David Grigorovich Khachaturov

Susceptibility of Hierarchical Reinforcement Learning to Adversarial Examples

Computer Science Tripos – Part II

Churchill College

Declaration of Originality

I, David Grigorovich Khachaturov of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed: David Grigorovich Khachaturov

Date: 13/05/2021

Proforma

Candidate Number:	
Project Title:	**Susceptibility of Hierarchical Reinforcement Learning to Adversarial Examples**
Examination:	**Computer Science Tripos – Part II, May 2021**
Word Count:	**11,800**[1]
Final Line Count:	**30,791**[2]
Project Originator:	Mr Dmitry Kazhdan
Supervisors:	Mr Ilia Shumailov, Mr Dmitry Kazhdan

Original Aims of the Project

The aim of the project was to implement reinforcement learning and hierarchical reinforcement learning agents, several existing attacks on these agents, and to quantitatively demonstrate the effect of these attacks on trained instances of the agents.

Work Completed

The core success criteria were all successfully achieved. Extensive further work was conducted in terms of explaining and interpreting the observed effects of the attacks. Novel architectures were proposed to address shortcomings of existing architectures that were highlighted during evaluation, and novel theory was put forward in order to explain the observed results. The main bulk of the project was written up as a paper and submitted to NeurIPS 2021[3].

Special Difficulties

None.

[1] Computed by `pdftotext diss.pdf -enc UTF-8 - | tr -cd '0-9A-Za-z \n' | wc -w`

[2] Python files and Jupyter Notebooks, many of which are copies with alterations between experiments.

[3] `https://nips.cc/`

Contents

Chapter 1

Introduction

1.1 Motivation

Reinforcement Learning (RL) deals with agents that learn how to act optimally in stochastic environments through trial-and-error. It has seen a surge in popularity recently, primarily attributed to a number of success stories. OpenAI used Deep Reinforcement Learning to beat the human world champion at Dota 2, a complex and real-time game [1]. AlphaZero, a program leveraging a general RL algorithm, managed to beat state-of-the-art (SotA) programs and top human experts at Go, chess, and shogi [2].

Barto and Mahadevan [3] and, more recently, Kulkarni et al. [4] have tried to address the scalability issues that plague more traditional approaches to RL by learning policies at different levels of abstraction. This approach is broadly known as Hierarchical Reinforcement Learning (HRL). HRL consists of learning multiple policies at different levels of hierarchy, similarly to how humans solve tasks by decomposing them into subtasks [5, 6]. An example of decomposition into subtasks is as follows: the overarching goal of writing a Part II dissertation can be split into writing chapters, which can in turn be split into writing individual paragraphs and so on.

Much like humans can be fooled with auditory and visual tricks, neural networks are also susceptible to these so-called *adversarial examples*. In 2012 Biggio et al. [7] investigated attacks against Support Vector Machines. The first attacks on neural networks were discovered and formalised by Szegedy et al. [5] in 2014. These were applied to *convolutional* neural nets. It was later discovered by Huang et al. [8] that networks based on RL techniques are also susceptible to adversarial examples. These work by introducing minor perturbations to the inputs that would result in the model producing unexpected results. The minor alterations were shown to greatly reduce the effectiveness of the trained agents at any one time-step, and cumulatively over the entire agent's lifetime, highlighting the relatively poor *robustness* of RL.

The notion of *generalisation* has been studied to great depth in the context of RL. It is generally accepted that RL agents often overfit to the presented task or environment [9]. This makes them perform poorly when faced with a slightly altered environment, which severely limits real-world use-cases for the technology. While there have been works, such as Igl et al. [10] and Ye et al. [11], that target this problem of overfitting, they apply complicated regularisation techniques inspired by supervised learning approaches and often result in significant slow-downs of the agent's training regime.

Robustness and generalisation are closely linked: intuitively, for an agent to be robust it needs to have low generalisation properties and vice versa. This implies that to be robust an agent would have to overfit to the specific task or environment at hand, limiting its ability to solve similar tasks. On the other hand, if an agent learns to cope with a general class of tasks, it becomes sensitive to adversarial examples. The generalisation bounds for learning algorithms, based on their robustness, have been derived theoretically by Xu and Mannor [12]. However, there are no existing works that consider HRL agents and their generalisation and robustness properties.

Combining these recent developments – the rising popularity of HRL and recently-discovered attacks on RL – naturally lends itself to the question of whether HRL would be susceptible to the same attacks. The human-inspired architecture of HRL also leads to some interesting intuition as humans are relatively adept at dealing with adversaries: the same trick would hopefully not fool someone twice, for example.

Due to the increased complexity and difference in model structure, one could speculate that HRL models should be less vulnerable to the same attacks that can fool RL models. The idea is that separation of a global task into subtasks should help generalisation and improve robustness of the agent. On the other hand, a number of characteristics would suggest otherwise. The underlying principles are very similar to traditional RL; and the hierarchical nature of HRL could suggest that fooling one level of the model would be enough to propagate the errors down to the output.

1.2 Project Summary

This project aims to experimentally test the susceptibility of HRL to adversarial examples and compare it to RL. This is done by implementing several existing HRL and RL architectures, training agents on a carefully-prepared set of environments, and quantitatively reporting the effect of several existing attacks on these agents. Extensive further work was conducted in terms of explaining and interpreting the observed effects of the attacks. Novel architectures were proposed to address some shortcomings of existing architectures that were highlighted during evaluation, and novel theory was put forward in order to explain the observed results.

Chapter 2

Preparation

I present an overview of the theory behind RL and HRL in Sections 2.1 and 2.2. This is followed up with a dive into the history and technicalities of adversarial examples in the context of ML in Section 2.3. Section 2.4 then talks about the metrics for quantitative evaluation of models, as well as the motivation behind their choice. The chapter is then concluded with a requirements analysis and a description of the starting point.

2.1 Reinforcement Learning

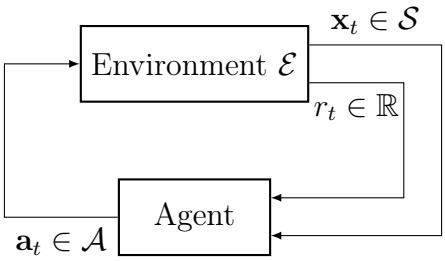


Figure 2.1: High-level overview of a generic RL architecture.

Consider an *agent* in an environment $\mathcal{E}$. At each time-step, the agent selects an action $\mathbf{a}_t$ from the environment's action space $\mathcal{A}$. $\mathbf{a}_t$ alters $\mathcal{E}$ in some way and the agent receives some *observation* $\mathbf{x}_t \in \mathcal{S}$, where $\mathcal{S}$ is the state space of the environment (usually $\mathcal{S} \subseteq \mathbb{R}^d$ for some dimensionality $d \in \mathbb{Z}$), and a reward $r_t \in \mathbb{R}$. An overview is given in Figure 2.1.

The aim of the agent is to maximise future-discounted rewards. Define the future-discounted *reward* at time-step t to be:

$$R_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} \tag{2.1}$$

where γ is the discount factor for future rewards. It then follows that the agent is trained to learn a *policy* π mapping states to actions ($\pi : \mathcal{S} \to \mathcal{A}$) to maximise the expected return from the start distribution $J = \mathbb{E}_{r_i, \mathbf{x}_i \sim \mathcal{E}, \mathbf{a}_i \sim \pi}[R_1]$.

Define the optimal *action-value* function $Q^*(\mathbf{x}, \mathbf{a})$ as the maximum expected return achievable over all possible strategies after observing state $\mathbf{x}$ and taking an action $\mathbf{a}$. It then follows that:

$$Q^*(\mathbf{x}_t, \mathbf{a}_t) = \max_\pi \mathbb{E}_{\substack{\mathbf{x}_{t+1:}\mathbf{x}_T, \\ \mathbf{a}_{t+1:}\mathbf{a}_T}}[R_t \mid \mathbf{x}_t,\ \mathbf{a}_t] \tag{2.2}$$

where $\mathbb{E}$ represents expectation. One can find that $Q^*(\mathbf{x}, \mathbf{a})$ obeys the *Bellman equation*. i.e. if the optimal value of $Q^*(\mathbf{x}_{t+1}, \mathbf{a}')$ for state $\mathbf{x}_{t+1}$ at the next time-step was known for all possible actions $\mathbf{a}'$, then the optimal strategy is to select the action $\mathbf{a}'$ that maximises the expected value of $r_t + \gamma Q^*(\mathbf{x}_{t+1}, \mathbf{a}')$, i.e.:

$$Q^*(\mathbf{x}_t, \mathbf{a}_t) = \mathbb{E}_{\mathbf{x}_t \xrightarrow{\mathbf{a}_t} \mathbf{x}_{t+1}}[r_t + \gamma \max_{\mathbf{a}'} Q^*(\mathbf{x}_{t+1}, \mathbf{a}') \mid \mathbf{x}_t,\ \mathbf{a}_t] \tag{2.3}$$

Reinforcement learning algorithms estimate the optimal action-value function by iteratively updating an approximation $Q_i(\mathbf{x}, \mathbf{a})$ using the above Bellman equation: $Q_{i+1}(\mathbf{x}_t, \mathbf{a}_t) = \mathbb{E}_{\mathbf{x}_t \xrightarrow{\mathbf{a}_t} \mathbf{x}_{t+1}}[r_t + \gamma \max_{\mathbf{a}'} Q_i(\mathbf{x}_{t+1}, \mathbf{a}') \mid \mathbf{x}_t,\ \mathbf{a}_t]$. The iteration algorithms would then converge at the optimal action-value function as $i \to \infty$ and hence $Q_i \to Q^*$. A purely value-based algorithm is impractical due to often very large state and action spaces and time/resource constraints. Hence, it is common to use a function approximator $Q(\mathbf{x}, \mathbf{a}; \theta) \approx Q^*(\mathbf{x}, \mathbf{a})$. The standard function approximator is a neural network with weights θ known as a Q-network [13].

Q-Learning [14] and subsequently Q-networks, use a *greedy* approach for determining the policy π:

$$\pi(\mathbf{x}_t) = \mathbf{a}_t = \text{argmax}_{\mathbf{a}} Q(\mathbf{x}_t, \mathbf{a}) \tag{2.4}$$

The Q-network can be trained to minimise a sequence of loss functions $L_i(\theta_i)$. The parameters from the previous iteration θ_{i-1} are fixed when optimising $L_i(\theta_i)$:

$$L_i(\theta_i) = \mathbb{E}_{\mathbf{x}, \mathbf{a} \sim \rho(\cdot)}\left[(y_i - Q(\mathbf{x}, \mathbf{a}; \theta_i))^2\right] \tag{2.5}$$

$$y_i = \mathbb{E}_{\mathbf{x}_t \xrightarrow{\mathbf{a}_t} \mathbf{x}_{t+1}}[r_t + \gamma \max_{\mathbf{a}'} Q^*(\mathbf{x}_{t+1}, \mathbf{a}'; \theta_{i-1}) \mid \mathbf{x}_t,\ \mathbf{a}_t] \tag{2.6}$$

where y_i is an adaptation of Equation (2.3) and the *target* for iteration i; and $\rho(\mathbf{x}, \mathbf{a})$ is the probability distribution over states $\mathbf{x}$ and actions $\mathbf{a}$. Note that while minimising the loss stochastic gradient descent (SGD) is often used, rather than a calculation involving the full expectations for the loss gradient [13].

2.1.1 Actor-Critic Model

The Q-network approach allows for high-dimensional observation spaces, but is only applicable to discrete action spaces. Most real-world tasks, such as robotic control, have continuous and high-dimensional action spaces. Q-networks rely on finding an action maximising the *action-value* function. In a continuous setting, this would involve iterative optimisation at every step, greatly slowing down decision-making.

A solution involving discretising the action space falls victim to the *curse of dimensionality* and does not allow for fine control of actions [15]. An in-depth overview of the difference between discrete and continuous action spaces is given in Section 3.1.

With these considerations in mind Lillicrap et al. [15] proposed a model-agnostic *actor-critic* algorithm called deep deterministic policy gradient (DDPG) that is able to learn high-dimensional, continuous action spaces.

The algorithm maintains a parameterised *actor* $\pi(\mathbf{x} \mid \theta^\pi)$, with $\pi : \mathcal{S} \rightarrow \mathcal{A}$, and a *critic* $Q(\mathbf{x}, \mathbf{a} \mid \theta^Q)$ that is learnt using the Bellman equation described in Equation (2.3).

The actor is updated by applying the chain rule to the expected return from the start distribution J with respect to the actor's parameters, using the critic's *learnt* Q-function [15]:

$$\nabla_{\theta^\pi} J \approx \mathbb{E}_{\mathbf{x}_t \sim \mathcal{E}}[\nabla_{\theta^\pi} Q(\mathbf{x}, \mathbf{a} \mid \theta^Q) \mid \mathbf{x} = \mathbf{x}_t,\ \mathbf{a} = \pi(\mathbf{x}_x \mid \theta^\pi)] \tag{2.7}$$

$$\approx \mathbb{E}_{\mathbf{x}_t \sim \mathcal{E}}[\nabla_{\mathbf{a}} Q(\mathbf{x}, \mathbf{a} \mid \theta^Q) \nabla_{\theta^\pi} \pi(\mathbf{x} \mid \theta^\pi) \mid \mathbf{x} = \mathbf{x}_t,\ \mathbf{a} = \pi(\mathbf{x}_x \mid \theta^\pi)] \tag{2.8}$$

For further details, please refer to Lillicrap et al. [15].

2.2 Hierarchical Reinforcement Learning

The main hurdle for traditional RL techniques is the task of exploration with sparse feedback. This implies that the agent will not be receiving 'useful' rewards regularly and will often be required to perform long sequences of steps without being explicitly rewarded for them. Since rewards are used for 'reinforcing' decisions, this proves to be an obvious obstacle. HRL attempts to overcome this by motivating agents to solve intermediate goals with attached intrinsic rewards that aid exploration. This is achieved by combining deep reinforcement learning with hierarchical value functions [4]. The model is broken down into two (or more [16]) stages, with each trying to optimise for different rewards. Without loss of generality, I consider a two-level HRL hierarchy in the remainder of this work, in line with the SotA [17].

The two-level structure has a lower-level controller (with corresponding policy μ^L), known as the **worker**, and a higher-level controller (with corresponding policy μ^H), known as the **manager**. The two controllers respectively receive $\mathbf{x}_t^L = \mathbf{x}_t^H$ from $\mathcal{E}$. While the worker produces $\mathbf{a}_t \in \mathcal{A}$, the manager would operate on coarser levels of abstraction and set *goals* $\mathbf{g}_t \in \mathcal{G}$ in the goal space $\mathcal{G}$ for the lower-level controller. The exact definition of a goal is left up to the specific architecture. The goal would be passed down to the worker together with the original observations.

In line with most existing HRL SotA [17, 16], I primarily consider cases where $\mathcal{G} \subseteq \mathcal{S}$ (state-space goals). I conduct some additional experiments where I consider $\mathcal{G} \subseteq \mathcal{A}$ (action-space goals); these can be found in Appendix B.

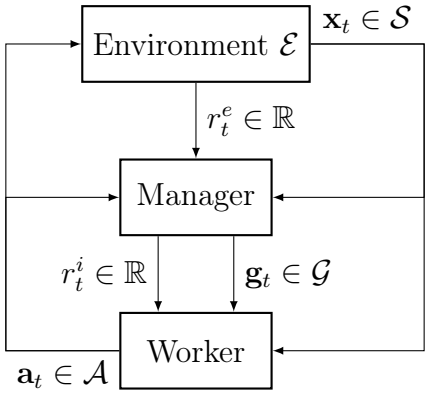


Figure 2.2: High-level overview of a generic HRL architecture.

The worker acts as a standard RL agent performing an action $\mathbf{a}_t$ based on the joint goal and observation. However, the extrinsic environment reward r_t^e (r_t in earlier notation) would only be visible to the manager. The worker would received an intrinsic, often parameterised [18], reward r_t^i from the manager. In HIRO [17], r_t^i is defined to be the negative L2-norm between $\mathbf{x}_t^H$ and $\mathbf{g}_t$:

$$r_t^i(\mathbf{x}_t, \mathbf{g}_t, \mathbf{x}_{t+1}) = -||\mathbf{x}_t + \mathbf{g}_t - \mathbf{x}_{t+1}||_2 \tag{2.9}$$

An overview of a generic HRL model is presented in Figure 2.2. As this is a developing field, the design of how the hierarchy is structured and how goal-setting is performed are both open problems. Two popular architectures, HDQN [4] and HIRO [17], are discussed in detail in the *Implementation* chapter.

2.3 Adversarial Examples

The effectiveness of *adversarial examples* against neural networks was first shown by Szegedy et al. [5]. Goodfellow et al. [19] focused on explaining and exploiting these adversarial examples when applied to classification tasks: a small carefully-chosen perturbation of an otherwise normal image can cause a model to misclassify the image with high confidence.

Huang et al. [8] showed the effectiveness of such adversarial examples and methods of generating them in the context of RL. The use of ML agents in the 'real world' is making the topic of agent robustness to such examples relevant. ML is often used in high-stakes environments, such as for computer vision to aid self-driving cars. Whenever an agent is deployed beyond a lab setting, it is automatically exposed to potential malicious actors. For example, Athalye et al. [20] produce a physical adversarial example in the form of a toy turtle that is detected as a rifle by image classification algorithms.

In general, any test-time adversarial attack consists of generating a *perturbation* η – a small element-wise augmentation – that is applied to the state $\mathbf{x} \in \mathcal{S}$ observed by an agent.

The adversarial example $\mathbf{x}' \in \mathcal{S}$ is then defined to be:

$$\mathbf{x}' \leftarrow \max(\mathbf{x}_{min}, \min(\mathbf{x}_{max}, \mathbf{x} + \eta)) \tag{2.10}$$

where $\mathbf{x}_{max/min}$ are the maximum and minimum possible values of $\mathbf{x}$, i.e. 1.0 and 0.0 in a normalised observation space. This clamping is done to ensure that $\mathbf{x}' \in \mathcal{S}$.

2.3.1 Random Noise

It was recently shown by Zhao et al. [21] that black-box random noise perturbations work quite well as an attack on RL systems, only marginally being outperformed by specially-crafted adversarial samples.

The perturbation generated with this method is simply:

$$\eta = \mathcal{U}(-\epsilon, \epsilon) \tag{2.11}$$

where $\mathcal{U}(a, b)$ is a function that samples a uniform distribution with minimum value a and maximum value b, with ϵ being the perturbation budget.

2.3.2 Fast Gradient Sign Method

Goodfellow et al. [19] introduced the white-box "fast gradient sign method" (FGSM) of generating adversarial examples, relying on gradient calculations of the model. There are two main flavours of FGSM:

Targeted FGSM generates perturbations that *minimise* network loss with respect to a given *target*. This is used, for example, when one wants a classification network to explicitly mislabel some input as a predetermined class y.

The perturbation becomes:

$$\eta = -\epsilon \text{sign}\left(\nabla_{\mathbf{x}} \mathcal{L}(\theta(\mathbf{x}), y)\right) \tag{2.12}$$

where y is the 'target' associated with $\mathbf{x}$, and $\mathcal{L}$ is a loss function suitable for use between the network's output and the chosen y. I use $\theta(\mathbf{x})$ as shorthand to indicate a forward pass of a network with parameters θ and input $\mathbf{x}$.

Non-targeted FGSM, in turn, produces perturbations which *maximise* network loss for a given input. This is used when a malicious actor just wants to degrade a model's performance.

The perturbation in this case becomes:

$$\eta = \epsilon \text{sign}\left(\nabla_{\mathbf{x}} J(\theta, \mathbf{x})\right) \tag{2.13}$$

where J is the network's loss function.

Non-targeted attacks usually result in higher performance degradation for the same perturbation budget as a targeted construction is an inherently harder optimisation task, requiring solving for an additional constraint. Note in both cases the required gradient $\nabla_{\mathbf{x}} \ldots$ can be computed efficiently using back-propagation.

2.4 Performance Indicators

I will consider three main performance indicators: safety and security from a security engineering perspective; and generalisation from an ML perspective. I will refer to these jointly as the **SSG** properties of an architecture or model.

2.4.1 Safety & Security

There are two main factors to consider when evaluating the performance of any system from a security engineering perspective. These are *safety* and *security* [22]. In general, safety is concerned with system performance in the presence of non-malicious behaviour, whilst security concerns itself with situations where explicit malicious intent is present: an adversary would be trying to manipulate the system.

In an ML setting, *safety* of a system is determined by the degradation of performance under the presence of *noise*. This noise is assumed to be non-adversarial, in the sense that it is non-discriminatory and non-targeted. I will be testing the safety of my trained agents using random noise, as defined in Section 2.3.1.

The *security* (or *robustness*) of a system is defined by the degradation of performance under the presence of *explicit adversarial perturbations* generated by a malicious actor. I will be testing the robustness of my agents under the presence of perturbations generated by FGSM, as defined in Section 2.3.2.

The above factors 'score' higher the less degradation happens for a given perturbation size.

2.4.2 Generalisation (Extension)

Generalisation in an ML context refers to how well a task is understood by a given model. It is harder to define and test for generalisation in a context-agnostic manner. I will assess this metric via model performance: in an augmented-action environment, and with an out-of-training-distribution starting state. A similar approach was undertaken by Igl et al. [10].

I define an 'augmented-action environment' to be an environment with the same structure and completion metrics as the environment an agent was trained on, but where actions have slightly different effects. For example, an action that resulted in moving an arbitrary distance of 2 units in the original environment would only move the agent 1 unit in the augmented-action environment.

The idea is that an agent with an in-depth understanding of the environment would be able to cope with both of these setups. This is due to the perceived state retaining its semantic meaning in both cases. Meanwhile, an agent that experienced overfitting would trip up even with slight augmentations.

2.5 Requirements Analysis

The project was implemented entirely in Python using the PyTorch [23] ML and autograd framework, relying on OpenAI's *gym* [24] framework for environment simulation. The majority of the work was run via Google Colaboratory [25], which I had to become familiar with. All experiments are easily reproducible with a single button click.

The project centres around RL and HRL theory, which had to be learnt and studied from scratch. Due to both topics being active research areas, there was a substantial amount of reading that had to be done to ensure that the work done is relevant and up-to-date.

I had to become particularly familiar with the research detailing the exact HRL and RL architectures that I planned to use, namely DQN [13], HDQN [4], TD3 [26], and HIRO [17].

There was a significant amount of adversarial ML theory to cover as well, which my supervisors helped with by pointing me towards the most relevant and recent papers.

2.5.1 Personal

I have some previous experience with ML, Python, and security both through personal endeavours and the Computer Science Tripos. I have completed a number of relevant Tripos courses: *Machine Learning and Real-world Data*, *Software and Security Engineering*, *Artificial Intelligence*, and *Security*. I was fortunate enough to gain work experience as an ML research intern at Speechmatics where I worked for 3-and-a-half months in the summer vacation following Part 1A. During my internship, I worked extensively with Python, specifically TensorFlow and PyTorch, in a production environment. The work was mainly focused on Natural Language Processing and analysis in contrast to this project's RL focus.

I have read through the notes of Part II's *Machine Learning and Bayesian Inference* course but have not taken the course. I, however, took *Deep Neural Networks* as a unit of assessment in Lent term. The course provided me with a deeper understanding of the fundamentals of neural networks. More tangentially related, I took the *Data Science: principles and practice* unit of assessment in Michaelmas term, which aided with quantitative evaluation of this project.

I take personal interest in the ML field and keep up-to-date with the latest developments by reading through recently-published papers on arXiv and following specialised blogs. Furthermore, I acquainted myself further with adversarial ML by writing a paper on the field's implications for inpainting; the paper was submitted to and accepted by ICML 2021[1].

[1] `https://icml.cc/`

2.5.2 Existing Work

A number of HRL and RL papers have open-source implementations on GitHub. However, most available HRL-specific code have deviations in either implementation or base assumptions compared to the papers they reference. Quite a lot of the available code also consists of incomplete paper implementations. As such, I will be re-implementing all the RL and HRL models for this project to allow for easier attack implementations. I use the existing code bases for correctness checking.

Attacks, such as gradient-based ones [8, 27], have been shown to work on RL agents. Their implementations are all relatively simple and described in detail in their respective papers. These have never been applied to HRL agents before in a formal setting.

Chapter 3

Implementation

The project's core implementation consists of three main components:

1. DQN, HDQN, TD3, and HIRO agent implementations.
2. Designing and implementing 3 HRL-specific *Point* gym [24] environments inspired by Nachum et al. [17]'s *Ant* environments.
3. Designing and running multiple experiments testing the safety, security, and generalisation (**SSG**) properties of RL and HRL models.

These were followed by the implementation of the extension components:

1. Interpretation of the SSG experiment results in terms of what they mean for the viability and future developments of HRL.
2. Proposing novel theory in regards to the factors that contribute to the SSG performance of RL and HRL. Implementing said theory in practice by means of designing new architectures.
3. Re-running the SSG experiments on the new architectures and comparing results to the standard RL and HRL models.

In addition to this work, a significant amount of implementation was not included in the main body of the dissertation due to its tangential nature. This work, however, was essential to my understanding of the field and without it I would not have been able to reach the milestones that I have. This work is presented in Appendix B. A brief summary is as follows:

1. Multiple existing gym environments have been modified and had experiments run on them.
2. Actions-as-goals for HRL have been briefly explored and experimented with. This is a novel idea.

3.1 Environment Overview

To understand when and where to apply certain models, one needs to understand the different kind of environments that an agent could attempt to solve. OpenAI's *gym* [24] environments, which are ubiquitous for RL experiments, have two types of environments classed by the action spaces: *discrete* and *continuous*.

In *discrete* action space environments, an agent executes one of N actions per time-step. In *continuous* action space environments, an agent executes an action vector of length N per time-step. Each element in the action vector corresponds to an independent action and the element's value represents the *intensity* of the performed action. In *gym*, the action space is constrained by two vectors `low` and `high`, each having length N and respectively defining the lower and upper bounds for each piece-wise element in the action vector.

The main focus of this project will be on continuous action space environments due to their increasing relevance in the field of robotics and the active research surrounding them in the field of HRL. I briefly explore discrete action space environments in Appendix B.2.

RL and HRL models usually solve one of these two environment types, with some exceptions where the architecture can be adapted to suit either type [28]. Various architectures and their corresponding environment types are presented in Table 3.1.

Discrete	Continuous
DQN [13]	DDPG [15]
DDQN [29]	TD3 [26]
HDQN [4]	HIRO [17]

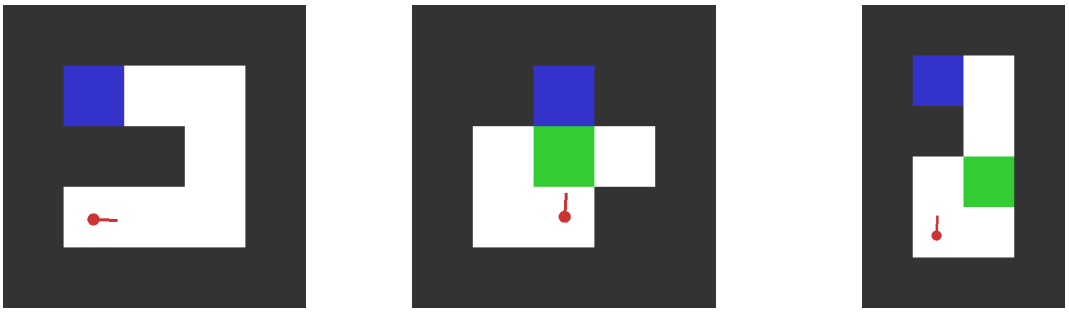
Table 3.1: Non-exhaustive list of common RL and HRL architectures binned by action space type. The first two rows are RL, the final row is HRL.

In lieu of this, I present environments available in *gym* with similar grouping in Table 3.2.

Discrete	Continuous
CartPole-v1	MountainCarContinuous-v0
Acrobot-v1	Pendulum-v0
MountainCar-v0	MuJoCo (all)
Atari (all)	

Table 3.2: Non-exhaustive list of *gym* environments binned by action space type. These environments were selected as they are related to the implemented experiments.

The initial work done for this project focused on these environments, but the lack of simple environments that had both numeric states (i.e. not a visual ones) and were complex enough to justify using state-space goals proved to be a major limitation in terms of usefulness in exploring the properties of HRL. I elaborate on this further in Appendix B.1. With this in mind, I chose to implement my own environments.



(a) PointMaze-v0 (b) PointPush-v0 (c) PointFall-v0

Figure 3.1: Visual representation of the various environments used.

It is worth noting that MuJoCo is a closed-source commercial physics engine that is used extensively in HRL research at the moment. Licensing involves producing a unique identifier tied to a physical machine (thus making it impossible to use on rented cloud server) and can be considered prohibitively expensive in cost. This concern has also been raised in a review at ICLR 2021 [30], which gave a paper a 'clear rejection' for using MuJoCo as a benchmark. The reasons cited were that MuJoCo is a commercial piece of software, without a free license for academic use, making it potentially inaccessible for under-represented researchers. I personally feel that this review, and the authors' subsequent adjustments in re-implementing their benchmarks using open-source alternatives is a positive trend in the field. This focus on inclusivity of RL research will surely make it more accessible in the near future, yielding better and more diverse research.

3.2 Environments

One novel aspect of this project is that I comparing performance of RL and HRL. This implies that I need to test on environments that are solvable by both architectures. The environments used to demonstrate HRL are usually completely unsolvable by traditional RL, making meaningful comparison of performances impossible. The more 'standard' HRL environments with sparse feedback are hence automatically ruled out. This presented an additional challenge, as the environments had be complicated enough that the use of HRL would be intuitively justified, but simple enough to be solvable by RL agents.

The main environments that were evaluated upon are the **Point** environments – **PointMaze**, **PointPush**, **PointFall**. These were implemented from scratch using boilerplate code from *gym* and are inspired by the MuJoCo *Ant* environments presented in Nachum et al. [17], which are used to benchmark the current SotA HRL architecture HIRO. The Point environments are open-source and publicly-available on this project's GitHub repository[1], hopefully contributing to more inclusive HRL research in the future.

The *Point* environments closely mimic the *Ant* ones. The agents control a 'point', which has infinitesimal size and a position and orientation. An action consists of a distance $0 \leq d \leq 1$ to move and an angle to change the orientation by $-\frac{\pi}{4} \leq \Delta\theta \leq \frac{\pi}{4}$.

[1] `https://github.com/davidobot/adversarial_hrl`

The environments have a *scale factor* $s > 0$, which is used to control difficulty. The distance actually moved by the agent within the environment is $\frac{d}{s}$, meaning that the *largest* distance an agent is able to move in a single step is $\frac{1}{s}$. Note that a unit distance (e.g. size of a single block) within these environments is 1. The final *Ant* results presented in Nachum et al. [17] use $s = 8$; I take $s = 4$ to allow RL models to solve the environments for the sake of comparison.

There is a 500 step limit to episodes in all *Point* environments, with a -0.1 reward for each timestep, $+100$ for reaching the goal. All environments are considered solved after the agent receives an average reward equal to or above 90 over the last 100 episodes. The environment provides observations in the form (x, y, θ, t), where x, y, θ are respectively the position and orientation of the agent and t is the number of steps taken thus far in the current episode.

PointMaze Figure 3.1a shows a visual representation of the environment. The agent is shown in red, with the orientation indicated by the 'pole'. The goal of the agent is to reach the blue tile, at which point the episode is terminated.

PointPush In Figure 3.1b one can see a similar agent. The agent is expected to push the green box from the left-hand side in order to gain access to the blue tile. If the agent naively pushes the box up, it blocks the path to the blue tile for the remainder of the episode.

PointFall In Figure 3.1c a similar agent can be seen. This time, the agent is expected to push the green box an entire tile up, at which point the box becomes traversable and the agent can proceed to the blue tile.

I will refer to these environments interchangeably with and without a "-v0" suffix. The suffix comes from gym environment naming conventions. I describe all additional environments that were used for testing actions-as-goals and initial investigation in Appendix B.2.

3.3 Agents

3.3.1 Reinforcement Learning Agents

All the hierarchical models use traditional RL models under-the-hood. Hence, to implement the HRL models, I had to first implement their underlying RL models. The RL model performance also served as a benchmark to compare HRL against.

The following RL agents were implemented:

1. DQN [13] was one of the first attempts of applying Deep Learning to the Q-learning [14] algorithm described in Section 2.1.

2. TD3 [26] uses an *actor-critic* model, as described in Section 2.1.1. The architecture addressed the pitfalls of earlier attempts at continuous action domain agents, such as DDPG [15]. Namely, it addressed the problem of Q-value overestimation.

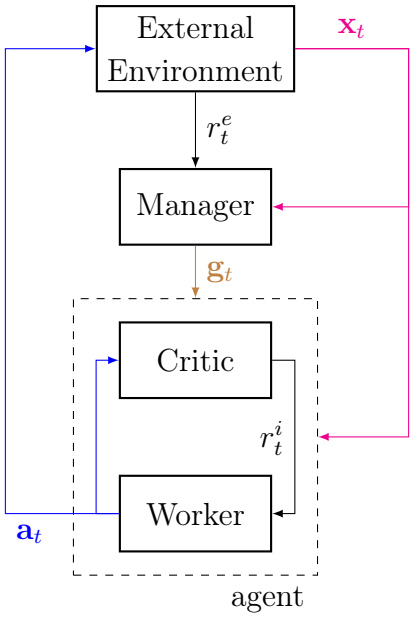


Figure 3.2: Structure of HDQN, proposed by Kulkarni et al. [4]. The controllers are implemented as deep Q-networks (DQNs) [13]. The critic is a deterministic algorithm for checking whether a goal is achieved and awarding an intrinsic reward.

3.3.2 Hierarchical Reinforcement Learning Agents

The HDQN [4] architecture, illustrated in Figure 3.2, has goals that are drawn from a pre-defined *entity space* to constrain the exploration space. The entity space is defined on a high-level as the individual *objects* in the environment, such as the agent, doors, keys, etc.. The critic is defined in the space $\langle entity_1, relation, entity_2 \rangle$, e.g. $entity_1 = agent$, $relation = reaches$, $entity_2 = door$. The presence of a critic allows for goals with non-trivial formulations. i.e. ones where checking whether the goal is reached requires more abstract notions of 'reached'. This setup lends itself to using hand-made goal spaces, as unsupervised detection of objects in scenes is an open problem in computer vision [4].

In contrast, as illustrated in Figure 3.3, HIRO's top-level controller produces goals in the *state space*, i.e. $\mathcal{G} \subseteq \mathcal{S}$, making for a very general notion of a goal. The intrinsic reward r_t^i is then defined as the negated 'distance' from the current state to the goal state: $r^i(\mathbf{x}_t, \mathbf{g}_t, \mathbf{a}_t, \mathbf{x}_{t+1}) = -||\mathbf{x}_t + \mathbf{g}_t - \mathbf{x}_{t+1}||_2$. This suggests that the manager nudges the worker towards a specific location via the set goals. 'Clearing' a goal would imply reaching the desired state $\mathbf{g}_t$. However, it is often enough for the goals to encourage the worker to make decisions that benefit environment exploration or completion, and so they do not need to be strictly 'cleared' before being reassigned.

Off-policy training usually involves some sort of *replay memory* to store past environment interactions to learn from at later stages; the updated policy is different from the behaviour policy. On-policy training has no replay memory and learns directly from its interactions with the environment; the learnt policy also controls the agent's behaviour. For further clarification, please see Gu et al. [31].

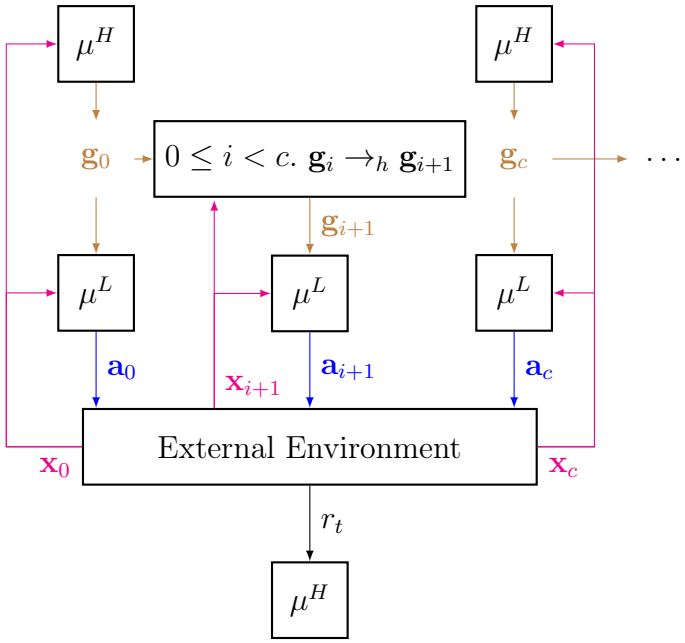


Figure 3.3: Structure of HIRO [17]. The controllers are TD3 agents [26]. The manager and worker have been respectively labelled as μ^H and μ^L. The meta-controller outputs *goals* $\mathbf{g}_t \in \mathcal{G}$ every c steps. On intermediate steps, a fixed goal transition function h determines the next step's goal. Intrinsic rewards are omitted.

Off-policy training is usually preferred to on-policy as it is more efficient. All the approaches mentioned in this work, both RL and HRL, use off-policy training as they are based on Q-learning [14]. However, HIRO's general goal-proposing methods would require on-policy training This is due to the lower-level policy changing underneath the higher-level policy [17].. Hence, a sample observed for a high-level action, e.g. a goal-proposal, may not yield the same low-level behaviour in the future and thus will not be a valid experience for training. This results in a non-stationary problem for the manager's policy. Nachum et al. [17] remedy this issue by introducing off-policy correction. This re-labels past experiences with an augmented high-level action (goal) that is chosen to maximise the probability of the past lower-level actions.

Aside from work on the optimal goal-proposing methods, there has also been work exploring partial information-hiding between the different hierarchies. Rather than fully concealing the extrinsic reward, Vezhnevets et al. [32] propose to optimise $R_t^{\text{extrinsic}} + \alpha R_t^{\text{intrinsic}}$ where α is a constant.

The following HRL agents were implemented:

1. HIRO [17] uses TD3 controllers for the manager and worker.
2. HDQN [4] uses DQN controllers. The model was used for initial investigations.

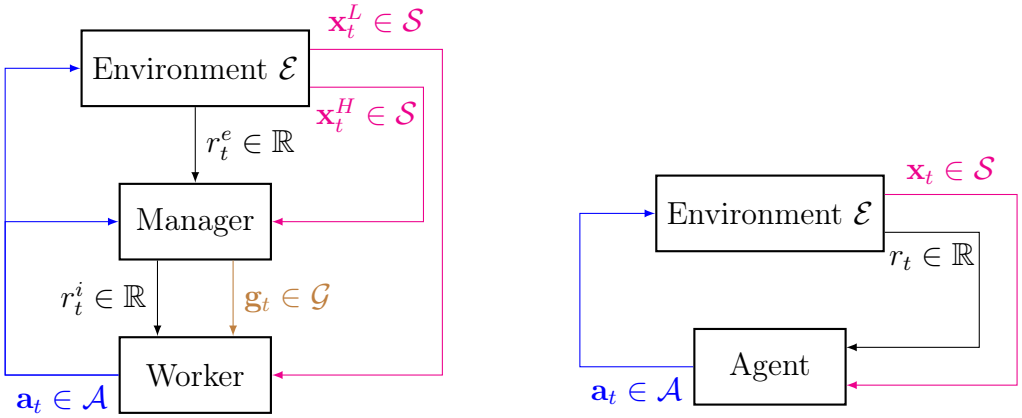


Figure 3.4: Generic HRL (Figure 3.4a) and RL (Figure 3.4b) architectures. Colours represent equivalent structure in the specific HRL architectures in Figures 3.2 and 3.3.

3.4 Attacks

In this section, I discuss the implementation details of the attacks outlined in Section 2.3.

The state observations $\mathbf{x}$ shown in magenta in Figure 3.4a are the main attack point. The action and goal lines are highlighted as these will be affected by the state perturbations. It is possible to apply attacks designed for traditional RL systems [21] to each controller separately, i.e. applying different perturbations to $\mathbf{x}_t^L$ and $\mathbf{x}_t^H$, or to the model as a whole. The former does not have a 'real-world' threat model equivalent, but as described later in Section 3.4.2 will be useful for exploring manager-worker dependency. In general, I will be applying the same perturbation to both $\mathbf{x}_t^L$ and $\mathbf{x}_t^H$ unless stated otherwise.

3.4.1 Random Noise

Random noise is relatively simple to implement in PyTorch. Listing 3.1 shows how I implement the perturbation as described in Section 2.3.1.

Listing 3.1: Random Noise in PyTorch

```
noise = torch.FloatTensor(state.shape).uniform_(-epsilon, epsilon)
state = state + state_range * noise
state = torch.max(torch.min(state, state_max), state_min)
```

3.4.2 Testing Manager-Worker Dependency

I apply the 'attacks' to a hierarchical model as a whole to test for the various SSG factors. However, to explore the dependency that exists between the manager and worker in various setups, I separate $\mathbf{x}_t^H$ and $\mathbf{x}_t^L$ and apply *noise* to each individually. I test four possible ways of doing so, calling them *types*: (1) apply the same noise to $\mathbf{x}_t^H$ and $\mathbf{x}_t^L$; (2) apply different noise to $\mathbf{x}_t^H$ and $\mathbf{x}_t^L$; (3) apply noise to $\mathbf{x}_t^H$, leaving $\mathbf{x}_t^L$ untouched; (4) apply noise to $\mathbf{x}_t^L$, leaving $\mathbf{x}_t^H$ untouched.

3.4.3 Fast Gradient Sign Method

The high-level implementations of the two flavours of FGSM as described in Section 2.3.2 – targeted and non-targeted – are presented in Listings 3.2 and 3.3. The target was chosen to be an appropriately-sized vector of 0s: this, however, is a semantically meaningful goal and action. Specifically, a **0** goal indicates that the manager wishes for the worker to remain stationary, and a **0** action would result in the agent remaining stationary. This implies that the effect of the targeted approach may be more sensitive to environment characteristics than the non-targeted approach. The non-targeted implementation varies depending on the architecture used, so the simpler DQN version is presented.

Listing 3.2: Targeted FGSM in PyTorch

```
state = state.clone().detach().requires_grad_(True)
# approximate the action-value function
Qfunction = agent(state)
# calculate loss
loss = -F.mse_loss(Qfunction, target)
agent.zero_grad()
loss.backward()
# generate and apply perturbation to minimise loss to target
state = state + epsilon * state.grad.data.sign() * state_range
state = torch.max(torch.min(state, state_max), state_min)
```

Listing 3.3: Non-targeted FGSM in PyTorch

```
...
# inverting produced action-value function inverts the argmax result
loss = -F.mse_loss(Qfunction, -Qfunction)
...
```

When applying FGSM, be that the targeted or non-targeted variant, one needs to choose which hierarchy to generate perturbations from since FGSM is a white-box attack and perturbations are obtained by performing backward propagation. This means that an adversary would have to choose whether to generate perturbations from the manager or the worker. The generated perturbations would then be applied to $\mathbf{x}_t$ and propagated down to both the manager and worker.

3.5 Generalisation Experiments (Extension)

3.5.1 Scaling Experiments

The *scaling factor* of the evaluated environments was set to $s = 4$ for training, as described in Section 3.2. In this experiment, I varied s from 1 (faster movement speed) to 7 (slower movement speed) in increments of 1 at test time. These changes preserve the semantic meaning of the observations.

This experiment demonstrates model understanding of the environment. For example, the observed time t is useful to the agent as a way of keeping state, crucial in the *Point-Push* and *PointFall* with movable blocks and hence a stateful environment. Changing the movement speed could exploit the dependency on t in a model and cause it to fail. Furthermore, at smaller scales an action would cause the agent to 'overshoot' and end up further along than was seen during training time. If the agent has poor generalisation properties, then it would get confused in this unseen state.

This experiment has roots in nature: Wittlinger et al. [33] showed that extending or clipping an ant's legs would cause it to over or undershoot its nest even when released in a familiar location. This was due to the ants using an internal odometer. Similar ideas have recently been incorporated into RL generalisation experiments by Igl et al. [10].

3.5.2 Starting Range Experiments

All environments see the agent's starting position and orientation randomized to $\mathcal{U}(-d, d)$ and $\frac{\mathcal{U}(-d,d) \bmod 2\pi}{2\pi}$ where $d = 0.1$ during training. In this experiment, I increased d from 0.1 to 0.5 in increments of 0.05 at test time. This increased starting range would upset a model that overfit to the possible starting positions. This change fully preserves the semantic meaning of the observations, hence it would be reasonable to expect a trained model to cope with such changes.

3.6 Improving SSG Results (Extension)

As will be shown in Section 4.2, HIRO and TD3 perform similarly in the SSG experiments proposed above. This suggests that the currently-accepted way of structuring an HRL architecture does not lead to an improvement in SSG properties, contrary to my initial intuition.

In lieu of this, I proposed four novel HIRO-inspired architectures that aim to improve the SSG properties of the resulting agents. In each case the general HIRO structure remains unchanged, with most changes being to the manager. The implementation details and intuitions behind these architectures are described below.

3.6.1 DQN Manager

I proposed using a **DQN** [13] agent as the manager, instead of the continuous-control TD3 manager classically used in HIRO. Lillicrap et al. [15] cites the drawbacks of discretising action spaces as motivation for continuous RL control. However, the intended coarseness of the goal space means that these drawbacks are not as severe as would be expected. This approach can be thought of as a deep variant of Ecoffet et al. [34]. The goal space is discretised such that the environment is broken into a 10×10 grid, with the manager being able to set goals at the vertices of this grid giving a total of 121 (a $n \times n$ grid has $(n+1)^2$ vertices) possible goals.

3.6.2 Freezing the Worker

A fact that became apparent when going through the manager-worker dependency results, as will be described in Section 4.4, was that the dependency of the worker on the manager was highly variable between different trained instances of the same architecture and on the same environment. Moreover, in the simpler environment cases of PointMaze and PointFall, there was a distinct low dependency on the manager. This could be explained by the relative simplicity of the environments not requiring use of the hierarchy. However, this meant, at least intuitively, that some model capacity was being wasted.

With this in mind, I proposed a training regime that **freezes** the parameters of the worker when the average environment reward over the last 100 episodes exceeds 70, i.e. before completion of the environment. This number was chosen through experimentation. The intuition behind this was that freezing the worker forces the manager to increase its contribution to the remaining 20 average reward needed to solve the task. This, in turn, forces higher dependency on the manager.

3.6.3 Pheromone Trail

Keeping with the nature-inspired experiments, I proposed a *deterministic* manager that operates on a principle similar to trail **pheromones** used by ants. This intuition was confirmed by looking into research on explainability and deterministic planning. The manager keeps a priority list of length 5 (arbitrarily chosen) of state sequences $\mathbf{x}^i = \{\mathbf{x}^i_0, \mathbf{x}^i_1, \dots \mathbf{x}^i_T\}$ obtained from a single run of episode i, ordered by the cumulative reward R^i obtained during that episode. When selecting a goal, the manager iterates over its priority list and finds the state $\mathbf{x}^i_t$ closest to the currently-observed state $\mathbf{x}_t$ by means of L2-distance on the x, y co-ordinates. The goal is then taken to be:

$$\mathbf{g}_t = k \sum_{i=0}^{4} R^i \mathbf{x}^i_{t+c} \tag{3.1}$$

i.e. the weighted sum of states c steps away from the closest-matching one. k is a normalisation factor. c is defined in Nachum et al. [17] to be the number of steps between goal predictions. Similarly to the original HIRO paper, I use $c = 10$ throughout all of the experiments. This approach reinforces paths that have been previously more successful by guiding the worker onto these paths in a 'soft' manner.

3.6.4 Random Manager

Finally, I explore what would happen if, using the standard HIRO setup, the manager is replaced with a *deterministic* component outputting **random** goals. That is, every c steps, the manager would randomly sample $\mathbf{g}_t \in \mathcal{G}$.

The very fact that this approach trains successfully, and, as will be shown later on, performs well in terms of SSG properties, is surprising! I justify and explain this behaviour in Section 4.5.

3.7 Concerns about RL-based Approaches

Despite the numerous advantages and promising results, there are currently several key concerns associated with RL-based approaches, which were picked up throughout the project. These will be outlined in the remainder of this section.

3.7.1 Reproducibility

There have been concerns recently about the reproducibility of results in the RL community, highlighted by Henderson et al. [35]. This ties into sensitivity of learnt policies. Slight alterations in input features, embedding model architectures, time discretization, reward function, and random seeds can significantly impact the learnt policies [36].

3.7.2 Catastrophic Forgetting

The DQN [13] architecture is used extensively in this project, both independently and as part of HDQN [4]. It is known that DQN models often suffer from "catastrophic forgetting": a phenomenon whereby a model's performance quickly drops over a few training episodes and seldom recovers [37]. Kirkpatrick et al. [37] propose methods for overcoming this, but implementation of these techniques is deemed beyond the scope of this project. The solution used in this project is to simply detect when this 'forgetting' happens and restart the training.

I ran into this issue repeatedly when initially training the hierarchical models. By plotting losses, I was able to see that the loss changed at a suspiciously quick rate. This was later tracked to τ, the variable responsible for the "soft" target updates. This method of updating the target network was proposed by Lillicrap et al. [15]: rather than directly copying networks weights over from the source network every N time-steps, the "soft" update is performed as follows:

$$\theta_t \leftarrow \tau\theta_s + (1-\tau)\theta_t \tag{3.2}$$

where θ_t represent the target network's parameters, and θ_s represent the source network's.

Mnih et al. [38] introduced the idea of having a separate target network for Q-learning updates. This was done to improve stability, with the explanation that "Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_j, making divergence or oscillations much more unlikely". Updating the target network too frequently hence can cause training to become unstable.

The exact value of τ that works varies between environments and agents, but it was generally observed that hierarchical models in sparser-rewarded environments required a smaller τ (slower target network updates) to prevent training collapse. The downside to a smaller τ value is that time to convergence increases.

3.7.3 Non-Stationary Problem for Manager's Policy

As discussed in the *Implementation* section, Nachum et al. [17] highlights a non-stationary problem for the manager's policy, resulting from the lower-level policy changing throughout training. This is a problem unique to hierarchical models, and was solved in the paper and this project by means of *off-policy correction*. Some of the environments used were relatively simple so it often sufficed to just allow for a longer pre-training period for the worker. During the pre-training period the manager does not do any learning, resulting in a more stable low-level policy for the manager to build upon.

3.8 Repository Overview

The structure of the repository is as follows:

- `envs` contains the various *gym* environments used during evaluation. A number of the environments were adapted with minor alterations from the default environments that *gym* provides. These are: `cartpole`, `complex_cartpole`, `continuous_acrobot`, `continuous_cartpole`, `continuous_complex_cartpole`. The other environments were written by myself, using boilerplate code from *gym*: `dsdp`, `gridworld`, `point_fall`, `point_maze`, `point_maze_wo_time`, `point_push`.
- `notebooks` contains the bulk of the implementation code, split into 4 folders:
  - `continuous` contains initial implementations of HIRO, TD3 and DDPG agents, as well as initial experiments performed on them.
  - `discrete` contains initial implementations of HDQN and DQN agents, as well as initial experiments performed on them.
  - `final` contains two main collections of notebooks:
    - `actions_as_goals` contains final implementations of HDQN, HIRO, DQN, and TD3 agents tested on the modified-gym environments listed above. These results are presented briefly in Appendix B.3.
    - `states_as_goals` contains final implementations of HIRO and TD3 agents tested on the *Point* environments, with the SSG experiments described above.
  - `random` contains miscellaneous notebooks used for correctness-checking of environments.
- `plotting` contains notebooks for plotting result graphs, as well as the `.json` files of the experiment results.

Each notebook is more-or-less self-contained – each has the necessary code to run all of the described experiments. There is some duplicate code between the notebooks.

As mentioned in Sections 2.5 and 2.5.2, I use PyTorch [23] and OpenAI's *gym* [24] framework. Plotting was done using *matplotlib* [39]. Code for the RL and HRL agents was written using the respective original research papers and with reference to the original authors' implementations that were available on GitHub. The adversarial attacks were written with reference to Goodfellow et al. [19] and Zhao et al. [21].

Chapter 4

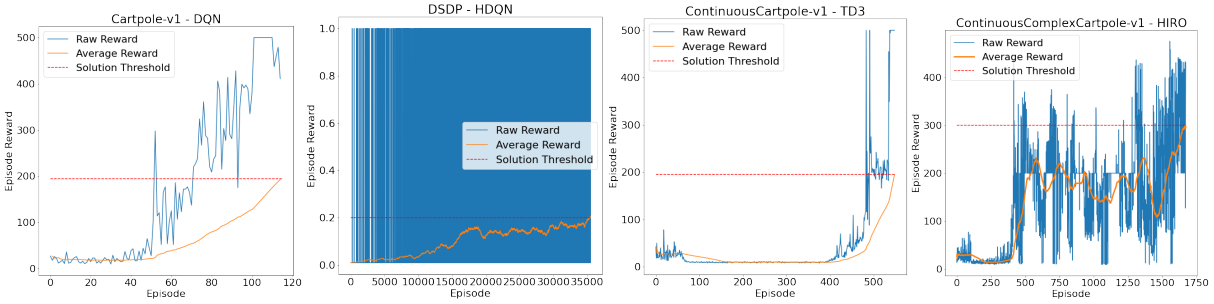
Evaluation

In this chapter I first demonstrate the correctness of implementation of the RL and HRL agents described in Section 3.3. In Sections 4.2 and 4.3 I present and describe the experimentally-obtained SSG properties of the various explored architectures. This is followed by a discussion of manager-worker dependency in Section 4.4. A general discussion of the proposed novel theory that concerns the factors that contribute to SSG performance of RL and HRL is presented in Section 4.5. The chapter concludes with Section 4.6: an evaluation of the performance of the novel architectures proposed in Section 3.6.

Evaluation of action-space goals (rather than state-space ones) can be found in Appendix B.3.

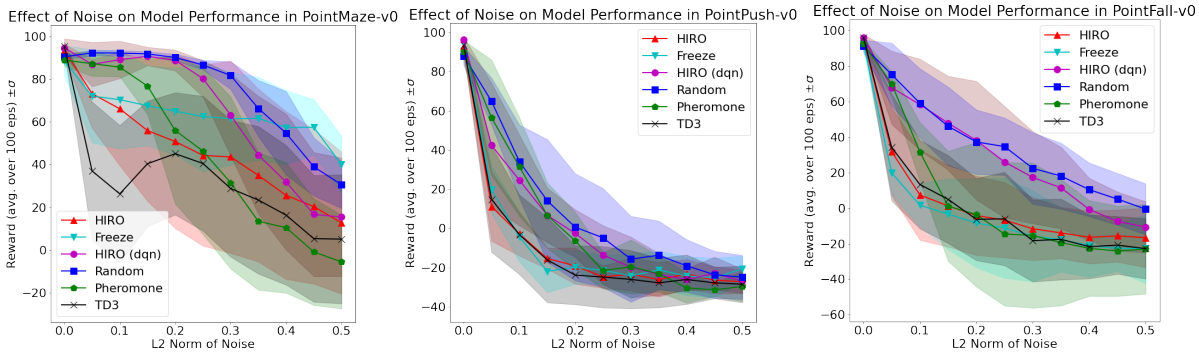
4.1 Agents

Figure 4.1 shows plots of raw (per-episode) and average (over 100 episodes [a, c, d]; over 1000 episodes [b]) episode rewards throughout training until a set solution threshold. This acts a proof-of-correctness of implementation as it shows that the agents are able to learn to solve tasks.



(a) DQN (b) HDQN (c) TD3 (d) HIRO

Figure 4.1: Proof-of-correctness of implementation of models by means of showing convergence on simple tasks. Environments are detailed in Appendix B.2.



(a) *PointMaze-v0* (b) *PointPush-v0* (c) *PointFall-v0*

Figure 4.2: Effect of random Gaussian noise.

4.2 Attacks

This section talks about the observed safety and security properties of the models being evaluated. Namely, HIRO and TD3, together with the four novel architectures being proposed: Freezing the worker, DQN manager, Random manager, and Pheromone manager.

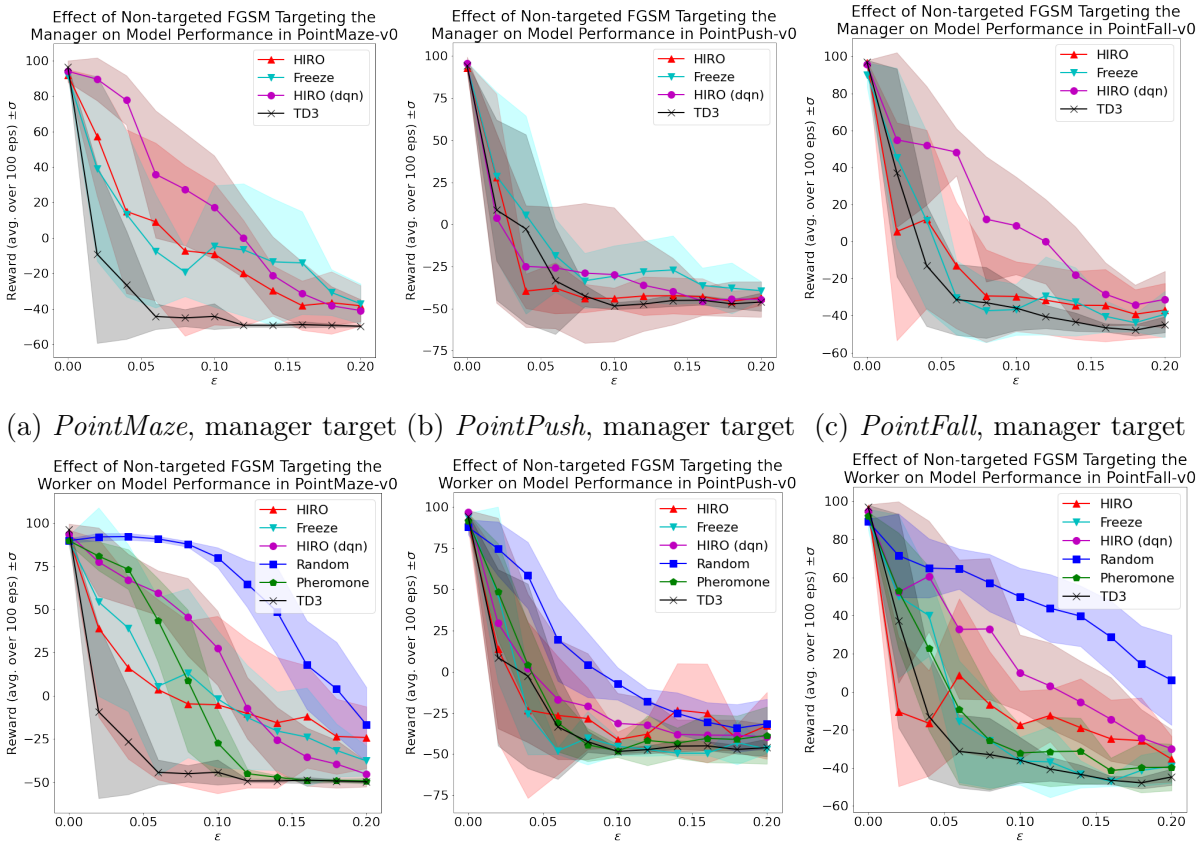
Each data point in the graphs is an average obtained over 100 episodes of a single model. The line itself shows the mean over five different trained instances of the same architecture, with the shaded area reflecting a single standard deviation in the results. That is, a 'tighter' shaded area implies more consistent behaviour across trained instances.

4.2.1 Safety

Figure 4.2 show the results of random noise on model performance. It is clear that the pattern of performance degradation is very environment-dependent. This is a welcome confirmation of the diversity of the environments being evaluated.

The next striking observation is that HIRO does not see any meaningful safety improvement compared to TD3. Indeed, across all three environments, it can be seen that the Random manager consistently out-performs the rest in terms of safety. It is also worth noting that the Pheromone manager seems to perform relatively well in lower-noise cases, but quickly decays to equivalent-or-worse behaviour compared to TD3. These surprising results are discussed later in Section 4.5. The reason I call these results 'surprising' is that the deep learning capacity of the Random and Pheromone architectures is equivalent to that of TD3: the manager is a deterministic algorithm, leaving the TD3 worker to be the only 'deep' component.

Freezing the worker seemed to do little to help with safety, aside from the results in PointMaze. However, PointMaze is also the simplest environment out of the three. The implication of this result is that an increased utilisation of the manager alone does *not* improve safety. In other words, the limiting factor in terms of safety perhaps lies in the interactions between hierarchies rather than the architecture set-up itself or the individual components.



(a) *PointMaze*, manager target (b) *PointPush*, manager target (c) *PointFall*, manager target

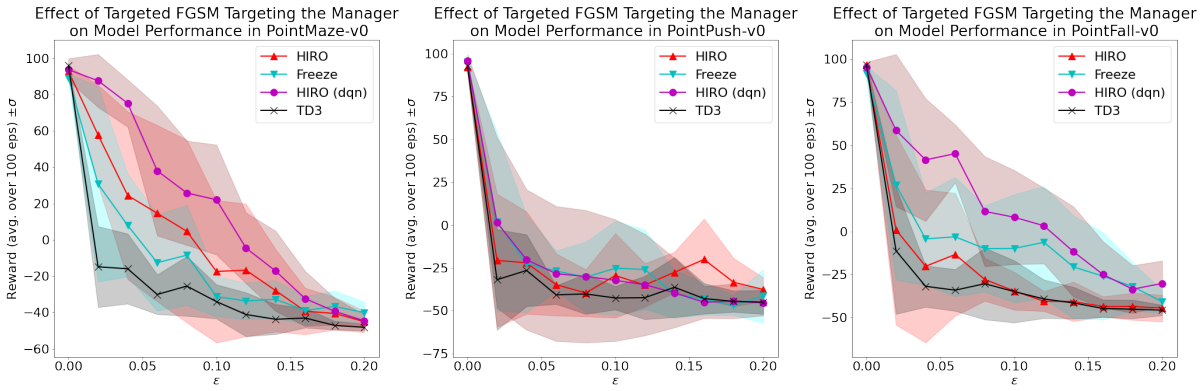
(d) *PointMaze*, worker target (e) *PointPush*, worker target (f) *PointFall*, worker target

Figure 4.3: Effect of non-targeted FGSM, targeting the either manager or worker. Note that Random and Pheromone results are not presented when targeting the manager due to their deterministic nature. TD3 is provided as a baseline.

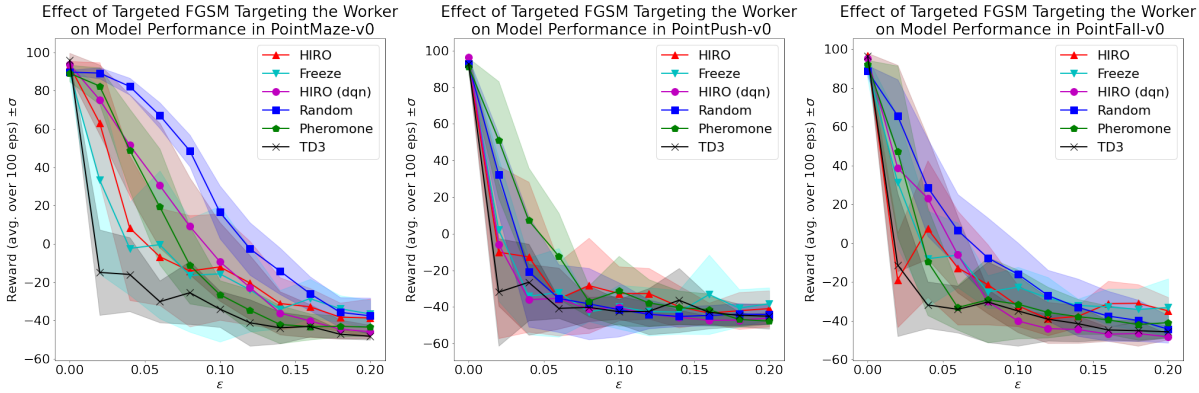
One can also see the DQN manager performing quite nicely. This suggests that my intuition about the benefits of discretising the goal space were justified. Specifically, the intuition was that due to the manager acting on coarser levels of abstraction, the advantages of quantising the state space outweigh disadvantages such as loss of structure of the action domain [15]. This finding also makes a case for 'deep' managers, as at first glance it seemed that the deterministic ones were performing better. Exploring this further with different deep managers is a direction for future work.

4.2.2 Security

Figures 4.3 and 4.4 show the results of non-targeted and targeted FGSM respectively, applied to my assortment of models. It is also worth pointing out that Figures 4.3a to 4.3c and Figures 4.4a to 4.4c generate perturbations by targeting the manager; while Figures 4.3d to 4.3f and Figures 4.4d to 4.4f generate them by targeting the worker. It hence follows that the deterministic managers – Random and Pheromone – are inherently not susceptible to this former attack, and are therefore excluded from the respective subplots.



(a) *PointMaze*, manager target (b) *PointPush*, manager target (c) *PointFall*, manager target



(d) *PointMaze*, worker target (e) *PointPush*, worker target (f) *PointFall*, worker target

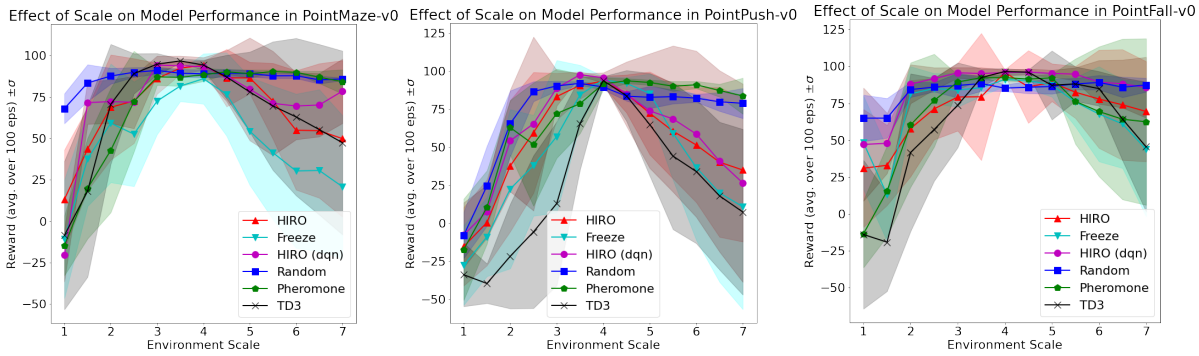
Figure 4.4: Effect of targeted FGSM, targeting the either manager or worker.

It is evident that HIRO exhibits better security properties than TD3. The fall-off when targeting either the manager or worker is similar for the architectures tested, implying that there is equal weighting on the decision-making processes of the two hierarchies. It is also clear that the Random manager performs exceptionally well in most scenarios, with the Pheromone and DQN managers being a close second, especially on lower perturbation sizes. It is worth noting that at higher perturbation sizes, the Pheromone manager converges with the TD3 results. This is not all that surprising as the deterministic managers just utilise the deep model capacity of the underlying TD3 agent.

Similarly to the safety results, freezing the worker does not improve security compared to the standard HIRO setup. This again suggests that solely increasing utilisation of the manager through better learnt policies does not lead to any tangible security benefits.

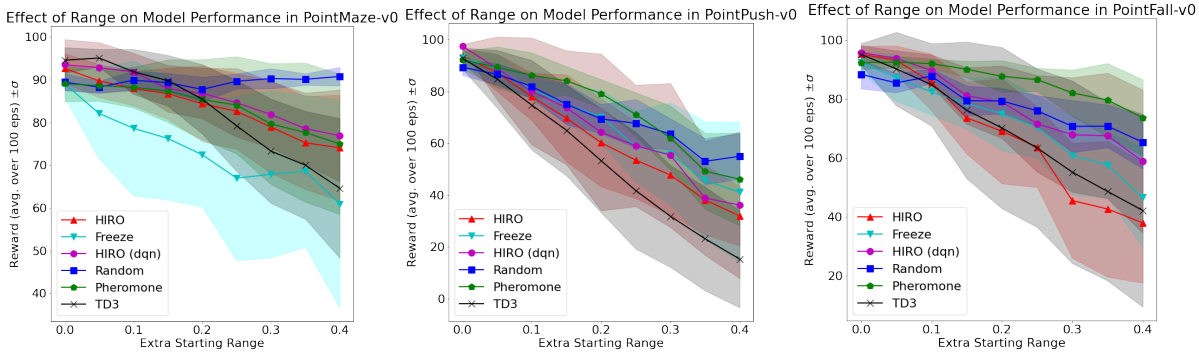
Manager-wise, the DQN manager has the smallest performance drops for a given perturbation size. This resilience is impressive and supports my intuition that discrete action spaces are more suitable for the coarser decision-making that the manager does.

Three architectures consistently outperform the standard HIRO setup at lower perturbation sizes: Random, Pheromone, and DQN managers. At larger perturbations, the performance is comparable to the others albeit with smaller standard deviation bounds, suggesting that the approaches lead to less variability in the resulting trained models.



(a) *PointMaze-v0* (b) *PointPush-v0* (c) *PointFall-v0*

Figure 4.5: Effect of scale on model performance. Note that the agents were trained on $s = 4$, the middle of the shown x-axis.



(a) *PointMaze-v0* (b) *PointPush-v0* (c) *PointFall-v0*

Figure 4.6: Effect of increased starting range on model performance.

4.3 Generalisation (Extension)

Figures 4.5 and 4.6 show the results from the generalisation experiments described in Section 3.5. In my opinion, these are the most interesting results obtained throughout this project for reasons that will become apparent.

Note, that larger scales (i.e. $s > 4$) result in an environment that is inherently a subset of the one being trained on. The agent is able to take smaller steps than the scale implies: as mentioned in Section 3.2, the *maximum* step size is $\frac{1}{s}$. It then follows that larger scales *decrease* the maximum step size, but the agent was able to take steps of this size in the trained environment already. Hence, a model with any sort of generalisation abilities should be able to cope with larger scales.

Smaller scales ($s < 4$) on the other hand, see the agent 'over-step' for a given input which can lead to irreversible environment changes, such as pushing a box too far, that the agent could not have anticipated during training. These smaller scales are outside of the training regime, so any sort of diminished performance fall-off is laudable.

Focusing on the scale experiments first, one can see that the two deterministic managers – Random and Pheromone – perform exceptionally well on higher scales. Moreover, the Random manager seems to consistently outperform the other models. Similarly to the safety and security experiments, HIRO usually generalises marginally better than TD3. Using a DQN manager further widens this performance gap.

Freezing the worker, on the other hand, has variable impact on generalisation much like in the other experiments: improving on HIRO's results marginally at times and performing poorly at other times. It is difficult to say which level of hierarchy is responsible for the overall agent's SSG performance. As mentioned earlier and highlighted again now, purely increasing the utilisation of one component does not improve SSG metrics. Perhaps the key lies in the interaction between the hierarchies and finding a way to better facilitate that; the implications of this are discussed in Section 4.5.

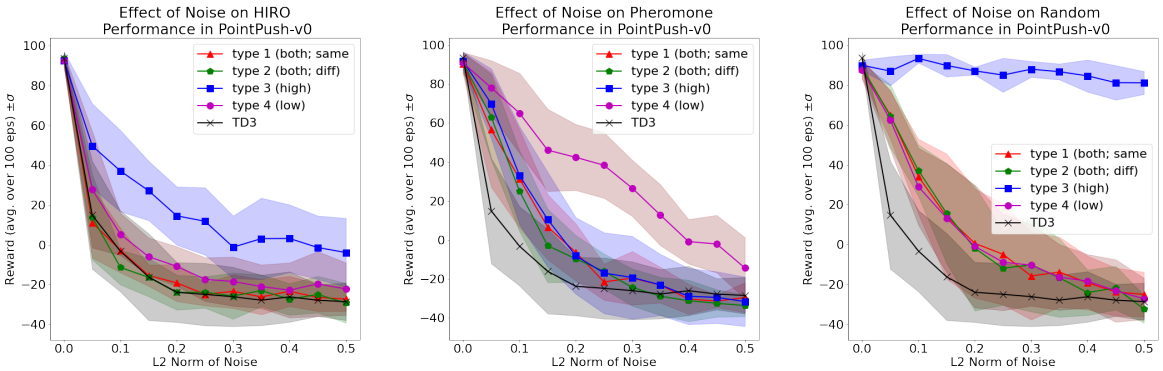
In the range experiment, it is still evident that TD3 has worse or equivalent performance to HIRO. Most of the results are similar to that of the scaling experiment: an encouraging result that backs up the intuition that both of these experiments test generalisability in as pure of a form as possible. The deterministic managers continue to outperform or match performance of the deep managers.

It is worth emphasising that in the case of Random and Pheromone, I train the models on $s = 4$ with the resulting models performing equally well on $s = 7$. This is an important result as training on larger scales takes significantly more time. Training generalisable agents on smaller environments can be used to save time and computational costs.

4.4 Manager-Worker Dependency (Extension)

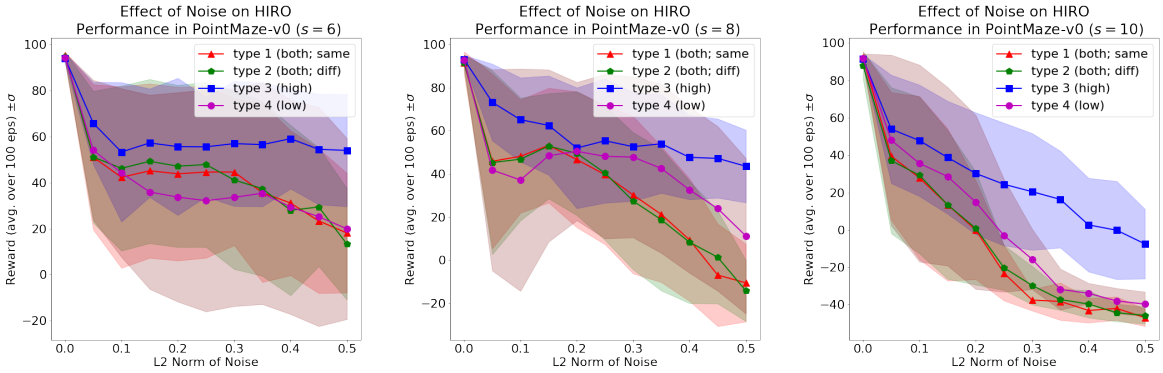
Figures 4.7 and 4.8 present partial results of the exploration I conducted into manager-worker dependency. The full results are presented in the Appendix as Figures A.1 to A.3. I found no better way of testing how much the worker 'listens' to the manager than applying noise to isolated $\mathbf{x}^L$ or $\mathbf{x}^H$ states via the 'breakdown' method described in Section 3.4.2, and seeing the respective performance degradation. As mentioned in Section 3.3.2, goals often merely encourage the worker to make certain decisions: it is not necessary, nor is it often desirable to 'clear' a goal by reaching the indicated goal state $\mathbf{g}$.

Figure 4.7 presents this breakdown for HIRO, Pheromone manager, and Random manager on PointPush-v0. Type 3 and 4 are the most telling lines: the larger the fall of one relative to the other, the more dependency exists on that hierarchy. For example, if model performance falls for type 3 (the blue line, applying noise to just the manager), but less so for type 4 (the magenta line, applying noise to just the worker) as is the case for Figure 4.7b, then there is a *higher* dependency on the manager. On the other hand, Figure 4.7c shows the other extreme, where there is virtually no dependency on the manager. Figure 4.7a presents a middle case, where both the worker and manager play some role in decision-making, as indicated by steep fall-offs of both type 3 and 4 at high noise levels.



(a) HIRO (b) Pheromone (c) Random

Figure 4.7: Demonstrating manager-worker dependency of different architectures in PointPush-v0 by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately. Detailed in Section 3.4.2.



(a) $s = 6$ (b) $s = 8$ (c) $s = 10$

Figure 4.8: Illustration of how *training* a HIRO agent on the PointMaze-v0 environment with different scales affects the dependency of the worker on the manager. We see that the larger scales, which pose more of a challenge, have a greater dependency on the manager.

In Figure 4.8 the effect of varying environment scales at *train* time on manager-worker dependency is presented. Larger scales make the environments harder to solve, keeping the maximum number of steps fixed. It is clear that there is an increased dependency between the manager and worker at higher scales, suggesting that an HRL system achieves some equilibrium of how much the worker 'listens' to the manager based on the difficulty of the task at hand. The manager's decision-making capacity is only utilised when it is needed, potentially 'wasting' the deep parameters' capacity.

These results, while hard to interpret in isolation, gave a critical insight into the ways I could increase this dependency, and hence make better use of the model capacity. I consider high dependency on both the manager and worker (i.e. type 3 and 4 should show similar behaviour) to be an indicator of 'better use'. Through the insight gained from Figure 4.8, I managed to come up with ways to control this dependency, which manifests itself in the existence of the two architecture extremes presented in Figure 4.7.

4.5 Discussion (Extension)

This section will focus on bringing together all of the results presented above and producing a coherent takeaway message.

4.5.1 Manager-Worker Dependency and Optimisation

I explain the peculiar observations of the manager-worker dependency through the semantic meaning of the intrinsic and extrinsic rewards. I believe the the key part of HRL is the information hiding inherent to the architecture and implicit message passing between the manager and worker through the intrinsic reward.

When using state-space goals (as is common practice at the moment) the semantics of the intrinsic and extrinsic rewards often match when the goal of the environment is to solve it in as little steps as possible: the worker receives a semi-constant reward per timestep and implicitly aims to minimise the number of steps until completion. I speculate that this gives rise to a very flexible situation from the perspective of the worker. It is given the freedom to optimise for two factors:

- reduce L2 distance to the set goals by maximising the intrinsic reward, i.e. listening to the manager. Semantically meaningful goals would encourage such behaviour.
- simply to reduce the number of steps until completion to implicitly maximise cumulative extrinsic reward. This is specific to an environment type where the aim is to solve the environment as quickly as possible; hence it suffices to just minimise the number of steps rather than maximising the intrinsic reward as such. 'Bad' managers that give 'random' (from the perspective of the worker) goals would intuitively push the worker towards optimising for this factor.

I claim that the factor that ends up being optimised for is primarily dependent on the learnt manager policy μ^H. I tested this proposal through the introduction of the novel managers presented in Section 3.6 that aim to control this behaviour of μ^H. Indeed, the two extremes presented were *Random* – fully adhering to the assumed 'bad' manager definition – and *Pheromone*, which provides exclusively meaningful goals. As shown in Figure 4.7, which compares the manager-worker dependency of HIRO to these two extremes, the dependency on the manager corroborates the above proposal.

I find evidence via the generalisation results of the Pheromone and Random manager that the closer a model adheres to either extreme, the better the generalisation properties of the trained models. Perhaps, this is because the agent is less 'confused' about what it is meant to optimise and hence the focus of it lies in environment understanding, rather than trying to make sense of the fed-in state and rewards.

4.5.2 Exploration Benefits of a Manager

The surprising fact that having a random manager leads to an agent that not only manages to solve the environment, but is also robust and able-to-generalise suggests a different way of viewing RL agent exploration and training. The vast majority of RL architectures have a temperature-controlled *action* exploration factor that decays over time [15, 13]. This bears an implicit assumption that extensively exploring the action space leads to 'good enough' state exploration. This does not always hold, especially for environments with a large state space and relatively small action space.

Having a manager with extrinsic reward hiding forces *explicit* state exploration. A *random* manager then results in exhaustive state exploration, mimicking the effect of the action exploration factor but in a more semantically meaningful way. Intuitively, this random manager causes the agent to run around the state space gaining a deep semantic understanding of it, before converging on the task solution. This notion was explored in a slightly different light by Colas et al. [40], but with the same conclusions.

4.5.3 Blurring the Line between RL and HRL

The main takeaway of these experiments also lies in the fact that deterministic managers seem to perform better than deep ones, especially for the resources involved in training and running the models. This could be a consequence of fairly simple environments – exploring more complex environments would be a good direction for future work. This takeaway also has implications for simplifying both the model structure and training. More importantly, these discoveries blur the line between RL and HRL.

This above results of '*blurring the line*' is perhaps a product of the generic nature of HRL. As a tool, it is perhaps overly powerful in the sense that the intrinsic reward mechanism acts as both a training mechanism akin to reward engineering, and as a control mechanism during inference time. This results in a very delicate game that a deep manager has to play. Combined with the changing lower-level policy (discussed in Section 3.7.3) that the manager has to influence, this makes training a deep manager difficult.

Another way of looking at the manager is as a learnt intrinsic reward function. The difficulty lies in learning this function while the 'executor' of actions, the worker, is also learning. A kind of meta-learning, perhaps. Using a continuous-action agent such as TD3 as the manager means that the intrinsic function search space is large. Quantising the search space and using a discrete-action agent such as DQN as the manager results in a much smaller search space, allowing convergence on an 'optimal' solution quicker.

Keeping with this analogy, the Pheromone deterministic manager learns this intrinsic reward function from the environment itself. The Random manager intuitively 'bootstraps' the uniform extrinsic reward (-0.1 per timestep) into a more-or-less uniform intrinsic reward that incorporates semantic 'distance' (in the environment sense) meaning into it while keeping the uniform aspect of the extrinsic reward. To some regard, this can be considered domain-knowledge injection and its usefulness is likely limited to environments where the aim is to solve them as quickly as possible.

4.5.4 Impact on Training

The ability of deterministic managers to train models that generalise to larger scales is also an important result for the speed and efficiency of training. Attempting to train a model at $s = 7$ requires significantly more time than at $s = 4$. Using the Random and Pheromone managers, it is possible to reduce the training time of a model designed to act on higher scales by training it first on smaller scales and then, if needed, simply fine-tuning it on the intended scale.

I have demonstrated that simpler, deterministic managers lead to significantly more robust and generalisable models in the context of the evaluated environments. This can potentially pave a new way for 'robustifying' RL with a HRL-inspired schemes.

4.6 Novel Architectures (Extension)

To summarise the effectiveness of the proposed architectures:

DQN Manager A favourable alternative to the standard setup of HIRO, that is faster to train due to swapping the actor-critic structure of TD3 for DQN. In almost all cases, SSG properties of this architecture were equivalent or better than that of HIRO.

Freezing the Worker Initially showed promising safety results, but had underwhelming robustness and generalisation properties. If nothing else, it was a worthwhile experiment to demonstrate that increased manager-worker dependency does not automatically translate into SSG properties, and that each of the three SSG factors are relatively independent.

Pheromone Trail Shows strong generalisation performance, often matching or slightly under-performing in terms of safety and robustness compared to HIRO. That said, this is a deterministic manager model, so the improvements compared to a standard TD3 model (which the worker still is) are impressive. If developed further, this approach could provide a solid *explainable* way of improving RL model training.

Random Manager The most surprising result of all, in my opinion. While relatively slow to train, this deterministic-manager architecture managed to score highly in all SSG experiments. This is most likely due to the extreme levels of exploration that the worker is forced to endure under this model. As mentioned in Section 4.5, this likely results in the model gaining a deep semantic understanding of the environment.

Future directions which would be worthwhile to explore are discussed in Section 5.2.

Chapter 5

Conclusions

This project demonstrated for the first time in a formal setting the safety, security and generalisation properties of HRL, and compared them with RL. This was achieved by proposing and implementing various experiments that test SSG properties. It was found that HRL marginally outperforms RL in most SSG experiments. The generalisation experiments are novel and give a nice insight into the tested properties. Evaluation was performed on three novel environments, created for the sake of more accessible HRL research. These environments have been open-sourced and released publicly.

This project also put forward four novel architectures inspired by HIRO [17]: HIRO with a DQN manager, freezing HIRO's worker, a nature-inspired deterministic trail pheromone manager, and a random manager. The SSG properties of these new models were surprising in most regards, with some consistently outperforming HIRO and TD3. Novel theory to explain the observed SSG properties of both the existing and novel architectures was put forward and partially verified with the conducted experiments.

The novel idea of actions-as-goals was explored at the beginning of the project. Many existing environments were modified and altered to suit the experiments I was conducting.

5.1 Lessons Learnt

The main personal take-away from this project is that RL can be difficult to work with, requiring a lot of tweaking. A lot of time was put into this project, both in terms of learning about RL and HRL, and in terms of experimentation. Training a single model instance took up to 12 GPU hours in some cases.

If I were to redo the project, I would start by producing my own environments that fit the experiments I would like to perform, rather than trying to modify existing ones to suit my needs. I was also alerted mid-project to the lack of well-defined HRL vs RL benchmarks in SSG settings. A lot of time was also spent initially experimenting with actions-as-goals: while novel and interesting in itself, it is rather tangential to mainstream HRL research and as so is less relevant.

5.2 Future Work

In my opinion, deterministic managers deserve more attention and exploration. They provide a simplified training regime, while maintaining explainability to some degree. There is increasing interest in explainable AI, especially in the fields of RL and Multi-Agent RL [41]. This interest comes from both a research angle and a commercial one, leading to a large number of publications on this topic in recent years [41, 42, 43]. However, there is a distinct lack of approaches to explainable HRL.

Concretely, I believe that some combination of the Pheromone and Random managers could lead to fruitful results. The Pheromone manager provides an explainable set of goals, based on past environment solutions; while the Random manager ensures high levels of state exploration as discussed in Section 4.5. I have not implemented this due to time constraints, but the basic theory would be as follows. Given the previous definitions found in Section 3.6.3, set the goals to be:

$$\mathbf{g}_t = (1 - \epsilon_t) \left(k \sum_{i=0}^{4} R^i \mathbf{x}^i_{t+c} \right) + \epsilon_t \mathcal{U}(\mathbf{x}_{\min}, \mathbf{x}_{\max}) \tag{5.1}$$

where ϵ_t is a 'temperature' factor controlling state *exploration* via random state-space goals versus *exploitation* of the pheromone trail. ϵ_t would decay with some form of scheduling from 1.0 to 0.0 over time. It is also important to clamp the goal appropriately to ensure $\mathbf{g}_t \in \mathcal{G}$.

The intuition behind this combination would be to ensure a high level of exploration reminiscent of the Random manager at the start of training, and then converge to the fast-training and explainable pheromone trail approach.

Other future work could include exploring more complex environments, especially ones with vastly different extrinsic reward schemes.

Bibliography

[1] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[3] Andrew Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications*, 13, 12 2002. doi: 10.1023/A:1025696116075.

[4] Tejas D. Kulkarni, Karthik R. Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation, 2016.

[5] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2014.

[6] Yannis Flet-Berliac. The promise of hierarchical reinforcement learning. *The Gradient*, 2019.

[7] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In *Proceedings of the 29th International Conference on International Conference on Machine Learning*, ICML'12, page 1467–1474, Madison, WI, USA, 2012. Omnipress. ISBN 9781450312851.

[8] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies, 2017.

[9] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

[10] Maximilian Igl, Kamil Ciosek, Yingzhen Li, Sebastian Tschiatschek, Cheng Zhang, Sam Devlin, and Katja Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. *arXiv preprint arXiv:1910.12911*, 2019.

[11] C. Ye, A. Khalifa, P. Bontrager, and J. Togelius. Rotation, translation, and cropping for zero-shot generalization. In *2020 IEEE Conference on Games (CoG)*, pages 57–64, 2020. doi: 10.1109/CoG47356.2020.9231907.

[12] Huan Xu and Shie Mannor. Robustness and generalization, 2010.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[14] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8 (3):279–292, May 1992. ISSN 1573-0565. doi: 10.1007/BF00992698. URL `https://doi.org/10.1007/BF00992698`.

[15] T. Lillicrap, J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2016.

[16] Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight, 2019.

[17] Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning, 2018.

[18] Carlos Florensa, David Held, Xinyang Geng, and Pieter Abbeel. Automatic goal generation for reinforcement learning agents, 2018.

[19] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[20] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples, 2018.

[21] Yiren Zhao, Ilia Shumailov, Han Cui, Xitong Gao, Robert Mullins, and Ross Anderson. Blackbox attacks on reinforcement learning agents using approximated temporal information, 2019.

[22] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2 edition, 2008. ISBN 9780470068526.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In

H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[24] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[25] Google Colaboratory. URL `https://colab.research.google.com/`.

[26] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[27] Yen-Chen Lin, Zhang-Wei Hong, Yuan-Hong Liao, Meng-Li Shih, Ming-Yu Liu, and Min Sun. Tactics of adversarial attack on deep reinforcement learning agents, 2019.

[28] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[29] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[30] AnonReviewer2. Concerns about accessibility for underrepresented groups. `https://openreview.net/forum?id=px0-N3_KjA&noteId=_Sn87qXh3el`, 2020.

[31] Shixiang Gu, Timothy Lillicrap, Zoubin Ghahramani, Richard E. Turner, and Sergey Levine. Q-prop: Sample-efficient policy gradient with an off-policy critic, 2017.

[32] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning, 2017.

[33] Matthias Wittlinger, Rüdiger Wehner, and Harald Wolf. The desert ant odometer: a stride integrator that accounts for stride length and walking speed. *Journal of Experimental Biology*, 210(2):198–207, 2007. ISSN 0022-0949. doi: 10.1242/jeb.02657. URL `https://jeb.biologists.org/content/210/2/198`.

[34] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021. ISSN 0028-0836. doi: 10.1038/s41586-020-03157-9.

[35] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters, 2019.

[36] MingYu Lu, Zachary Shahn, Daby Sow, Finale Doshi-Velez, and Li wei H. Lehman. Is deep reinforcement learning ready for practical applications in healthcare? a sensitivity analysis of duel-ddqn for hemodynamic management in sepsis patients, 2020.

[37] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks, 2017.

[38] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015. doi: 10.1038/nature14236.

[39] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[40] Cédric Colas, Pierre Fournier, Olivier Sigaud, Mohamed Chetouani, and Pierre-Yves Oudeyer. Curious: Intrinsically motivated modular multi-goal reinforcement learning, 2019.

[41] Erika Puiutta and Eric M. S. P. Veith. Explainable reinforcement learning: A survey. In Andreas Holzinger, Peter Kieseberg, A Min Tjoa, and Edgar Weippl, editors, *Machine Learning and Knowledge Extraction*, pages 77–95, Cham, 2020. Springer International Publishing. ISBN 978-3-030-57321-8.

[42] Prashan Madumal, Tim Miller, Liz Sonenberg, and Frank Vetere. Explainable reinforcement learning through a causal lens. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2493–2500, Apr. 2020. doi: 10.1609/aaai.v34i03.5631. URL `https://ojs.aaai.org/index.php/AAAI/article/view/5631`.

[43] Dmitry Kazhdan, Zohreh Shams, and Pietro Liò. Marleme: A multi-agent reinforcement learning model extraction library, 2020.

[44] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

[45] A. W. Moore. Efficient memory-based learning for robot control. 1990.

[46] Alborz Geramifard, Christoph Dann, Robert H. Klein, William Dabney, and Jonathan P. How. Rlpy: A value-function-based reinforcement learning framework for education and research. *Journal of Machine Learning Research*, 16(46):1573–1578, 2015. URL `http://jmlr.org/papers/v16/geramifard15a.html`.

[47] Jacob Rafati and David C. Noelle. Learning representations in model-free hierarchical reinforcement learning, 2019.

[48] Juarez Monteiro, Nathan Gavenski, Roger Granada, Felipe Meneguzzi, and Rodrigo Barros. Augmented behavioral cloning from observation, 2020.

Appendix A

Full Results

This appendix is dedicated to the remainder of the experiment results, which did not fit in the main project, either due to space or position constraints.

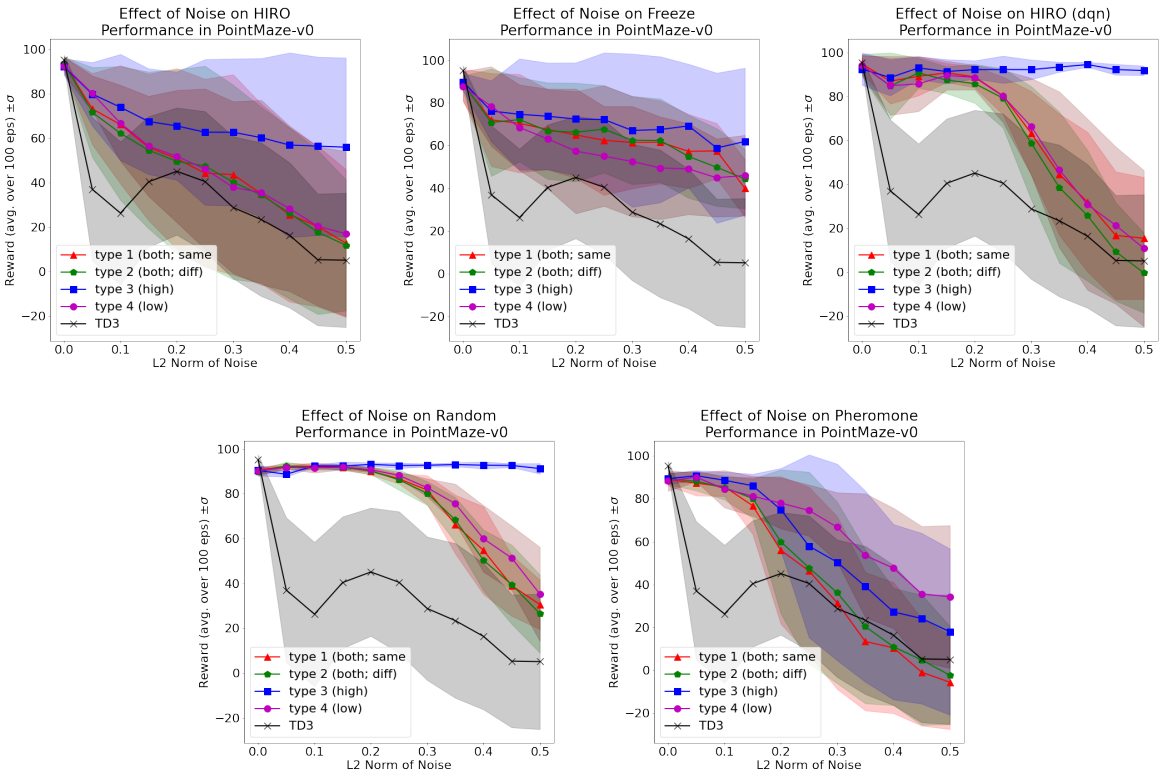


Figure A.1: Demonstrating manager-worker dependency of different architectures in PointMaze-v0 by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately.

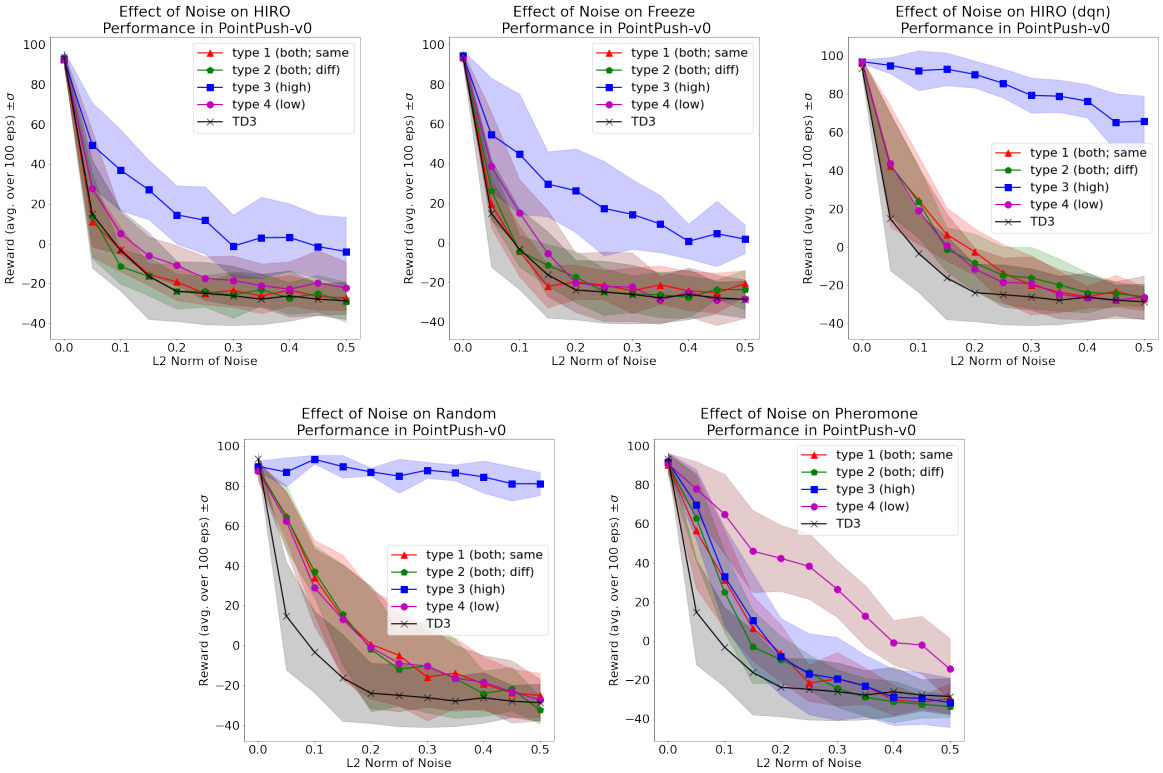


Figure A.2: Demonstrating manager-worker dependency of different architectures in PointPush-v0 by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately.

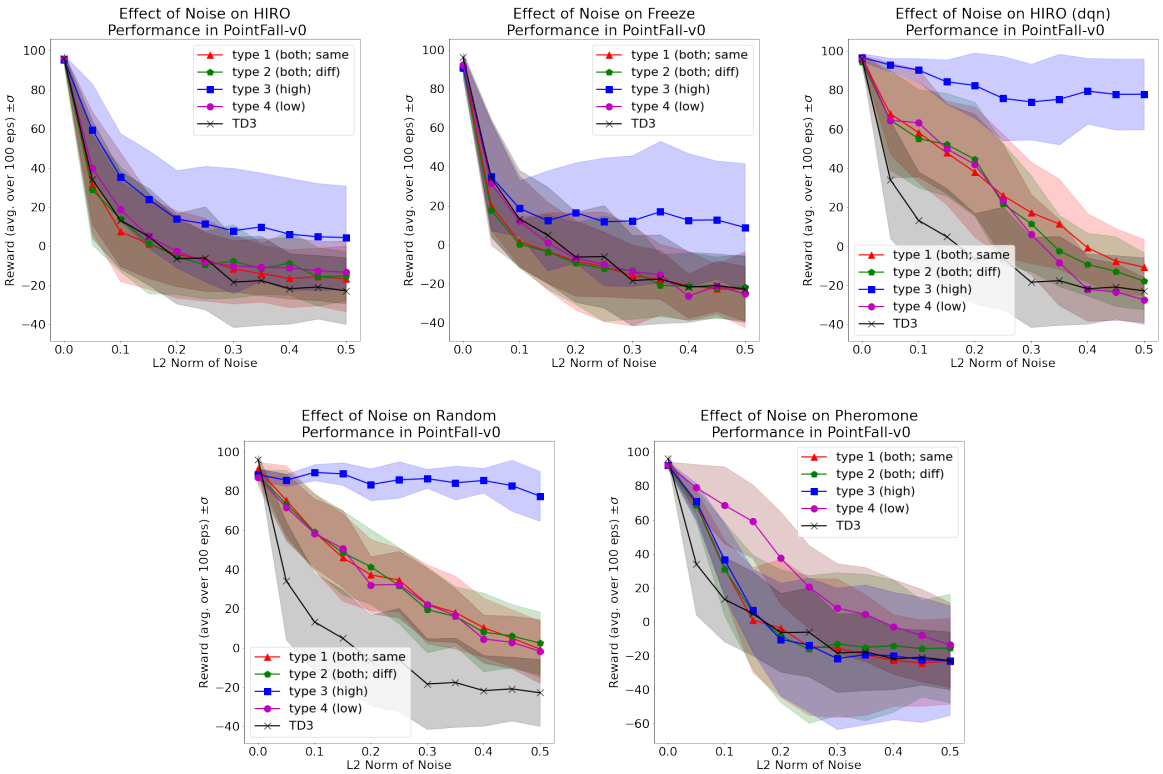


Figure A.3: Demonstrating manager-worker dependency of different architectures in PointFall-v0 by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately.

Appendix B

Actions as Goals (Extension)

B.1 Overview

In the initial exploration phase of the project, I used actions as goals (i.e. $\mathcal{G} \subseteq \mathcal{A}$). This was primarily due to a lack of environments that had both numeric states and were complex enough to justify using state-space goals. There were a few more complex environments provided by *gym* where the state observations would be images, such as the Atari environments. However, this would require using a convolutional layer in the models and would slow down training considerably.

While HIRO provided an intrinsic reward based on L2 distance from the observed state and the goal state, there was no precedence set for intrinsic rewards when using actions as goals. As such, in a discrete case, the intrinsic reward was set to be 1 if the agent performed the goal action, and was 0 otherwise. This meant that workers were not explicitly penalised for disobeyed the manager, which, at least in theory, should lead to more opportunities for exploration.

In a continuous action space, two main approaches were trialled. Either the intrinsic reward was set to be 1 if the *sign* of the action matched the goal's sign, and 0 otherwise; or the intrinsic reward was set to be the negative L2 distance between the goal and the worker's action.

Furthermore, unlike in HIRO, where the goal is set every c ($c = 10$) steps, this setup required a distinct indicator of when the goal was cleared. In a discrete case, the goal was re-sampled (and hence cleared) when the action matched the goal. In a continuous case, I set an environment-specific L2 threshold between the goal and action to determine when a goal was cleared and could be re-sampled.

The architectures evaluated are HIRO [17], TD3 [26], HDQN [4], and DQN [13].

B.2 Additional Environments

Below is a list of all additional environments that were used for testing actions-as-goals and initial investigation. All environments below have a discrete action space, unless their name prefixed or suffixed with *Continuous*. In cases where an environment was implemented from scratch, an *(I)* is suffixed:

1. `CartPole-v1` [44] was used extensively. `ContinuousCartPole-v1` was created to use a continuous action space. This was to enable TD3 and HIRO to solve the environment, so that their performance could be compared to that of DQN and HDQN.

2. `MountainCar-v0` [45] and the equivalent continuous action space version `MountainCarContinuous-v0`.

3. `Acrobot-v1` [46]. `ContinuousAcrobot-v1` was created to use a continuous action space.

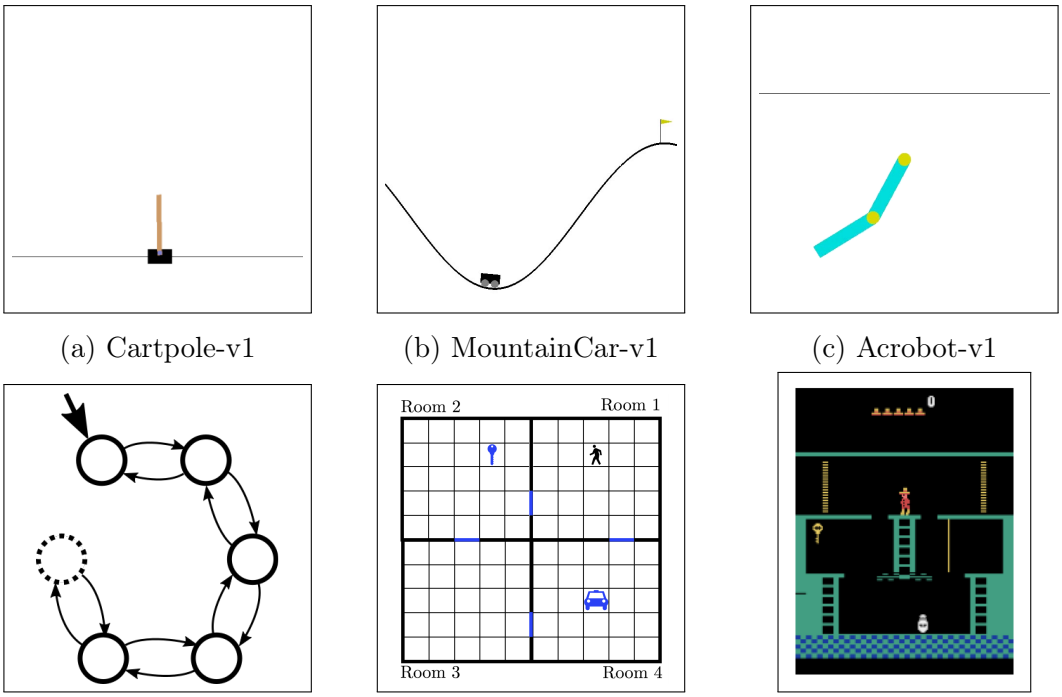
4. Six-state discrete stochastic decision process (DSDP) as described in Kulkarni et al. [4]. This was used as a proof-of-correctness of implementation of HDQN. *(I)*

5. 4-room environment with a key and a lock, inspired by [47]. This was a lead-up to the *Point* environments. The authors provide a GitHub repository (`https://github.com/root-master/unified-hrl`) for their code, but it was found that there were a number of inconsistencies between the parameters and settings that were described in the paper and the presented code. My implementation of the room environment allows for scaling of the room size and the number of rooms. This allowed for experiments with smaller or different room configurations. The grid-like world allowed HDQN to predict states without resorting to discretising the state space. However, the discrete state observations made noise and FGSM experiments difficult and unrealistic. *(I)*

6. MontezumaRevenge-v0, an Atari game, was used in Kulkarni et al. [4]. I initially attempted to recreate the paper's work for this and completed all the necessary implementation work for it, including the *entity space* system mentioned in Section 3.3.2. However, the incredibly long training time required (GPU weeks) meant that it was unrealistic for me to pursue this path further.

The environment that allowed a transition into the more standard states as goals (i.e. $\mathcal{G} \subseteq \mathcal{S}$) is **Complex-Cartpole**: a modified version of the original Cartpole environment [44], altered to have a continuous action space and designed to be more suited to HRL agents. I accomplished this by modifying the reward to be proportional to the distance away from the starting position.

The reward per timestep is defined to be $1 + \lfloor 4\min(1, |\frac{\mathbf{x}_t}{2.4}|) \rfloor$, where $\mathbf{x}_t$ is the cart's position along the x-axis and 2.4 is the threshold value for $\mathbf{x}_t$. Hence, the reward r_t has range $1 \leq r_t < 5$. The maximum episode length is retained at 200, but *ComplexCartpole-v1* is considered solved with an average reward of 300 or more over the last 100 episodes. This



(a) Cartpole-v1 (b) MountainCar-v1 (c) Acrobot-v1

(d) DSDP [4] (e) 4-room environment with a key and a lock [47] (f) MontezumaRevenge-v0

Figure B.1: Visual representations of the additional environments used.

makes it impossible to solve by staying central and requires extensive exploration of the state space.

All models were trained to convergence before evaluation. The 'solution threshold' is described in terms of average episode reward over a given window:

1. `CartPole-v1` defines success as achieving an average reward of at least 195 over the last 100 episodes.

2. `MountainCar-v0` defines success as achieving an average reward of at least -110 over the last 100 episodes. `MountainCarContinuous-v0` has a slightly different definition of success, requiring an average of at least 90. This is due to the difference in giving out rewards: -1 per timestep until reaching the goal, in the discrete case; and $-||action||$ per timestep and 100 for reaching the goal, in the continuous case. The rewards were normalised to be comparable.

3. `Acrobot-v1` is an unsolved environment, meaning that there is no specific reward threshold at which it is considered solved [48]. I set the threshold to be an average reward of at least -90 over the last 100 episodes. This number was obtained through experimentation.

4. For `DSDP` I set the threshold to be 0.2 over the last 1000 episodes. This was derived by performing experiments with 'ideal' players.

A visual representation of all mentioned environments is presented in Figure B.1.

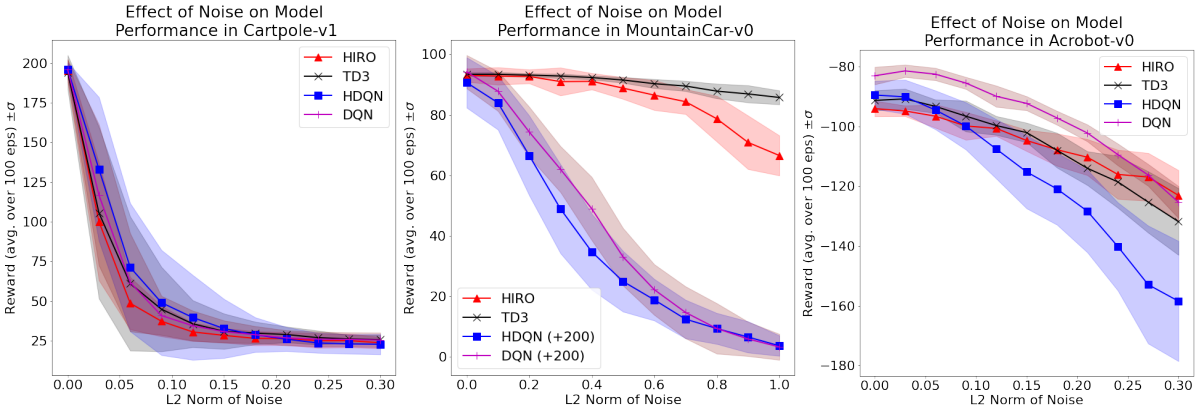


Figure B.2: Effect of random noise. Note that the continuous and discrete version of the MountainCar-v0 environment are semantically very different.

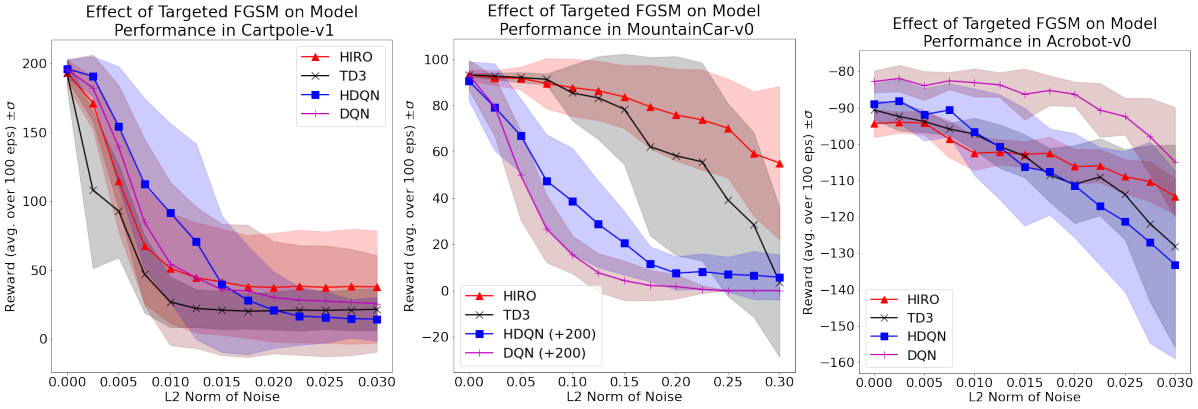


Figure B.3: Effect of targeted FGSM. The FGSM targets the manager when generating perturbations for $\mathbf{x}^H$, and the worker for perturbations for $\mathbf{x}^L$. As such, there are no separate results for targeting either hierarchy. This is a less realistic threat model than the one described in the main project.

B.3 Results

The results for this initial investigation are lacking the depth that is seen in the main body of the project. Most investigation, barring the generalisation experiments and non-targeted FGSM, is present to some degree. An additional type of experiment was run to gauge the *agreement* between the worker and manager. This is a quantitative measure, unlike the manager-worker dependency discussed in Section 3.4.2. With actions as goals, it is trivial to tell whether the worker executed the intended action. The nuance of the word *agreement* is that if the actions do match the goals, it is not possible to tell whether the worker relied on the manager to make this decision, or if it made the same decision independently. I formally define agreement in a single episode to be:

$$\text{agreement} = \frac{\text{№ steps when the manager and worker agree}}{\text{total episode steps}} \tag{B.1}$$

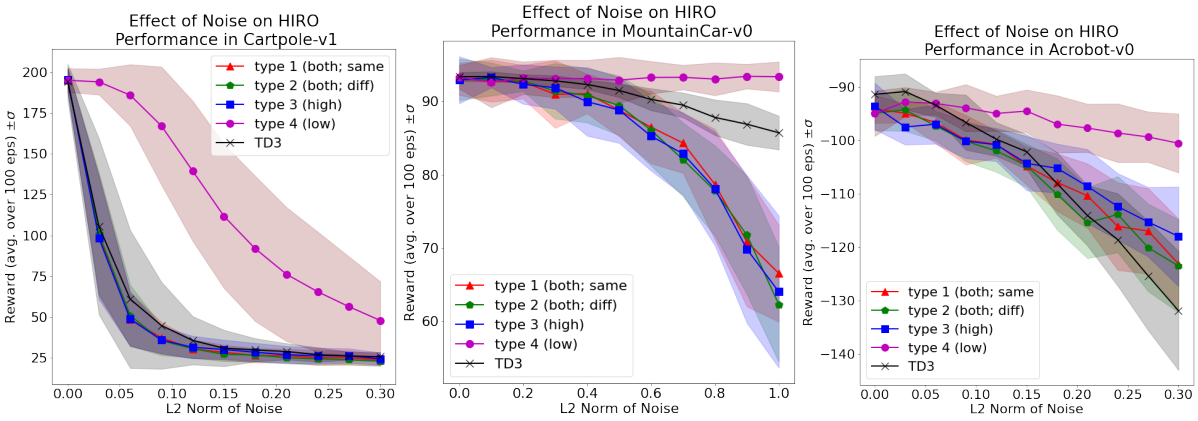


Figure B.4: Demonstrating manager-work dependency of HIRO by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately as discussed in Section 3.4.2.

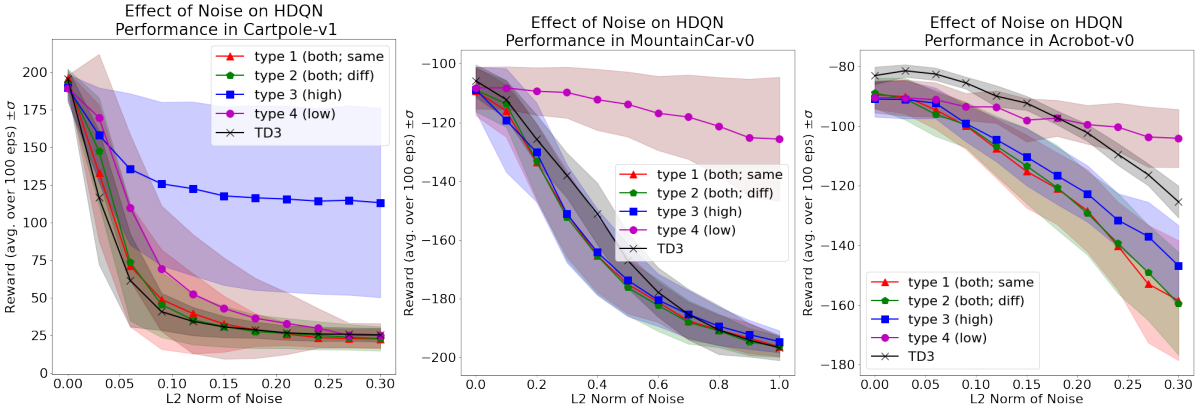


Figure B.5: Demonstrating manager-work dependency of HDQN by targeting $\mathbf{x}^H$ and $\mathbf{x}^L$ separately as discussed in Section 3.4.2.

B.3.1 Safety

Figure B.2 shows the safety properties of the evaluated models. Much as with the states-as-goals, there is no clear advantage to using HRL over RL in this context.

B.3.2 Security

Figure B.3 shows the security properties of the evaluated models. Encouragingly, and similarly to Figure 4.4, HRL marginally outperforms RL in most cases.

B.3.3 Manager-Worker Dependency

Figures B.4 and B.5 demonstrate manager-worker dependency. In most cases, there is a very high reliance on the manager as indicated by the prevalence of the magenta (type 4; attacking the worker only) line compared to the blue (type 3; attacking the manager only) line. This makes intuitive sense as having goals in the action space renders the worker relatively obsolete, useful only for repeating the set goals as actual actions to be performed in the environment.

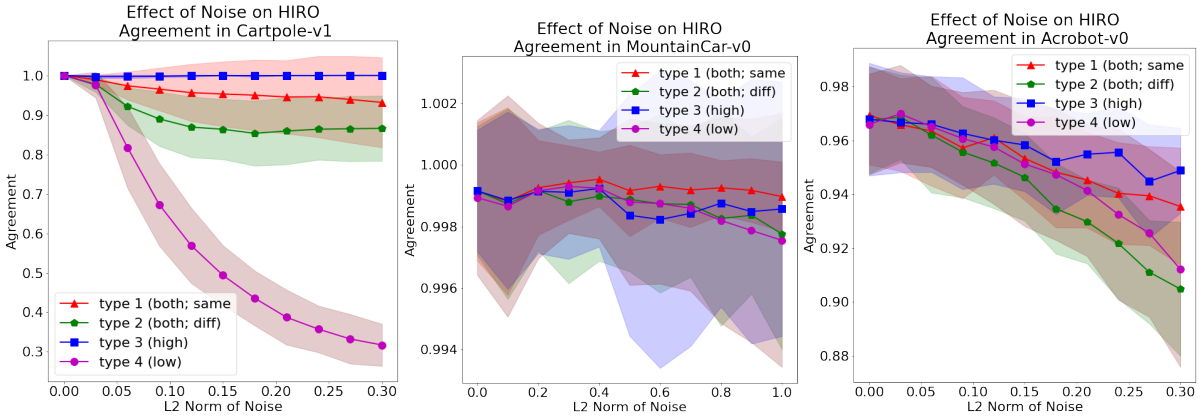


Figure B.6: Demonstrating manager-work agreement of HIRO as discussed in Appendix B.3.

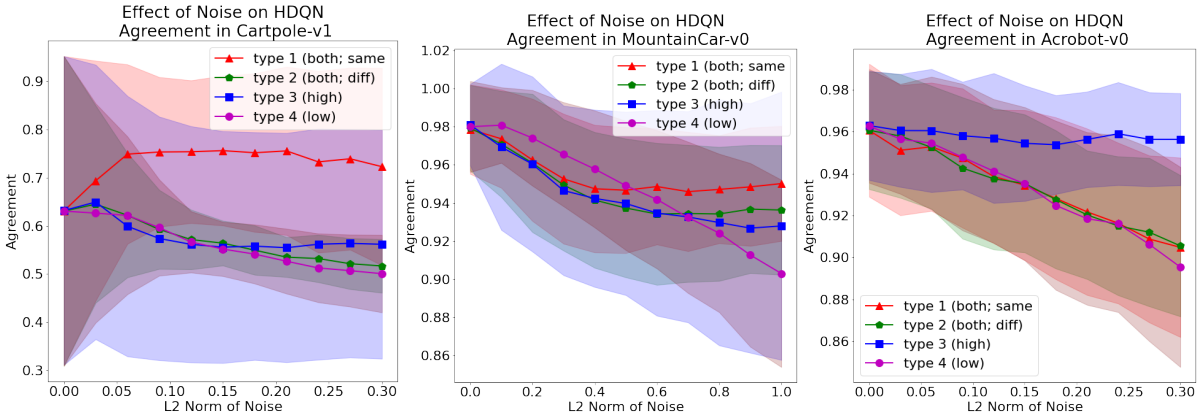


Figure B.7: Demonstrating manager-work agreement of HDQN as discussed in Appendix B.3.

B.3.4 Agreement

Figures B.6 and B.7 demonstrate the *agreement* score for HIRO and HDQN for increasing amounts of random noise. In all cases, targeting the worker (type 4) decreases the agreement more compared to other instances. This suggests that the worker does do some 'sanity checking' of the goals presented to it via the observed state, i.e. if the state suggests an entirely different action that does not match the action-goal, then the worker would choose to ignore the manager. This is an interesting result and deserves further separate investigation, especially given that without noise there is almost always perfect agreement.

Project Proposal

Susceptibility of Hierarchical Reinforcement Learning to Adversarial Examples

Part II Project Proposal

supervised by Ilia Shumailov (is410) and Dmitry Kazhdan (dk525)

October 2020

Introduction

Reinforcement Learning (RL) deals with agents that learn how to act optimally in stochastic environments through trial-and-error. It has seen a surge in popularity recently, primarily attributed to a number of success stories. OpenAI used Deep Reinforcement Learning to beat the human world champion at Dota 2, a complex and real-time game [1]. AlphaZero, a program leveraging a general RL algorithm, managed to beat state-of-the-art programs at Go, chess, and shogi [2].

Barto and Mahadevan [3] and more recently, Kulkarni et al. [4], have tried to address the scalability issues that plague more traditional approaches to RL by learning policies at different levels of abstraction. This approach is broadly known as Hierarchical Reinforcement Learning (HRL). HRL consists of learning multiple policies at different levels of hierarchy, similarly to how humans solve tasks by decomposing them into subtasks [5, 6].

It was discovered by Huang et al. [8] that networks based on reinforcement learning techniques are also susceptible to adversarial attacks [5]. These work by introducing minor perturbations to the inputs that would result in the model producing wrong results. Those minor alterations were shown to greatly reduce the rewards of the trained agents.

Combining these two recent developments – the rising popularity of HRL and recently-discovered attacks on RL – naturally lends itself to the question of whether HRL would be susceptible to the same attacks.

Due to the increased complexity and difference in model structure, one could speculate that HRL models should be less vulnerable to the same attacks that can fool RL models. The idea is that separation of a global task into subtasks should help generalisability and improve robustness of the agent. However, a number of characteristics would suggest

otherwise. The underlying principles are very similar to traditional RL; and the hierarchical nature of HRL could suggest that fooling one level of the model would be enough to propagate the errors down to the output. Each level receives strictly less data than in a traditional RL model due to intentional data hiding.

This project's goal would be to implement both RL and HRL models and the attacks described in Huang et al. [8] and Lin et al. [27]. As an extension, I would then explore both of those possibilities and figure out which attacks HRL systems are susceptible to, to which degree, and whether there are similarities in working attack vectors between traditional RL systems and HRL ones.

Starting Point

Personal

I have some previous experience with machine learning, Python, and security both through personal endeavours and the Computer Science Tripos. There are a number of relevant Tripos courses in Part 1 that I have attended: *Machine Learning and Real-world Data*, *Software and Security Engineering*, *Artificial Intelligence*, *Security*. I was fortunate enough to gain work experience as a machine learning research intern at Speechmatics where I worked for 3-and-a-half months in the summer vacation following Part 1A. During my internship, I worked extensively with Python, specifically TensorFlow and PyTorch, in a production environment. The work was mainly focused on Natural Language Processing and analysis though.

I have read through the notes of Part II's *Machine Learning and Bayesian Inference* course but will not be taking the course. I will be however taking *Deep Neural Networks* as a unit of assessment in Lent term. I believe this will be quite relevant, especially for deeper understanding of fundamentals of neural networks. More tangentially related, I will be taking the *Data Science: principles and practice* unit of assessment in Michaelmas term, which should aid with quantitative evaluation of this project.

I take personal interest in the ML field and keep up-to-date with the latest developments by reading through recently-published papers on arXiv and following specialised blogs.

Background

A number of HRL and RL papers have open-source implementations on GitHub. However, most available HRL-specific code have deviations in either implementation or base assumptions compared to the papers they reference. Quite a lot of the available code also consists of incomplete paper implementations. As such, I will be re-implementing all the RL and HRL models for this project to allow for easier attack implementations. I will use the existing code bases for correctness checking.

Attacks that have been shown to work on RL agents, such as gradient-based [8, 27] ones, have never been tried on HRL agents before. However, the papers that apply these attacks

on RL agents have clear descriptions of how to implement them.

Substance & Structure

The project can be split into two main components:

- The implementation of RL and HRL agents, based on a number of possible algorithms. These include DQN [13], HDQN [4], and HIRO [17].

  The models will be trained using OpenAI's gym environments [24]. I will be making use of some pre-existing environments that previous papers have used as benchmarks, as well as writing my own environments to demonstrate correctness of implementation for papers that don't use gym. Some preliminary environments that seem promising:

  – the 6-state Markov decision process presented in Kulkarni et al. [4] seems to provide a quick-to-train sparsely-rewarded environment that makes a very clear divide between RL and HRL

  – *Taxi-v3*, an existing environment in [24], is specifically designed to illustrate issues in HRL

  – *ant maze* figures prominently in some papers such as Nachum et al. [17]

  Evaluation of this component will be primarily a quantitative proof of correctness of implementation. I would demonstrate that the RL/HRL agents can solve their respective tasks. This can be done by comparing average reward graphs with relevant benchmarks present in existing papers.

- The implementation of adversarial attacks, such as FGSM [8] and other gradient-based attacks [27].

  It is important to make a distinction between test and training-time attacks. The project will focus on implementation and testing of *test*-time attacks, and as such will exclude techniques like *poisoning*.

  Evaluation will primarily consist of quantitative measurements of the difference in model performance with and without attacks applied. Performance here refers to environment-specific measures of reward. We would expect the average reward over a period of time to be statistically smaller in the case of the attack being applied compared to the same time period without the attack.

Extensions

The implementation of the extensions will depend on the progress of the primary success criteria in the next section and any discoveries that I might make along the way. To give some indication of what possible extensions could consist of:

- Adapting existing attacks on RL agents to be more effective against HRL agents

- Devising new attacks on HRL agents
- Explaining the effectiveness of RL attacks on HRL
- Investigating whether there exists some HRL-specific attack that traditional RL systems would not be vulnerable to

Evaluation of the extensions would be strictly quantitative and similar to the main success criteria, with the exception of the explainability extension that lends itself to a more qualitative evaluation.

Success Criteria

1. Implementation of several existing RL and HRL algorithms including: DQN [13], HDQN [4], and potentially others such as HIRO [17].
2. Implementation of one or several existing attacks on these agents, such as simple FGSM [8] or attacks on higher-level controllers.
3. Quantitatively showing the successful implementation of the RL/HRL by demonstrating their ability to solve their respective tasks, by observing an increase in their reward throughout training time.
4. Quantitatively showing the effect of the existing implemented attack(s) on the implemented agent(s), by demonstrating the effect of the attack on the reward obtained by the agent(s).

Plan of Work

To make the best use of potential feedback, I will be meeting with my supervisors on a weekly basis to present the current work and discuss the project and relevant topics.

10th October – 23rd October

Begin literature review of the topic; proposal work.

Milestone

- Write and submit project proposal

Deadlines

- *12th October (15:00)*, send Phase 1 Report Form to overseers
- *16th October (12:00)*, send draft proposal to overseers
- *23rd October (12:00)*, proposal deadline

24th October – 6th November

Setup of necessary programming environments. Finishing up literature review.

Milestone

- Have a working basic implementation of a RL agent

7th November – 20th November

Further implementation of agents.

Milestones

- Working implementation of HRL agent
- Open AI gym environments implemented (where needed) and integrated with the rest of the code
- Success criteria 1 and 3 achieved

21st November – 4th December

Initial work on crafting adversarial examples.

Milestone

- Working implementation of random noise attack on RL agent

5th December – 25th December

Finish core code base.

Milestones

- Working implementation of simple FGSM attack on RL agent
- Success criteria 2 and 4 achieved

26th December– 8th January

Slack period for polishing off core code. If completed early, looking into extensions.

Milestone

- Core code base completed: at least two implementations of RL/HRL, and at least two implemented attacks

9th January – 22nd January

Evaluation of attacks on HRL.

Milestone

- Quantitative results on attack effects on HRL agent

23rd January – 5th February

Progress report work.

Milestone

- Progress report written and submitted

Deadline

- *5th February (12:00)*, progress report deadline

6th February – 19th February

Rehearsal for project presentations. Possible extension work.

Milestone

- Project presentation written and performed

Deadline

- *11th, 12th, 15th, or 16th February (14:00)*, progress report presentations

20th February – 5th March

Extension work. Begin writing dissertation.

Milestones

- First extension completed and evaluated
- Introduction and background/preparation chapters written

6th March – 19th March

Write implementation chapter.

Milestone

- Implementation chapter written

20th March – 2nd April

Write evaluation chapter.

Milestone

- Evaluation chapter written

3rd April – 16th April

Write conclusion chapter.

Milestone

- Conclusion chapter written

17th April – 30th April

Edit dissertation based on received feedback. Final clean-up of source code.

Milestone

- Draft of dissertation finished

1st May – 14th May

Slack time.

Milestones

- Dissertation written and submitted
- Source code submitted

Deadlines

- *14th May (12:00)*, dissertation deadline (electronic)
- *14th May (17:00)*, source code deadline (electronic copies)

Resource Declaration

I plan to use my own computers for the project. The specifications are as follows:

1. *Desktop*; 3.4 GHz CPU, 16 GB RAM, 250 GB SSD, 1 TB HDD, GTX 770, Windows 10
2. *Laptop*; 1.6 GHz CPU, 8 GB RAM, 500 GB SSD, GTX 920M, dual boot between Ubuntu 18 and Windows 10

I will also be making use of GPUs, access to which will be arranged by my supervisor Ilia Shumailov (is410). In the case that these GPUs end up being the departmental GPU servers, I will need a departmental Linux account and a corresponding Kerberos password to be able to ssh into them.

My contingency plans against data loss are using git on GitHub to version control all relevant files, with backups to Google Drive. Furthermore, the data will be locally stored both on my *Desktop* and *Laptop*. My contingency plans against hardware failure include being able to transition from working on either one of my computers to the other. In dire circumstances, it would also be possible to move my entire workload onto the MCS machines.

I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.