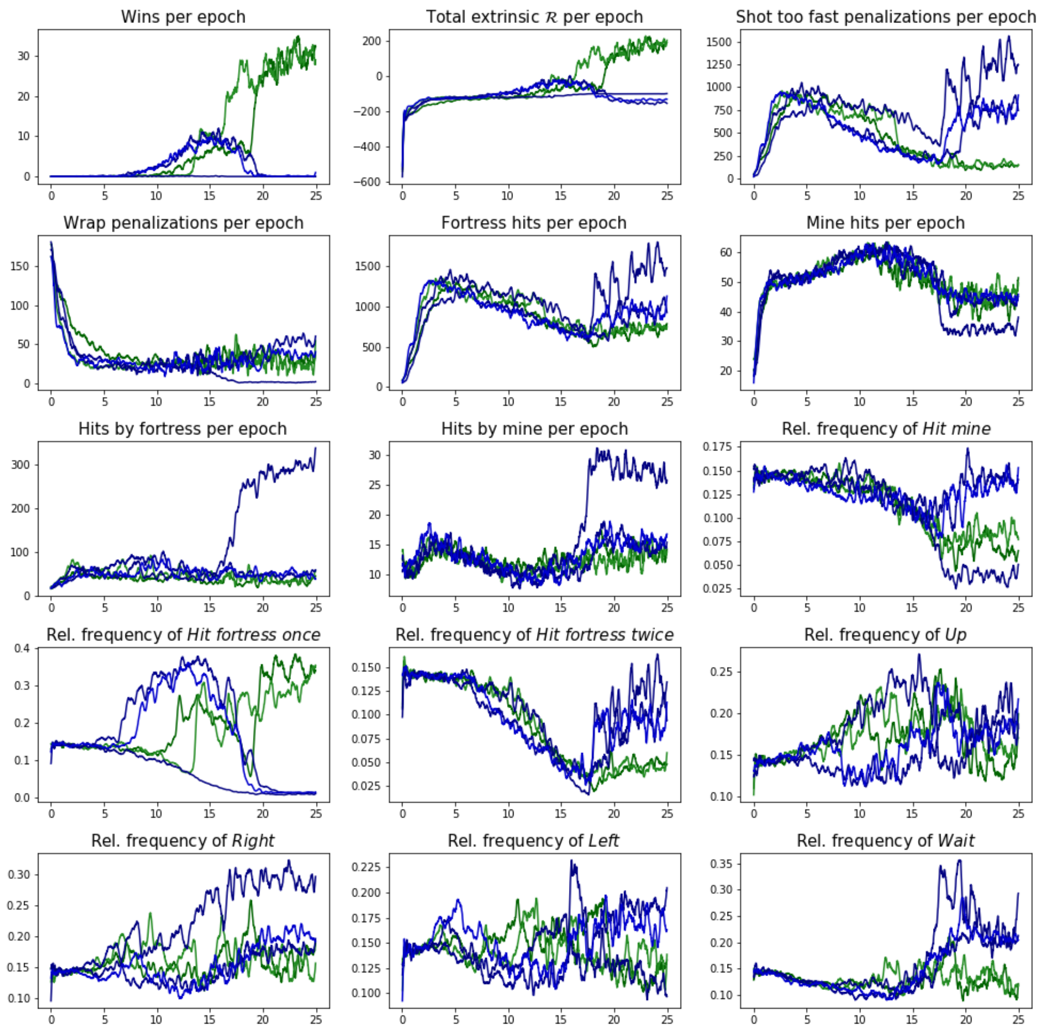

Figure 1: Scalar indicators monitored during the training of five hDQN $\mathcal{G}_2$ agents with different seeds each on SF with $\mathcal{R}^d$. Successful runs are colored with green tones and the unsuccessful ones with blue tones.

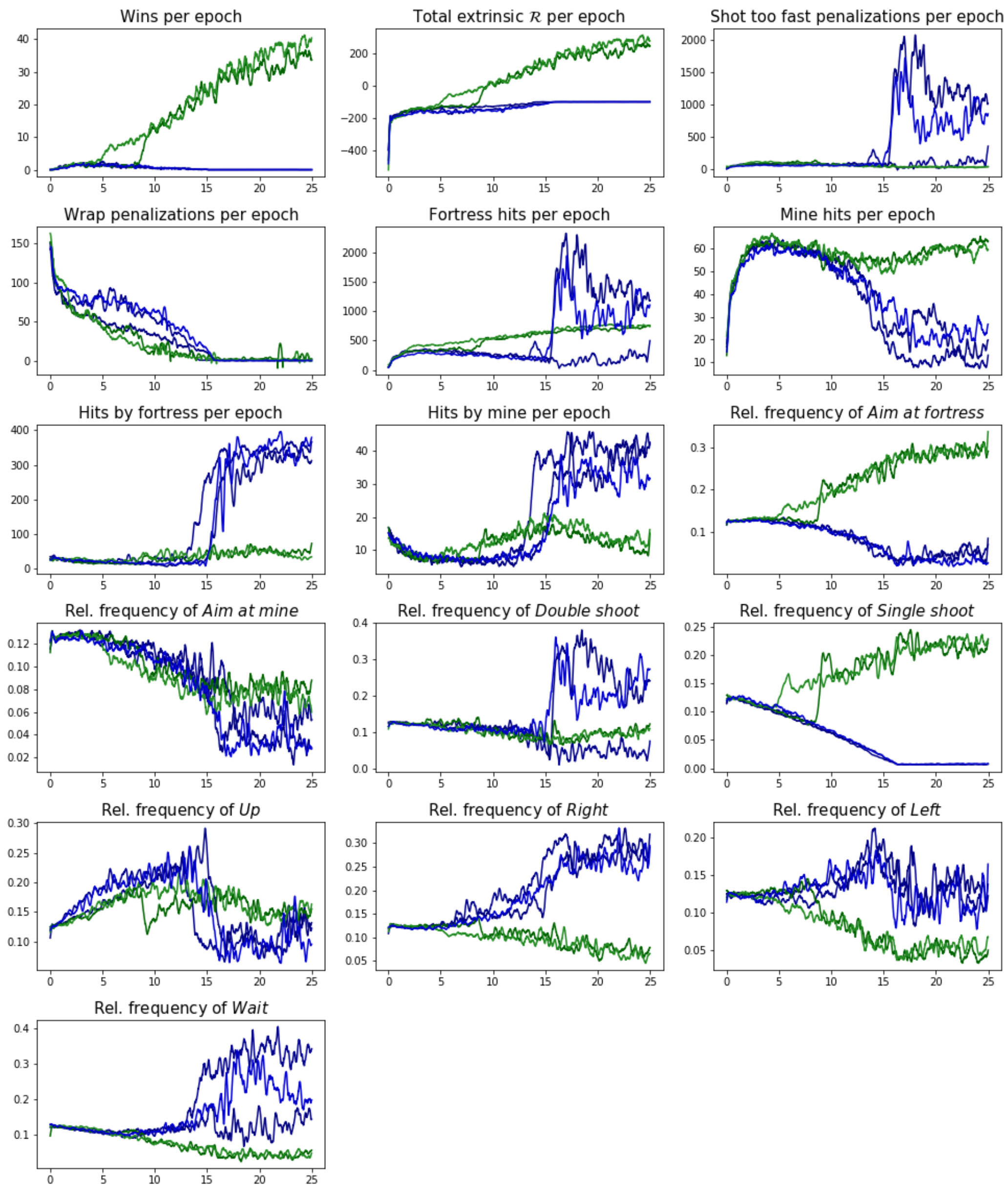

Figure 2: Scalar indicators monitored during the training of five hDQN $\mathcal{G}_1$ agents with different seeds each on SF with $\mathcal{R}^d$. Successful runs are colored with green tones and the unsuccessful ones with blue tones.