Hovedoppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Patrick Nitschke

Reinforcement Learning for Fast, Map-Free Navigation in Cluttered Environments Using Aerial Robots

Hovedoppgave i MSc. Cybernetics and robotics
Veileder: Prof. Dr. Kostas Alexis
Medveileder: Dinh Huan Nguyen, Mihir Kulkarni
Juli 2022

NTNU
Kunnskap for en bedre verden

Patrick Nitschke

Reinforcement Learning for Fast, Map-Free Navigation in Cluttered Environments Using Aerial Robots

Hovedoppgave i MSc. Cybernetics and robotics
Veileder: Prof. Dr. Kostas Alexis
Medveileder: Dinh Huan Nguyen, Mihir Kulkarni
Juli 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU
Kunnskap for en bedre verden

NTNU
Kunnskap for en bedre verden

Department of Engineering Cybernetics

TTK4900

Cybernetics and robotics, Master's Thesis

Reinforcement Learning for Fast, Map-Free Navigation in Cluttered Environments Using Aerial Robots

Patrick Nitschke

July 11, 2022

Supervisor: Prof. Dr. Kostas Alexis
Co-supervisers: Dinh Huan Nguyen, Mihir Kulkarni

Abstract

Autonomous navigation in increasingly complex domains presents new challenges that question the efficiency and capability of traditional model-based methods. Though traditional approaches have been successful for unstructured environments in the past, uncertain, sensor-degraded, or dynamic environments cannot be modelled and thus be solved by these approaches. Instead, learning-based methods have become increasingly popular due to their ability to learn complex behaviour without explicit programming, where multiple components can be combined into a single model to tackle the perception, prediction and motion task of autonomous navigation.

In this theme, this thesis explores the use of reinforcement learning for autonomous navigation of a quadrotor through cluttered environments, with only a depth camera. We propose a two-part deep neural network model comprised of an encoder-CNN and MLP, where the CNN serves as the perception module while the MLP is the optimal controller. With this framework, our model receives a quadrotor state and depth image as input and maps this to a velocity and yaw rate reference to reach a specified goal in three dimensions.

To solve the task, we present the problem as an unsupervised representation learning and reinforcement learning task. The CNN is trained as an encoder of VAE that learns to reconstruct depth images, while the MLP learns to utilise the VAE latent code as a depth representation of the environment, so to be able to navigate the environment. We introduce a custom reconstruction error for the VAE to specify collision-specific features that should be prioritised in the depth encoding. We also introduce a novel reward function for the reinforcement learning agent that motivates both waypoint navigation and collision avoidance.

By further utilising large-scale parallelism, we present the training and evaluation of our final reinforcement learning policy, which achieves a 92.5% success rate averaged across four known 20×10 environments with varying degrees of clutter. The agent demonstrates good robustness when a Gaussian multiplicative noise $\epsilon_n \sim \mathcal{N}(1, 0.2)$ is applied to all states and actions, with an 87.5% success rate across the four environments. However, we identify some constraints with our model – namely dependence on accurate depth representations and a poor generalisation to larger environments. Finally, as further work, we should train our modules to handle noisy depth images, add modifications to account for generalisation, and add a prediction module in the form of an LSTM or Transformer to further improve performance.

Sammendrag

Autonom navigering i stadig mer komplekse domener byr på nye utfordringer og stiller spørsmål ved effektiviteten og kapasiteten til tradisjonelle modellbaserte metoder. Selv om tradisjonelle metoder har vært vellykkede for ustrukturerte miljøer i det siste, kan usikre, sensor-degraderte eller dynamiske miljøer ikke modelleres og dermed løses med disse metodene. I stedet har læringsbaserte metoder blitt stadig mer populære på grunn av deres evne til å lære kompleks atferd uten eksplisitt programmering, der flere komponenter kan kombineres til en enkelt modell for å takle persepsjons-, prediksjons- og bevegelsesoppgaven til autonom navigering.

I dette temaet utforsker denne oppgaven bruken av forsterkende læring for autonom navigering av en drone gjennom hinderfylt miljøer, med kun et dybdekamera. Vi foreslår en todelt dyp nevrale nettverksmodell som består av en koder-CNN og MLP, der CNN fungerer som persepsjonsmodulen mens MLP er den optimale kontrolleren. Med dette rammeverket mottar modellen vår en dronetilstand og et dybdebilde som input og kartlegger dette til en hastighets- og girhastighetsreferanse for å nå et spesifisert mål i tre dimensjoner.

For å løse oppgaven presenterer vi problemet som en uovervåket representasjonslærings- og forsterkende læringsoppgave. CNN er opplært som en koder for VAE som lærer å rekonstruere dybdebilder, mens MLP lærer å bruke VAE latent kode som en dybderepresentasjon av miljøet, for å kunne navigere i miljøet. Vi introduserer en tilpasset rekonstruksjonsfeil for VAE for å spesifisere kollisjonsspesifikke funksjoner som bør prioriteres i dybdekodingen. Vi introduserer også en ny belønningsfunksjon for forsterkende læringsmiddel som motiverer både veipunktnavigasjon og kollisjonsunngåelse.

Ved ytterligere å bruke storskala parallellisme, presenterer vi opplæringen og evalueringen av vår endelige forsterkende læringspolicy, som oppnår en suksessrate på 92,5% i gjennomsnitt over fire kjente miljøer på 20 $ganger10$ med ulik grad av rot. Agenten viser god robusthet når en Gaussisk multiplikativ støy $\epsilon_n \sim \mathcal{N}(1, 0, 2)$ brukes på alle tilstander og handlinger, med en suksessrate på 87,5% på tvers av de fire miljøene. Imidlertid identifiserer vi noen begrensninger med modellen vår – nemlig avhengighet av nøyaktige dybderepresentasjoner og en dårlig generalisering til større miljøer. Til slutt, som videre arbeid, bør vi trene modulene våre til å håndtere støyende dybdebilder, legge til modifikasjoner for å ta hensyn til generalisering, og legge til en prediksjonsmodul i form av en LSTM eller transformator for å forbedre ytelsen ytterligere.

Preface

This master's thesis symbolises the end of 5-years at the Norwegian University of Science and Technology (NTNU) – the last page of what has been an exciting educational chapter. I had the pleasure of writing it under the guidance of Prof. Dr. Kostas Alexis and PhD. candidates Dinh Huan Nguyen and Mihir Kulkarni, who are all a part of the Autonomous Robots Lab (ARL). As a result, the theme of this thesis follows from their goal – to develop intelligent robotic systems that can complete tasks under any possible conditions in complex, dynamic and diverse environments.

The thesis was also part of the lab's initiative to explore more learning-based methods in their work and a new simulation framework released last year, Isaac Gym. This meant that the bulk of the approach had to be written and integrated with Isaac Gym from the ground up, where I had to learn an entirely new machine learning framework, PyTorch. Admittedly, this has been worth it. I have been quite fortunate to receive such an exciting topic that builds on current state-of-the-art methods and tools. Looking forward, there is much to be improved in the implementation, so hopefully, this thesis (and the code) can come to good use for future students.

Furthermore, this thesis builds on the project thesis [1] – essentially a crash course in reinforcement learning for robotics – where we trained a quadrotor for waypoint navigation with no obstacles present. Unfortunately, the results were not that promising, which resulted in a sceptical and conservative development process during this thesis, being the primary motivation for the *curriculum*, which in hindsight served it well.

Since a significant focus of the project was on the theory, multiple sections in the reinforcement learning chapter are taken from the thesis and marked with (*). This is so that the theory can be presented from the absolute fundamentals, as it is not a part of NTNU's cybernetics curriculum. Otherwise, this is not the case for the deep learning aspect, as this thesis assumes that fundamentals here should be known: neural networks, gradient descent, backpropagation, etc. We also assume the same for estimation and machine learning theory, e.g. maximum likelihood estimation, Bayesian networks and inference.

Acknowledgements

Without help, writing a master's thesis with a scope as large as this would be impossible. Thus, I give credit where credit is due. Firstly, I would like to thank Prof. Dr. Kostas for his tips, insight and expertise during our meetings, which have laid the foundation for this thesis. Secondly, I would like to thank my PhD. co-supervisors, Huan and Mihir, for their extensive help: Huan, for helping me with tools for learning PyTorch, designing the VAE, and providing tips on improving it; Mihir, for creating the base of the quadrotor task and feedback on how to design the agent experiment. Both have also provided important feedback on the PPO reward, sported long after-hour meetings, and shared support. Also, I thank the cybernetics department, particularly Gunnar Aske, for lending me a powerful PC to run Isaac Gym.

Then, I must show appreciation to my friends at *sal* who have provided feedback, late-night company, and also moral support. Finally, I thank my family and partner, who have facilitated the last few weeks of this thesis.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

Autonomous navigation of robotic vehicles is a complicated task that has challenged the cybernetics community since its inception. It is a research field of substantial focus particularly in recent years, given its novel applications within commercial sectors [2–4], transportation [5], search and rescue [6], and defence [7]. As new possibilities for autonomy are increasing, so is the need to find new, innovative, robust and safe solutions to meet this demand.

The difficulty of autonomous navigation in cluttered and dynamic environments arises from the combination of three separate tasks: first to perceive the local environment directly from on-board sensors, then to predict how the environment will evolve, and finally to decide on a safe and intelligent action based on the inferred information [8]. Each of these tasks presents its challenges, such as dealing with noise or uncertainty in sensor data or feasible trajectory planning, but separating these tasks into a threefold process ultimately leads to an increased latency and compounding of errors in the pipeline [9].

Moreover, as robotic use-cases are becoming more advanced – such as in underwater [10], forested [9] and subterranean environments [11] – autonomous robots now have to contend with environments that are: sensor-degraded with limited illumination; long, narrow and multi-branching; unpredictable and unstructured; isolated from external communications. Though localisation and mapping techniques based on 3D perception have been successful in unstructured environments in the past [12], the characteristics of these newer domains present new challenges for traditional approaches. Specifically, these environments make it difficult to maintain an accurate map of the environment, puts the tractability of trajectory planning into question and limits the amount of computational resources available to the robot [9, 13–15].

Not to mention, navigation within cluttered environments requires fast, accurate and careful planning of versatile vehicles – such as in multirotor aerial vehicles (quadrotors) [16]. This requires a quick mapping from sensor data to action, which makes a high-latency solution non-

viable because of the inherent difficulty of pose estimation when travelling at high speeds [9, 14].

Therefore, these issues prompt the consideration of learning-based methods to directly infer actions from raw sensor input, as an alternative to the three-subtask, model-based pipeline. The idea is to remove the necessity for accurate maps, though retaining essential features, and using this to plan feasible trajectories even in complex edge-case scenarios. Utilising a data-driven approach should allow an agent to capture the system's dynamics and the environment's uncertainties without the need for any explicit programming [17] – thus removing the need for feature engineering or heuristics to make the navigation task tractable [13]. The processing time dramatically decreases due to this direct mapping, as sensor data does not need to be preprocessed into higher dimensional information [18, 19], maps will not have to be generated or queried, and exhaustive collision checking is avoided [15]. Instead, a learning approach will be used to extract high-dimensional information directly, capable of filtering out redundant information in LiDAR and depth data. Then, the agent can learn collision avoidance based on experience from these high-level features [13].

The apparent limitation of using learning-based methods is that the amount of data required to solve complex tasks is proportional to its complexity [20], where varied environments are often also required in the learning process [21]. Due to this, the question of whether or not a task was learnable through reinforcement learning became simply a question of time or computational resources. If these resources were unavailable, more sample-efficient methods had to be explored, for example: supervised learning through clever use of datasets [22, 23], engineering of action spaces and learning these in a self-supervised manner [24], or by imitation learning using an expert planner [9, 15, 25].

However, until recently, the research community has been developing parallel end-to-end hardware-accelerated (GPU) simulators, such as *Isaac Gym*, which have provided the opportunity to simulate tens of thousands of environments in parallel and "enables the solving of tasks with a single GPU that were previously only possible on massive CPU clusters" [26]. This has opened up a multitude of possibilities for autonomous navigation using reinforcement learning, thus being the motivation for this thesis.

1.2 Scope

This thesis explores how a mapless navigation policy for collision avoidance can be learned on a quadrotor without expert demonstrations by leveraging novel ideas in learning-based autonomous navigation combined with a massively parallel learning scheme (Isaac Gym). Specifically, the aim is to infer a reference velocity and steering angle for the quadrotor given its state and depth image from a forward-facing depth camera. With this policy, an agent should be able to navigate through a cluttered environment to some goal specified in three-dimensional space,

such that it can be combined with some global path planner.

To learn this policy, this thesis will present the theory and implementation of a two-part deep neural network module. The first module, a variational autoencoder (VAE), is tasked with extracting the essential information or features from a depth image. The second module will be a fully-connected neural network (or multi-layer perceptron) and serves as the reinforcement learning agent. The agent will receive the essential information of the depth image (i.e. its features), along with the quadrotor state, and decide on a velocity and yaw rate reference for the quadrotor. To optimise the VAE, we use the Auto-Encoding Variational Bayes (AEVB) algorithm, while agent is optimised through Proximal Policy Optimisation (PPO). We also define a custom loss function for the VAE to define important features to be prioritised, and define a reward function that is conditions an agent to be collision avoidant.

Additionally, as the VAE and MLP modules are written from scratch, the design choices and training for each module will be presented in detail. For the VAE, this includes the choice of architecture and loss functions for improvements in the reconstructed images. For the MLP, this includes simulation setup and gradual training steps (curriculum) to achieve a stable training process and eventually collision avoidance. Finally, this thesis will evaluate the performance of the trained policy in a series of standardised tests, ranging from known environments with varying difficult, robustness tests to noise and finally generalisation to larger domains. We also present the results for the VAE, training though these are not evaluated to the same degree.

1.3 Outline

The outline of this thesis will be as follows:

- **Chapter 1: Introduction**
  The motivation for this thesis is presented, along with the scope of the task and the content of this thesis.
- **Chapter 2: Theoretical Background** The underlying theory for VAE is presented through the lens of unsupervised representation learning, where fundamental deep learning theory (like gradient descent) is assumed to be known. Then, a full introduction to reinforcement learning is provided, so to understand PPO.
- **Chapter 3: Related Works** The approaches of previous works within learning-based motion planning are presented to get an overview of the different methods to solve the task. The motivation for reinforcement learning is put forward in comparison.
- **Chapter 4: Problem Formulation** The learning task of this thesis is presented from a technical perspective. The problem is split into two parts – a reinforcement and representation

learning task.

- **Chapter 5: Proposed Approach** The two-part deep neural network model is proposed. The custom rewards, losses, training setup and architecture of the VAE and MLP are presented and discussed.
- **Chapter 6: Implementation** The method to prepare and train the proposed network is presented, along with various software tools and frameworks. This includes the gathering of data for VAE, the simulation setup and implementation details for optimising our reinforcement learning agent.
- **Chapter 7: Navigation Policy Evaluation Studies** The step-by-step procedure for training the agent is presented and the results analysed. The agent performance is then evaluated in known environments, when exposed to noise, and in larger environments.
- **Chapter 8: VAE Evaluation Studies** The results of VAE training with different loss functions are presented and their reconstructions analysed.
- **Chapter 9: Discussion** We explore how a reinforcement learning agent is able to solve a collision task and why it sometimes fails, and discuss the differences between the loss functions for the VAE.
- **Chapter 10: Conclusions** The overall approach, results and discussion points are summarised.

Chapter 2

Theoretical Background

In this chapter, we will cover the relevant theory required to understand our two-part model, namely a *Variational Autoencoder* (VAE) combined with a reinforcement learning agent – a multi-layer perceptron (MLP) – trained with *Proximal Policy Optimisation* (PPO).

First, a gentle introduction of *unsupervised learning* will be given, along with its use in representation learning. Then, we will look into *autoencoders* as a way of extracting features in a dataset before finally exploring the theory behind VAEs. Later, in section 2.3, we will begin with a recollection of fundamental reinforcement learning concepts such as Markov Decision Processes, returns, value functions and policies. This will serve as a stepping stone to understanding the topic of policy optimisation, particularly *actor-critics* and how PPO builds on this.

2.1 Unsupervised Learning

Machine learning algorithms are broadly classed into four categories: supervised, semi-supervised, unsupervised and reinforcement learning – depending on what kind of experience the algorithms are allowed to have during the learning process. Though supervised learning has been one of the most powerful tools of AI, it requires labour-intensive feature engineering and labelling in order to create datasets in areas such as vision, audio and text [27]. In contrast, unsupervised techniques learn from *unlabelled* datasets, where, in a deep learning context, the aim is to learn the useful properties of the structure of this dataset or even its underlying probability distribution [28]. The key idea is that by learning the useful properties of our data, we can use this as a better, more compact *representation* of our input, significantly reducing its complexity.

2.2 Representation Learning

Representation learning refers to the unsupervised learning of a dataset's useful structures or probability distribution in order to make a subsequent learning task easier [28]. In [29], it is

highlighted that "the performance of machine learning methods is heavily dependant on the choice of data representation," and that to "understand the world around us," it must "learn to identify and disentangle the underlying explanatory factors hidden in low-level sensory data." To explore these ideas, we focus on the use of autoencoders and variational autoencoders as a means of *learning a representation* of a dataset.

2.2.1 Autoencoders

Autoencoders are a class of neural networks whose learning objective is an identity mapping from input to output, under some specific constraint [30]. The mapping is described with two parts, first a function to describe an encoding $\boldsymbol{z} = f(\boldsymbol{x})$, and then a function to describe a decoding $\boldsymbol{r} = g(\boldsymbol{z})$. Here, $\boldsymbol{x}$ refers to the input data, while $\boldsymbol{z}$ is the hidden layer of a neural

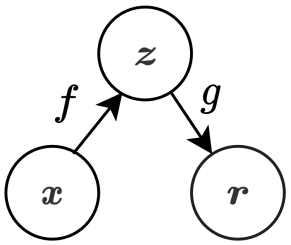


Figure 2.1: The general structure of an autoencoder showing how an input $\boldsymbol{x}$ is encoded into a latent code $\boldsymbol{z}$, before being mapped to a reconstruction $\boldsymbol{r}$ [28].

network that represents a *code* or *latent representation*, and $\boldsymbol{r}$ is the reconstructed input mapped from the code $\boldsymbol{z}$.

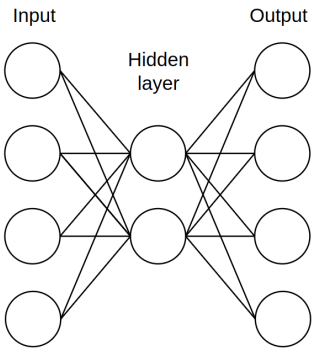
The constraint on an autoencoder is typically placed through its *architectural design* or its *learning process*, where the idea is to restrict autoencoders to prioritise only parts of the information in the input when reconstructing the input (mapping $\boldsymbol{x}$ to $\boldsymbol{z}$ and $\boldsymbol{z}$ to $\boldsymbol{r}$). By doing so, the prioritised parts of the input will become a more useful, alternative representation of the input $\boldsymbol{x}$ [28].

In this thesis, we focus on autoencoders that are architecturally constrained. Architecturally constrained autoencoders are referred to as *undercomplete* autoencoders as they are designed to lack the representational power to map inputs to outputs perfectly. Practically, this means that the hidden layer $\boldsymbol{z}$ has a dimension less than the dimension of the input $\boldsymbol{x}$ or reconstruction $\boldsymbol{r}$, as seen in Figure 2.2a. As a result, undercomplete autoencoders are forced to capture the most prominent features of the input data in its hidden layer [28].

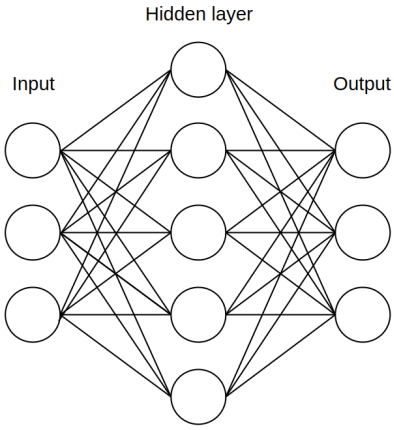
To train an undercomplete autoencoder, we minimise a loss function,

$$\mathcal{L}(\boldsymbol{x}, g(f(\boldsymbol{x}))) \tag{2.1}$$

where we set the target values of the neural network to be equal to the input $\boldsymbol{x}$. The loss function



(a) An undercomplete autoencoder: the dimension of the smallest hidden layer is less than the input dimension.



(b) For comparison, a sparse autoencoder has sufficient hidden units but has a sparsity constraint imposed on its hidden layer.

Figure 2.2: The difference between an undercomplete and sparse autoencoder.

is normally chosen to be the mean-squared error (MSE) between the input and the reconstructed input,

$$\mathcal{L}_{MSE} = \sum_{x_i \in \boldsymbol{x}} \left[x_i - g(f(x_i)) \right]^2 \tag{2.2}$$

and the neural network is optimised through a standard optimisation algorithm, such as mini-batch stochastic gradient descent. The loss can also be chosen as the binary cross-entropy between input and reconstruction depending on the task, for example, when encoding Bernoulli distributed data or one-hot encoded text. For the sake of completeness, a sparse autoencoder has almost the same objective, but with an added penalty term $\Omega(\boldsymbol{z})$ (e.g. L1 loss) that enforces sparsity (keep weights close to 0) in the hidden layer $\boldsymbol{z}$ [31].

2.2.2 Variational Autoencoders

Variational autoencoders (VAEs) [32, 33] are related to autoencoders in terms of architecture, but are in the family of *structured probabilistic models*. This means that they also have an encoder-decoder neural network structure but, in contrast, make assumptions on the underlying probability distribution of the input $\boldsymbol{x}$ and latent code $\boldsymbol{z}$ and wish to model it.

A Parametrised Probabilistic Model

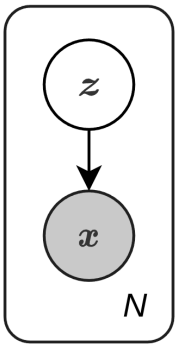


Figure 2.3: A graphical model (Bayes net) of the probability model in the VAE. The assumption is that our observed input data $\boldsymbol{x}$ is generated from the conditional distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$, with true parameters $\boldsymbol{\theta}^*$ unknown. Here, the latent representation $\boldsymbol{z}$ is hidden, while the input $\boldsymbol{x}$ is observed (grey). N is the number of repetitions (in plate notation), equivalent to the number of i.i.d. samples of some dataset.

VAEs contain a probabilistic model, with parameters $\boldsymbol{\theta}$, that aims to estimate a joint probability distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}, \boldsymbol{z})$ as shown in Figure 2.3. It assumes that our latent variables $\boldsymbol{z}$ are generated from some prior distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{z})$ and that the input $\boldsymbol{x}$ is generated from the conditional distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$. Also, it is assumed that the prior $p_{\boldsymbol{\theta}^*}(\boldsymbol{z})$ and likelihood $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$ both come from parametric families of distributions $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ and $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$, though their true parameters $\boldsymbol{\theta}^*$ are unknown [32].

The main use of learning this joint distribution is so that the probabilistic model can perform *probabilistic inference* – computing the posterior distribution of the hidden nodes, given the values of observed nodes [34]:

$$p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}) = \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})p_{\boldsymbol{\theta}}(\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})} \tag{2.3}$$

From a representation learning perspective, being able to learn how to perform posterior inference requires that a probabilistic model learns the distribution of the data $p_{\boldsymbol{\theta}}(\boldsymbol{x})$, which then allows the model to infer the hidden structure or latent code $\boldsymbol{z}$ that best explains our input $\boldsymbol{x}$ [35].

However, exact inference is often intractable due to $p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \int p_{\boldsymbol{\theta}}(\boldsymbol{z})\, p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ requiring a marginalisation of over the latent variables (which is exponential in the number of hidden nodes) and because of the scale of large datasets [32]. Therefore, VAEs instead make use of *variational inference*: attempting to approximate the true posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$ with a restricted family of distributions $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, where the goal is to find the settings of the variational parameter $\boldsymbol{\phi}$ that best approximates the true posterior. This transforms a complex inference problem into a high-dimensional optimisation problem [28, 35].

So, in machine learning terms, the encoder neural network of a VAE parametrised by $\boldsymbol{\phi}$ represents the approximate posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, and generates a family of distributions – such as a set of means and variances for Gaussians – for a given datapoint $\boldsymbol{x}$. Similarly, the decoder neural network, parametrised by $\boldsymbol{\theta}$, then represents the corresponding likelihood distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ that generates a distribution over values of $\boldsymbol{x}$ for a given latent code $\boldsymbol{z}$. In the literature, the probabilistic encoder is also referred to as an *inference network* or *recognition model*, while the probabilistic decoder is referred to as a *generative network* or *generative model*.

Optimising the VAE

Now, in order to optimise the parameters of the encoder and decode, we want to find an objective function to update parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ so that our encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ best approximates the true posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, and that our generative decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ best approximates the true likelihood $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$. To do this, we first have to consider a non-negative similarity measure, the *Kullback-Leibler (KL) divergence*, that can be used to measure the difference between two distributions $P(x)$ and $Q(x)$ [28]:

$$D_{KL}(P \parallel Q) = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{x\sim P}\left[\log P(x) - \log Q(x)\right] \tag{2.4}$$

Then, with the goal of minimising the difference between the approximate $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and true posterior $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})$, we can substitute these for $P(x)$ and $Q(x)$ to get:

$$\begin{aligned} D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) &= \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) - \log p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})\right] \\ &= \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) - \log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})}{p_{\boldsymbol{\theta}}(\boldsymbol{x})}\right] \\ &= \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})\right] - \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right] + \log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \end{aligned} \tag{2.5}$$

However, this term cannot be computed directly due to the marginal likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ being intractable as mentioned above. To get around this, we instead rewrite the marginal likelihood as:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) = D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x})) + \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}) \tag{2.6}$$

Since the KL divergence is non-negative, the term $\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x})$ is referred to as the *variational lower bound* or *evidence lower bound (ELBO)* since it sets a lower bound on the marginal likelihood (or evidence) [32]:

$$\log p_{\boldsymbol{\theta}}(\boldsymbol{x}) \geq \mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[-\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{x},\boldsymbol{z})\right] \tag{2.7}$$

This can be rewritten further:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}) =& \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[-\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z}) + \log p_{\boldsymbol{\theta}}(\boldsymbol{z})\big] \\ =& -\mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) - \log p_{\boldsymbol{\theta}}(\boldsymbol{z})\big] + \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\big] \\ =& -D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel \log p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \mathbb{E}_{\boldsymbol{x}\sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\big[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\big] \end{aligned} \tag{2.8}$$

This ELBO term is particularly interesting because *maximising* it is equivalent to *minimising* the KL divergence between the approximate and true posteriors. Therefore, we choose to optimise this term through a standard optimisation algorithm, such as mini-batch stochastic gradient *ascent*. For a single datapoint with L samples, an estimator for the ELBO loss in (2.8) is:

$$\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}^{(i)}) = -D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)}) \parallel \log p_{\boldsymbol{\theta}}(\boldsymbol{z})) + \frac{1}{L}\sum_{l=1}^{L}\big(\log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}|\boldsymbol{z}^{(i,l)})\big) \tag{2.9}$$

This can be extended to be an estimator for the ELBO loss for the full dataset:

$$\mathcal{L}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}^{(i)}) \simeq \widetilde{\mathcal{L}}^{M}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}^{(i)}) = \frac{N}{M}\sum_{i=1}^{M}\big(\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi};\boldsymbol{x}^{(i)})\big) \tag{2.10}$$

where M is the number of datapoints per mini-batch and N the total datapoints. By choosing M large enough (e.g. $M = 100$) allows us to use have one sample per datapoint ($L = 1$) when computing (2.9) [32]. We also note for later reference that the first term in (2.9) can be referred to as a *regularisation loss*, while the second term can be referred to as a *reconstruction loss*.

The Reparametrisation Trick

However, since $\boldsymbol{z}$ is a random variable sampled from the distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, how are we able to find *deterministic* gradients of the ELBO with respect to the parameters $\boldsymbol{\phi}$ of the distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$? To solve this, [32] introduced a *reparametrisation trick* where, for a datapoint $\boldsymbol{x}^{(i)}$, the continuous *random* variable $\boldsymbol{z}^{(i,l)} \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}^{(i)})$ is expressed as a *deterministic* variable:

$$\boldsymbol{z}^{(i,l)} = g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}^{(i,l)}, \boldsymbol{x}^{(i)}), \qquad \text{where} \qquad \boldsymbol{\epsilon}^{(l)} \sim p(\boldsymbol{\epsilon}) \tag{2.11}$$

and $\boldsymbol{\epsilon}$ being a noise variable with marginal distribution $p(\boldsymbol{\epsilon})$. By reparametrising $\boldsymbol{z}$ through a differentiable transformation $g_{\boldsymbol{\phi}}(\boldsymbol{\epsilon}^{(i,l)}, \boldsymbol{x}^{(i)})$ and $\boldsymbol{\epsilon}$, we ensure that the parameters of the distribution still remain learnable while the VAE remains stochastic through $\boldsymbol{\epsilon}$. To give an example, in most cases, we assume the variational approximate and true posteriors to be a multivariate Gaussian. To sample $\boldsymbol{z}^{(i,l)}$ using the reparametrisation trick would be done as:

$$\boldsymbol{z}^{(i,l)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(l)}, \qquad \text{and} \qquad \boldsymbol{\epsilon}^{(l)} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{2.12}$$

with $\odot$ meaning the element-wise product. In this example, the mean $\boldsymbol{\mu}^{(i)}$ and standard deviation $\boldsymbol{\sigma}^{(i)}$ of the Gaussian would then become learnable parameters of the encoder, each having their own deterministic gradients that can be calculated in backpropagation.

Understanding ELBO and the VAE Latent Space

Finally, using a VAE gives the learned latent representation $\boldsymbol{z}$, for a given input $\boldsymbol{x}$, some desirable properties that distinguish it from ordinary autoencoders.

First, by observing the structure of (2.9), we note that to maximise the ELBO loss requires that we minimise the KL divergence between the approximate posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ and the latent prior $p_{\boldsymbol{\theta}}(\boldsymbol{z})$. When we assume both to be multivariate Gaussians, $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$ is essentially penalised for being "different" from a Gaussian distribution. Then, by forcing the approximate posterior distribution to be similar to some pre-decided prior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z})$, we can sample $\boldsymbol{z}$ from this prior and generate synthetic data through the decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$. To visualise how the latent space influences the generated data, we can also perform a systematic sampling of the latent space, such as in a $[-1, 1]$ area for a Gaussian two-dimensional latent space, to generate a manifold (or prior predictive distribution) as shown in Figure 2.4. Therefore, by placing the prior assumption

(a) Learned Frey Face manifold **(b)** Learned MNIST manifold

Figure 2.4: The predicted manifolds generated from sampling latent variables $\boldsymbol{z}$ in a linearly spaced $[-1, 1]$ area when $p_{\boldsymbol{\theta}}(\boldsymbol{z})$ is 2D Gaussian. Both images are obtained from [32].

on the underlying distribution of our approximate posterior and latent space distribution, VAEs are capable of being *generative models*.

To understand the ELBO loss further, [32] refers to the first KL divergence term in (2.9) as a *regulariser*, while the second term can be thought of as a *reconstruction error*. Looking first at the reconstruction term, we recall that the objective of most machine learning models is that of maximum likelihood estimation (MLE) (minimising the negative log-likelihood might sound

familiar). The goal of MLE is to find the parameters of a model that maximises the likelihood that they generated the data – or, in this case, finding the best $\boldsymbol{\theta}$ so that $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$ most accurately describes the process that generated our data, which was assumed to be the unknown likelihood distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$. So, by maximising the log-likelihood in (2.9), we are doing MLE, which is equivalent to finding the best model parameters that minimises the reconstruction error between a target – the input generated from $p_{\boldsymbol{\theta}^*}(\boldsymbol{x}|\boldsymbol{z})$ – and output generated from our decoder $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$.

Next, the KL divergence term can be considered a regulariser by first remembering that it penalises the encoder for being "different" from a Gaussian prior. Further, given that it is not entirely likely that the true unknown posterior $p_{\boldsymbol{\theta}^*}(\boldsymbol{z}|\boldsymbol{x})$ follows a Gaussian distribution, it will be natural to incur an information loss (and so a high KL divergence) when we assume that our approximate posterior is. Therefore, there is a trade-off between learning a proper input reconstruction (when the approximate posterior deviates from a Gaussian) versus staying close to the Gaussian prior. This constraint can be thought of as limiting the generative capacity of the VAE when compared to having no KL divergence loss in the ELBO, but its regulatory effect places emphasis on the VAE having to learn *meaningful and statistically independent latent factors* [36]. In other words, it is expensive for the latent variables to deviate from the Gaussian prior, so the latent variables that do deviate should hold meaningful and independent information – i.e. they each describe their own features of the input – so that a reduction in the reconstruction error compensates for their incurred cost.

To summarise, the goal of VAEs is to minimise the KL divergence between the true and approximate variational posteriors, $D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x}) \parallel p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{x}))$. This was intractable due to the marginal likelihood $p_{\boldsymbol{\theta}}(\boldsymbol{z})$, and so it was shown that this was equivalent to maximising the ELBO loss in (2.9). The effect of formulating the objective function in this way had two consequences. Firstly, the VAE achieved a desirable, well-formed latent space (most often close to a Gaussian prior). This allowed VAEs to be generative since, to generate "synthetic" data, we could sample latent variables following the prior distribution and compute $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$. Second, the KL divergence term served as a regulariser which motivated latent variables in the VAE to hold meaningful representations of the input distribution.

2.2.3 Convolutional Variational Autoencoders

Convolutional variational autoencoders are VAEs that utilise *convolutional neural networks* (CNNs) to parametrise the encoder and some *deep generative deconvolutional network* (DGDN) to parametrise the decoder [37]. The theory behind this is that for structured data, convolutional operations are best suited for feature extraction, being "tremendously successful" in practice [28]. Therefore, when tasked with learning an unsupervised representation of (for example) images, we can use these instead of fully-connected VAEs.

Convolution

CNNs are simply NNs that use the convolution operation ($*$) instead of general matrix multiplication in at least one of their layers [28]. If we consider a function $x(t)$, convolutions can be understood as a an operation $(w * x)$ that describes how one function $x(t)$ is affected by a second function $w(t)$. Intuitively, we can think of $x(t)$ as our input and $w(t)$ as some weighting for our input $x(t)$, where the weights $w(t)$ are the weights of our neural network. Explicitly, convolution for one-dimensional inputs where both functions are a function of time t, produces a function s [28]:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)\, w(t-a) \tag{2.13}$$

Note that if $w(t)$ is a weighted average function, convolution resembles *Bayesian smoothing*.

Convolutions can also be extended to two-dimensional inputs. In this case, we normally refer to the first argument as input I, and the second as a *filter* or *kernel* K. This gives a *feature map* S:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)K(i-m, j-n) \tag{2.14}$$

Though in practice, we actually use *cross-correlation* to represent convolution [28]:

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i+m, j+n)K(m,n) \tag{2.15}$$

This operation is perhaps most familiar to us, where we recognise that for a kernal K of size $m \times n$, the output at index (i,j) is given by a simple matrix multiplication of the kernal with a specific region of the input. This is also how [38] represents convolution:

$$\boldsymbol{y}_l = \boldsymbol{W}\boldsymbol{x}_l + \boldsymbol{b}_l \tag{2.16}$$

where $\boldsymbol{x}$ is an $m \times n$-by-1 vector that represents co-located $m \times n$ pixels, and $\boldsymbol{W}_l$ is a d-by-$m \times n$ matrix where d represents the number of kernels for convolution.

Deconvolution

Mathematically, deconvolution is defined as the inverse of convolution. However in the literature, deconvolution is also used to describe a series of *convolution-unpooling (convolution-upsampling)* operations [37] or *transposed convolutions*[1]. For the purpose of this thesis, we focus on the use of transposed convolutions to represent deconvolution.

Transposed convolutions are designed by swapping the forward and backward passes of a

[1]The authors of [39] provide a very tidy visualisation of transposed convolutions at: `https://github.com/vdumoulin/conv_arithmetic`

convolution. As mentioned, the forward pass of a convolution operation can be expressed as the matrix multiplication of a set of co-located pixels $\boldsymbol{x}_l$ with a kernel $\boldsymbol{W}_l$. When calculating the gradient of $\boldsymbol{y}_l$ w.r.t. the kernel $\boldsymbol{W}_l$ in backpropagation, this becomes instead a multiplication of the feature map $\boldsymbol{y}_l$ with the transposed kernel $\boldsymbol{W}_l^\top$. So, in another sense, transposed convolutions can be thought of as a function applied on a feature map such that its output is the initial input used to create the initial feature map [39]. Therefore, transposed convolution layers make use of this idea as a sort of pseudo-inverse of convolution and allows it to also recover the shape of the input.

The Encoder and Decoder of a Convolutional VAE

As mentioned earlier, convolutional VAEs are comprised of a CNN as an encoder and some deconvolution network that serves as the decoder. Here, the CNN uses the convolution operation in place of a full matrix multiplication, while the deconvolution network applies transposed convolutions as a pseudo-inverse to the convolution operation. Apart from this technical distinction, convolutional VAEs are optimised in exactly the same way as fully-connected ones – using the loss specified in (2.9) and (2.10) – such that the CNN parametrises the approximate posterior distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})$, while the deconvolution network parametrises the likelihood distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})$.

2.2.4 A Practical Note on the VAE Reconstruction Loss

When implementing the VAE loss function (2.9), we can represent the reconstruction loss either through the *binary cross-entropy (BCE)* or *mean-squared error (MSE)*, though these are not entirely equivalent. The reason for this is that both loss functions are actually founded in MLE and minimising them is also equivalent to minimising the negative log-likelihood of our data $p_{\text{data}}(\boldsymbol{x})$ given our model $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ [28]. The main difference for their use lies in the what we assume the distribution of our data $p_{\text{data}}(\boldsymbol{x})$ to be, as this assumption lays the ground for how we optimise for $\boldsymbol{\theta}$. To see how we can extend the theory from Section 5.5 of [28] to two fundamental examples.

Essentially, in MLE we define the conditional maximum likelihood estimator $\boldsymbol{\theta}$ for predicting observed targets $\boldsymbol{y}^{(i)}$ from samples $\boldsymbol{x}^{(i)}$ to be:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}) \tag{2.17}$$

where m is the number of samples available. When assuming $p_{\text{data}}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)})$ follows a Gaussian distribution $\mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}^{(i)}, \sigma)$, where $\hat{\boldsymbol{y}}^{(i)}$ is the predicted mean for a Gaussian distributed $\boldsymbol{x}^{(i)}$ (with fixed σ for simplicity), we can rewrite the log-likelihood based on the well-known Gaussian

probability distribution function (PDF):

$$\sum_{i=1}^{m} \log p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) = m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||}{2\sigma^2} \tag{2.18}$$

From this we can identify the similarity with the MSE loss:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{m} \sum_{i=1}^{m} \frac{||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||}{2\sigma^2} \tag{2.19}$$

Alternatively, if we assume that our data-generating distribution $p_{\text{data}}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)})$ follows a Bernoulli distribution $\text{Ber}(\boldsymbol{y}^{(i)}; \hat{p}^{(i)})$, with $\boldsymbol{y}^{(i)} \in \{0, 1\}$ as the observed class and $\hat{p}^{(i)} \in (0, 1)$ as the predicted probability of $\boldsymbol{y}^{(i)}$ in class 1, its less-known probability mass function (PMF) is given by:

$$p_{\boldsymbol{\theta}}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}) = \hat{p}^{(i)^{\boldsymbol{y}^{(i)}}} (1 - \hat{p}^{(i)^{1-\boldsymbol{y}^{(i)}}}) \tag{2.20}$$

Then, we can rewrite the maximum likelihood objective in (2.17) using this:

$$\sum_{i=1}^{m} \log p_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = \sum_{i=1}^{m} \log \left(\hat{p}^{(i)^{\boldsymbol{y}^{(i)}}} \left(1 - \hat{p}^{(i)^{1-\boldsymbol{y}^{(i)}}} \right) \right) \tag{2.21}$$

$$= \sum_{i=1}^{m} \boldsymbol{y}^{(i)} \log \hat{p}^{(i)} + (1 - \boldsymbol{y}^{(i)}) \log(1 - \hat{p}^{(i)}) \tag{2.22}$$

From which we identify the BCE loss:

$$\mathcal{L}_{\text{BCE}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{y}^{(i)} \log \hat{p}^{(i)} + (1 - \boldsymbol{y}^{(i)}) \log(1 - \hat{p}^{(i)}) \tag{2.23}$$

So from these examples, we see that *BCE* is generally used when we assume our data is *Bernoulli distributed*, e.g. a (black-white) pixel value is either 0 or 1, while *MSE* is used when we assume that our data is *Gaussian distributed*, e.g. the heights of students in Trondheim.

To apply this to VAEs, since we wish to optimise the parameters of our generative network $\boldsymbol{\theta}$ to reconstruct our input $\boldsymbol{x}^{(i)}$, we replace the observed targets $\boldsymbol{y}^{(i)}$ with our input $\boldsymbol{x}^{(i)}$, while the reconstructions $\boldsymbol{r}^{(i)} \sim p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$ serve as the model prediction $\hat{\boldsymbol{x}}^{(i)}$. The choice for the loss function, however, is unclear. Whether one chooses to assume that the data-generating distribution $p_{\boldsymbol{\theta}^*}(\boldsymbol{x} \mid \boldsymbol{z})$ with unknown parameters $\boldsymbol{\theta}^*$ is best approximated through a family of Gaussians or family of Bernoulli distributions is left as a design choice, as we have no indication of the true shape of its distribution. Though as consolidation, the optimal parameters $\boldsymbol{\theta}$ for our model $p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid \boldsymbol{z})$ are the same no matter which loss function is used, only the loss values are different [28].

2.3 Reinforcement Learning

Reinforcement learning is a learning-based method for discovering complex behaviour without explicit programming. It allows an agent to capture the dynamics of a system and the uncertainties of the environment through a data-driven approach, using a high-abstraction, evaluative feedback or *reward* [17].

To get an overview of the basics of reinforcement learning, we can begin by looking into its definition and goal. Sutton and Barto [40] states, “reinforcement learning is learning what to do – how to map situations to actions – so as to maximise a numerical reward signal.” Further, [17] adds, “reinforcement learning enables a robot to autonomously discover an optimal behaviour through trial-and-error interactions with its environment." From these, there are many terms that could be examined, such as, *situations*, *actions*, *rewards*, *behaviour*, *trial-and-error interactions* and *environment*. We can ask ourselves, “what exactly is an optimal behaviour and how to we express this?”. Throughout this section, we will cover these central concepts, where in the end, we will explore how *temporal-difference learning*, *policy-gradient methods*, *actor-critics* and *proximal policy optimisation* algorithms enable us to discover an optimal behaviour.

As reinforcement learning was the central theme of the project thesis, much of the fundamental theory in the following sections is shared with that in the project [1]. As a result, many of the following subsections are taken from the project thesis, being only partially rewritten and marked with an asterisk (*).

2.3.1 Finite Markov Decision Processes*

Finite Markov Decision Processes (MDPs) are a way of formalising how an agent interacts with the environment, serving as a standardised learning framework for reinforcement learning. MDPs are often depicted as an iterative diagram as shown in Figure 2.5, comprising of five elements: the *agent*, *environment*, *state* S_t, *action* A_t and *reward* R_t. MDPs are an extension of

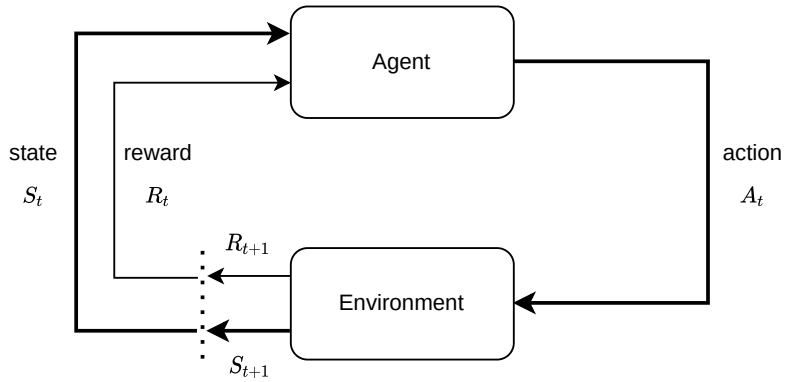


Figure 2.5: The interaction between agent and environment in an MDP, from [40].

stochastic Markov chains, where state transitions from S_t to S_{t+1} are now influenced by a choice in action A_t, and each transition also yields a reward R_t.

To go into detail, an *agent* has a task of *learning* how to solve a specific task, whereby improving its performance through *experience* – simply a set of repeated state-action interactions with the environment. The *environment* contains the task to be solved and essentially everything an agent interacts with, for example, the system dynamics and the rewards for being in certain states. From a cybernetics perspective, the agent serves as a controller while the environment can be understood as the plant or process to be controlled. The situations that an agent finds itself in is referred to as *states* or *observations*. Each of the states in an MDP also has the Markov Property, which means that all the information in state S_t is only dependent on the previous state S_{t-1} and action A_{t-1}. For any state $S_t \in \mathcal{S}$, the agent can take an *action* $A_t \in \mathcal{A}(s)$, where $\mathcal{S}$ represents the state-space or set of possible states and $\mathcal{A}(s)$ the action-space or set of possible actions for S_t. Actions generally refer to anything that transitions an agent into a new state and may vary largely from task to task. An example of this could be a control signal from a controller, such as the torques for each rotor on a quadrotor, though it could also be reference signals from an optimal controller, such as velocity references for a controller to follow.

Moreover, the states that an agent finds itself is determined by the *initial state distribution* $p(s)$, while the states the agent moves to follows the *state-transition distribution* $p(s', r \,|\, s, a)$. The state-transition distribution is a function of four arguments that captures the dynamics of the MDP and describes the probability of a specific transition within the environment [40]:

$$p(s', r \,|\, s, a) = P(S_t = s', R_t = r \,|\, S_{t-1} = s, A_{t-1} = a) \tag{2.24}$$

Here, S_t, R_t and A_t are random variables with well-defined probability distributions while the lower case letters s, r and a refer to specific values of these variables. The value s' is commonly used to denote the value of the next state from s.

Lastly, we see that for each state transition an agent receives a numerical *reward* $R_t \in \mathcal{R} \subset \mathbb{R}$, which can be interpreted as an evaluative feedback for a choice of action A_t. The reward is commonly a function of the current state and action values, $R(s, a)$, and is often the main tool used to shape agent behaviour in the environment [17].

2.3.2 Returns*

Now that we have formalised agent-environment interactions, we can imagine that by trying enough actions in different states, the agent will discover an optimal behaviour - essentially choosing the "best" action for every possible state $S_t \in \mathcal{S}$. Yet, what exactly is the "best" action for each state S_t? Is it the state-action pair with the highest reward R_t?

To answer this, we can start by defining the goal of reinforcement learning. As mentioned at the beginning of the section, reinforcement learning aims to, informally, "maximise a numerical

reward signal". The reward signal, in this case, is a scalar received every time step at each new state and serves as an immediate feedback for the agent's action. Yet, if we consider the formulation of MDPs more carefully, taking a certain action in a particular state does not only affect the state the agent transitions to in the next time step, but also all consequent states and rewards for all following time steps. This illustrates the concept of *delayed reward*, which suggests that receiving a high reward now does not necessarily mean receiving a high reward later.

Therefore, it is important to define the goal of reinforcement learning in the context of reward more precisely. As such, the return G_t is defined as a function of a specific sequence of rewards, R_{t+1}, R_{t+2}, R_{t+3} .... For example, the simplest return can be defined as the sum of the reward sequence [40]:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.25}$$

With this, we generally seek to maximise the *expected* return over some time horizon:

$$J = \mathbb{E}[G_t] = \mathbb{E}\left[\sum_{k=t+1}^{T} R_k\right] \tag{2.26}$$

Here, T represents the time of termination and is a random variable that normally varies with each *episode*. An episode in this case refers to a sequence of timesteps for which an agent is performing a task until the agent reaches a *terminal state*.

However, in control tasks, we see that there is often no defined terminal state but rather an indefinite continuing process, such as in process control. Therefore, it is common to instead maximise the *discounted return* [40]:

$$J = \mathbb{E}[G_t] = \mathbb{E}\left[\sum_{k=0}^{T} \gamma^k R_{k+t+1}\right] \tag{2.27}$$

where $\gamma^k \in [0, 1)$ is referred to as the *discount factor* – an exponentially decreasing weight on *future rewards*, often chosen to be 0.99. The discount factor ensures that for any timestep t the infinite sum of rewards is finite, allowing the agent to evaluate returns that are defined for each time step. Another interesting property that should be noted – and will be important for concepts later – is the recursive expression for the return G_t:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} ... \\ G_t &= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} ...) \\ G_t &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.28}$$

So, with the goal of reinforcement learning defined more clearly in terms of maximising the

expected discounted return, we now need to have a formal definition for the agent behaviour before we can optimise it.

2.3.3 Policies and Value Functions*

First, the behaviour that an agent learns is referred to as a policy π. Policies define a mapping of states to probabilities of selecting each possible action, $\pi(s \mid a) = P(a \mid s)$ [40]. Despite this, a policy π can also be deterministic, resulting in the same action for a state every time, written as $a = \pi(s)$ [17].

Initially, we can imagine that policies are quite random and non-idea. Then throughout the learning process, the agent will receive rewards R_t, accumulate returns G_t, and update its policy π consequently. So, when we think about an optimal policy π^*, we imagine an agent performing the actions that result in the best result. Hence, we can say that in order to solve the reinforcement learning problem, we need to "solve" the MDP by finding this optimal policy π^*.

The method forward is to define a *state-value function* V^π, which indirectly tells us how good it is to be in a particular state. For MDPs, the state-value function can be defined as the expected return for being in a state s and following a policy π thereafter [40]:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} \,\middle|\, S_t = s\right] \forall s \in \mathcal{S} \tag{2.29}$$

Similarly, we can define an *action-value function* (or *Q-function*), as the expected return for being in a state s and taking an action a, and following a policy π thereafter [40]:

$$Q^\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^t R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{2.30}$$

In other words, these value functions represent the total reward you can expect by following a policy π (e.g. until the end of an episode), from a particular state s. Informally, the state-value function is simply referred to as the value function $V(s)$, and is how we refer to it throughout this thesis. The difference between the value function V and the Q-function is that the value function gives the expected return for a state s assuming the agent takes the action a decided by π, whereas the Q-function evaluates the expected return for different choices of actions a, at a state s.

A fundamental property of value functions is that they also possess the recursive property

similar to that of returns in (2.28):

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}[G_t \,|\, S_t = s] \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s] \\ &= \sum_a \pi(a\,|\,s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_{\pi}[G_{t+1} \,|\, S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a\,|\,s) \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma V^{\pi}(s') \Big] \end{aligned} \tag{2.31}$$

as shown in [40]. First, we expand according to (2.28). Then, we expand the expectation to R_{t+1} and G_{t+1}, and use the definition of the expectation. This yields a sum over the possible actions, next states and rewards for a certain state, where the return for that outcome is weighted with its probability. The weight for each possible outcome is given by the product of the probability of selecting action a given s, $\pi(a\,|\,s)$, and transitioning to state s', or $p(s', r\,|\,s, a)$. Finally, we recognise the value function term for the next state s'. Equation (2.31) is also referred to as the *Bellman equation for* V^{π} [41].

2.3.4 Optimal Policies and Value Functions*

Optimal policies can be defined as a policy π whose expected return is greater than or equal to all other policies π' $\forall s \in \mathcal{S}$ [40]. By the theory of [41], we know that there is at least one optimal value function that follows π^*. We denote an optimal value function as:

$$V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s) \tag{2.32}$$

while an optimal Q-function is denoted:

$$Q^{\pi^*}(s, a) = \max_{a} Q^{\pi}(s, a) \tag{2.33}$$

This optimal Q-function is defined as the expected return of a state s and taking an action a, then following the optimal policy thereafter:

$$Q^{\pi^*}(s, a) = \max_{\pi} \mathbb{E}\Big[R_{t+1} + \gamma\, V^{\pi^*}(S_{t+1}) \,\Big|\, S_t = s, A_t = a \Big] \tag{2.34}$$

Going further, we see that the optimal value function is identical to the optimal Q-function when taking the best action:

$$V^{\pi^*}(s) = \max_{a \in \mathcal{A}(s)} Q^{\pi^*}(s,a) \tag{2.35}$$

$$= \max_a \mathbb{E}\left[R_{t+1} + \gamma V^{\pi^*}(S_{t+1}) \,\middle|\, S_t = s,\ A_t = a\right]$$

$$= \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma V^{\pi^*}(s')\Big] \tag{2.36}$$

where the last step comes from the definition of the expectation, similar to (2.31). Following the same reasoning for the action-value function Q, we get:

$$Q^{\pi^*}(s,a) = \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma \max_{a'} Q^{\pi^*}(s', a')\Big] \tag{2.37}$$

which comes from using (2.36) and inserting (2.35) for the optimal value function. Equations (2.36) and (2.37) are known as the *Bellman optimality equations*.

2.3.5 Temporal-Difference Learning

Temporal-difference (TD) learning is a method used for solving the Bellman Optimality Equations. It exists at the intersection between dynamic programming and Monte Carlo methods, using a combination of both ideas in order to effectively estimate the value functions $V(s)$ or $Q(s,a)$.

A Brief Note on Dynamic Programming and Monte Carlo Methods

The Bellman optimality equations are essentially a set of nonlinear equations, one for each state in an MDP. If we had full access to the state-transition dynamics p, we could be able to solve the whole MDP through *dynamic programming* (DP), essentially iterating the state space multiple times and improving our estimate of the value functions according to (2.36), seen in the update rule (2.38) for the *policy iteration* algorithm:

$$v_{k+1}(s) = \mathbb{E}_\pi\left[R_t + \gamma v_k(S_{t+1}) \,|\, S_t = s, A_t = \pi(s)\right]$$

$$= \sum_{s',r} p\big(s', r \,|\, s, \pi(s)\big)\big[r + \gamma v_k(s')\big] \tag{2.38}$$

or the update rule (2.39) for *value iteration* [40]:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}\big[R_t + \gamma v_k(S_{t+1}) \,|\, S_t = s, A_t = a\big] \\ &= \max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma v_k(s')\big] \end{aligned} \tag{2.39}$$

After iterating enough times and find the optimal value function $V^*(s)$, the policy π would then reduce to a greedy strategy where we choose the action a, that yields the highest expected return for a state s [40]:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V^*(s')\big] \tag{2.40}$$

However, one of the problems that exist for DP methods is that we often only have an imperfect model of our system and lack the knowledge of the state-transition dynamics for this system. Hence, the task of learning the value function through (2.36) is no longer possible, as the state-transition dynamics p is no longer available. We normally refer to these problems of *incomplete knowledge* as *model-free* problems, where model-free methods do not rely on a priori information in the form of transition dynamics and the reward structure of an MDP.

In these types of problems, methods will have to instead rely on *sampling* to estimate the value function $V(s)$ for a given state, based on the return it achieves after visiting that state. The difference with DP is that in *model-based* problems, having access to p allows us to update $V(s)$ considering the expected return for all possible next states, as we see in (2.36). On the other hand, Monte Carlo methods have to instead visit each individual state in order to *sample* the return before it can update the value function. This difference is also seen through each method's *backup diagram* as in Figure 2.6.

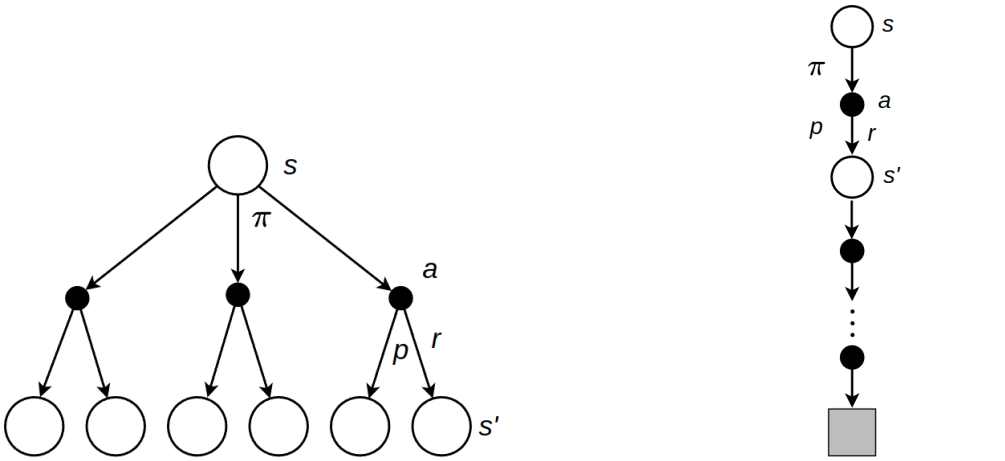
By sampling the return, Monte Carlo methods are then able to update their value function $V(S_t)$ – the expected return for being in a state S_t – using the sampled return from that state:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G - V(S_t)\Big] \tag{2.41}$$

where α is some step-size parameter.

TD Value Prediction

So, to generate an estimate for the value function $V(s)$, we saw that DP methods use one-step updates for $V(s)$ while Monte Carlo methods update $V(s)$ for each state only at the end of an episode. TD methods are similar to Monte Carlo methods since they sample states to update $V(s)$. However, instead of waiting until the end of an episode, TD methods update $V(s)$ at every



(a) DP methods **(b)** Monte Carlo methods

Figure 2.6: The backup diagrams for $V(s)$ for DP and Monte Carlo methods. These diagrams show how information transfers back to a state from its successor states. DP methods update $V(s)$ using information from one-step transitions, while Monte Carlo samples a return G from an entire episode, stopping only at a terminal state. [40].

timestep, similar to DP methods. This is seen more clearly in the simplest TD update [40]:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{2.42}$$

where the value estimates of consecutive timesteps are used as an update rule. The backup diagram of this is shown in Figure 2.7. The idea of updating an estimate through another es-

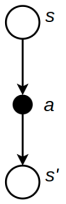


Figure 2.7: Backup diagram for TD methods. Value estimates for state s, $V(s)$, are bootstrapped to $V(s')$.

timated value is called *bootstrapping*, where in this case, we say that the current state value estimate $V(S_t)$ is bootstrapped to the next-state value estimate $V(S_{t+1})$. Bootstrapping is at the core of DP methods, where we see that the Bellman optimality equations in (2.36) and (2.37) also follow this bootstrapping form [40]. The term in the brackets in (2.42) is also referred to as the *TD-error* δ_t:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.43}$$

which will be important later in Section 2.3.10.

To summarise, the point of the update rules for $V^{\pi}(s)$ in equations (2.38), (2.39), (2.41) and (2.42), is that we wish that our value function reaches the optimal value function $V^{\pi}(s) \rightarrow V^{\pi^*}(s)$, after enough updates. This is similar to any optimisation problem, where we wish to minimise the error between our current value estimate $V^{\pi}(s)$ and some target value estimate $V^{\pi^*}(s)$. For TD learning, this error is the TD-error (2.43), where the target value is the estimate $R_{t+1} + \gamma V(S_{t+1})$, which exploits the recursive nature of the optimal value function given by the Bellman optimality equations in (2.36), similar to DP methods. For Monte Carlo methods, we can see that the target value for the expected return $V(s)$ is the sampled return G in (2.41); and since the sampled return G is not an estimate per se, Monte Carlo methods also do not bootstrap.

2.3.6 Exploration versus Exploitation

Now that we have seen how agents can learn optimal value functions, it is important to specify that model-free methods more commonly rely on learning the action-value function $Q(s,a)$ instead of $V(s)$. The reason is that learning the Q-function allows the agent to solve the reinforcement learning problem by simply selecting the action with the greatest return, without needing to consider possible next-states or the dynamics of the environment [40].

However, learning the Q-function in sampling-based methods requires that we specify what action to take in our target Q-function estimate. Unlike model-based methods, sampling-based methods require an adequate exploration of all actions $a \in \mathcal{A}(s)$ in a given state s in order to achieve an accurate state-action value $Q^{\pi}(s,a)$ for that state. However, the goal of the reinforcement agent is also to maximise its expected return, which means to *greedily* choose actions a as shown in (2.40) in value iteration. The question of whether and when an agent should exploit its current knowledge of the value of actions, or attempt to explore actions that might yield a higher return in the long run, is referred to as *exploration versus exploitation* [40]. For TD learning, handling this question brings us to the topic of *on-policy* and *off-policy* methods.

2.3.7 On-policy and Off-policy methods*

Generally speaking, the difference in on-policy and off-policy methods lies in the action-value function update step, specifically how the current action value estimate is bootstrapped to the action value estimates of consecutive states. The best way to visualise this is through the use of an example, where we will look at two fundamental methods, *SARSA* [40] and *Q-learning* [42]:

On-policy – The update step for SARSA:

$$Q^{\pi}(S_t,A_t) \leftarrow Q^{\pi}(S_t,A_t) + \alpha\Big[R_{t+1} + \gamma Q^{\pi}(S_{t+1},A_{t+1}) - Q^{\pi}(S_t,A_t)\Big] \tag{2.44}$$

Off-policy – The update step for Q-learning:

$$Q^{\pi}(S_t, A_t) \leftarrow Q^{\pi}(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q^{\pi}(S_{t+1}, a) - Q^{\pi}(S_t, A_t) \right] \tag{2.45}$$

In both cases, the policy π is an arbitrary policy, e.g. choosing a random action with ϵ probability and the max action with probability $1-\epsilon$ (ϵ-greedy). SARSA is dubbed an *on-policy* algorithm as the next state-action value estimate is based on *exactly the same policy* as the agent's current one. This means that when updating the current state-action value pair, the value of the next state-value pair is evaluated to be the expected return $Q^{\pi}(S_{t+1}, A_{t+1}$ from a state S_{t+1} after taking an action A_{t+1} under the *same behaviour* policy π. It can also be said that the *target policy* for on-policy methods is the same as its current policy, if we think of the update as an error between the current and target policies.

In contrast, Q-learning uses an update step of $\max_a Q^{\pi}(S_{t+1}, a)$, while its behaviour policy could be for example, ϵ-greedy. As a result, the actions chosen in the subsequent states will always be *greedy* choices and *not actions chosen under its behaviour policy* π, which was only greedy with probability $1-\epsilon$ (ϵ-greedy). So, methods that bootstrap current value estimates (under behaviour policy π) to estimates derived from a *different* target policy π' are referred to as off-policy methods. In these cases, the behaviour policy is often denoted as β, while the target policy can be denoted as $\pi(a|s)$ if stochastic or $\pi(s)$ if deterministic.

Going back to the question of exploration versus exploitation, the result of updating $Q^{\pi}(S_t, A_t)$ with a greedy target policy, like in off-policy Q-learning, is that the learned policy is more likely to suggest taking actions that follow this "optimal", greedy trajectory, rather than exploring the action space more in order to find another more optimal approach. Off-policy methods try to overcome this by explicitly choosing a behaviour policy β that is exploratory by nature, for example, a random policy or one with added noise. The after-effect of choosing a highly exploratory action is that many obvious "bad" actions could be taken – unnecessarily slowing learning. This is especially true when state and action spaces are large, as we want the agent to explore actions close to an optimal trajectory rather than waste time exploring actions far from the optimal solution. In contrast, on-policy methods avoid this issue by not updating estimates with greedy strategies in the first place, though they still have the potential to suffer from insufficient exploration. A reverse problem for on-policy methods is that, by definition, they do not have a behaviour policy that they can specify explicitly. Hence, they are limited to defining a target policy that incorporates some degree of randomness so as to prevent converging to some locally optimal solution.

2.3.8 An Extension to Continuous Control*

So far, the methods that we have seen have only been applicable to MDPs; *tabular solution methods* like DP, Monte Carlo, and TD learning only work in environments with a relatively small, finite set of *discrete* states and actions [40]. However, for most of the interesting control problems in cybernetics, we deal with *continuous* state and action spaces. In these problems, a simple question of how one should represent the state space $\mathcal{S}$ or the action space $\mathcal{A}(s)$ can quickly become challenging. For example, the discretisation of these continuous spaces has a severe limitation: namely the *curse of dimensionality*, where the number of both state and action combinations grows exponentially by the number of degrees of freedom [41]. Thus, methods that rely on finding the maximum action in order to update its action-value function, such as in (2.45) in Q-learning, will then require an iterative optimisation process at each step due to the large set of possible actions [43], which is impractical if not infeasible.

As a result, these high-dimensional continuous state and control problems strictly restricts us to finding an only *approximate* solution for the policy and value functions, which motivates the use of *approximate solution methods* – methods that focus on *generalisation through function approximation* [40]. This restriction is also a motivation for the relevance of *policy search*, as policies require less representational power than a value function approximation [44] and can frankly, just be simpler to approximate [40]. So, finding an adequate parametrisation of these functions has also become a key focus in reinforcement learning in recent years. Fortunately for us, there has been a clear parametrisation of choice for both value function and policy in recent years, through the use of *neural networks* (NNs) as universal function approximators, initially inspired by the success of *Deep Q-Networks* (DQNs) [45] and later, *Deep Deterministic Policy Gradients* (DDPG) [43]. In the literature, we refer to all methods that learn a parametrised policy $\pi_{\boldsymbol{\theta}}(a \,|\, s)$, through the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to parametrised policy parameters $\boldsymbol{\theta}$ as *policy gradient methods*.

2.3.9 Policy Gradient Methods

Policy gradient methods are a form of policy search, where we have a vector of d parameters $\boldsymbol{\theta} \in \mathbb{R}^d$ that parametrises our policy π:

$$\pi_{\boldsymbol{\theta}}(a \,|\, s) = P\left(A_t = a \,|\, S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\right) \tag{2.46}$$

Policy gradient methods have a goal of optimising a certain objective function $J(\boldsymbol{\theta})$, with respect to these parameters $\boldsymbol{\theta}$. A typical objective function could be the expected return at a particular state under the parametrised policy $\pi_{\boldsymbol{\theta}}$, or simply the value function (2.31):

$$J(\boldsymbol{\theta}) = V^{\pi_\theta}(s) \tag{2.47}$$

As the objective function also depicts a performance measure that we wish to maximise, rather than a loss, we aim to maximise it through *gradient ascent* [40]:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)} \tag{2.48}$$

Here, the gradient term of the objective $J(\boldsymbol{\theta})$, with respect to the policy parameters $\boldsymbol{\theta}_t$, is represented by $\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}$ and is a stochastic estimate. So, we describe all methods that use a parametrisation of the policy π as policy gradient methods, and what normally varies is how we define the objective function $J(\boldsymbol{\theta})$.

If we use the value function as the performance measure as in (2.47), we obtain the key result in policy gradient methods, the *policy gradient theorem*, where we can express the gradient of the objective function, with respect to the policy parameters $\boldsymbol{\theta}$ as [40]:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s p(s) \sum_a Q^{\pi}(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid s) \tag{2.49}$$

$$= \mathbb{E}_{\pi, S_t \sim p(s)} \left[\sum_a Q^{\pi_\theta}(S_t, a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a \mid S_t) \right] \tag{2.50}$$

where $p(s)$ is the *on-policy state distribution*, as mentioned in Section 2.3.1, which represents the probability distribution of states under π, and can be thought of as the fraction of time spent at a state s, where $\sum_s p(s) = 1$ [40]. Then, as $p(s)$ serves as a weighting for states s, we see that (2.49) can be simplified to just an expectation in (2.50).

The policy gradient theorem in (2.50) is a central result in reinforcement learning because it summarises how a parametrised policy $\pi_{\boldsymbol{\theta}}(a \mid s)$ can be optimised without the need for any model information. Intuitively, the policy performance should depend on both the probability of actions and the distribution of states for which the actions are taken in, i.e. the state distribution p – though this is impossible to know in a model-free setting. Nevertheless, we see that the gradient does not depend on the state distribution p but only on the expected return and the gradient of the policy parameters.

As a result, algorithms have been derived that attempt to estimate this expectation through a sample-based approach, though a question for policy gradient methods has been on finding a good sample for $Q^{\pi_\theta}(s,a)$ [46]. The issue is that the sampled returns for episodes can vary greatly, leading to high-variance updates in the policy space that can lead to convergence difficulties. This makes it difficult to simply use the sampled return as an estimate for $Q^{\pi_\theta}(s,a)$ [40]. Thus, as an extension to mend this problem, we have actor-critic methods.

2.3.10 Actor-Critics*

Actor-Critic methods follow the same idea as policy gradient methods but also include a parametrisation of the action-value function $Q(s,a)$. In other words, these methods aim to concur-

rently learn a policy $\pi(a\,|\,s)$ and the action-value function $Q(s,a)$. The *actor* refers to the parametrisation of the policy $\pi(a\,|\,s)$ through $\boldsymbol{\theta}$, while the *critic* refers to the parametrisation of the action-value function $Q(s,a)$ through a vector $\boldsymbol{w}$ [40]. The critic could also parametrise the value function $V(s)$ instead, as in PPO [47].

Updating the Actor Parameters

To make use of this critic, we incorporate it into the update step for the actor in the form of a *baseline*, $b(s)$. By comparing the estimate to this baseline, we can reduce the variance of the actor updates significantly and beneficially [40]. A simple baseline can be added to the gradient as so:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \sum_s p(s) \sum_a \Big(Q^{\pi_\theta}(s,a) - b(s)\Big) \nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}\,(a\,|\,s) \tag{2.51}$$

However, to get this expression into the form we desire, we have to simplify the expression into just an expectation, similarly to (2.50). This is done by adding the missing weight for each actions, which is the stochastic policy $\pi(a|s)$:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi, S_t \sim p(s)} \left[\sum_a \pi_{\boldsymbol{\theta}}\,(a\,|\,S_t) \Big(Q^{\pi_\theta}(S_t,a) - b(s)\Big) \nabla_{\boldsymbol{\theta}} \frac{\pi_{\boldsymbol{\theta}}\,(a\,|\,S_t)}{\pi_{\boldsymbol{\theta}}\,(a\,|\,S_t)} \right] \\
&= \mathbb{E}_{\pi, S_t \sim p(s), A_t \sim \pi} \left[\Big(Q^{\pi_\theta}(S_t,A_t) - b(s)\Big) \frac{\nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)}{\pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)} \right] \\
&= \mathbb{E}_{\pi, S_t \sim p(s), A_t \sim \pi} \left[\Big(G_t - b(s)\Big) \frac{\nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)}{\pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)} \right]
\end{aligned} \tag{2.52}$$

Here, G_t is the return with the same expectation as the $Q^{\pi_\theta}(S_t,A_t)$ value. Yet, two observations should be made to this result: first, this gradient assumes we can sample the return (like a Monte Carlo method), and second, the baseline is strictly not a critic in the fact that it does incorporate information from the consecutive time steps [40]. Thus, there is one more step that has to be taken in order to achieve the result we desire. To both bypass the need to sample the return and be considered an actor-critic method, actor-critics borrow ideas from TD learning by bootstrapping its action value estimate to the next-state action value estimate:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi, S_t \sim p(s), A_t \sim \pi} \left[\delta_t \frac{\nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)}{\pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)} \right] \tag{2.53}$$

$$\delta_t = R_{t+1} + \gamma Q^{\pi_\theta}_{\boldsymbol{w}}(S_{t+1},A_{t+1}) - Q^{\pi_\theta}_{\boldsymbol{w}}(S_t,A_t) \tag{2.54}$$

where we recognise δ_t as the TD error from (2.58). So finally, the actor update can be represented as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha^{\boldsymbol{\theta}} \delta_t \frac{\nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)}{\pi_{\boldsymbol{\theta}}\,(A_t\,|\,S_t)} \tag{2.55}$$

Updating the Critic Parameters

As for the critic, we aim to take a step in the direction that reduces the error between the approximate value $Q^{\pi_\theta}(s,a \mid \boldsymbol{w})$ and the true value $Q^{\pi_\theta}(s,a)$. This is known as the *Mean Squared Value Error*, $\overline{VE}$ [40]:

$$\overline{VE}(\boldsymbol{w}) = \mathbb{E}_{\pi, S_t \sim p(s), A_t \sim \pi}\Big[\big(Q^{\pi_\theta}(S_t, A_t) - Q^{\pi_\theta}_{\boldsymbol{w}}(S_t, A_t)\big)^2\Big] \tag{2.56}$$

The way forward is quite similar to the value prediction methods we have seen before, where we have to choose how to represent our target $Q^{\pi_\theta}(s,a)$. We can choose this to be the Monte Carlo sample G_t, though we instead choose it to be the bootstrapping target $R_{t+1} + \gamma Q^{\pi_\theta}_{\boldsymbol{w}}(S_{t+1}, A_{t+1})$.

Then, we can minimise this error through stochastic gradient descent, where we take the gradient of the $\overline{VE}$ with respect to the parameters $\boldsymbol{w}$. This gives us the update rule in *Semi-gradient TD(0)*:

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \alpha^{\boldsymbol{w}} \delta_t \nabla_{\boldsymbol{w}} Q^{\pi_\theta}_{\boldsymbol{w}}(S_t, A_t) \tag{2.57}$$

$$\delta_t = R_{t+1} + \gamma Q^{\pi_\theta}_{\boldsymbol{w}}(S_{t+1}, A_{t+1}) - Q^{\pi_\theta}_{\boldsymbol{w}}(S_t, A_t) \tag{2.58}$$

with δ_t as the TD-error. So, by using a "bootstrapping critic", we can introduce a bias – injecting information based on the assumption that our critic should be in a value function form, i.e. it follows a recursive nature with optimal form as (2.37). Moreover, this allows for every-step updates as mentioned in Section 2.3.5, and, "typically enables significantly faster learning" of the value function [40].

Lastly, compared to Monte Carlo policy gradient methods that sample the return, we also see that by taking the TD-error in the actor gradient, the same benefit of reduced variance of gradient updates and accelerated learning is applied [40].

2.3.11 Proximal Policy Optimisation*

In light of the advancements within deep reinforcement learning from [45] and [43], a new family of methods was developed to curb the problem of instability and divergence when training agents using NNs. These are called *trust-region* based policy optimisation methods, stemming from Trust-region Policy Optimisation (TRPO) [48]. PPO [47] is heavily inspired by this, where the overarching idea is that in order to maintain stability in training, the new, updated policy should be within a specific *trust-region* of the old policy, hence the name *proximal* policy. As for its other characteristics, PPO is also a model-free, on-policy and actor-critic method, parametrising a stochastic policy $\pi(a \mid s)$ and a value function $V(s)$. Finally, it introduces a uniquely defined objective function to optimise this.

The Advantage Function

Earlier, we defined the policy gradient in (2.53). This can be implemented in practice as:

$$\widehat{\nabla_{\boldsymbol{\theta}} J_t(\boldsymbol{\theta})} = \hat{\mathbb{E}}_t \left[\frac{\nabla_{\boldsymbol{\theta}}\, \pi_{\boldsymbol{\theta}}(a \,|\, s)}{\pi_{\boldsymbol{\theta}}(a \,|\, s)} \hat{A}_t(s,a) \right] \tag{2.59}$$

where $\hat{\mathbb{E}}_t$ represents the empirical average over a finite batch of samples of actions a and states s. Also, we introduce a colloquial term $\hat{A}$, called the *advantage* function $A : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, that represents how well an action did compared to some *baseline* estimate:

$$\hat{A}(s,a) = \underbrace{Q(s,a)}_{\text{discounted return}} - \underbrace{V_{\boldsymbol{\theta}}(s)}_{\text{estimate}} \tag{2.60}$$

The baseline in this case is the parametrised value function $V_{\boldsymbol{\theta}}$. Note that the first term shows the rewards that was actually received, while the second is an estimate of what we expected to receive in that state – the critic. Hence, the advantage gives an idea of whether the action performed was better or worse than expected by the critic.

PPO-Clip Objective

PPO ensures that the new policy is close to the old one by using one of two tricks: clipping or an adaptive KL divergence penalty term. Primarily, the one that is used is the *clipped surrogate objective* version of PPO, which is also used in this thesis. In this version, the authors prevent substantial changes to the policy parametrisation by basically flattening the policy objective function to a certain maximum value. To visualise this, we can look at the novel objective function used.

PPO takes the surrogate objective function used by TRPO, whose gradient is equivalent to (2.59):

$$J_t(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\frac{\pi_{\boldsymbol{\theta}}(a \,|\, s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \,|\, s)} \hat{A}_t(s,a) \right] = \hat{\mathbb{E}}_t \left[r_t(\boldsymbol{\theta}) \hat{A}_t(s,a) \right] \tag{2.61}$$

and clips it by a maximum value bounded by a hyperparameter ϵ to obtain the *actor objective function*:

$$J_t^{CLIP}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\min\left(r_t(\boldsymbol{\theta}) \hat{A}_t(s,a),\, \text{clip}\left(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon\right) \hat{A}_t(s,a) \right) \right] \tag{2.62}$$

This clipping aims to remove the incentive from deviating more than ϵ away from the old policy [47] and can be visualised in Figure 2.8. Also, the ratio $r_t(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a \,|\, s)}{\pi_{\boldsymbol{\theta}_{\text{old}}}(a \,|\, s)}$ can be understood as the change in probability of selecting actions compared to the old policy, $\pi_{\boldsymbol{\theta}_{\text{old}}}(a \,|\, s)$, such that $r_t(\boldsymbol{\theta}_{\text{old}}) = 1$.

The way to understand the clipped surrogate objective is to first remember that we are performing gradient ascent in the objective function, w.r.t. the policy parameters $\boldsymbol{\theta}$. This means that

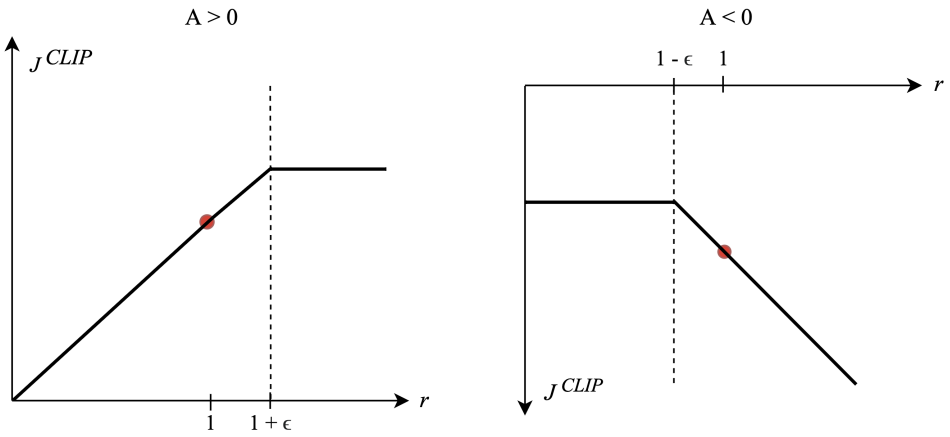


Figure 2.8: Visualising the clipped surrogate objective function for positive and negative advantages A. The figure is recreated from the original paper [47].

we essentially choose the direction in which $r_t(\boldsymbol{\theta})$ should move in (2.62) and Figure 2.8. When the agent performs better than expected, i.e. the advantage is positive, we wish to increase the probability of doing those actions again, which is equivalent to increasing $r_t(\boldsymbol{\theta})$. So, we adjust the parameters $\boldsymbol{\theta}$ such that the probability ratio $r_t(\boldsymbol{\theta})$ moves to the right. However, since we are taking the minimum and the objective is clipped at $1+\epsilon$, there is no added benefit of increasing $r_t(\boldsymbol{\theta})$ beyond this clipped point. Similarly, when the agent performs worse than expected, i.e. the advantage is negative, we wish to lower the probability of those actions happening again, which is equivalent to reducing the probability ratio $r_t(\boldsymbol{\theta})$. Again, since we clip the value at $1-\epsilon$ and the objective function is taking the minimum, there is no benefit of decreasing $r_t(\boldsymbol{\theta})$ beyond the clipped point. Therefore, by aiming to maximise the performance objective in (2.62) by gradient ascent, the authors manage to prevent the new policy from deviating too far from the old one, as seen by $r_t(\boldsymbol{\theta})$ being discouraged from moving beyond $[1-\epsilon, 1+\epsilon]$.

Actor-Critic Structure

With the key idea from PPO presented, we can delve into the actor-critic structure of PPO. As shown above, the policy gradient is based on the advantage term $\hat{A}_t$. As a result, PPO uses a parametrisation of the value function $V(s)$ as a critic, so to produce an estimate for the advantage $\hat{A}_t$ in (2.60). Similarly to before (2.56), the parametrised value function can be optimised by minimising the mean squared value error $\overline{VE}$:

$$J_t^{VF}(\boldsymbol{\theta}) = \mathbb{E}_t\Big[(V_{\boldsymbol{\theta}_t}(s) - V_t^{\text{targ}})^2\Big] \tag{2.63}$$

where the target value function V_t^{targ} that was implemented in the code of [47] is defined as:

$$V_t^{\text{targ}} = R_t + \gamma V_{\boldsymbol{\theta}_t}(s') \tag{2.64}$$

Lastly, there is an entropy term $S[\pi_{\boldsymbol{\theta}}](s)$ that is also added to the objective function, which serves as an exploration term.

So combined, the overall objective function for PPO with the actor objective function in (2.62), critic loss in (2.63) and entropy term, is:

$$J_t^{CLIP+VF+S}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t\Big[J_t^{CLIP}(\boldsymbol{\theta}) + c_1 J_t^{VF}(\boldsymbol{\theta}) + c_2 S[\pi_{\boldsymbol{\theta}}](s)\Big], \tag{2.65}$$

where c_1 and c_2 are coefficients. PPO also assumes that some automatic differentiation software is used, such that the software is able to keep track of how each objective function is computed in order to backpropagate the gradients appropriately. This also allows PPO to simply combine the objective functions like above.

In the PPO implementation, the parameters $\boldsymbol{\theta}$ characterise the whole actor-critic model, though "under the hood" it can also be two different NNs that receive their own respective gradients, such as in [20] or [49]. However, they have made this generalisation in the case where parameter-sharing is desired, where the "bottom" hidden layers are the same, and the network heads are different – such as in this thesis, which will be discussed briefly in Section 5.1.3. Moreover, since the actor is parametrising a stochastic policy, the head of the actor network outputs the parameters of the policy distribution, which for continuous cases is normally chosen to be a Gaussian distribution.

Furthermore, one of the things to keep in mind is that PPO is also an on-policy algorithm. This means that when the agent is sampling experiences, these samples are gathered under its current policy $\pi_{\boldsymbol{\theta}}(a|s)$. Hence, in its actor-critic implementation, a sample *trajectory* $\boldsymbol{T}$ is first collected under a policy $\pi_{\boldsymbol{\theta}}(a|s)$ for T timesteps before updates are made to the actor and critic parameters $\boldsymbol{\theta}$. This means after T timesteps, we have also received the rewards for each timestep and can compute the advantage estimates $\hat{A}_t$ for every timestep $t = 1, 2, ..., T$. Then, when we are optimising the performance objective, we have the opportunity to define how many epochs K were in that trajectory of size T, which means that we can specify how many gradient updates to do using the same batch of experiences. After the updates are completed, we discard this trajectory of experiences and begin sampling a new one to ensure that the new experiences occur under the new policy $\pi_{\boldsymbol{\theta}}$. Conceptually, we can view the whole update process in Figure 2.9. Here, we first see that a trajectory is sampled based on the current policy given by the actor before the loss terms are calculated. Finally, the gradient of the objective function w.r.t. to the actor and critic weights is used to update the actor and critic networks.

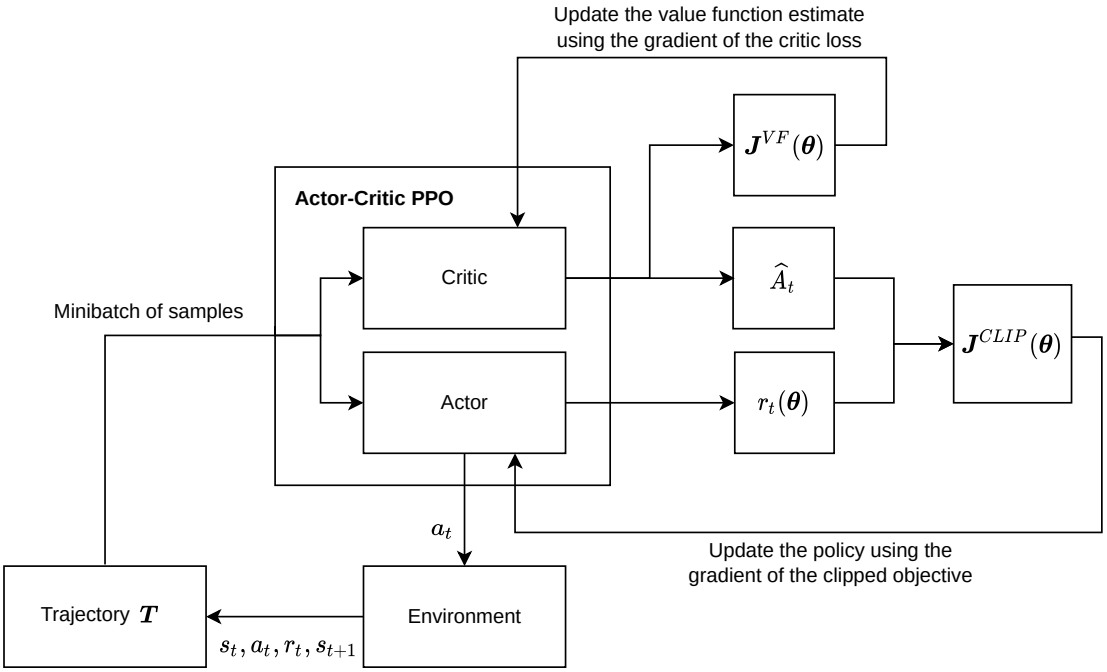


Figure 2.9: An overview of how the actor and critic networks are updated in PPO. Once the loss terms are calculated, the gradients $\nabla J_t^{CLIP}(\boldsymbol{\theta})$ and $\nabla J_t^{VF}(\boldsymbol{\theta})$ are used to update the actor and critic respectively.

Summary

PPO can solve a vast variety of continuous reinforcement learning problems by being a model-free, actor-critic method and using neural networks as function approximators. It is also an on-policy method, meaning the sampled trajectory of experiences is collected under its current policy $\pi_{\boldsymbol{\theta}}(a|s)$. The algorithm is able to achieve state-of-the-art performances through its adaptation of the trust-region based method, TRPO, where the idea is to take the largest possible step in the right direction while ensuring that the new policies, after an update, stay close (or are *proximal*) to the old one. In turn, as stated in TRPO [48], this should guarantee a monotonic improvement of the policy.

In terms of implementation, it is relatively simple compared to its counterpart, TRPO. It uses a clipped surrogate objective to define its proximal policy aim rather than a hard constraint that requires second-order methods to optimise. Also, as it assumes we are using an automatic differentiation software, we can combine the objective functions for both the policy and the value function parametrisations, where the software is able to keep track of how to compute the gradients (perform backpropagation) for the respective parameters.

Also, since it is on-policy, it retains its high data efficiency and reliable performance. This is particularly significant for problems with high-dimensional state and action spaces since the agent can focus on exploring actions along its current policy instead of calculating less gradients

from actions in states that are very uncommon. This is also why it is more data-efficient, as it can converge to an optimal behaviour faster.

Though conversely, since PPO is on-policy, the *overall* degree of exploration is based mainly on its stochastic policy $\pi_{\theta}(a|s)$, with the exception of the entropy term. As discussed in Section 2.3.8, this means that the agent may suffer from a lack of exploration if its policy does not incorporate some degree of randomness. Yet, by the nature of optimisation, PPO will progressively increase probabilities of doing "good" actions and decrease probabilities of doing "bad" ones, based on its estimate of advantage and value function. This means that over time, the agent will exploit the environment more, irrespective of how accurate its estimate of the policy and value function is, and could become trapped in a local optimum.

Chapter 3

Related Works

In the complex task of autonomous navigation, learning what to do or which actions to take – such as the optimal velocity and steering angle for a given input image – can be challenging to express explicitly. In this chapter, we will explore some related learning-based methods to tackle this problem, covering methods that utilise expert demonstrations in supervised or imitation learning, or those that rely on a reinforcement learning approach to learn an optimal behaviour.

3.1 Motion Planning with Supervised Learning

The first learning-based method for navigation is to supervise the learning process by compiling a set of expert demonstrations. The idea here is that we have some deep NN – typically a CNN – that will learn to directly predict optimal actions (or action sequences) based on some input – typically an image from an RGB or depth camera or laser range findings from a LiDAR sensor. In this approach, the navigation problem is essentially a prediction problem. The main challenge is defining the optimal target action for each input when compiling the dataset of expert demonstrations.

In [25], they proposed the first target-oriented end-to-end navigation model for a robotic platform, capable of predicting steering commands – translational and rotational velocities – from raw 2D-laser range findings and a target position. They use a CNN with residual blocks inspired from [50], which takes in the LiDAR data and concatenates this feature output with the target position to serve as an input to a fully connected NN. Then, comparing the action prediction from this model to targets from a global planner – serving as an expert – allowed [25] to train their model in an end-to-end fashion. By using a global planner, the authors also avoided requiring a human expert to tediously provide steering commands at a large scale.

In [23], they introduced *DroNet* – an efficient CNN with a ResNet8 architecture also from [50] – that can guide a quadrotor in urban environments by predicting steering angles and corresponding collision probabilities from single image inputs. To manage this, [23] utilised

a dataset created from cars and bicycles to train their CNN instead of compiling their own dataset. From this, they achieved safe navigation that was highly generalisable and avoided the high sample complexity required for reinforcement learning algorithms.

In autonomous drone racing, one requires a fast perception system capable of real-time detection and pose estimation of gates. This problem can also be approached with a CNN, which then requires the design of a custom dataset due to highly varying race tracks and possible actions. In [51], they proposed a method of generating a labelled dataset using an expert trajectory and policy. First, an optimal trajectory can be generated through all gates if their poses are known. With this trajectory, an expert policy can be used to generate a desired direction and speed to follow the trajectory. Finally, by collecting sampling state estimates and corresponding images, one could use the expert policy to label the images to create a labelled dataset. This allowed [51] to train a CNN – a DroNet from [23] – to learn the desired direction and speed output for each image input.

3.2 Motion Planning with Semi-Supervised Learning

Self-supervised learning techniques are closely related to supervised methods, requiring a labelled dataset so to be able to learn desirable actions for inputs. However, this approach does not depend on some expert planner's demonstrations. In contrast, these methods rely solely on a robot's retrospective self-experience to learn the environment's physical attributes. For example, [52] relies on learning a navigation policy through crashing. They equipped a 720p camera to a quadrotor and tasked it with flying in a straight line until collision. Based on the experience gathered from this simple behaviour, they compiled an image dataset full of positive and negative collision examples and used this to train a CNN to predict whether or not to go straight. By further cropping the images into left and right halves, they created a turning mechanism that allowed the quadrotor to navigate cluttered indoor environments.

The work done in [53] takes this concept further, teaching a mobile ground robot through self-supervision to navigate in "real-world urban and off-road environments with geometrically distracting obstacles" with only a camera. The Berkeley autonomous ground robot, *BADGR*, gathers off-policy data in real-world environments from a random control policy and uses this to train a deep model – a CNN and Long Short-Term Memory [54] model – to predict all future navigational events, such as reaching a goal, collisions or driving over bumpy terrain. Based on the predicted future events, BADGR then finds an optimal action sequence using a stochastic optimiser [55] and executes the first action in an MPC-like fashion.

Following a similar approach, [24] proposed *ORACLE*, a motion primitives-based navigation planner for a quadrotor using a deep collision predictor. The deep collision predictor is an uncertainty-aware NN model that predicts the collision cost of a predefined set of motion-primitives (action sequences), given some depth image and only the quadrotor linear and an-

gular velocities. As for data collection, a quadrotor is deployed in simulation with a random velocity and steering angle (within the quadrotor field-of-view). Data is collected until collision, and the sequence of state-actions and their collision labels are recorded. This dataset then allows the end-to-end training of the deep collision predictor, where it learns to predict the collision probability of an action for each time step of the action sequence. Further, it makes its predictions uncertainty-aware by filtering depth image observations, taking the unscented transform for the quadrotor partial state, and having Monte Carlo dropout in the CNN. By doing this, [24] also achieved a successful sim-to-real transfer.

3.3 Imitation Learning using Expert Planners

In the cases where we have direct access to expert planners, we can also apply supervised learning ideas to reinforcement learning to achieve *imitation learning*. This is a sequential task where, given a dataset of demonstrations, an agent tries to find the best way to learn a policy that achieves an action that is most similar to the expert [56]. Ideally, this should be very sample efficient and should allow the agent to instantly generalise its policy to new situations of the same task.

In 2013, [57] used a novel imitation learning technique with data aggregation, or *DAgger* [58], to train a reactive heading policy for a quadrotor based on the demonstration of an expert human pilot. In contrast to a supervised technique, their approach iteratively learned and exploited corrective input from a human pilot to boost the overall performance of the predictor. This meant that initially, the agent learns a policy based on the data provided by the expert, but it replays this policy in several training iterations to gather more data, specifically where a human pilot could provide correct steering commands when observing undesired behaviour from the quadrotor, e.g. when it turns towards a tree. Using this method, [57] managed to learn a policy capable of navigating a quadrotor through cluttered forest environments at 1.5ms^{-1} using only a single, cheap camera.

In 2021, [9] proposed an end-to-end approach for high-speed flight in natural environments, training solely from simulation. Their student policy is learned via *privileged learning*, whereby imitating an expert with access to privileged information. In this case, the expert is a sampling-based planner with access to the perfect state and 3D map and samples a set of collision-free trajectories conditioned towards some global collision-free trajectory. Then, from this set, the best three trajectories are chosen the student policy. The result is a CNN that learns how to map simulated noisy depth images directly to collision-free trajectories in a range of real-world environments, including a forest and urban environment where the quadrotor had 100% success rates for speeds up to 5ms^{-1} and 7ms^{-1} respectively.

3.4 Motion Planning with Reinforcement Learning

The approach most related to this thesis is reinforcement learning for motion planning. This approach does not rely on the existence of an expert planner or the need to define target actions for some supervised prediction model. Instead, this is an iterative training process where the reward function is used to define the optimal desired behaviour.

Using reinforcement learning for navigation is not a new concept, with [59] demonstrating in 2005 a method for high-speed obstacle avoidance using monocular vision and reinforcement learning. In this work, a linear model was first used to learn depth from encoded images, which was then combined with a reinforcement learning algorithm to learn steering commands for an RC car. Since then, [60] extended this work to deep learning, using a CNN to predict depth from monocular images, and proposed a duelling architecture based deep Q-Network (D3QN) to output command steering and linear velocities for a robot from the depth images. Using this approach, they could train a model entirely in simulation capable of navigating a ground robot in cluttered real-world environments, even with very noise depth predictions.

Another paper that achieves zero-shot transfer from simulation to reality is the work done in [22]. Here, the authors propose a learning-based algorithm for directly mapping monocular images to collision-free quadrotor motor commands, called CAD^2RL. Here, a reinforcement learning agent chooses one of 41x41 image grid bins to travel to, which is then transformed into a velocity vector. To learn the correct actions, the agent learns a Q-function – parametrised by a CNN with VGG16 architecture [61] – via a custom-made policy iteration algorithm, whereby simulating multiple-step rollouts and performing a Monte Carlo policy evaluation.

The methods mentioned above make use of discretised action spaces to simplify the reinforcement learning problem, though as discussed in Section 2.3.8, this does not scale to high-dimensional problems with many degrees of freedom. To amend this, the authors in [62] introduced a method for continuous control for mapless navigation of mobile robots, using an end-to-end asynchronous deep reinforcement learning approach. They use an asynchronous form of DDPG [43], where a sparse set of 10 range findings, previous action and target position are mapped to continuous steering commands in the actor-network.

However, in a more complex, dynamic environment, having a state representative to represent the world can be beneficial for control tasks [63]. This is especially relevant for problems where an agent's states are only partially observed – known as partially observed MDPs (POMDPs) – as not all the information of the environment can be deduced in one observation as not all states hold the Markov property. For these cases, a recurrent neural network (RNN), such as a Long-Short Term Memory (LSTM) [54], can be used to encode the temporal relationship of observation sequences.

The work done by [64] leveraged this idea, proposing the use of a Long-Short Term Memory to encode the observations of an arbitrary number of other agents in an environment into a

fixed-length vector. To achieve collision avoidance on their ground robot in the presence of other agents, they combined this representation with their robot's own state vector and used this as an input to their actor-critic networks. From this, the agent was able to command steering angles and speeds to be able to navigate amongst humans at walking speed.

Having the same idea for image observations, the authors of [65] proposed a network architecture that combines a VAE with a Mixture Density Network (MDN) and RNN (MDN-RNN) to create a representation of OpenAI Gym [66] environments. Using this architecture, they feed the latent code of the VAE and the RNN hidden state to a very simple linear model that outputs an action. With this, [65] managed to solve a range of tasks, among them a race car navigation problem from pixels that had previously not been solved.

Then, in [67], the authors proposed a principled training procedure for unifying latent representation learning with reinforcement learning. In contrast to end-to-end learning methods, [67] suggests separating the two tasks: first relying on variational inference to learn a latent representation, then training the reinforcement learning agent using the learned latent space. From this, their experiments show that their algorithm successfully learns complex continuous control tasks from raw images in the OpenAI Gym and DeepMind Control Suite Environments.

Taking inspiration from [65], the authors in [14] also proposed a three-part deep model, but for robotic navigation in dynamic human environments. Their model included a VAE to reconstruct a LiDAR state, an LSTM to predict future state sequences and a 2-layer perceptron taking in the representations of the first two modules as input. Their work focused on testing different variations of this network for the navigation task: altering the LiDAR representation, switching the LSTM to a transformer [68], and training their model jointly in an end-to-end approach or separately as in [65], and [67]. Finally, they demonstrated the performance of one of their models on a real robot, where it reached its goal 100% of the time, albeit with some room for improvement in its behaviour.

Lastly, the work that has been most inspirational for this thesis the most is that of [13]. Here, the authors learn a state representation from depth images and camera trajectories and use this to train a policy for navigation in cluttered and dynamic environments. They use a VAE to encode depth images and feed the latent vector with camera trajectories to an LSTM to generate a hidden state. Then, a simple MLP acting as the policy receives this state representation and a goal as input, decides on velocities in x and y, and a yaw rate. This model achieved only a 3% failure rate in their tests and was successful in sim-to-real transfer. A key feature of [13] is that they trained their VAE to perform simple depth completion by comparing reconstructed depth images with a filtered target depth image. By doing so, they minimised the shortcomings of real depth images compared to simulated ones and reduced the complexity of depth images. This idea is also useful when we wish to limit the generative capacity of the VAE to only a small latent dimension. Finally, other similarities of this work with our thesis include using Isaac Gym as a simulation environment and PPO as the reinforcement learning algorithm of choice.

Chapter 4

Problem Formulation

The overarching task for this thesis is the autonomous navigation of an aerial robot in a cluttered environment without any access to a global map but equipped with some device for perception: monocular, RGB or depth camera, or LiDAR. We also assume that we have access to the quadrotor state but not its shape nor its dynamics, and that there exists some goal state to reach (e.g. a specified waypoint from some global planner). In this context, we wish to design a safe, local planner that is capable of both collision avoidance and reaching a specified goal in 3-dimensional space.

From the previous chapter we have seen that there are many ways of approaching this problem, with some of them either requiring a dataset for training some prediction network [23], an offline map of the environment to generate a reference trajectory [51], access to expert planners [9, 59], or access to a model-based control library [24]. Otherwise, some approaches were also to a finite set of discrete actions [52, 59, 60]. However, in this thesis, we do not assume to have access to these resources and choose to not limit ourselves to discrete action spaces. The latter will serve to not impede the scalability of our method to higher dimensional problems (e.g. navigation in 3-dimensions) and also upholds the versatility and navigational efficiency of the aerial robot to a higher degree. Last, we aim to maintain a simple and flexible simulation environment that does not need to be heavily customised, e.g. with high texture [9, 22], to demonstrate the robustness of the learning method.

To address the above-mentioned points, we model the problem as two-fold: an unsupervised representation learning task, followed by a model-free reinforcement learning task in simulation. Specifically, we first wish to learn a representation of the depth images and then find a control policy that is capable of guiding a quadrotor towards a goal in an obstacle-filled environment. The control policy should then be capable of deciding the motion of the quadrotor through a reference velocity and a steering angle, based on a depth image representation and the quadrotor state. Framing the problem in this way guarantees a minimum amount of feature engineering (with the exception of the reward function) and allows the training methodology

to be generalised to other domains.

4.1 The Reinforcement Learning Task

The learning task for navigation in a reinforcement learning context is to find the parameters $\boldsymbol{\theta}$ of a stochastic policy, $\pi_{\boldsymbol{\theta}}(a \mid s, z)$, that maximises the expected return G_t under the induced trajectory distribution p_π. The policy should map observations S_t to a probability distribution across actions A_t, where the sampled actions A_t should allow our quadrotor state $\boldsymbol{s}_t$ to converge towards some goal state $\boldsymbol{s}^*$ as $t \to T$, where T is the end of an episode.

Since we are dealing with a navigation task with access to the quadrotor state and a depth image representation, we can specify the agent observations and actions as:

$$S_t = \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{z}_t \end{bmatrix} \in \mathbb{R}^{77}, \qquad A_t = \begin{bmatrix} \boldsymbol{v}_t^d \\ r_t^d \end{bmatrix} \in \mathbb{R}^3 \tag{4.1}$$

Here, $\boldsymbol{s}_t \in \mathbb{R}^{13}$ is the quadrotor state, $\boldsymbol{z}_t \in \mathbb{R}^{64}$ is the depth image representation, $\boldsymbol{v}_t^d \in \mathbb{R}^2$ is the desired velocity in the x and z axes of the body frame, $r_t^d \in \mathbb{R}$ is the desired yaw rate.

The quadrotor state $\boldsymbol{s}_t$ consists of the position $\boldsymbol{p}_t \in \mathbb{R}^3$, velocity $\boldsymbol{v}_t \in \mathbb{R}^3$, orientation $\boldsymbol{q}_t \in \mathbb{R}^4$ and angular velocity $\boldsymbol{\omega}_t \in \mathbb{R}^3$:

$$\boldsymbol{s}_t = \begin{bmatrix} \boldsymbol{p}_t \\ \boldsymbol{v}_t \\ \boldsymbol{q}_t \\ \boldsymbol{\omega}_t \end{bmatrix} \in \mathbb{R}^{13} \tag{4.2}$$

where (following notation from [69]),

$$\boldsymbol{p}_t = \begin{bmatrix} x_{ib}^b \\ y_{ib}^b \\ z_{ib}^b \end{bmatrix} \in \mathbb{R}^3, \quad \boldsymbol{v}_t = \begin{bmatrix} u_i^b \\ v_i^b \\ z_i^b \end{bmatrix} \in \mathbb{R}^3, \quad \boldsymbol{q}_t = \begin{bmatrix} \eta \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \end{bmatrix} \in \mathbb{R}^4, \quad \boldsymbol{\omega}_t = \begin{bmatrix} p_i^b \\ q_i^b \\ r_i^b \end{bmatrix} \in \mathbb{R}^3 \tag{4.3}$$

The position $\boldsymbol{p}_t$, linear velocity $\boldsymbol{v}_t$ and angular velocity $\boldsymbol{\omega}_t$ are expressed in the body frame $\mathcal{B}$ with respect to the goal frame $\mathcal{I}$, while the orientation $\boldsymbol{q}_t$ denotes the rotation of quadrotor body frame $\mathcal{B}$ with respect to the goal frame $\mathcal{I}$, represented in quaternions. Also, the goal frame $\mathcal{I}$ is stationary and represents an approximate inertial frame. To simplify the task of reaching the target, we choose the desired goal state to always be the centre of the goal frame, such that $\boldsymbol{p}_t$ should converge to $\mathbf{0}$ as $t \to T$.

As for the action space, the desired velocity $\boldsymbol{v}_t^d$ is:

$$\boldsymbol{v}_t^d = \begin{bmatrix} u^d \\ w^d \end{bmatrix} \in \mathbb{R}^2 \tag{4.4}$$

Where u^d and w^d are the desired velocities of the quadrotor along the body x and z axes, with respect to the goal frame $\mathcal{I}$ (the notation is omitted for clearness).

4.2 The Representation Learning Task

Next, to obtain a good representation for the depth images $\boldsymbol{d}$, we can first assume that these depth images are generated from the conditional distribution $p_{\boldsymbol{\theta}}(\boldsymbol{d}|\boldsymbol{z})$, where $\boldsymbol{z}$ is some hidden continuous random variable. Based on this assumption, we are interested in finding $\boldsymbol{z}$ as this code serves as a latent representation of our depth data $\boldsymbol{d}$. To do this, we attempt to learn the true posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{d})$ through variational inference, where we attempt to approximate $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{d})$ with a family of parametric distributions $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$. We then aim to optimise for the variational parameters $\boldsymbol{\phi}$, such that the approximate posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$ best approximates the true posterior distribution $p_{\boldsymbol{\theta}}(\boldsymbol{z}|\boldsymbol{d})$.

Furthermore, since our intention is to use this model in combination with a reinforcement learning agent for fast navigation on a small, robotic platform – we are restricted to a light-weight inference network for representing $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$. Moreover, to ensure training a relatively quick convergence of our navigation policy, we also wish to restrict the dimension of our latent space to a maximum of $\mathbb{R}^{64}$ so that we minimise the complexity of the observation-action mapping space. As a result of this, a model will have a constrained representational capacity and lack the ability to represent all details of a given depth image – and so the ability to reproduce them in reconstruction. Therefore, we identify a list of some important characteristics of depth images that are essential for collision avoidance for which a model should prioritise in its latent representation:

1. **Geometric shapes** – Certainly, a depth reconstruction should resemble its original depth image input to some degree. Thus, we expect that a model should be able to estimate the rough shape of seen obstacles. This should also help the model to generalise to unseen obstacles.
2. **Clear, close obstacles** – In a very cluttered environment, it may be too demanding to expect the reconstructions of all shapes in the environment. For collision avoidance, it is critical that *at least* very close objects are detected and represented clearly.
3. **Thin obstacles** – Wires, poles and thin pillars are examples of obstacles that could easily be missed in a depth reconstruction. It is important that a quadrotor is not blind to these when it approaches them.

As it is impossible to meet all these demands simultaneously, we can compromise that the reconstruction fidelity should be a function of its distance – simply put, we wish to observe shapes generally, but the closer it is, the sharper we wish the reconstruction to be. Equally, we accept that obstacles far away can *go missing*, such that a model instead prioritises those that are close.

As for thin obstacles, we can reason that this might be a too difficult task in general as reproducing a thin wire can be considered a very fine detail. Thus, we can also compromise in this aspect by stating that it is not important that reconstructions of these thin obstacles are clear, only that these are not missed.

Chapter 5

Proposed Approach

In this chapter, we present the methodology for solving the autonomous navigation task outlined in the previous chapter. Overall, we propose a CNN-MLP model where, given a depth image $\boldsymbol{d}_t$ and the quadrotor states $\boldsymbol{s}_t$, it decides a continuous velocity $\boldsymbol{v}_t^d$ and steering r_t^d command that should avoid obstacles in a cluttered environment, while travelling towards some goal in 3-dimensional space. We will present and discuss this two-part model's design choices, starting with the MLP module for learning the navigation policy, then later the encoder-decoder-based CNN inference network used for representation learning. An overview of the model is shown in Figure 5.1.

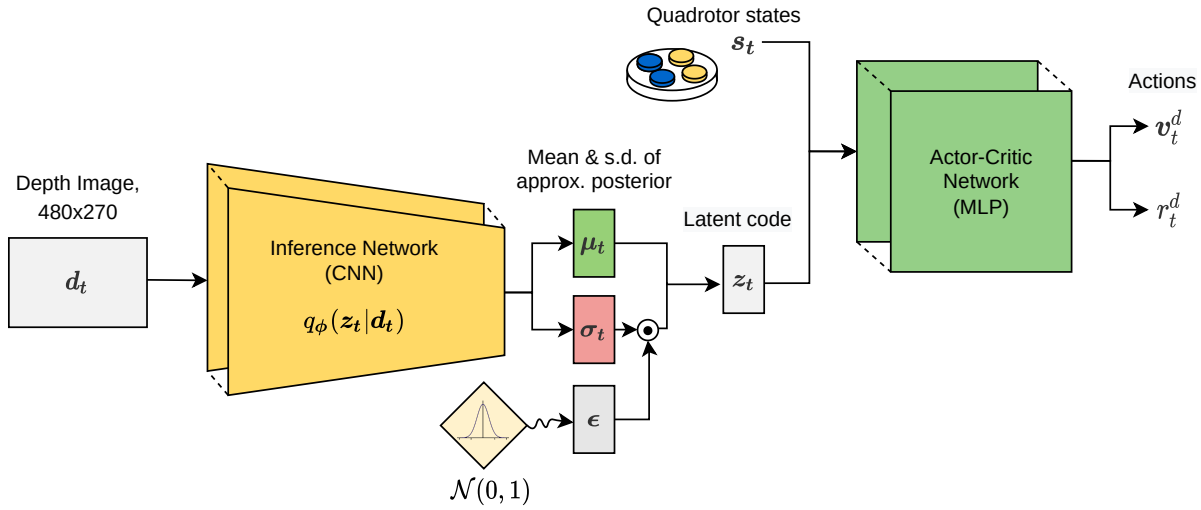


Figure 5.1: An overview of our two-part model. The inference network (encoder) learns an approximate posterior distribution $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$, parametrised by a set of Gaussian distributions with an input-dependent mean $\boldsymbol{\mu}_t$ and diagonal covariance matrix $\boldsymbol{\sigma}_t$. We then sample this to get the latent representation $\boldsymbol{z}_t$. With $\boldsymbol{s}_t$ and $\boldsymbol{z}_t$, our agent – a model-free actor-critic – learns a policy $\boldsymbol{\theta}_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{z}_t)$ which outputs a desired velocity $\boldsymbol{v}_t^d$ and yaw rate r_t^d.

5.1 Learning the Navigation Policy

Starting first with the navigation task, we solve the reinforcement learning problem through the use of PPO. Following the theory in Section 2.3.11, we aim to concurrently learn a critic value function $V_{\boldsymbol{\theta}}(s)$ and actor policy $\pi_{\boldsymbol{\theta}}(a\,|\,s, z)$, parametrised with parameters $\boldsymbol{\theta}$. To train our actor-critic for collision avoidance, we define a custom reward function and incorporate a *curriculum* [20] during training.

5.1.1 Reward Function

The reward function is one of the most powerful design tools within reinforcement learning. Using it to formalise the idea of a goal is also one of its most distinctive features [40], since by deciding the reward structure of the environment, we can indirectly manipulate the learned agent behaviour to match the behaviour we hope to observe in testing. Though, for a navigational task, proper care must be taken to balance the rewards for different behaviours, such that an agent remains careful but also navigationally efficient. This is because the complex behaviour that an agent learns is based directly on the idea of maximising the total reward, which includes exploiting the environment and its reward function. Hence, any undesired behaviour that is observed during test time is often a consequence of a poorly designed reward function.

By weighing these considerations, we construct a reward function that builds on the implementation in [26], but with additional rewards R and penalties P to shape the agent behaviour for collision avoidance. First, we motivate the agent to minimise its distance to goal by rewarding its inverse distance to goal:

$$R_{\text{pos}}(S_t) = \frac{K_{\text{pos}}}{1 + ||\boldsymbol{p}_t||_2} \tag{5.1}$$

Then, we define a set of desired behaviours we wish to see when the agent is close to goal: remain still R_{vel}, stay upright R_{up} and do not spin R_{spin}:

$$R_{\text{vel}}(S_t) = \frac{K_{\text{vel}}}{1 + ||\boldsymbol{v}_t||_2} \tag{5.2}$$

$$R_{\text{up}}(S_t) = \frac{K_{\text{up}}}{1 + \left|1 - \mathcal{I}_z(\boldsymbol{q}_t)\right|^2}, \qquad \mathcal{I}_z(\boldsymbol{q}_t) = \frac{\epsilon_2}{\sqrt{1-\eta^2}} \tag{5.3}$$

$$R_{\text{spin}}(S_t) = \frac{K_{\text{spin}}}{1 + r^2} \tag{5.4}$$

Here, we use $\mathcal{I}_z(\boldsymbol{q}_t)$ to denote the upwards-ness of the quadrotor, where the idea is to represent the quadrotor orientation $\boldsymbol{q}_t$ in axis-angle form and then find the normalised z-component of the axis. For R_{spin}, r denotes the quadrotor yaw rate as shown in (4.3).

As for the conservative behaviour, we specify three penalties terms: one for velocities in the

blind directions of the quadrotor (vertical and backward) P_{vel}, one for being too close to an obstacle P_{depth}, and the last for collision $P_{\text{collision}}$:

$$P_{\text{vel}}(S_t) = \alpha_{\text{vert}} \cdot w^2 + \alpha_{\text{back}} \cdot v_{\text{back}}^2 \qquad v_{\text{back}} = \begin{cases} v & \text{if } v \leq 0 \\ 0 & \text{else} \end{cases} \tag{5.5}$$

$$P_{\text{depth}}(S_t) = \mu_{\text{dist}} \cdot \max\left(0, d_\epsilon - d_{\text{obst}}(S_t)\right)^2 \tag{5.6}$$

$$P_{\text{collision}}(S_t) = \begin{cases} K_{\text{collision}} & \text{if } d_{\text{obst}}(S_t) \leq d_{\text{collision}} \\ K_{\text{collision}} & \text{if contact force detected} \\ 0 & \text{else} \end{cases} \tag{5.7}$$

The depth penalty P_{depth} is a simple one-sided quadratic barrier function taken from [70], consisting of a scaling parameter μ_{dist} and safety margin d_ϵ. Intuitively, this means that quadrotor receives no penalty if it is further than d_ϵ, but is penalised an obstacle within d_ϵ is in sight. Thus, this should motivate the agent to stop moving closer to an obstacle, and to turn elsewhere. Visually, the depth penalty is shown in Figure 5.2.

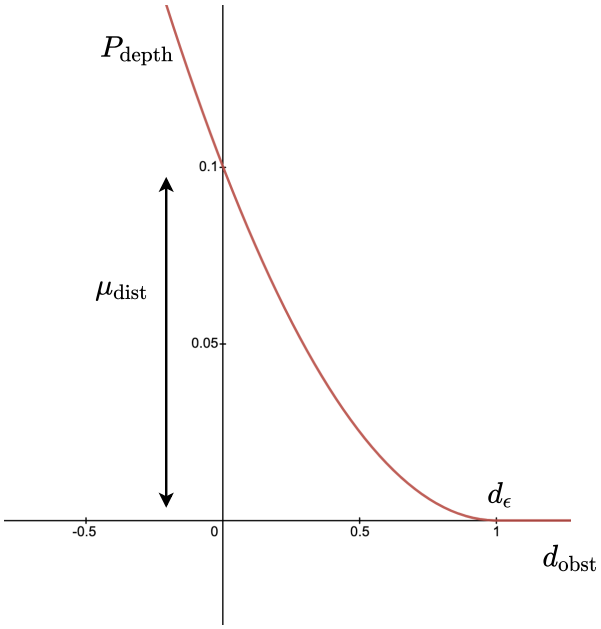


Figure 5.2: The depth penalty P_{depth} displaying the penalty for when an obstacle is closer than $d_\epsilon = 1.0$m, when $\mu_{\text{dist}} = 0.1$. Diagram recreated from [70].

To find the distance to the closest obstacle d_{obst}, we project a point cloud from a depth image and take the norm of each point in the point cloud with the camera position and find the minimum. However, since we can only detect forward-facing collisions with the depth camera,

we also detect collisions with a force sensor in simulation.

From these, the full reward function is the sum of all rewards and penalties:

$$R(S_t, A_t) = R_{\text{pos}} + R_{\text{pos}}(R_{\text{vel}} + R_{\text{up}} + R_{\text{spin}}) + P_{\text{vel}} + P_{\text{depth}} + P_{\text{collision}} \tag{5.8}$$

where we specify R_{vel}, R_{up} and R_{spin} to only be important near goal by multiplying it with R_{pos}. Finally, the reward gains, penalty coefficients and distance parameters used for the reward function are listed in Table 5.1.

Reward Parameter	Value
K_{pos}	2.0
K_{vel}	1.0
K_{up}	1.0
K_{spin}	1.0
$K_{\text{collision}}$	-2
α_{vert}	-0.1
α_{back}	-0.01
μ_{dist}	-0.1
$d_{\text{collision}}$	0.2

Table 5.1: List of reward gains, penalty coefficients and distance parameters used in the reward function.

We note that each of the reward terms are at maximum when $\boldsymbol{p}_t, \boldsymbol{v}_t = \mathbf{0}$, $r = 0$ and the quadrotor is upright. At this point, the quadrotor is directly on the goal, with a reward of $R_t^{\max} = K_{\text{pos}} + K_{\text{pos}}(K_{\text{vel}} + K_{\text{up}} + K_{\text{spin}}) = 8$ at every timestep, assuming that we avoid all penalties. However, when the quadrotor is very far from the goal, e.g. > 10m, this goal-motivating reward begins to be very sparse. Combined with the conservative penalties, we can imagine that it will be difficult to train a policy end-to-end in a large cluttered environment due to negligible positive rewards for flying towards a goal and considerable negative rewards for going near obstacles. This is also why we propose to use curriculum learning, which will be discussed in the next section.

5.1.2 Curriculum Learning

The success of this thesis' approach can largely be attributed to the setup and procedure for training the reinforcement learning agent. The term *curriculum* was introduced by [20] and is used to describe the idea of training a policy at levels of increasing difficulty. For collision-free navigation, we leveraged this idea by training the quadrotor in progressively larger environments with an increasing density of obstacles. The reason for this can be justified by two reasons: first, training a randomly initialised policy in a very difficult environment with sparse rewards can be extremely time-consuming if not impossible; and second, collision avoidance is very generalisable – once a quadrotor has learned to avoid one obstacle, it should not be difficult to extend

this knowledge to two, and eventually many.

We assert that before a quadrotor can learn collision avoidance, it must first learn to fly towards the goal. Our primary concern is that reinforcement learning is generally considered sample-intensive, where learning a complicated policy may just come down to waiting for a lucky sequence of actions to be repeatedly executed. In this thesis, we aim to minimise this "luck factor" and instead propose a three-step process that should *guarantee* successful training:

1. First, learn to fly towards the goal with no obstacles present.
2. Then, learn basic obstacle avoidance by spawning the quadrotor and goal on opposite sides of *one* obstacle, with the quadrotor facing the obstacle.
3. Last, gradually increase the number of obstacles, the environment size and the episode length T to obtain an advanced collision avoidance policy.

5.1.3 Network Architecture

Moving on, the actor-critic network is chosen to be a shared three-layer MLP with two separate output heads: the policy head and the value function head. Its architecture is shown in Figure 5.3. The policy $\boldsymbol{\theta}_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{z}_t)$ is modelled by the actor-network, which outputs a Gaussian distribution over actions for a given quadrotor state $\boldsymbol{s}_t$ and latent code $\boldsymbol{z}_t$. The value function is parametrised by the critic network that predicts the expected return (state value) for the same input.

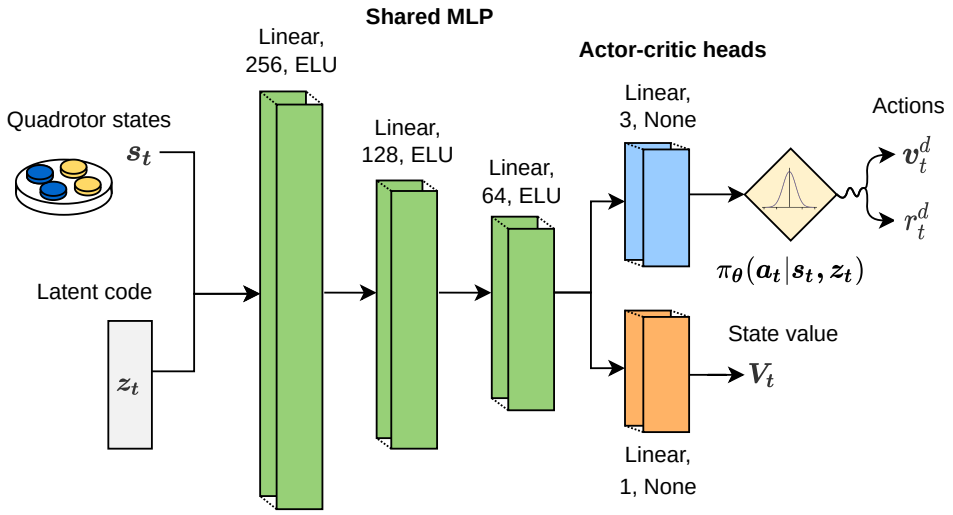


Figure 5.3: The actor-critic network architecture. The actor and critic are parameterised by a shared base network comprised of a three-layer MLP with output dimensions $[256, 128, 64]$ and ELU activation functions, and two linear (fully-connected) output heads with linear activation functions. The actor parametrises a policy $\boldsymbol{\theta}_{\boldsymbol{\theta}}(\boldsymbol{a}_t|\boldsymbol{s}_t, \boldsymbol{z}_t)$ that outputs a Gaussian distribution over actions, while the critic parametrises a value function $V_{\boldsymbol{\theta}}(\boldsymbol{s}_t, \boldsymbol{z}_t)$ which predicts the expected return V_t.

Shared Parameters in the Actor-Critic

The concept behind using a shared structure in the actor-critic networks is that the *base network* learns the *features* our input so that these can be used by the *network heads* for *task-specific prediction or classification*. Essentially, it is the same as representation learning, where the last layer of the shared MLP contains the representation of features of the input. We can assume that in order to produce a correct velocity and yaw rate reference, the model should have some understanding of the agent-relative surroundings. Though in a similar vein, to be able to predict the expected return for a particular state requires the same understanding. So, in general, if two output tasks are largely related but utilise the same input, it makes sense to have a shared parametrisation for the base network with two task-specific heads.

Size of the Network

The size of the network was largely chosen according to other baseline models in [26], and from previous experience from the project, thesis [1]. From [26], we noted that many examples utilise much larger networks, but these were also applied to tasks with "more difficult" observation-action mappings [49, 71] – in essence, just having a much higher dimensional observation and action space. This idea of using larger networks is that they have a larger *generalisation potential*, thus enabling them to do well in more complicated tasks. Moreover, recent research also states that over-parametrization of neural networks might even be necessary to have robust results [72]. However, from experience in the project thesis, we found that larger networks take longer to train and do not necessarily produce better results immediately, therefore motivating a more conservative approach which is more in line with machine learning teachings: starting simple and increasing the complexity underway. From this, we found that a base network with size [256, 128, 64] was reasonable, along with 64 neurons for each head.

Activation Functions

In Figure 5.3, we note the use of two types of activation functions, the exponential linear unit (ELU) and linear (None), were most noteworthy is the choice of the linear activation function for the final network layer. Traditionally, we choose the final activation function based on the type of problem we have – like *sigmoid* for logistic regression (prediction or binomial classification tasks), or *softmax* for multinomial logistic regression (multi-class classification). For continuous control problems with normalised action spaces, we often wish to limit our actions to a range of $[-1, 1]$, which actually makes *tanh* the most suitable activation function. Nevertheless, the decision to use the linear activation function was large as a result of the baseline implementations in [26]. Instead, as an implementation detail, actions were left unbounded from the network but were clipped if their values exceeded $[-1, 1]$.

Next, the exponential linear unit [73] was also used due to being the default implementation in [26] – with it also being used to solve other difficult tasks [20, 71]. An alternative for MLPs is, of course, the widely popular rectified linear unit (ReLU) [74], but the ELU differs slightly as it has negative values for inputs less than zero. This is shown more clearly in (5.9) and Figure 5.4.

$$f_{\text{ELU}}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}, \tag{5.9}$$

$$f_{\text{ReLU}}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{5.10}$$

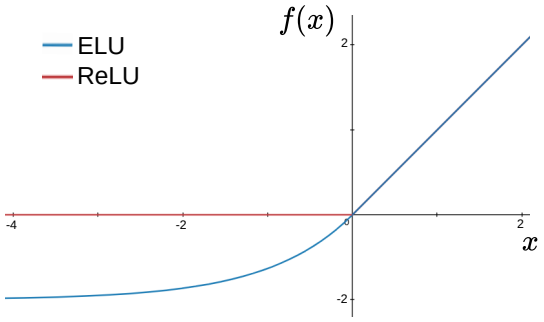


Figure 5.4: Visualising the difference between the exponential linear unit (ELU) (with $\alpha = 1$) and rectified linear unit (ReLU) activation functions.

The primary reason for using ELU rather than ReLU is that one of the most significant problems of ReLU is that of *dead neurons*. This problem occurs when a neuron is pushed to a negative weight (e.g. from a large update), because the gradient of the ReLU activation (its output) w.r.t the negative neuron weight will always be zero. To illustrate clearly, consider the perceptron $y = \text{ReLU}(Wx + b)$. When the a *positive input* x is multiplied with *negative weights* W, the output y is zero, and so too the gradient $\frac{\delta y}{\delta W}$. Conversely, if the *input* x and weights W are *negative*, the output y is non-negative but the gradient of the output w.r.t the weights $\frac{\delta y}{\delta W} = \text{ReLU}'(x)$ is still zero because the ReLU gradient ReLU$'$ is zero for negative inputs. As a result, the network weight will never be able to update itself as the gradient will be zero indefinitely, irrespective of the input data. The benefit of ELU is that its gradient is non-zero for negative inputs close to zero. This helps to solve the *dead-neuron problem* as it produces non-zero gradients that help to nudge network weights in the right direction, despite them having negative inputs.

However, it can be argued that having some sparsity in the network (due to dead neurons) is actually an advantage, which is why ReLU is the default recommendation by [28] for modern neural networks and is also used in [49] for its actor-critic MLP. Then again, too much sparsity can result in the network losing some of its generalisation capacity.

5.2 Learning the Depth Representation

To tackle the unsupervised representation learning problem, we use a convolutional VAE to learn the latent representation for the depth data $\boldsymbol{d}$. We use the method presented by [32] for optimising the VAE, whose theory is outlined in Section 2.2.2. Additionally, to both deal with the constrained dimension of the latent space and make our VAE suitable for collision avoidance, we introduce a custom loss function that allows us to specify which depth characteristics the VAE should prioritise in its reconstructions and choose a lightweight network architecture inspired from [24], and [75].

5.2.1 Ideal Depth Reconstruction With a Customised Reconstruction Loss

We first recall that the *reconstruction loss* in (2.9) defines the learned behaviour of our generative network $p_{\boldsymbol{\theta}}(\boldsymbol{d}|\boldsymbol{z})$. In a vanilla VAE, it defines that the decoder $p_{\boldsymbol{\theta}}(\boldsymbol{d}|\boldsymbol{z})$ should learn to reconstruct $\boldsymbol{d}_t$ from $\boldsymbol{z}_t$, where $\boldsymbol{z}_t$ is the sampled latent code from our encoder $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$ for a given $\boldsymbol{d}_t$. Now, the key insight is that in order to properly reconstruct $\boldsymbol{d}_t$, any features of $\boldsymbol{d}_t$ that should be reconstructed must be present in $\boldsymbol{z}_t$ – from which the VAE learned to do through the joint optimisation of the encoder and decoder in (2.10). This implies that if we define certain features of $\boldsymbol{d}_t$ to be more costly than others, their loss will be over-represented in the reconstruction loss, and the VAE would prioritise learning these in its latent space so that these specific features could be reconstructed, and the VAE loss minimised.

Thus, our approach is to *alter the reconstruction loss* such that the VAE learns which features of the depth image distribution to prioritise in its latent space. With this, we attempt to prioritise the features in Section 4.2 in the latent space by presenting the following modifications:

1. **Filtered targets** – Instead of using the input $\boldsymbol{d}_t$ as the target reconstructions of our generative model $p_{\boldsymbol{\theta}}(\boldsymbol{d}|\boldsymbol{z})$, we use a filtered depth image $\boldsymbol{d}^f$ as the target. This means that for a given depth image $\boldsymbol{d}_t$, the generative network instead learns a probability distribution $p_{\boldsymbol{\theta}}(\boldsymbol{d}_t^f|\boldsymbol{z}_t)$, where $\boldsymbol{d}_t^f = f(\boldsymbol{d}_t)$ is given by a deterministic filtering process f of the depth image $\boldsymbol{d}_t$, and $\boldsymbol{z}_t \sim q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$ is the sampled latent code from our encoder with input $\boldsymbol{d}_t$.
2. **Depth weighting** – We weigh the pixel-wise reconstruction error by a function of its observed depth. This means that the reconstruction error for pixels showing close obstacles in $\boldsymbol{d}_t^f$ are weighed more than pixels of far obstacles.
3. **Added edge loss** – We add the an additional mean-absolute error (MAE) term to the reconstruction error of filtered-obstacle edge pixels.

To go more in detail, the filtering process is the IP-Basic algorithm [76] for dilation and hole closing, as used in [13] and [24]. Its implementation is a result of a few benefits for our task. First, minimising the complexity of depth images by rounding shapes emphasises only learning rough shapes. Also, as dilation increases obstacle sizes, filtering provides an extra layer of safety

regarding collision avoidance. Finally, filtering also removes noise, which can be important when testing this framework on a real robotic system. Then, to avoid the extra computational load of pre-filtering depth images on the robot, we use filtered images as reconstruction targets for some depth input, such that the VAE learns to implicitly perform the filtering process in its forward-pass [13].

As for the depth weighting, we multiply the pixel-wise error of a reconstruction with the bounded depth gain $K_{\text{depth}}(d_{i,j})$, a function of the filtered-depth pixel value $d_{i,j}$:

$$K_{\text{depth}}(d_{i,j}) = \alpha_{\text{depth}} \cdot \min\left(\frac{1}{d_{i,j} + 0.5}, 1\right) \qquad \text{for} \quad i, j \in \dim(\boldsymbol{d}_t^f) \tag{5.11}$$

which is illustrated in Figure 5.5. The idea for the depth weighting is that if we increase the re-

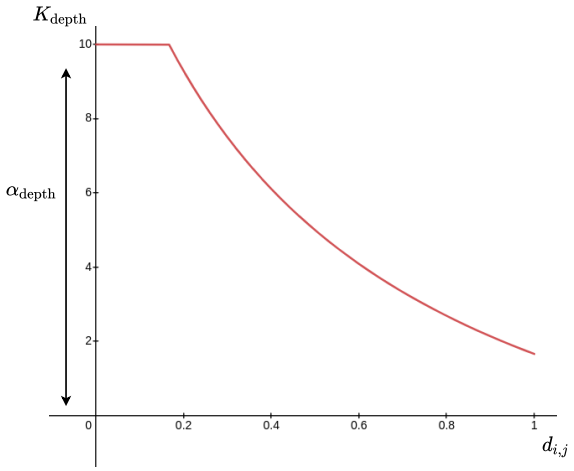


Figure 5.5: The depth gain to weigh the pixel-wise reconstruction error. Reconstruction errors for very close obstacles (pixel values $d_{i,j} < 0.16$) are weighed the same, with $\alpha_{\text{depth}} = 10.0$.

construction error according to its closeness, the pixel-wise loss of close obstacles should dominate the pixel-wise loss of obstacles far away. So close obstacles should be prioritised in the latent space representation. Intuitively, this is particularly important for collision avoidance: we wish to distinguish between pixel-wise errors for obstacles far away compared to those immediately nearby. For example, a 1m error for an obstacle 7m away should be considered less important than the same as a 1m error for an obstacle 1.5m away.

We also see that thin obstacles are expected to be reconstructed when they are close by combining depth weighting with filtered targets. This is because their reconstruction targets are more prominent, as many more pixels will produce a high loss, particularly at close range.

Finally, we used a simple homemade procedure to implement the added edge loss. First, we used a Canny edge detector [77] to find the obstacle edges in $\boldsymbol{d}_t^f$. After, we used a Gaussian filter to dilate the edges – taking any pixel value over 0 as an edge. Finally, the pixel-wise MAE of

filtered depth reconstruction was multiplied by this image-edge mask to achieve the edge loss. Since we motivated this for clearer reconstructions, the Gaussian filter had to be used to dilate the edges of the image-edge mask. This is because, for the MAE loss to account for the object's shape, it is essential to add the pixel-wise error along the edges of obstacles and the pixel-wise error of neighbouring pixels.

5.2.2 Network Architecture

With the loss function covered, what remains is the architecture of the VAE. As mentioned, our VAE design was mainly inspired by the work of [24] and [75], where we utilise a convolution-based encoder-decoder structure. The overall structure of the VAE is shown in Figure 5.6 and its parameters are detailed in Appendix B.

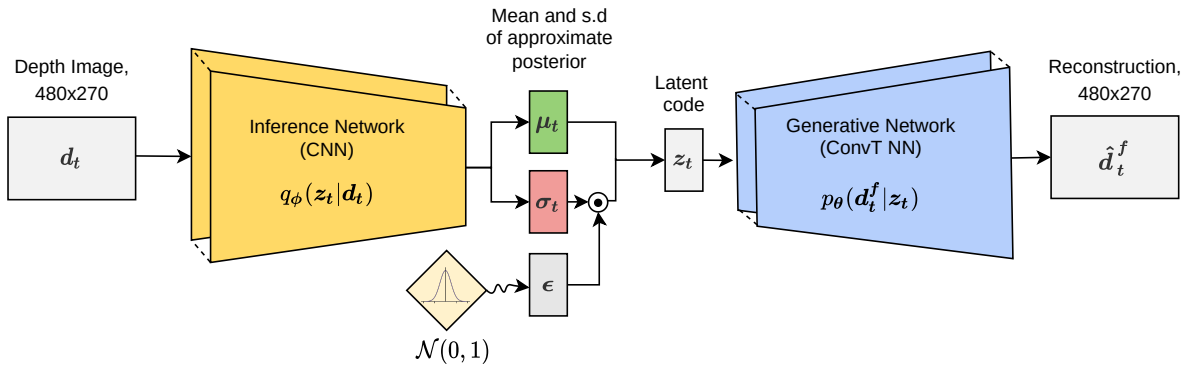


Figure 5.6: The VAE network architecture. The encoder is parametrised by a CNN, while the encoder is parametrised by a transposed CNN (ConvT NN). Given a depth image $\boldsymbol{d}_t$, we can sample from the inference network $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$ to obtain a latent code $\boldsymbol{z}_t$. The generative network $p_{\boldsymbol{\theta}}(\boldsymbol{d}^f|\boldsymbol{z})$ then learns to construct the filtered depth image $\boldsymbol{d}_t^f = f(\boldsymbol{d}_t)$ from the latent code$\boldsymbol{z}_t$.

Inference Network

Our encoder is follows the CNN design of [24], though utilises instead two convolution layers before a ResNet8 [50], with two fully-connected layers at the end. Its structure is shown in Figure 5.7, while the residual blocks are depicted in Figure 5.8.

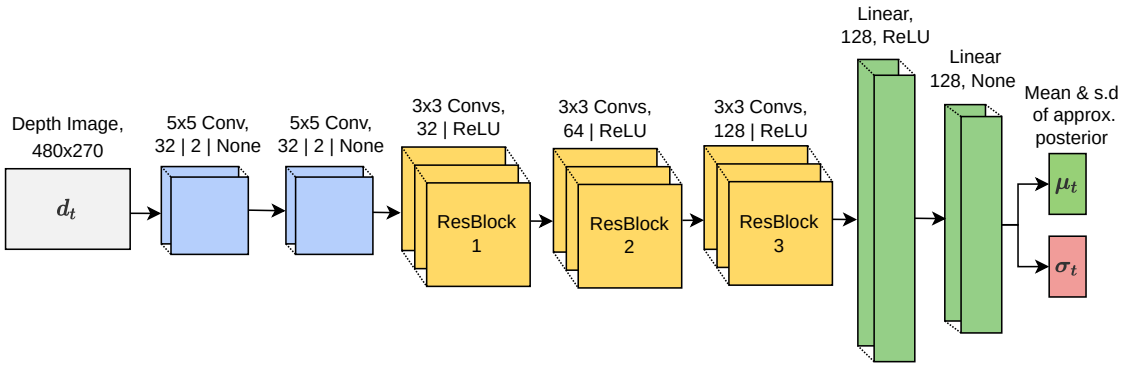


Figure 5.7: The encoder network architecture. It comprises two convolution layers, with 32 5×5 filters with a stride of 2, three residual blocks with $[32, 64, 128]$ output filters, and two fully connected layers with output dimensions $[128, 128]$. The dimension of the feature map is (roughly) halved for each convolution layer and residual block due to the 2-strided convolutions. The size of the feature map after the last convolution is 15×9. (See Appendix B.1 for details)

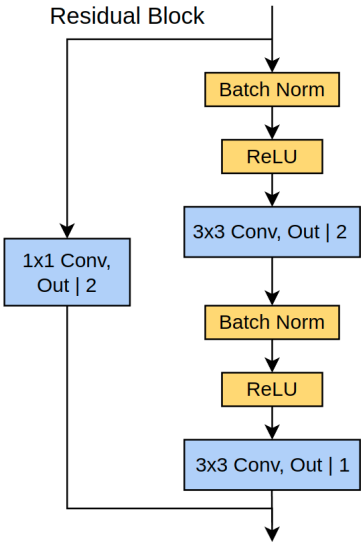


Figure 5.8: The residual block architecture. A convolution by a 1×1 filter (kernel), with stride 2 is applied to the shortcut connection. *Out* represents the number of filters of the residual block.

Regarding our design choices, these were primarily motivated by two factors – we desired a lightweight network and wished to reduce the dimension of the feature map to be small enough but not too small. The ResNet8 was suitable to satisfy the first aspect, while the depth of the network (through stridden convolutions) was decided to satisfy the other. We also found during testing that reducing the feature dimension below 15×9 (to 8×5) resulted in a much worse reconstruction output, discouraging the use of another residual block or stridden convolution layer. However, this choice resulted in a dramatically sized linear layer that accounted for 86% of the encoder weights – $(128 \cdot 15 \times 9) \cdot 128$ connections. Nevertheless, this was unavoidable since

128 was the minimum number of neurons in the linear layer (64 means and log-variances), which left the alternatives: to either reduce the feature dimension or the number of filters. Though, both options were tested and did not improve results, leaving the conclusion that this is a feature and not a disadvantage of our encoder.

Generative Network

The generative network is inspired from [75], with an additional linear layer and transposed convolution layer. Its architecture is shown in Figure 5.9. The primary design rule for autoen-

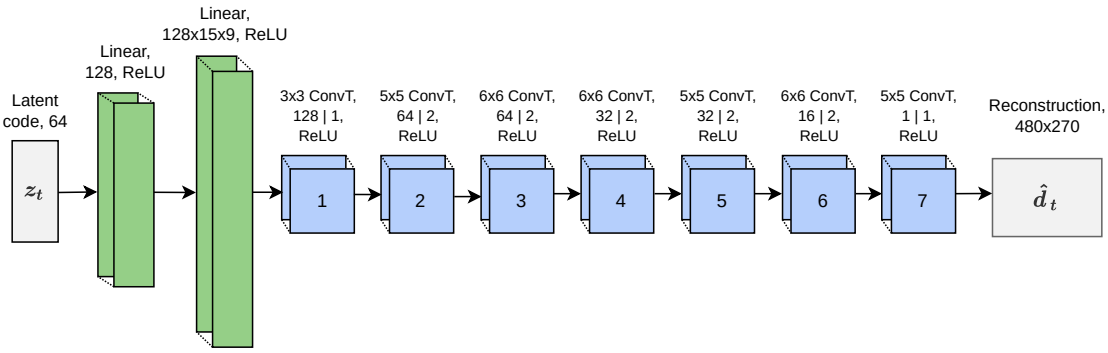


Figure 5.9: The decoder network architecture comprises two linear layers and seven transposed convolution layers. Each strode transposed convolution roughly the dimension of the feature map to achieve the final dimension feature map dimension 480×270. (See Appendix B.2 for details)

coders is that the decoder should be a mirror of the encoder so that the network resembles an *hourglass*. As a result, we followed the encoder with two linear layers, including the large linear layer that connects 128 neurons to 128, 9×5 filters. As for the transposed convolutions, this is a relatively simple but effective design. The only design choice here was the filter size and number of filters, though these were chosen primarily to match the layer-wise feature dimensions of the encoder and their overall sizes.

Vigorous testing was also done with a ResNet8 decoder, which was designed to mirror our encoder. However, despite its more advanced structure – with batch normalisation and shortcut connections – no significant performance gain was observed in training. Conversely, checkerboard artefacts and divergence in training were often observed when training on the whole dataset due to uncertain reasons, though it could be to the layer-wise stacking of stridden transposed convolutions with identical filter dimensions [78]. Therefore, a final decision was made to use the decoder in Figure 5.9.

Chapter 6

Implementation

This chapter presents the implementation details and software frameworks used in this thesis. Beginning with the training setup for our VAE and MLP, we include the procedure for collecting a depth images dataset for VAE training, and how we define our quadrotor task in the Isaac Gym framework. Finally, we present how we define our CNN-MLP model using a reinforcement learning library, *RL Games* [79], and discuss some details of the PPO implementation. Our code is written using *PyTorch* as a machine learning software framework, being the framework most used in the research community, and allowing for easy integration with Isaac Gym.

6.1 Collecting a Filtered Dataset of Depth Images

To collect the depth for VAE training, we used a pre-existing simulator that had been developed in [24]. This is the RotorS [80] simulator, which is an extension to the simulation framework, Gazebo [81]. Here, Gazebo is an open-source physics-based robotics simulator that allows us to design and test custom robotic models. RotorS is then a quadrotor simulator that builds on this, providing various packages ranging from quadrotor models to controllers to sensors.

To use RotorS for their task, [24] added a quadrotor model resembling their real-life resilient micro-flyer (RMF) to RotorS, equipped with a simulated depth camera with the same specifications as an Intel RealSense Depth Camera D455m, which has camera properties:

- Resolution: 480×270
- Field-of-view: 1.5 rad (85.94°)
- Minimum range: 0.2
- Maximum range: 10.0

Using this, their quadrotor was flown with random velocities and steering angles in a randomised environment in Gazebo until collision to generate a dataset. While [24] was interested in the state-action-collision labelled image dataset, we simply used this framework to collect

the depth images.

Then, since we are only interested in the depth images, we flip these along their vertical axis to double the total. With over 200,000 depth images, we filtered each of these with the IP-basic algorithm [76] and batched them into TensorFlow Record files. Finally, we wrapped the record files in a PyTorch Iterable Dataset to ensure that not all depth images would be loaded into memory when iterating through the dataset.

6.2 Quadrotor Task in Isaac Gym

We utilise *Isaac Gym* [26] as a large-scale parallel hardware-accelerated (GPU) simulator to initialise and run 512 environments simultaneously. Its end-to-end GPU simulation avoids performance bottlenecks in CPU-GPU data transfers and allows for a high performance and simulation throughput on a single GPU.

To use Isaac Gym for our task, we first have to define an MDP-like *quadrotor task* that we can perform reinforcement learning in – similar to any standard gym environment in OpenAI Gym [66]. Essentially, we have to create a quadrotor model and then define its states, actions, the state-transition dynamics and the rewards an agent receives per state-action timestep. Fortunately for us, there was a related task, namely *Quadcopter* example in Isaac Gym, that we could adapt to our task. From this, we could use the existing quadrotor model and its states but redefine its observation and action spaces, the reward function and environment.

6.2.1 Agent

Beginning with the agent, we wished to have an observation space $S_t = [\boldsymbol{s}_t, \boldsymbol{d}_t]$ consisting of the quadrotor state and depth images. To obtain the depth images, Isaac Gym allows us to add camera sensors with specific properties. Choosing its properties to model the same depth camera used to collect the VAE images, we attached the camera to the quadrotor body using Isaac Gym's API with a follow transform to obtain depth images $\boldsymbol{d}_t$ in simulation.

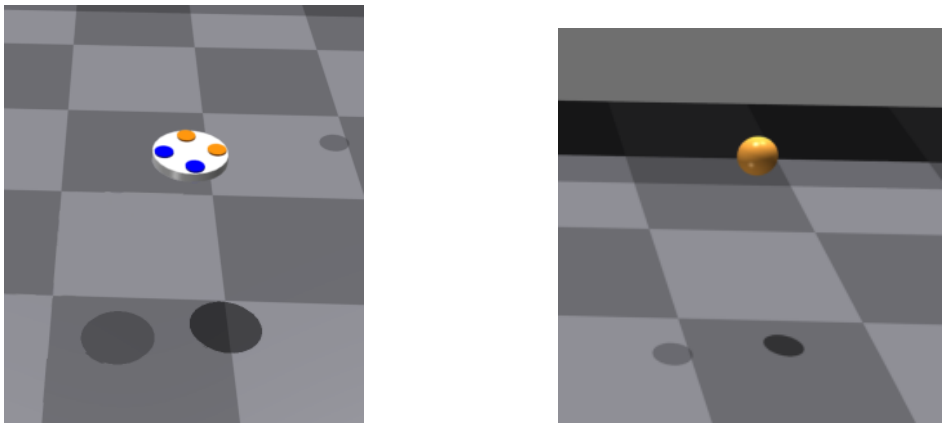
As for the state of the agent $\boldsymbol{s}_t$, Isaac Gym's API allows us to obtain the states of all *actors* (Isaac Gym assets) through a *root tensor*, where every state contains 13 floats, matching (4.2): 3 floats for the position, 4 for quaternion, 3 for linear velocity, and 3 for angular velocity. By indexing the quadrotor asset, we could then access the true quadrotor state at every time step.

As for the action space, Isaac Gym allows us to control an *actor* by specifying its motion directly or through its control tensors, such as by applying forces to a robot's degrees of freedom. Though not recommended, we opted for the first option due to simplicity – simply altering the quadrotor state in the root tensor to be the desired action velocity and yaw rate. This is, of course, non-physical behaviour since we override Isaac Gym's physics engine, meaning that changes in velocities would be instantaneous. To provide some compensation for this, we adjus-

ted the simulation timestep to be $dt = 0.2$ seconds, which ensures that there is a suitable bandwidth separation between the reinforcement learning agent "action frequency" and an eventual closed-loop control system bandwidth, so that we can expect that an underlying control system has time to set $\boldsymbol{v}_t = \boldsymbol{v}_t^d$ at each timestep. Nonetheless, since our thesis aims to demonstrate motion planning, this choice was considered reasonable (given the time constraints).

6.2.2 Environment

With the agent ready, we now had to define the environment in which the quadrotor was to operate. The obstacles used in this thesis were loaded as Isaac Gym assets through Unified Robotic Description Format (URDF) files. Most importantly, these described the visual and collision geometry of various shapes, including our quadrotor model and goal. In Figures 6.1 and 6.2 we show the quadrotor, goal and added obstacles to our environment.

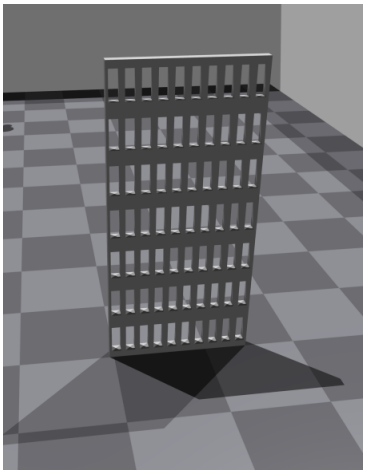


(a) Quadrotor **(b)** Goal

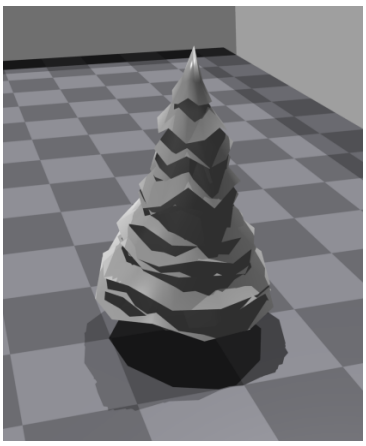
Figure 6.1: Quadrotor and goal assets in our Isaac Gym environment. These are both standard assets already present in Isaac Gym, and are borrowed for this task. The quadrotor does not collide with the goal, nor is the goal visible in the depth images.

With defined collision geometries, we could then enable collisions in the environments by placing obstacles in the same environment in the same collision group and setting their collision filters to be 1. Since we do not want collisions with the goal, we removed its collision geometry, rendering it invisible to the depth camera and removing the possibility for collision.

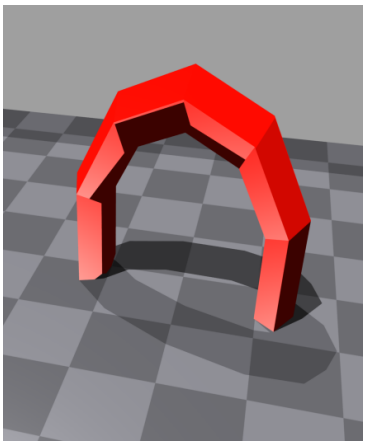
Finally, to place these obstacles nicely in our environment, we could change the object dimensions in the URDF files and do further scaling of objects in Isaac Gym. Then, through the root tensor, we adjusted their poses through rotations and randomised their position.



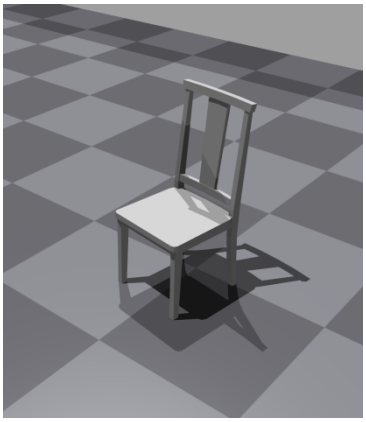
(a) Fence



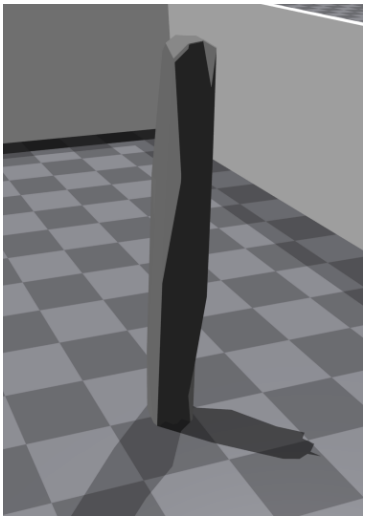
(b) Pine tree



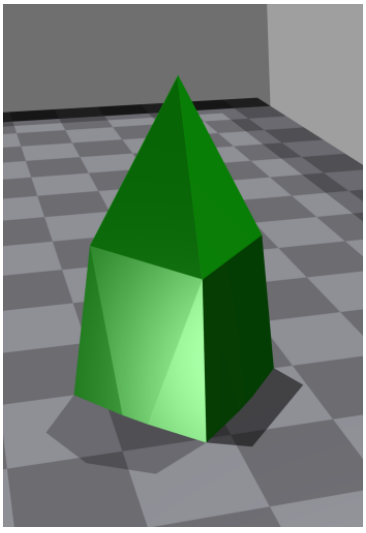
(c) Simple U-shape



(d) Chair



(e) Simple stone



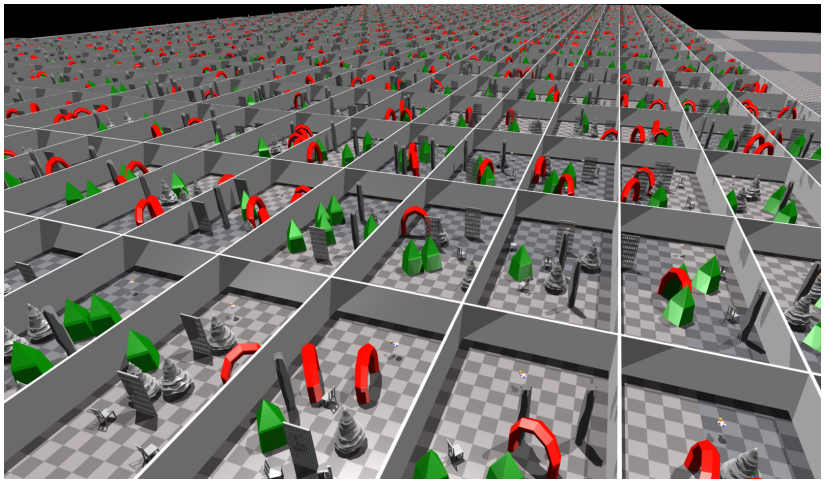
(f) Simple pyramid

Figure 6.2: Various obstacles are imported to our Isaac Gym environment through their URDF files. Their defined collision geometries allows them to be visible in the depth images and allows the physics engine to simulate collisions with them. By altering their dimensions and *root tensors*, we could fit them nicely in our environments, with randomised poses.

6.2.3 Parallel Initialisation

Regarding the parallel initialisation of the environments, this depends on which stage of the curriculum we are training on. In the first scenario, with no obstacles, we initialise each environment to have a quadrotor and goal with random positions in an area $x, y \in [-3, 3]$, with $z \in [-0.2, 2]$. Furthermore, we add a ground plane at $z = 0$ and mimic "far obstacles" by confining each quadrotor to a 16×8 enclosure. This serves to bias the agent to initially ignore the depth representation input while it learns to map the quadrotor states to correct actions.

Then, for 1-obstacle environments, the quadrotor and goal for each environment are now placed at each end of their enclosure, with an obstacle placed in the middle. Their initialised x positions are fixed (along the long axis) but are randomised in y (short axis). Also, the quadrotors and goals are slightly randomised in z. Then, we extend this design to n-obstacle environments, where we gradually increase the number of obstacles between a quadrotor and goal while ensuring that a quadrotor always has 3m of open space before the first obstacle. An example is shown in Figure 6.3b.



(a) 512 environments are initialisation in parallel, with randomised obstacles along y.

(b) Quadrotor, goal and obstacle positions are randomised in y for each environment.

Figure 6.3: An example of the parallel initialisation of environments and their obstacle placements with Isaac Gym [26]. We simulate 512 environments in parallel, where the quadrotor and goal is initialised at each end of a corridor-like enclosure with obstacles placed in between. Positions of each item are fixed in the x (long axis), but randomised in y (short axis). Quadrotor and goal heights (in z) are also slightly randomised.

6.3 Model Implementation and PPO in RL Games

Now that the reinforcement learning framework was prepared, we used a reinforcement learning library called *RL Games* [79] to implement our CNN-MLP network to be trained with PPO. The RL Games integration with IsaacGym follows a strictly hierarchical structure where all specific algorithms and models inherit from general base implementations. In this way, the training setup is defined through configuration files managed by *Hydra*[1].

6.3.1 Implementing our Model

To integrate our model with RL games, we identified that the main challenge is not how to implement PPO with our model but how to integrate our custom network into RL Games. If we could solve this, we could then specify that our model should use our custom network in configuration, and this should allow it to be seamlessly optimised with PPO. So to solve this, we first wrapped our custom network in a *network builder*. From here, we could use a *model builder* to register our network in RL Games' network registry during initialisation. As a result, our network was now a part of RL games and could be chosen for any reinforcement learning task.

Then, to use it in our custom quadrotor task, we define two configuration files, one for the setup of the quadrotor task and the other to define the model. Here, the task configuration is quite standardised, including the dimension of observation and action space, simulation parameters like *dt*, but also the number of obstacles, environment size and episode length. In contrast, the training file provided our model definition, consisting of the optimisation algorithm and its hyperparameters (batch size, learning rate, trajectory length, etc.), and our network and its hyperparameters (size, activation functions, normalised actions, separate actor-critic networks, etc.).

6.3.2 Normalised and Clipped Observation and Action Space

Next, there are some noteworthy details of our model implementation that is relevant to our task. The first is that normalising the observation and action spaces is of particular importance in RL Games, first introduced in [43] and also iterated in the documentation of Stable-Baselines3, "normalising input features may be essential to the successful training of an RL agent" [82]. *Normalising* can mean two different things in reinforcement learning, either by scaling observations to $[0, 1]$ or to have 0 mean and 1 standard deviation. The perspective this thesis takes is that image values are scaled while everything else is normalised. So, to normalise the observation space in our quadrotor task, RL Games keeps track of our running mean and standard deviation through its *RunningMeanStd* class, which calculates its values throughout the training

[1] See how IsaacGymEnvs uses Hydra at `https://github.com/NVIDIA-Omniverse/IsaacGymEnvs`.

process. We also alter the code slightly to ensure that our depth images are not affected. The consequence of this is that when we gradually increase the environment size to larger values, i.e. greater $\boldsymbol{p}_t$ from the goal, this should not alter the policy performance. For the actions, these are not normalised but instead clipped if their values exceed 1, which was also why we avoided using a non-linear activation function for our actor-critic. For good measure, the observations space is also clipped to a max value of 5 before being normalised, in case of, e.g. anomaly observations.

6.3.3 Experience and Optimisation

In order to perform optimisation, recall from Section 2.3.11 that PPO requires data (experience) sampled by its current policy in a trajectory $\boldsymbol{T}$, where each *experience* element is given by a tuple $< S_t, A_t, R_t, S_{t+1}$. In the parallel learning scheme, observations $S_t = [\boldsymbol{s}_t, \boldsymbol{d}_t]$ from each environment is compiled into an observation buffer $\boldsymbol{O}_t$ for each time timestep. As mentioned, these observations are then clipped and normalised before being sent to the actor to provide actions for the current policy π. Though we simulate 512 environments, there is only one network – for each timestep, we batch the 512 observations into an input tensor of shape $[512, 129613]$ and compute the forward-pass for the whole batch. The batched depth images are sent separately to the VAE, which produces the latent code with dimension $[512, 64]$. Then, by concatenating it with the quadrotor state tensor to get $[512, 64]$, our actor-critic outputs a batched action tensor of shape $[512, 3]$. Finally, our agent observes rewards R_t with dimension $[512, 1]$ to which it can compare to its value estimates V_t to calculate the advantage estimate $\hat{A}_t$.

Throughout this process, an *experience buffer* collects the state-action pairs as the trajectory. We define the *horizon* or size of the *trajectory* $\boldsymbol{T}$ to be 8, such that our batch size is $512 \cdot 8 = 4096$. We also define a minibatch size of $M = 1024$ and a number of mini-epochs $K = 8$. Overall, we first simulate our environment in parallel for 8 timesteps to collect a trajectory. With this, we optimise for our PPO loss in mini-batches and repeat the optimisation on the trajectory for $K = 8$ mini-epochs. Finally, we discard the trajectory and run the simulation for 8 timesteps to collect a new trajectory and so on. Lastly, when calculating the gradients for the trajectory, we also utilise truncated gradients where the gradients are scaled according to Pytorch's *GradScaler* and clipped by their norm. These gradients are then used to optimise our network weights through the *Adam* optimiser [83].

6.3.4 Reset Handling

When calculating gradient updates, the PPO critic value is used to calculate the advantage $\widehat{A}_t = Q(s,a) - V_{\boldsymbol{\theta}}(s)$, where $V_{\boldsymbol{\theta}}(s)$ is the predicted infinite sum of discounted rewards (predicted return). In our experiments, we specify a max episode length T that resets environments at this timestep. However, from what is seen in the experience buffer, resets count as a state-"transition"

for which the action performed suddenly leads to a drastic change in reward. Since these resets cannot be predicted, this can lead to inferior critic updates as the prediction error in (2.63) for these resets are severely incorrect. To handle this, the target of the critic is bootstrapped with its own prediction during resets, such that the critic error is only the reward obtained for the first timestep in the next episode.

Chapter 7

Navigation Policy Evaluation Studies

This chapter presents and evaluates our approach to developing an efficient collision-free navigation policy. Our approach is presented in the context of the curriculum, where we share the step-by-step decisions and thought processes for training our policy. The final policy is then evaluated, where we assess its performance in its training environment and its ability to generalise to new environments.

For our thesis, training and simulation were done on an Intel(R) Core(TM) i9-10940X CPU @ 3.30GHz desktop PC with a NVIDIA GeForce RTX 3090 GPU.

7.1 Policy Performance along the Learning Curriculum

In this section, we analyse the training characteristics of our navigation policy in progressively difficult environments. In total, our curriculum is composed of 7 levels, with an initial pretraining stage. An overview of the training time and environment setup is shown in Figures 7.1 and 7.2 respectively. For each environment, the policy performance will be briefly analysed in the

Level	No. of Obstacles	Iterations	Time
0	0 - Pretraining	97	12m
1	0	95	11m
2	1	808	1h 40m
3	3	652	1h 22m
4	5	1498	3h 12m
5	9	3500	7h 38m

Table 7.1: The learning curriculum for the navigation policy. The total time for training is all seven policies is 14h 15m, with 6650 total iterations. One iteration is given by the length of the trajectory $\boldsymbol{T}$, defined as 8 simulation timesteps.

context of *average return* across all environments, and the *episode end information* for the last

Level	No. of obstacles	Env. Size	Obst. Area	Space per obst. (m^2 / obst.)
0 & 1	0	16, 8	-	-
2	1	16, 8	32	32.0
3	3	20, 10	80	26.67
4	5	20, 10	80	16.0
5	9	20, 10	80	8.89

Table 7.2: Environments used in the curriculum. We specify the "openness" per environment as a more intuitive measure for clutter rather than obstacle density. The obstacle area begins 6m from each end of the enclosure to allow space for the quadrotor and goal. Thus, the obstacle area is given by $A = (X - 12) \cdot Y$.

1000 episode ends.

Understanding the Average Return and End Info Plots

The average return, in this case, is given by the average accumulated rewards across all environments, indicating the robot's overall navigational efficiency as the reward function is centred around the quadrotor reaching its goal. However, to ensure that robots are constantly exploring the environment dynamics and learning collision avoidance, we specify an end episode timestep T that resets the environment and places the quadrotor back to its initial position. This significantly changes the average rewards in the scenario that multiple environments are reset, producing sharp drops or spikes in the average return plots.

Since the reward is centred around reaching a goal, there is a good chance that the learned policy is not exactly ideal in terms of collision avoidance. Thus, we include the episode end label as a continuous abstract indicator of the collision rate for the current policy. We can expect that if both the collision rate and average return are high, the quadrotor is quite aggressive in flying towards the goal, sacrificing some collisions to exploit high rewards near the goal. In contrast, if both are low, we can imagine that the policy does not have an idea of how to exploit the reward system but avoids collisions, meaning it is overly conservative and does not reach the goal, despite not crashing. In the extreme case where the reward is very low, and the collision rate is high, this suggests that the policy has *diverged* – i.e. the weights of the network are so far from its optimal configuration space that it is unable to perform meaningful actions. Most clearly, the desired policy is the opposite of this: with maximum return and minimal collision. In addition, since the end information is averaged over 1000 episodes, there is a small *delay* when observing the collision avoidance properties from a current policy. As a result, the policies that are collision avoidant should either have positive slopes for timeout episodes or maintain their low collision rates for subsequent episode ends.

Finally, we specify bounds on the quadrotor height, where the quadrotor is reset if z-position

goes beyond $[0, 3]$. Hence, there are three ways an episode can end: through this *out of bounds* in z, through *timeout* after T timesteps, or *collisions* where an obstacle is $< 0.2m$ away from it, or there is contact on the sides.

7.1.1 Without Obstacles

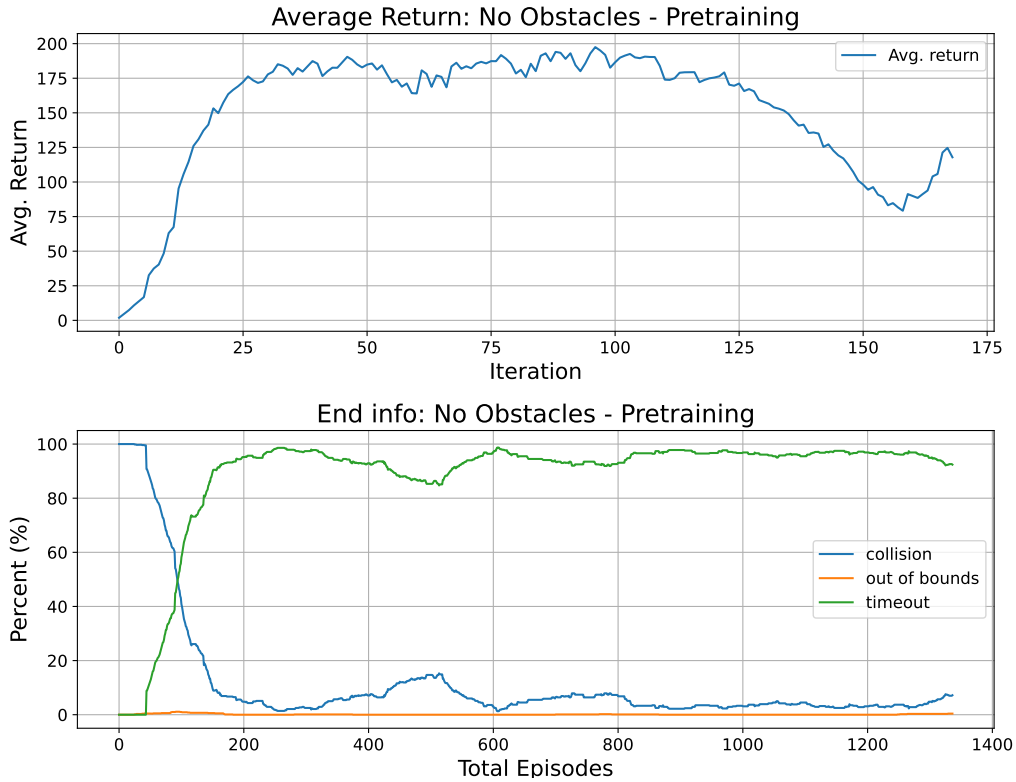


Figure 7.1: Initialising the navigation policy. The quadrotor and goal is randomly initialised in a $[-3, 3]$ area with heights $z \in [0.2, 2.0]$. The best model is selected at 97 iterations.

As stated in Section 5.1.2, before a quadrotor learns collision avoidance, it must first learn to fly towards a goal. Given that our observation-action mapping space is quite large ($\mathbb{R}^{77} \rightarrow \mathbb{R}^3$), our initial guess was that navigating toward a waypoint could already be quite challenging, given that we are in the continuous domain. With this expectation, we trained our policy with results in Figure 7.1. From this figure, we can observe that pretraining is very successful, where the quadrotor steadily increases its average return at the same time its collision rate drops. By just 30 iterations (4 minutes), we see that the agent reaches its goal 100% of the time and maximises its potential reward for most of the episode.

The next step taken is to initialise the quadrotor and goal in its new environment setup, where each are placed at the end of the environment. This is a relatively simple training stage

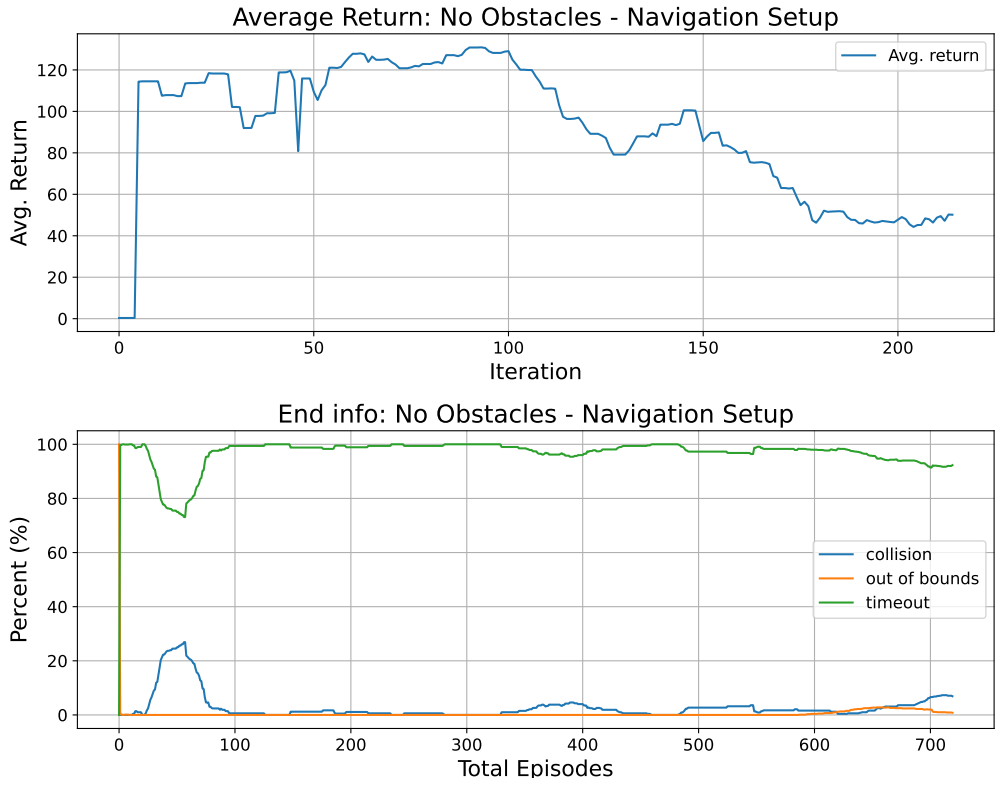


Figure 7.2: Conditioning the policy to its new state distribution. The best model is selected at 95 iterations.

that we expect should be completed with ease. From Figure 7.2, we see that this is true, where we mostly observe timeouts.

7.1.2 1 Obstacle

Before the first obstacle is added, the agent does not necessarily have to learn to avoid close obstacles. In this situation however, obstacles are placed directly in between the quadrotor and goal, such that to achieve a low collision rate, some degree of collision avoidance is necessary. Despite this new challenge, we see that the quadrotor does well to cope – steadily decreasing its collision rate to about 4% in about 160 iterations (20 minutes), and to a minimum of about 2% at 600 iterations. Though, in regards to policy performance, we note that the average return follows a very oscillatory behaviour which is reflected very slightly in the collision rate. Intuitively, this can be explained by the agent exploring its state-space, which consequently can be good or bad – the quadrotor cannot know that colliding with an obstacle always leads to a negative reward until it has experienced it repeatedly from various positions. However, since we did surround the quadrotor in a rectangular enclosure, it was observed that the quadrotor managed to learn to not crash into walls quite early, albeit to a small degree. Just simply stopping and turning,

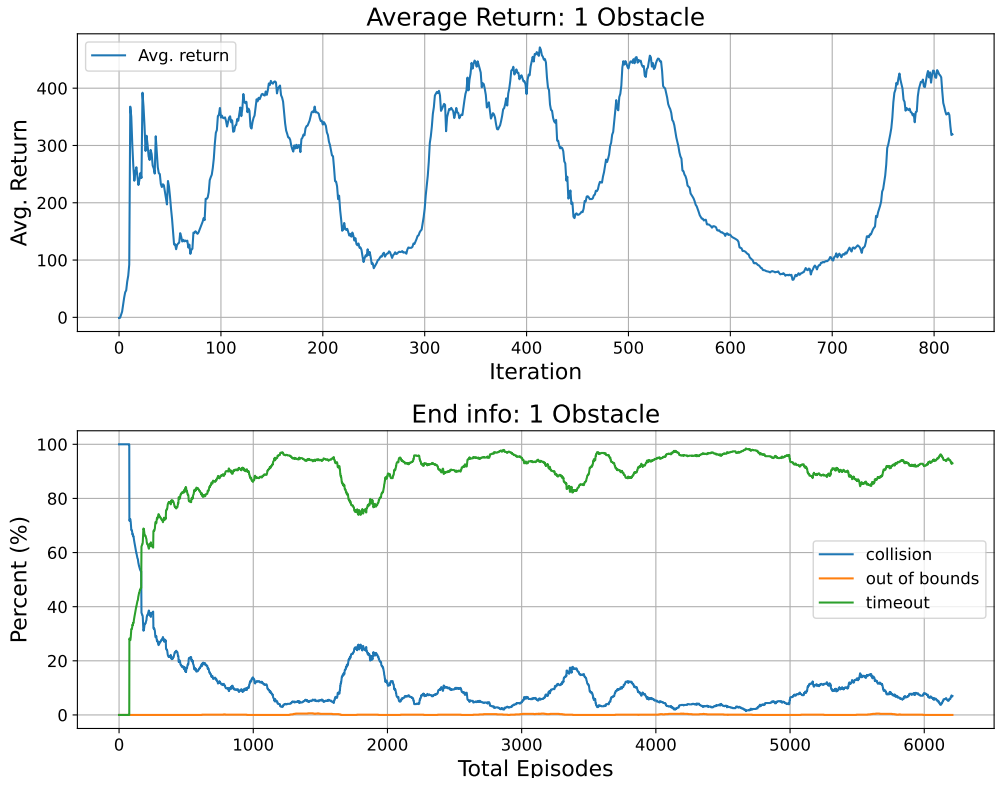


Figure 7.3: Training with 1 obstacle. The best model is selected at 808 iterations, with an average return of 422 and timeout rate of 95.6%.

though, is enough to pass this level.

We observe that it is not until the third oscillation in the average return that the agent finds an optimal policy which combines high total rewards with low collision rates. After this, we do observe that the collision probability falls even lower, but so too does the reward gained. Thus, we remind ourselves that we want *both* navigational efficiency and conservativeness in our policy. For this training stage, we selected the model at 808, which had a good combination of both.

An early behaviour that was also observed during training is that the quadrotor learned to pick a "favourite side" when avoiding obstacles – sometimes it always passed on the left, other times always on the right, despite it being more open on the other side. To try and explain this, we have to remember that neural networks are function approximators. In some sense, we can think of "seeing the obstacle" as a variable that is sometimes positive (or zero) in the latent code. If it is not there, we go straight, but if it is there, we turn to one side – deterministically. We can imagine that if the agent is directly in front and in the middle of an obstacle unless the policy is randomised to a significant extent, the same values in the input should result in the same action.

7.1.3 3 Obstacles

Continuing onwards to three obstacles, we expect that, similar to one obstacle, we should observe oscillations in the return as the agent explores different approaches to solving the navigation task. We also expect that since the obstacle has learned how to avoid one obstacle, it should be straightforward to generalise this to three – particularly when the objects are placed far apart. From Figure 7.4, we see that these expectations are largely matched – with the aver-

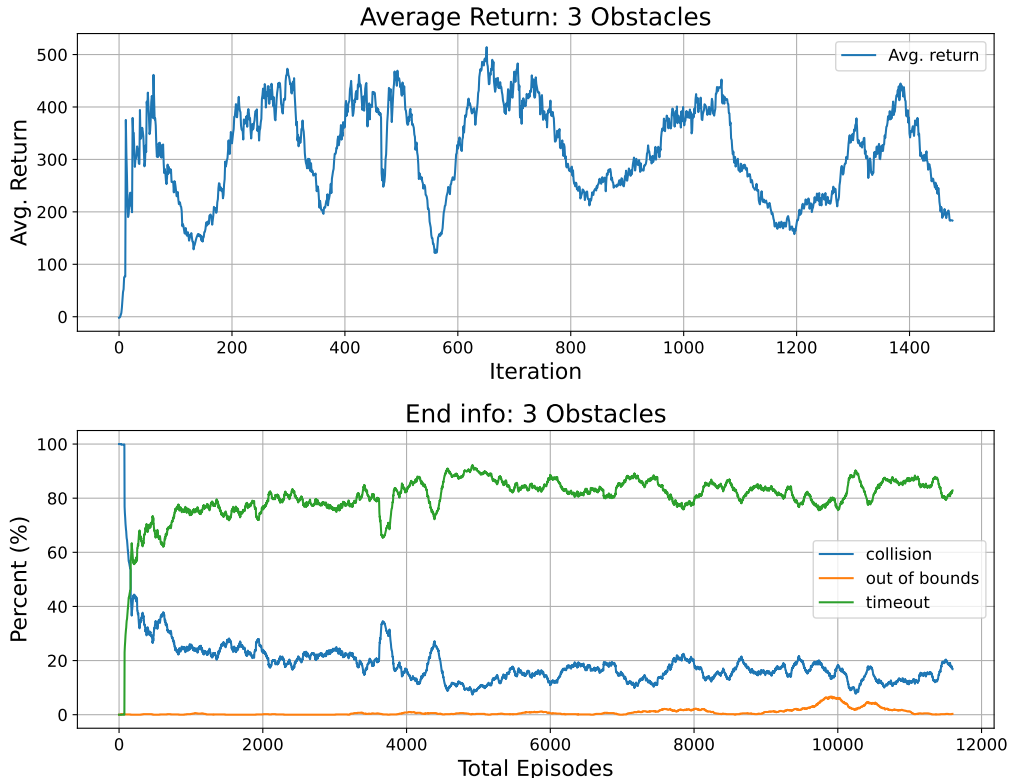


Figure 7.4: Training with 3 obstacles. The best model is selected at 652 iterations, with an average return of 514 and timeout rate of 89.7%.

age accumulated reward varying significantly and the collision rates steadily increasing. From this, we observe that the good combination of both is just about between 600 and 700 iterations, such that we choose the best model at 652 iterations.

One of the most pertinent differences that we can observe in this scenario is that the highest timeout rates are consistently lower than before. We can accept that since the agent is essentially re-learning a more complex state-action mapping – e.g. meeting obstacles at the edge of the environment after turning – we do not expect high timeout rates immediately. Yet, eventually, we do expect that if it can reach the goal $> 95\%$ of the time for 1-obstacle environments, it should be able to do the same here, given a similar "optimal" policy. However, this reasoning is slightly unfair since the agent now has to perform three times more collision-free manoeuvres

than in the previous case. Thus, if the agent can manoeuvre past one obstacle with 95% success, it is only fair to assume a success rate of $95\%^3 \sim 85\%$ for the three obstacle environment. From this perspective, we then see that the policy is improving since it does achieve an over 90% timeout rate at its best point.

7.1.4 5 Obstacles

When approaching more cluttered environments, an effective navigation policy has to learn to carefully judge actions as a function of more than one visible obstacle. It cannot simply navigate to one side of an obstacle but base the extent and direction of its turns from what it sees. This is a natural consideration that even takes time to learn, even for humans. We thus expect that policy learning will progressively take longer in more complex environments as the agent learns to fine-tune this behaviour. Based on this assumption, we train the policy for significantly longer ($\approx$ 6800 iterations) and increase the end episode timestep T to compensate for manoeuvring time. The result of which is shown in Figure 7.5. Once again, we observe from the figure a sim-

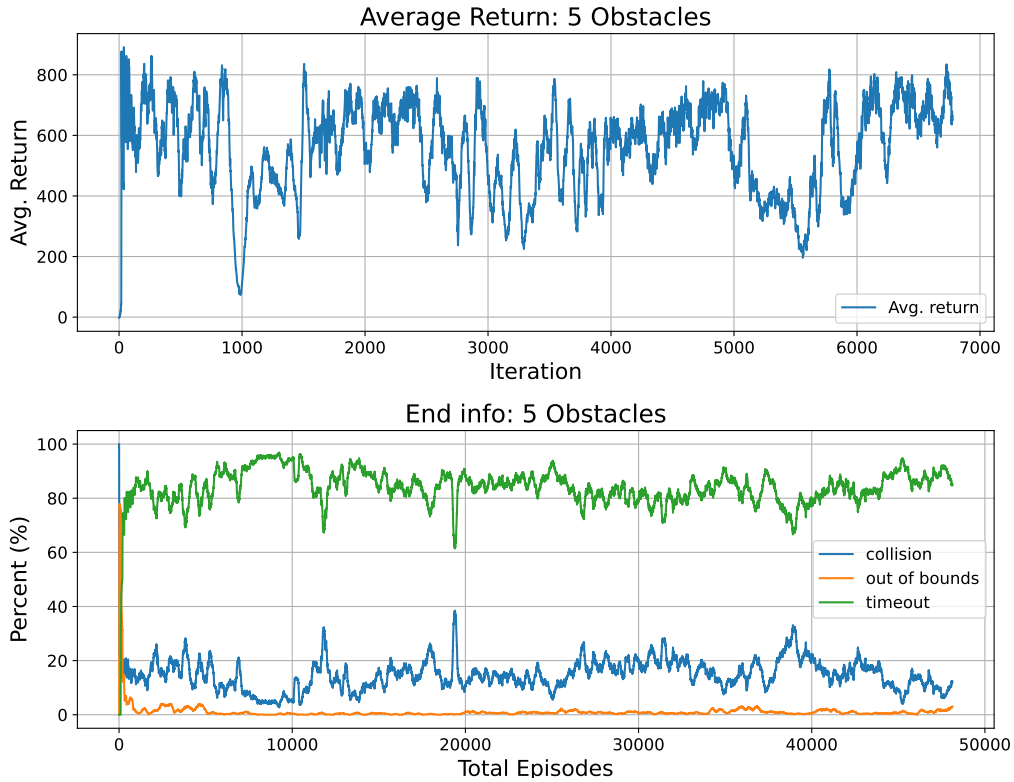


Figure 7.5: Training with 5 obstacles. The best model is selected at 1498 iterations, with an average return of 720 and timeout rate of 94.8%.

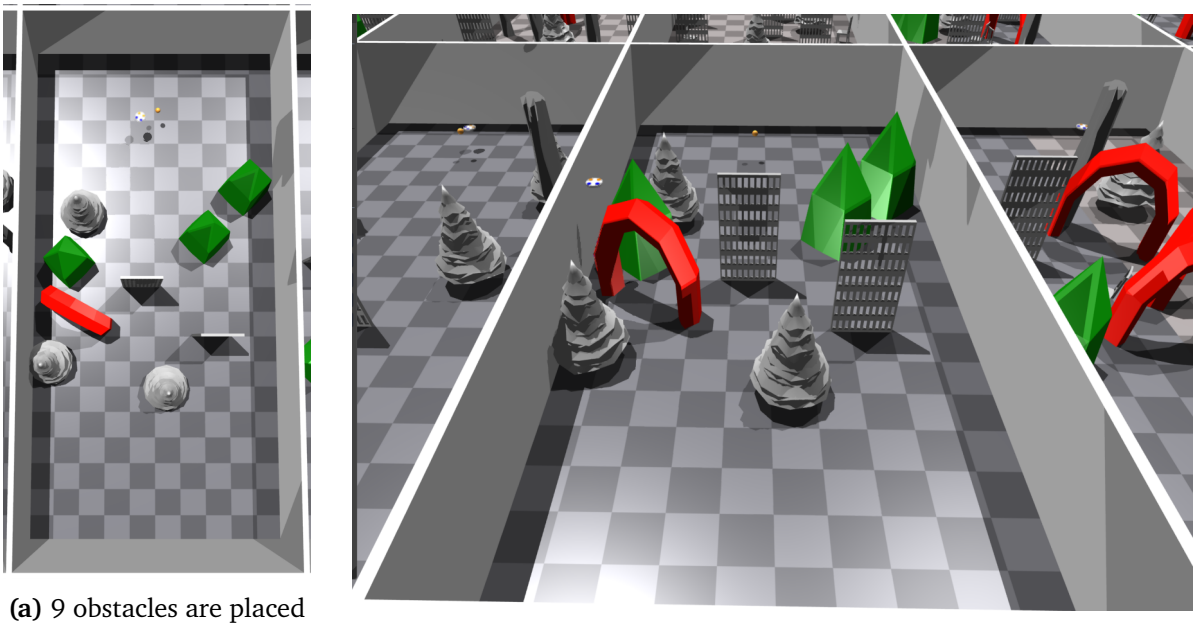
ilar pattern for the return in the first 1000 iterations. However, unlike the 3-obstacle case, we

recognise that the environment is indeed more difficult, resulting in a more complex optimisation space that causes higher fluctuations in the accumulated rewards when the agent explores different trajectories. This is also more pronounced due to the longer episode times, such that if a policy does well to reach the goal, it will accumulate significantly more rewards than one that does decent but does not reach the goal as closely or as often.

Another promising feature of this training plot is that the timeout rates are consistently higher than for the 3-obstacle case, even when the reward is high. The reason for this is quite indirect, though it will be discussed later in Section 9.1.2. Keeping this optimistic perspective, we can then move on to 9-obstacles.

7.1.5 9 Obstacles

The most difficult environment of the training process consists of 9 obstacles in a 20×10 environment. Accounting for the quadrotor and goal – a 6m spacing as seen in – the effective area for obstacles is an 8×10 square, or an obstacle every metre for 8m. For perspective, we provide two images in Figure 7.6.



(a) 9 obstacles are placed from along $x \in [-4, 4]$.

(b) Agent must display collision avoidant properties to pass.

Figure 7.6: An example environment with 9 obstacles, with size 20×10.

Following the same idea for the 5 obstacle case, we provided ample time for policy training and kept the episode length T the same to obtain the results in Figure 7.7. Here, we see that the average return follows an upwards trend to about 1500 iterations, which reflects the difficulty of the environment as the agent takes a much longer time to fine-tune its behaviour. Interestingly,

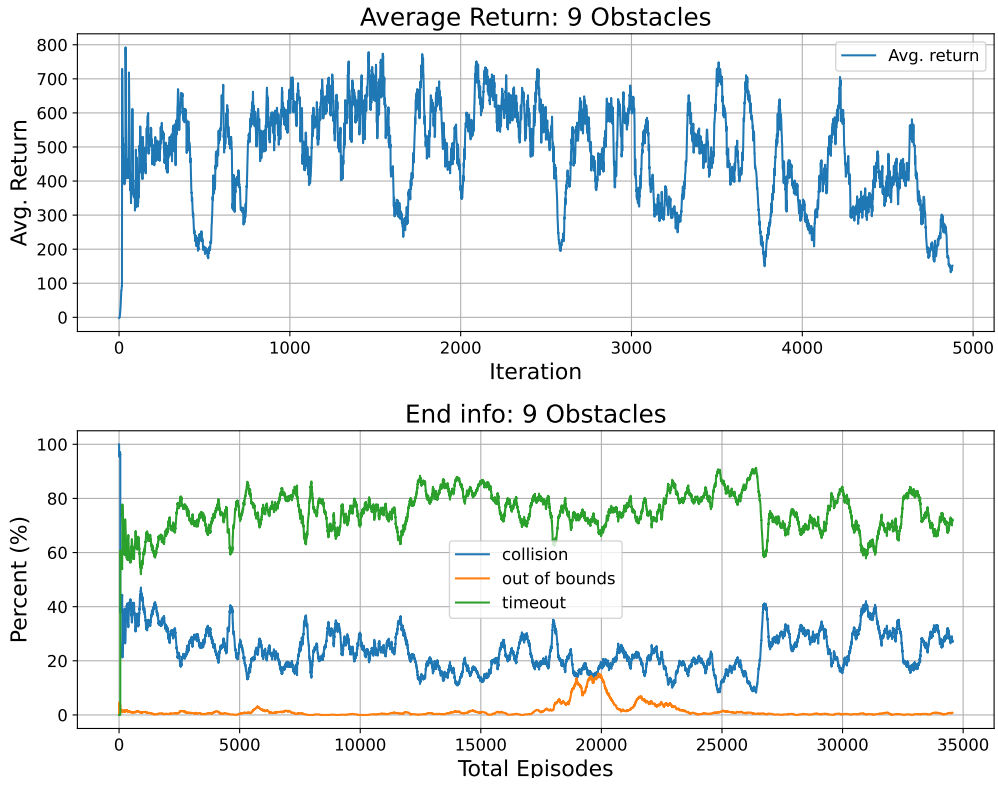


Figure 7.7: Training with 9 obstacle. The best model is selected at 3500 iterations, with an average return of 700 and timeout rate of 84.7%.

we see that this trend also applies to the timeout rate but only to about 1000 iterations. This suggests that the rewards for collision avoidance and goal-reaching are aligned to some degree, but at some point, the rewards motivate the agent to learn risky behaviour – prioritising to reach the goal in situations where it could be more careful.

However, after extensive sampling, we see that the agent policy begins to learn very promising collision avoidance attributes between iterations 3200 $\sim$ 3600, where we select our best model at 3500 iterations. In this situation, we also note that the average return varies dramatically. To reason for this, we can assume that when the agent is learning to navigate carefully, i.e. the policy is aware of state-actions pairs that cause collisions, the conservative behaviour could mean reaching a goal is unlikely. This is in contrast to the policy at around 1500 iterations which learned to "consistently" reach the goal, though ironically with crashes. So in our situation, we can view the model at 3500 iterations as having a good balance of both, where the policy has "stumbled upon" a favourable locally optimal solution with good navigational abilities.

The reason that conservative policies have a hard time reaching goals is also linked to the observation that collision rates are relatively high. Unfortunately, this is expected and can be explained by the agent having to manoeuvre past more obstacles in the previous case. Moreover,

due to the more cluttered nature of the environment, the agent can also experience *getting stuck* in between obstacles where there is no clear way forward. An example of this is shown in Figure 7.6, where we see a corner-type situation on the left side in between the tree and U-shape and can observe a quadrotor flying. Of course, we can think that a quadrotor getting stuck in this situation is unlikely, seeing that it could go right, for example. Yet, with 512 randomly generated environments, we can imagine that these corner-type situations could exist in many forms, both on the left and right side. We can also argue that after passing an obstacle on the left, the optimal path may be to turn and look to the right, but the quadrotor will not see this due to a limited field-of-view (86°) of the camera sensor. As a result, it will enter a corner situation, (and possibly pass) as the agent shown in Figure 7.6b.

So, from this corner situation, if the policy is conservative, an agent will lie and wait as to avoid collision. If it is more aggressive, it will enter and risk collision. Thus, on average, a policy that remains stuck in these unlikely situations will gather much less return than one that collides, and proceeds to find the goal in the next episode – which provides an explanation to why higher timeout rates do not equal higher rewards. However, by chance, a policy that is both conservative *and* can escape corner situations could appear, which is the case for our policy at 3500 iterations shown in Figure 7.6b.

7.2 Evaluating the Learned Navigation Policy

With a policy selected from the curriculum, we can now evaluate its performance in the form of standardised tests in known and unknown environments, along with assessing its robustness to noise. For the known environments, we aim to present three variations of the environment with an increasing degree of clutter, while for unknown environments we increase the environment size. From this, we should be able to gauge its overall performance and more interestingly its ability to generalise to unseen domains and uncertainty.

7.2.1 Known Environments

For the known environments, we test the agent in an environment of size $[20, 10]$, where we increase the number of obstacles from 7 to 12, as shown in Table 7.3. Unlike training, we use a non-randomised environment such that we can visualise the behaviour of different policies in a standardised context. We then run our policy in each environment for 1000 episodes and document the episode end label.

From the table, it is clear that the reinforcement learning agent is extremely successful in navigating *known* cluttered environments, though its performance is bounded by the available space to some extent. Compared to our expectations from training, these are certainly impressive results, given that the timeout rate for the 9-obstacle environment is 96.5% timeout rate

Level	No. of obstacles	Space per obst. (m^2 / obst.)	Timeouts	Collisions	Out. of bounds	Timeout rate (%)
Easy	7	11.4	980	21	0	97.9
Medium	9	8.89	967	33	2	96.5
Hard	12	6.67	771	230	0	77.0

Table 7.3: For the known environments, we evaluate the agent's response to a variation in the cluttered-ness of the environment. We choose an environment size of $[20, 10]$, which is an 8×10 or $80m^2$ effective obstacle area.

compared to 84.7% in Figure 7.7. An explanation for why we did not see such a high timeout rate in training is because the end information plots are *averaged* over 1000 episode ends. Following this, an indication which shows very good collision avoidance in the policy is that the *slope* of the timeout rate is very high at 3500 iterations – indicating a much higher timeout rate under the current policy, which pushes the average up. Otherwise, it could also be that the environments in training are randomised, which suggests that the environments used for testing are "too easy". To see for ourselves, we can look at Figures 7.8, 7.9 and 7.10.

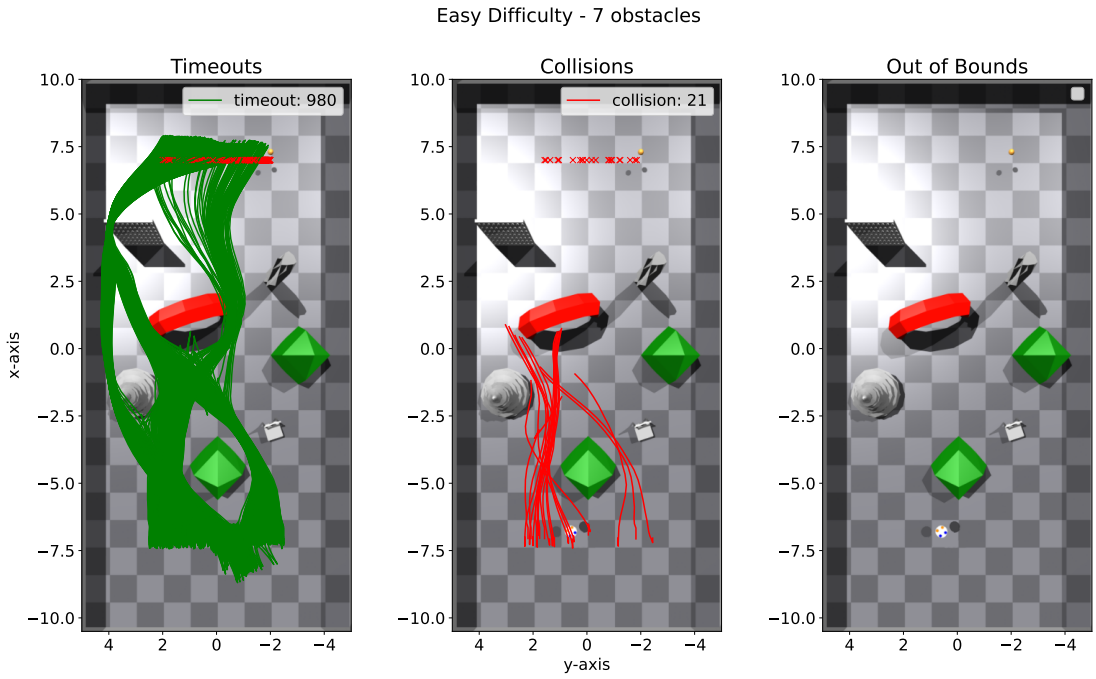


Figure 7.8: The environment with an easy difficulty with a timeout rate of 97.9%. There are 7 objects in this environment, where multiple are placed in the the agent's line-of-sight so that collision-avoidance is necessary for all trajectories. Despite this, openings are relatively spacious compared to difficult environments.

From these, it is evident that the degree of clutter is increasing, but also the number of

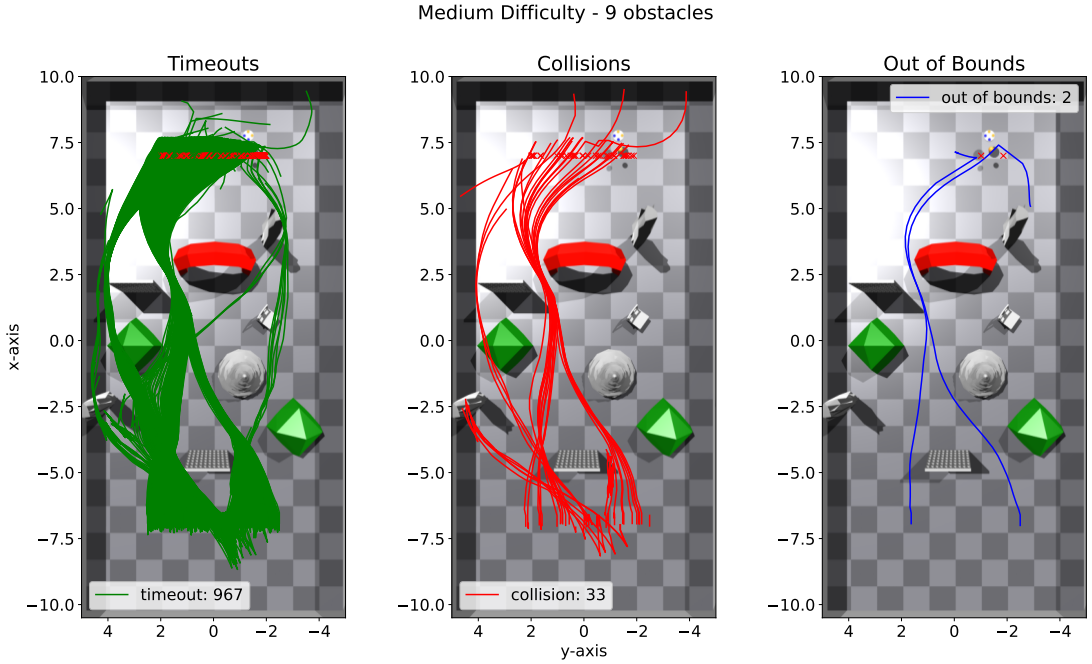


Figure 7.9: The environment with an medium difficulty with a timeout rate of 96.5%. 9 obstacles are placed so to reduce the size of openings and increase the average number of obstacles to pass per trajectory. This results in a large trajectory distribution and roughly 50% more collisions than the easy environment.

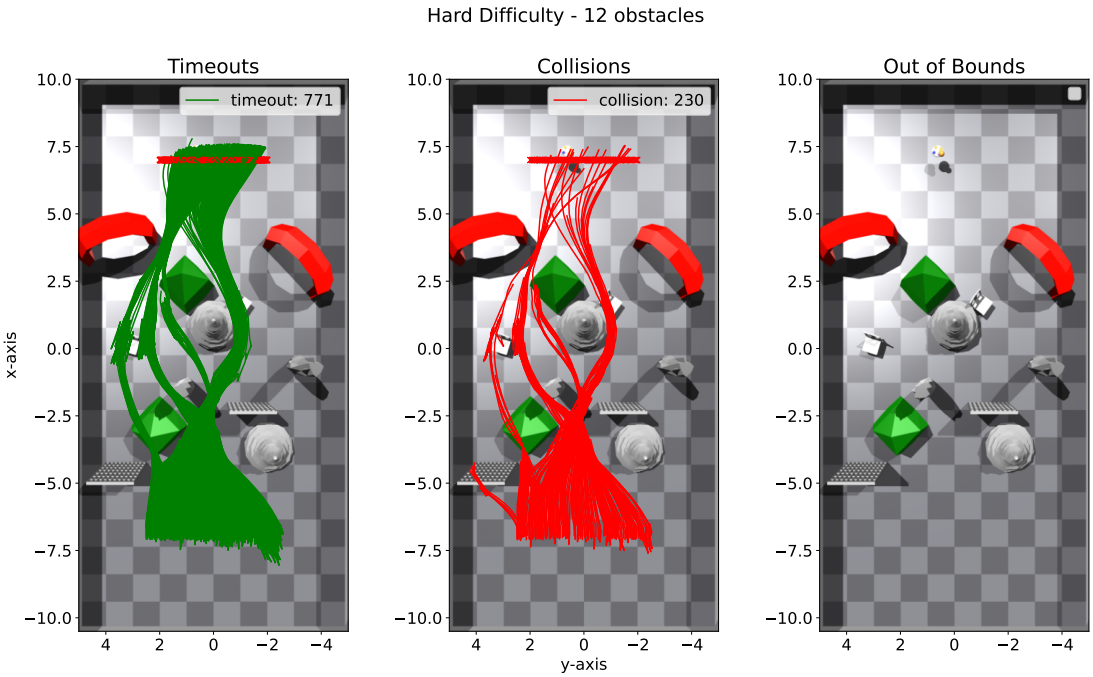


Figure 7.10: The environment with an hard difficulty with a timeout rate of 77.0%. We test the agent's leftward bias by reducing opening sizes further on the left side. Turns are also much sharper, which induces many pass-by collisions.

obstacles that the agent is forced to avoid in order to reach the goal. For the easy environment, we see that any chosen trajectory must avoid on average 3-4 obstacles, which increases to roughly 4 in the medium environment, and to roughly 5 in the most dense environment. We also note that the size of the openings along the agent trajectories significantly decreases for each environment. However, we do recognise that there are no corner-like situations, where we intentionally trap the quadrotor. From these observations, it is plausible that the existence of these corner-situations account for the majority of collisions, since we see that the policy is more than capable of navigating past 4 obstacles with a timeout rate of 96.5%.

To look more into the detail of the quadrotor behaviour, we observe that the policy is heavily biased to making left-turns, despite it being unnatural or completely unnecessary. This can be seen the easy plot, where after avoiding the first pyramid the simple-u in the center, the quadrotor decides to turn extensively to the left despite being able to move straight towards the goal. The result of this bias is of course collisions, as we can see in Figure 7.8, where all collisions are caused by forcing entry through the tight space between the simple-u and the pine tree.

Another, quite unexpected, behaviour that the quadrotor has learned is to be able to *reverse*. We see when the quadrotor spawns directly in front of the simple pyramid, quite a few of the trajectories first *go backwards*, before reaching goal and being marked as green. This behaviour is unexpected because we provide no rewards for reversing, except for a < 1m from obstacle penalty. In fact, we actually provide penalties for reversing – thus actually discouraging it. So surprisingly, we see that this behaviour has been learned completely learned through sampling, where the policy decides that if its vision is completely blocked by an obstacle – it should reverse in order to reach goal.

The obvious consequence of this behaviour, however, is that it induces potential crashes due to blindness. This is most visible in Figure 7.10, where the quadrotor collides with the chair on the left side after reversing due to the tight placement of obstacles. Perhaps unclear, in Figure 7.9, some of the collisions are a result of the quadrotor reaching goal, but reversing slightly to adjust its placement. However, sometimes this results in excessive reversing, to which the the quadrotor collides with the wall.

Another common reason why collisions occur is not due to the necessity of collision avoidance, but rather to an uncareful approach to goal. As we see in Figure 7.9, many trajectories do reach goal but seem to result in collisions. By doing a quick investigation, it can be seen that these are either from reversing or from descending. The need for descending is a result of the quadrotor attempting to fly as high as possible during the obstacle course, as most obstacles are become thinner with height, while the goal is between $z \in [0.5, 1.5]$. Yet, though it is effective to fly above these obstacles, its decent is still imperfect, which results in collisions either with the ground near goal, or with e.g. a chair in Figure 7.10.

However, the two most dominating causes for collisions is a result of *pass-by* collisions and *tight* collisions. The pass-by collisions can be described as when the agent is forced to fly around

an obstacle when approaching it parallelly, while the tight collisions are when the quadrotor aims to enter a narrow opening, but executes it imprecisely. Examples of these are seen in Figure 7.10, where the agent crashes with the fence, simple stone and pyramid in the early center and the pyramid at the end. To identify possible reasons why these occur, we can imagine that in pass-by collisions, just about when the quadrotor passes an obstacle, it disappears from its field of view. Normally this is fine, but when the quadrotor approaches it parallelly, the obstacle is so close that when the quadrotor turns or descends while moving forward, a collision can occur. Tight collisions are very related, as they require an agent to pass directly through the center from a perpendicular approach. If coming from an angle, this requires turning through the center but may result in a pass-by collision. We can link these two through the concept of *space*, which we discuss in the next section.

Effects of Obstacle Placement

As mentioned, one of the most significant differences between the hard environment compared to the easy and medium ones is not the number of obstacles, but rather the availability of space when passing obstacles. We see this quite clearly in Figure 7.10, where the agent chooses among four very tight trajectories on the left, and does not exploit the open space on the right. This is of course a result of its learned behaviour, where it simply maps various obstacle representations in its latent space to a positive yaw rates.

We can expect that therefore, if we create an artificial corner on the left side – i.e. obstacles tightly placed together – we can dramatically reduce its performance as we exploit the policy's weakness. Conversely, if we facilitate navigation on the left side through slightly larger openings, we can postulate that the collision rates will significantly drop, regardless of the number of obstacles it must pass on the way. To evaluate this theory, we perform another test on the hard environment, where we switch the simple stone (pillar) with the less obstructive chair on the left side (for reference see the out of bounds plot in Figure 7.10). From just this modification, keeping everything else constant, we obtain the results in Table 7.4. The results do confirm

Level	No. of obstacles	Space per obst. (m^2 / obst.)	Timeouts	Collisions	Out. of bounds	Timeout rate (%)
Hard, swapped	12	6.67	987	14	0	98.6

Table 7.4: By replacing the simple stone with the chair, we allow a larger opening in the center that significantly impacts our results. We note specifically a drop in the number of collisions, from *230* down to only *14* when simulating for 1000 episodes.

our hypothesis, and to a very large extent. We can also see its effect on the distribution of trajectories in Figure 7.11. As a result of the switch, almost all the trajectories pass through the center opening, directly above the chair. Following this, the quadrotor is no longer forced into

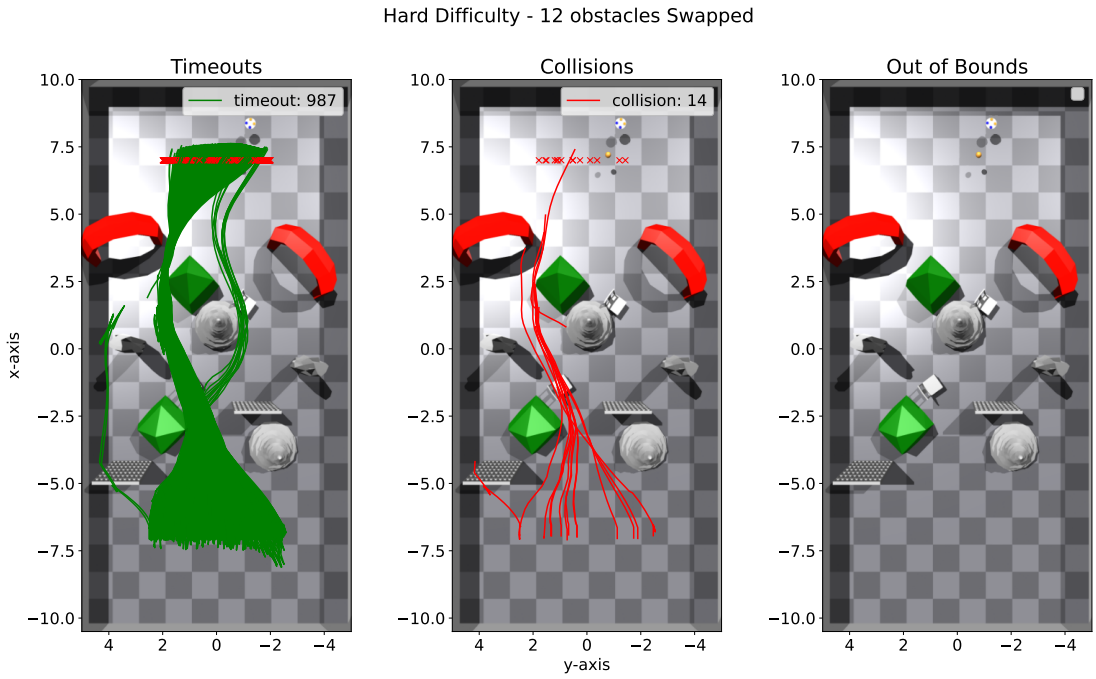


Figure 7.11: The hard environment where the chair on the left is swapped with the simple stone in the centre. This results in a timeout rate increase from 77.0% to 98.6% despite the agent having to account for the same number of obstacles.

a position where it has to pass the end pyramid from a parallel approach, which eliminates the collision probability substantially. We can also assume that simply shifting the simple stone half a metre (half a square) to the right, we will obtain a similar result. This shows that facilitating space along an an apparent quadrotor path is more important than the number of obstacles the quadrotor has to pass.

Of course, it will be interesting to evaluate the performance of the policy further – blocking off paths, etc. – though in light of the purpose of this thesis, i.e. we must remind ourselves that we are designing a policy for local motion planning in cluttered environments and not a path planner through extremely dense ones. From a practical aspect, we can thus emphasise that it is important to weigh the strengths of a local motion planner when for example combining it with some global waypoint planner.

7.2.2 Robustness to Noise in States and Actions

With overall very successful results, an important evaluation that must be done is one of the performance of the quadrotor when presented with uncertainty. To add noise to our quadrotor

states and actions, we follow the documentation in Isaac Gym[1] and emulate Gaussian noise $\epsilon_n \sim \mathcal{N}(1, 0.2)$ with mean of 1 and variance of 0.2, which we multiply to all quadrotor observations and actions. Multiplicative noise was chosen as opposed to additive due to the fact that we do not normalise our states directly (we leverage IsaacGym's implementation as mentioned in Section 6.3.2), such that we obtain we ensure proper scaling of noise to all variables. We also ensure that the depth images are noise-free at this stage. With this added noise, we obtain the results in Table 7.5.

Level	No. of obstacles	Space per obst. (m^2 / obst.)	Timeouts	Collisions	Out. of bounds	Timeout rate (%)
Noisy Easy	7	11.4	987	13	0	98.7
Noisy Medium	9	8.89	959	34	7	95.9
Noisy Hard	12	6.67	804	262	0	75.4
Noisy Hard, swapped	12	6.67	831	207	4	79.8

Table 7.5: For the noisy environments, we evaluate the agent's response to induced noise in all of the quadrotor states and actions, where each state is multiplied with $\epsilon_n \sim \mathcal{N}(1, 0.2)$. We perform this test to all known environments.

Surprisingly, the agent does very well to adapt to noise, having more-or-less the same collision statistics. We note that the main difference is in the hard-swapped environment, where the number of collisions has increased from 14 to 207. Otherwise, we also see a slight increase in the number of out-of-bounds events. To get a better understanding, we can examine Figures 7.12, 7.13, 7.14 and 7.15.

The first consequence of noise is that we observe a greater distribution of trajectories taken towards goal. If we consider what states the quadrotor has, the ones which are the most significant for waypoint navigation are the position, velocity and yaw rate. Since we add noise to these states, we can imagine that the quadrotor applies corrective behaviour to its action in response to changes in its state, which explains a more raggedness in the trajectories of the quadrotor. However, the a possible risk is the combination of corrective actions along with noise in the action – which could lead to imperfect manoeuvring and thus collision.

From the easy environment in Figure 7.12, we note that the increased risk of collision is almost non-existent, as the collision rates have actually dropped in comparison to before. This affirms the quadrotor conservative behaviour where it does not necessarily take risky behaviours and maintains a reasonable distance from obstacles. However, in more cluttered environments, we see that the quadrotor is more susceptible to collisions simply because it is forced to take narrow routes. This is also apparent in Figure 7.15, where previously it could navigate fine

[1]See domain randomisation: `https://github.com/NVIDIA-Omniverse/IsaacGymEnvs/blob/main/docs/domain_randomization.md`

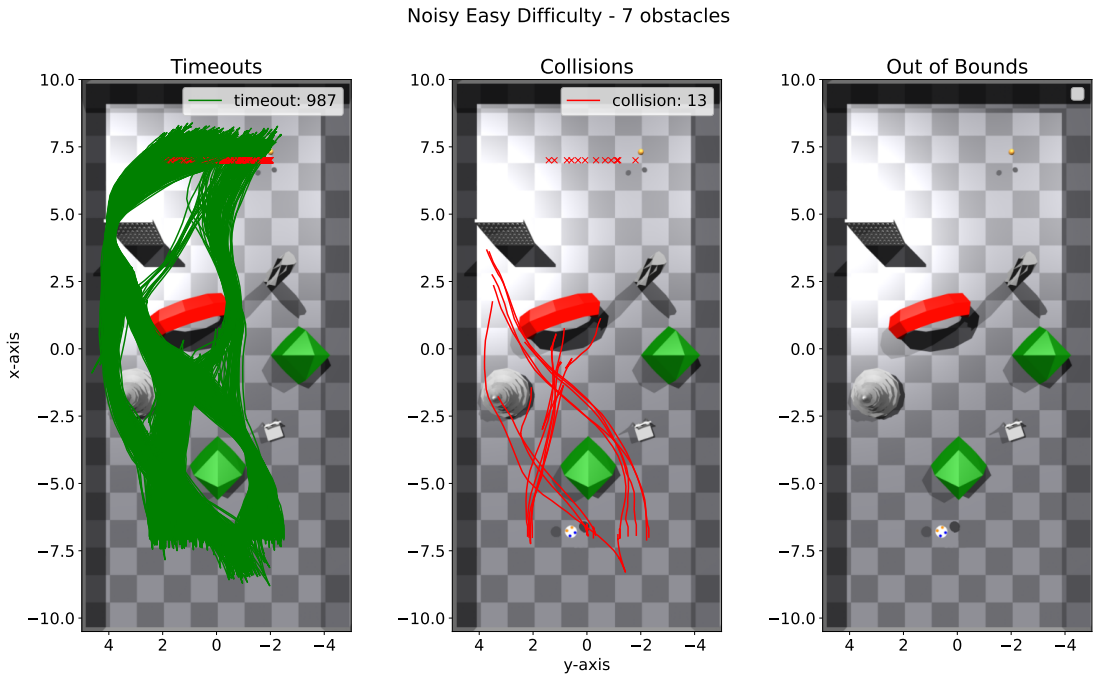


Figure 7.12: Noisy easy environment, with a timeout rate of 98.7%. The significance of noise is not as prevalent in the collision statistics for open-space environments.

through the opening at the end, though suffers the bulk of its collision there in the noisy case.

Another feature of noise is that we see a greater display of reversing, particularly in the hard environment by comparing Figure 7.10 and Figure 7.14. This suggests that the quadrotor has learned not only to reverse when the path ahead seems blocked, but also in the case of poor quadrotor placement in regards to obstacles. Rather than risking a collision, it was observed that the agent would simply oscillate back and forth as an attempt to reorient itself or find new paths. This also illustrates why a conservative policy would gather less rewards, while managing to maintain low collision rates.

Despite the reasonable noise in the agent states and actions – a $> 60\%$ chance of more than a 10% noise factor – the quadrotor still outperforms our expectations, where see that it is only in the very tight spaces where we observe the adverse effects of noise. As mentioned, this can be attributed to the quadrotor deciding on conservative paths to avoid being close to obstacles and it backing away if it perceives the path to be too risky. However, another justification is that PPO makes use of a stochastic policy from which actions are sampled. This means that there is already a layer of randomness in the agent actions, which the policy has has accounted for to ensure robust performance, which therefore minimises the effects of additional noise in the action space.

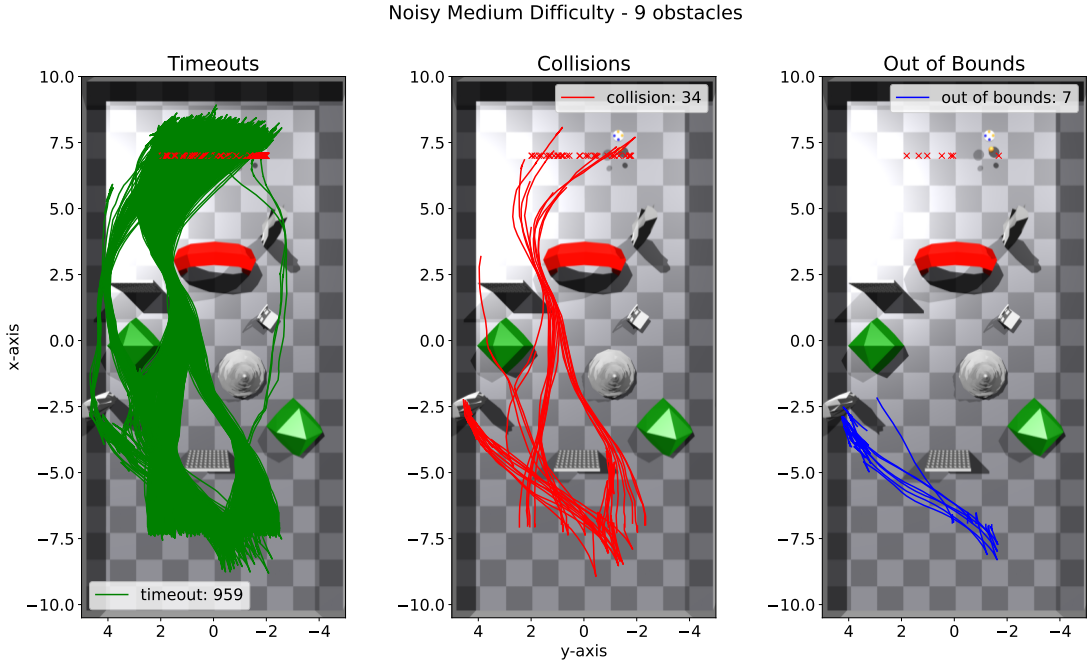


Figure 7.13: Noisy medium environment, with a timeout rate of 95.9%. The policy is still capable of entering tight spaces when approaching them directly.

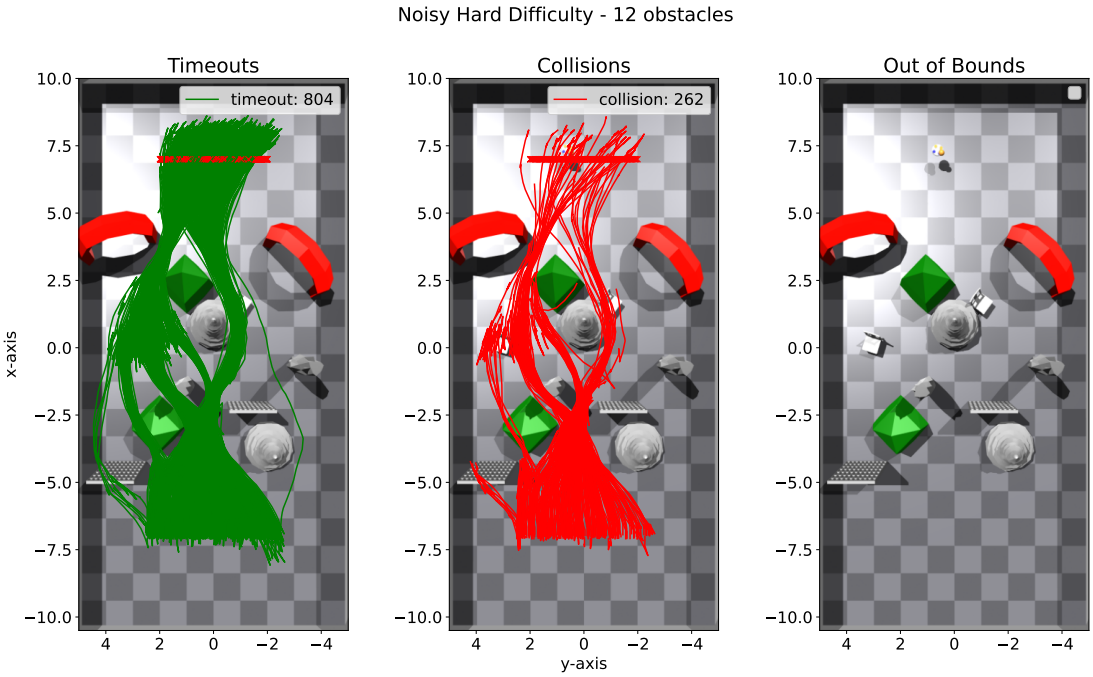


Figure 7.14: Noisy hard environment, with a timeout rate of 75.4%. The effects of noise are more prevalent when careful navigation is required. A larger distribution of trajectories is observed along with much more reversing.

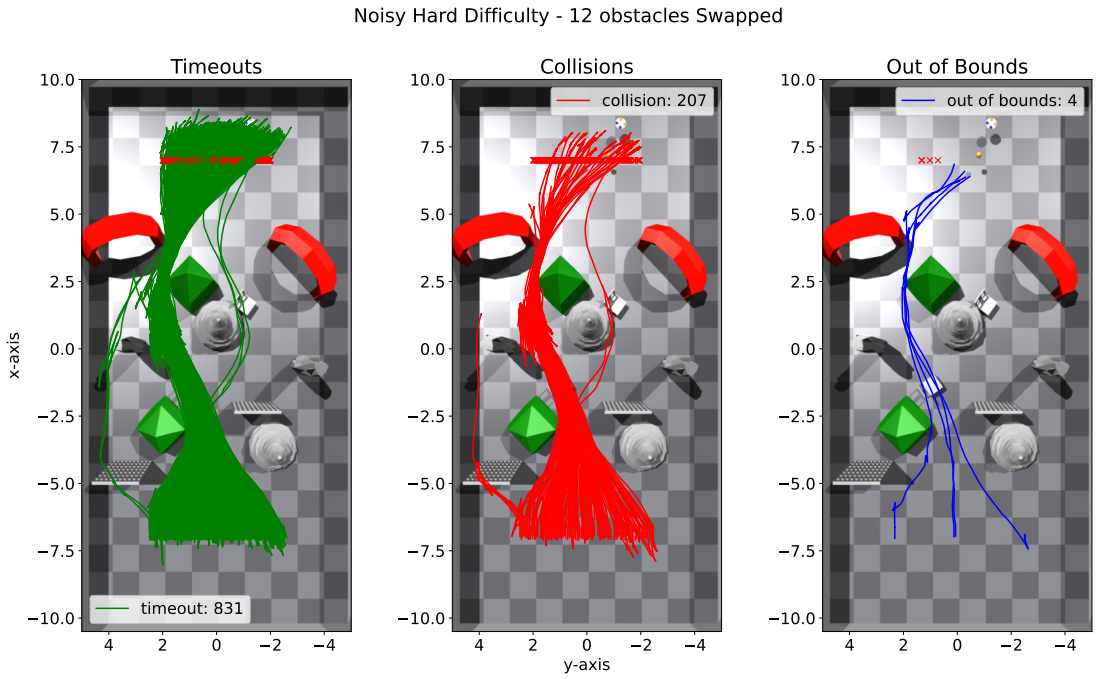
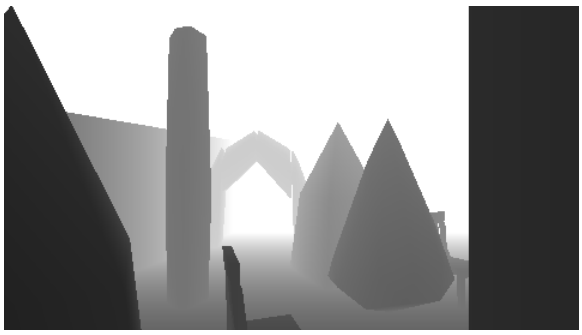


Figure 7.15: Noisy hard environment with swapped simple stone and chair, and a timeout rate of 79.8%. The policy performance in very tight spaces is significantly impacted due to small variations in observed state and actions. As a result, many more collisions occur in along the formerly optimal path.

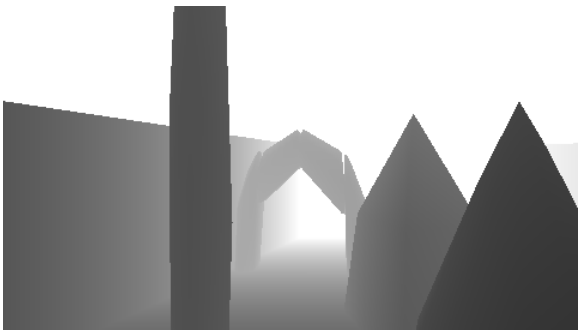
7.2.3 Robustness to Noise in Depth Images

Another possible reason for why the policy adapted well to noise in the previous chapter is because the depth images were still perfect. We justified that the positions, velocities and yaw rate observations were perhaps not that important since the quadrotor has planned good paths that account for stochastic behaviour. But what happens if we add noise to the agent's main tool for motion planning? To explore this, we perform another robustness test, this time to both the depth images, and the state and actions of the agent. We also choose the noise to be *additive* white Gaussian noise instead of multiplicative, with the noise parameters $\epsilon_n \sim \mathcal{N}(0, 0.05)$. Effectively, due to the "naive" processing of depth images in our agent, this results in many white and black speckles in the depth images received by our VAE, seen clearly in Figure 7.16.

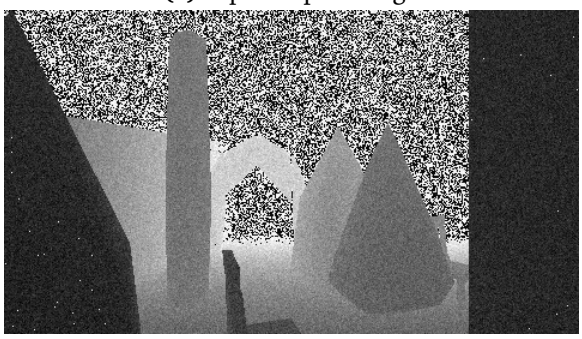
We can observe that the simulated noise is not that realistic, as compared to e.g. the ones in [9], as is by no means an example on how to train a quadrotor for zero-shot transfer to real environments. Yet, since the agent has never experienced any effects of noise to depth images, it serves to illustrate the consequence of suddenly experiencing it.



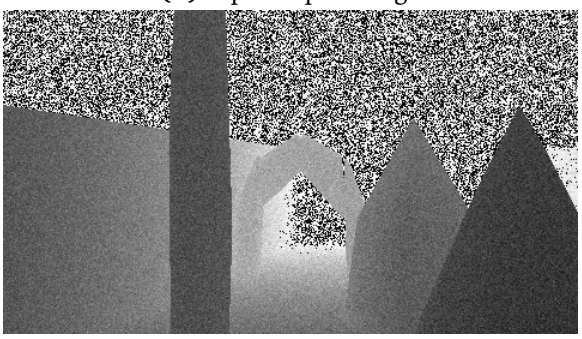
(a) Input depth image



(b) Input depth image



(c) Noise added



(d) Noise added

(e) Decent reconstructed output

(f) Poor reconstructed output

Figure 7.16: The effects of additive Gaussian white noise $\epsilon_n \sim \mathcal{N}(0, 0.05)$ to our depth images received by our network. Their reconstructed depth images are also shown. Recall that the VAE reconstructs a filtered version of inputs. We observe that in some cases performs decently, while other times imagining "phantom obstacles" that are very close.

We see the adverse effect of noise in Figure 7.17, where the performance of the agent is severely impeded. First, we can note a very varied trajectory distribution and substantial amount of collisions. We see that this test demonstrates the importance of accurate depth images for motion planning whereby the agent struggles to map the noisy depth images to proper actions. As a result, we observe excessive turning in all directions, and also a lot of reversing. This results in a lot of collisions as the experiences many pass-by collisions and reversing collisions.

Interestingly, a rare behaviour that was not previously observed was the policy's ability to

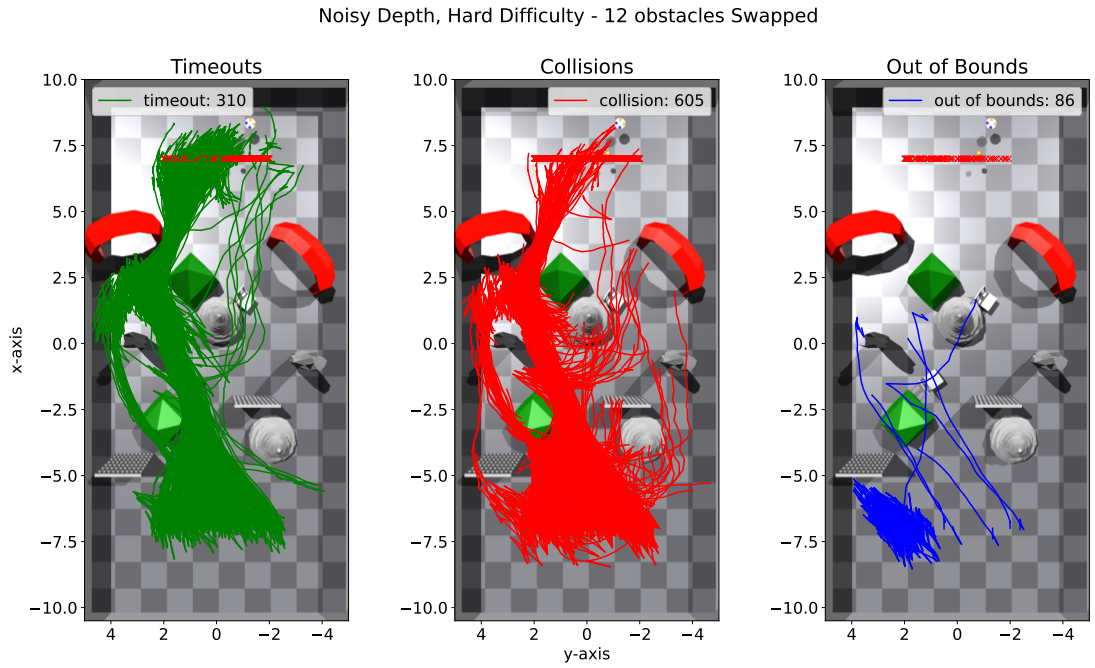


Figure 7.17: The swapped hard environment with Gaussian white noise $\epsilon_n \sim \mathcal{N}(0, 0.05)$ added to depth images, agent states and actions. The timeout rate is now at 31.0%, dramatically reduced from previous cases.

travel backwards (opposite of $\boldsymbol{p}_t$) so to navigate past obstacles and reach the goal. This can be seen in the timeout plot where the agent frequently attempted to make its way around a pine tree. However, we see that in many cases this resulted in crashing directly into the top of the tree.

Overall, this performance is expected since we have never exposed either our VAE our navigation policy to noisy depth images. This leads to a propagation of error through our model, since we are unaware of the consequence this noise has in our latent space, nor the effect this noise-in-the-latent-space has on our agent actions. Nevertheless, similarly to [13], this can be mitigated by training the VAE to also learn how to filter real or noisy depth images, such that it performs both dilation and denoising. Otherwise, we can also simulate noise our simulation depth images and train a policy which receives these noisy depth images as input, such that it learns to account for this in some degree, like in [9].

7.2.4 Larger Environments

For our final evaluation study, we attempt to run the policy in larger environments. This should demonstrate the generalisation capability of the policy when introduced to a new state-space distribution. Particularly, we have observed that the agent is capable in decently sized environ-

ments, adept at navigating through 8m of obstacles. In this case, we set the environment size to be 30×15, such that the effective obstacle area is $18 \cdot 15 = 270$, and set the number of obstacles to be 25. With this, we obtain the results in Figure 7.18.

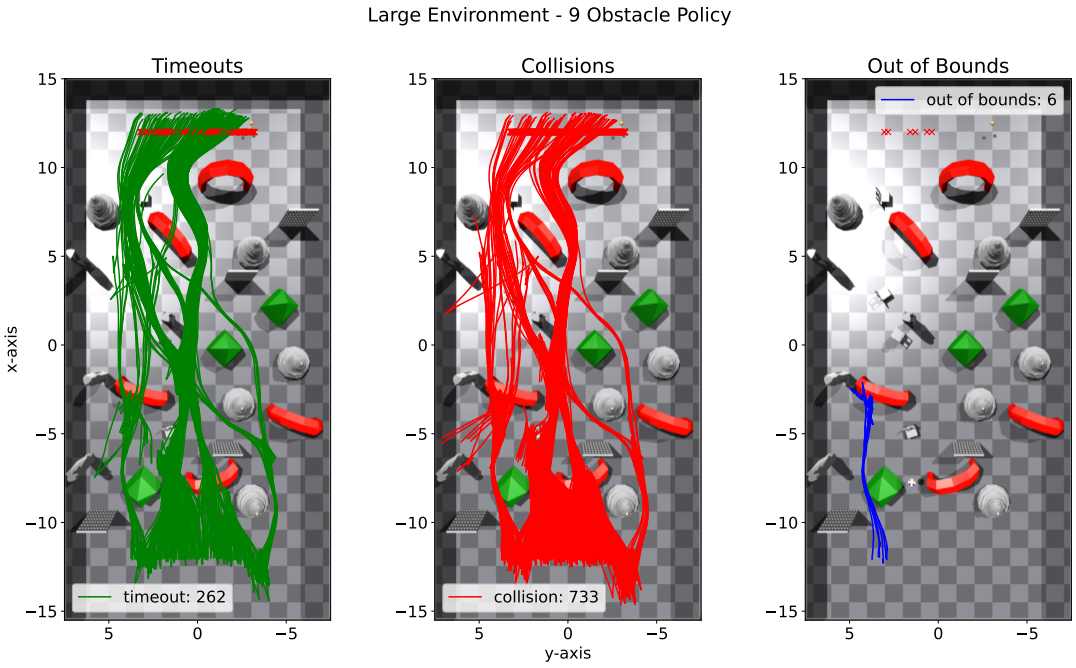


Figure 7.18: The large environment with 25 obstacles in an area $[30, 15]$ area. The timeout rate is 26.2%.

Based on the results, we see that the policy has performed poorly, colliding with obstacles 73.3% of the time. There could be several reasons for this, the first of which is a poor generalisation to larger environments, for example through seeing obstacles relatively early and much later. This means that because there is a lack of experience in these states-obstacle combinations, we cannot expect that the policy provides as finely-tuned actions as in the normal cases. We also can observe that the trajectories taken by the agent are mostly confined in $y \in [-5, 5]$. This further suggests that our agent has learnt to exploit the small environment – essentially "over-fitting" it – which comes at the cost of lack of generalisation.

A more straightforward reason could be that we overestimated the policy ability to navigate past obstacles, such that when tasked to avoid roughly seven obstacles per trajectory more mistakes occur along the way. Though not so easily visible, we should also note that many of the collisions were early on as a result of trying to fly over the simple-u but not having the height to manage it. This could be due to the low spawn point of the quadrotor at roughly 1m while needing to ascent to about 3m to pass above the simple-u. An alternative to passing the simple-u is to reverse and descend, such that it can go under the it. However, since the quadrotor is so

low during the start, this also resulted in many crashes. This behaviour can also be seen by the simple-u on the left, where we see many back-and-forth motions as the quadrotor descends, unfortunately into the chair.

Nevertheless, we expected the policy to be much better than observed here, since it had essentially no problems before – apart from too-tight spaces, though this is not the case here. So, to demonstrate the potential of this approach, we added two more levels to the curriculum, this time in environments of size $[24, 12]$ and later $[30, 15]$, as shown in Figure C.2. By testing the new policy in the same environment, we obtained the result in Section 7.2.4.

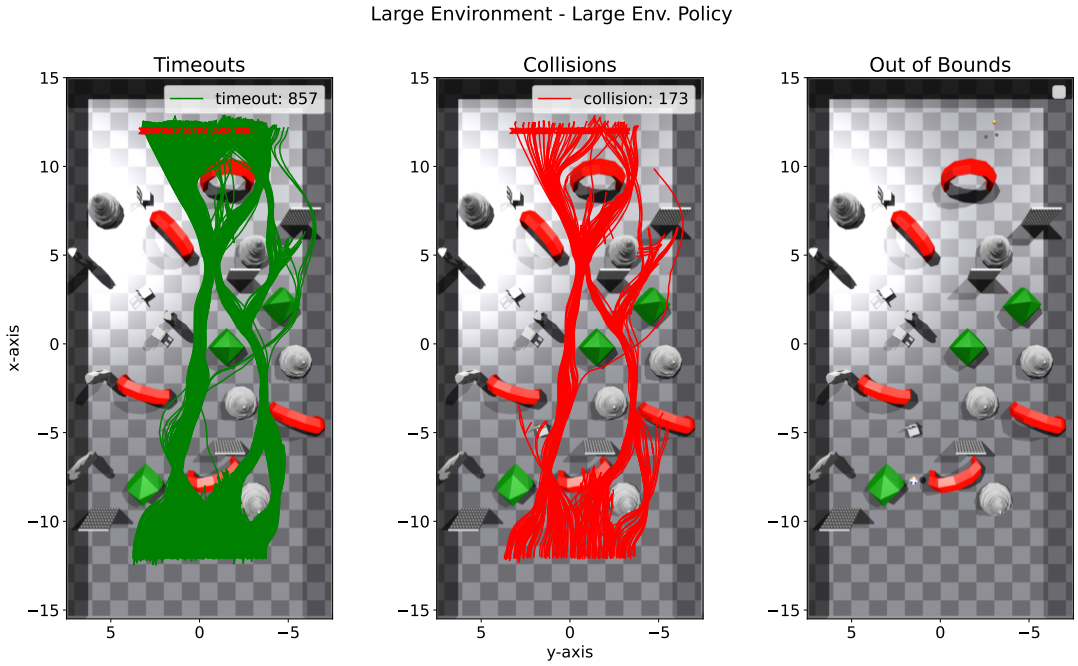


Figure 7.19: The new policy in a large environment, with timeout rate of 85.7%. Training it required 7500 extra iterations compared to the 9-obstacle policy.

Here, we observe a large progression in policy performance, where the agent attains a timeout rate of 85.7%. Despite not matching the statistics from the earlier tests, we reason that due to the increased number of additional turns per trajectory, this is acceptable. From the figure, we note that collisions are still caused by similar factors which we have discussed before – namely attempting to fly over or under simple-u's, reversing into obstacles and corner situations such as on the right at $[5, -6]$. Otherwise, another similar behaviour that it has is a favourite side, though this time predisposed to making right turns.

Chapter 8

VAE Results

In this section, we will briefly evaluate the the performance of our VAE model in regards to its reconstruction ability. We aim to explore the effects of the different modifications to the loss function, as presented in Section 5.2.1. To do this, we will begin with a plain reconstruction error, before adding a depth weighting, and finally edge loss. Moreover, we explore these implementations to both MSE and BCE loss.

An overview of the best models with their modified losses, best epoch and their time-to-train is shown below:

ID	Model	Epoch	Time
1	Vanilla MSE	40	19h 4m
2	Vanilla BCE	50	1d 14h 36m
3	Depth Weighted MSE	60	3d 4h 52m
4	Depth Weighted BCE	60	1d 13h 54m
5	Depth Weighted MSE with Edge Loss	100	1d 22h 49m
6	Depth Weighted BCE with Edge Loss	100	3d 3h 37m

Table 8.1: List of VAE models.

8.1 Training

We train our networks on 202,558 depth images for 100 epochs, where we present their training and test plots. We also train these two at a time, which slightly reduces their training times. Overall, not much can be seen from the training plots, apart from ensuring that training is stable and the models do not over-fit. Normally, the approach would be to compare the losses of each side by side, but since the scale of the losses are dissimilar, we provide this as an indication of how the changes to the reconstruction affect the loss values and training plots.

8.1.1 Vanilla Loss Function

Beginning first with the vanilla reconstruction loss, the overall loss function for a batch B of m of samples is:

$$\widetilde{\mathcal{L}}_B(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{d}^{(i)}, \boldsymbol{d}^{f(i)}) = \frac{1}{m} \sum_{i=1}^{m} \left(\log p_{\boldsymbol{\theta}}(\boldsymbol{d}^{f(i)} | \boldsymbol{z}^{(i,l)}) - D_{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z} | \boldsymbol{d}^{(i)}) \parallel \log p_{\boldsymbol{\theta}}(\boldsymbol{z})) \right) \tag{8.1}$$

Then, the reconstruction loss $\mathcal{L}^{(i)}_{\text{REC}} = \log p_{\boldsymbol{\theta}}(\hat{\boldsymbol{d}}^{f(i)} | \boldsymbol{z}^{(i,l)})$ can be replaced with the vanilla MSE and BCE, which is implemented in practice as:

$$\mathcal{L}^{(i)}_{\text{MSE}} = \left\| \hat{\boldsymbol{d}}^{f(i)} - \boldsymbol{d}^{f(i)} \right\| \tag{8.2}$$

$$\mathcal{L}^{(i)}_{\text{BCE}} = \boldsymbol{d}^{f(i)} \log \sigma \left(\hat{\boldsymbol{d}}^{f(i)} \right) + (1 - \boldsymbol{d}^{f(i)}) \log \sigma \left(1 - \hat{\boldsymbol{d}}^{f(i)} \right) \tag{8.3}$$

where σ is used to denote the *sigmoid* activation function. With this, we present the training curves in Figure 8.1. From the figures, we observe a slight over-fitting for both models, where

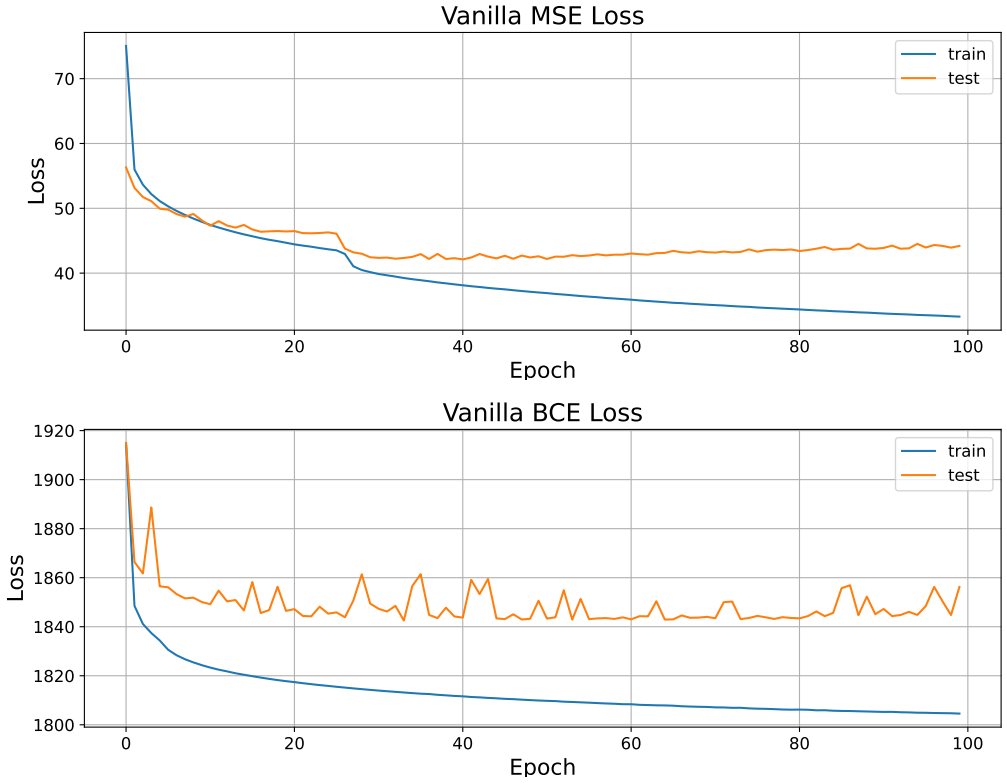


Figure 8.1: Training and test loss for the vanilla MSE and BCE models.

the MSE begins to over-fit beyond 40 epochs, and the BCE around 50.

8.1.2 Depth Weighted Loss

With the vanilla models trained, we now implement the depth weighting according to the input depth image $\boldsymbol{d}^{f^{(i)}}$:

$$\mathcal{L}_{\text{MSE}}^{(i)} = K_{\text{depth}}(\boldsymbol{d}^{f^{(i)}}) \cdot \left\| \hat{\boldsymbol{d}}^{f^{(i)}} - \boldsymbol{d}^{f^{(i)}} \right\| \tag{8.4}$$

$$\mathcal{L}_{\text{BCE}}^{(i)} = K_{\text{depth}}(\boldsymbol{d}^{f^{(i)}}) \cdot \left(\boldsymbol{d}^{f^{(i)}} \log \sigma \left(\hat{\boldsymbol{d}}^{f^{(i)}} \right) + (1 - \boldsymbol{d}^{f^{(i)}}) \log \sigma \left(1 - \hat{\boldsymbol{d}}^{f^{(i)}} \right) \right) \tag{8.5}$$

where the depth gain K_{depth} is given by Equation (5.11). Using this loss, we train our models to obtain the plots in Figure 8.2. Following a very close pattern with the vanilla training plots, we

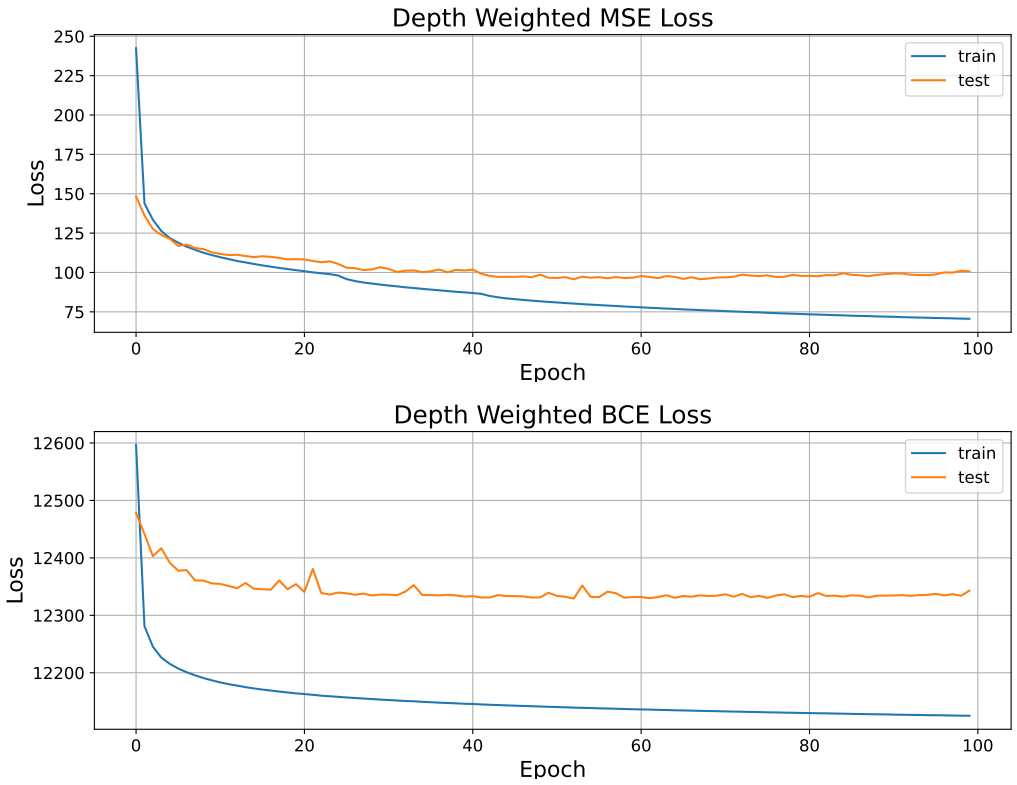


Figure 8.2: Training and test loss for the depth weighted MSE and BCE models.

observe essentially a doubling in the total loss for the MSE, while the BCE loss increases by a factor of 10. Otherwise, both models also show a slight tendency of over-fitting, at around 60 epochs.

8.1.3 Depth Weighted Loss With Edge Loss

Finally, to implement the depth weighted losses with edge loss, we add the mean absolute error (MAE) between the edges of the depth filtered image and reconstruction:

$$\mathcal{L}_{\text{MSE}}^{(i)} = K_{\text{depth}} \cdot \boldsymbol{d}^{f^{(i)}} \cdot \left(\left\| \hat{\boldsymbol{d}}^{f^{(i)}} - \boldsymbol{d}^{f^{(i)}} \right\|_2 + \mathcal{L}_{\text{edge}}^{(i)} \right) \tag{8.6}$$

$$\mathcal{L}_{\text{BCE}}^{(i)} = K_{\text{depth}} \cdot \boldsymbol{d}^{f^{(i)}} \cdot \left(\left(\boldsymbol{d}^{f^{(i)}} \log \sigma \left(\hat{\boldsymbol{d}}^{f^{(i)}} \right) + (1 - \boldsymbol{d}^{f^{(i)}}) \log \sigma \left(1 - \hat{\boldsymbol{d}}^{f^{(i)}} \right) \right) + \mathcal{L}_{\text{edge}}^{(i)} \right) \tag{8.7}$$

where the edge loss is given by:

$$\mathcal{L}_{\text{edge}}^{(i)} = K_{\text{edge}} \cdot E\left(\boldsymbol{d}^{f^{(i)}} \right) \cdot \left\| \hat{\boldsymbol{d}}^{f^{(i)}} - \boldsymbol{d}^{f^{(i)}} \right\|_1 \tag{8.8}$$

where $E\left(\boldsymbol{d}^{f^{(i)}} \right)$ is a function that finds the edge pixels of the filtered depth image through

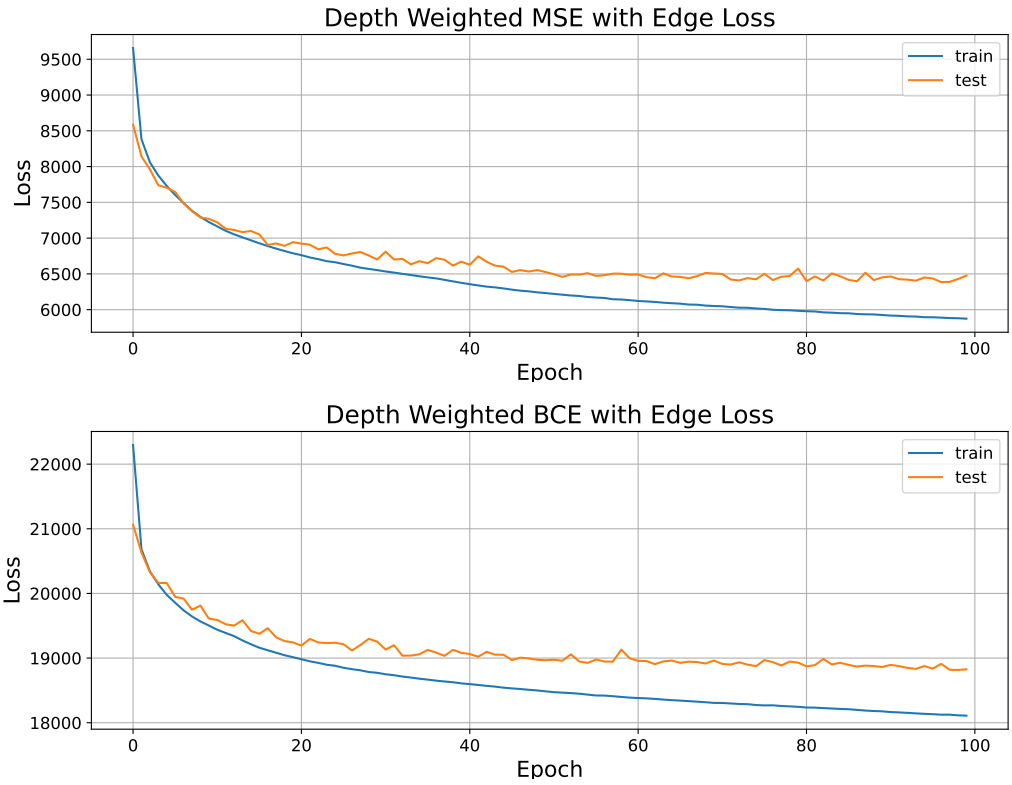


Figure 8.3: Training and test loss for the depth weighted MSE and BCE models when adding an edge loss term.

a canny edge detector [77], applies a Gaussian filter over it, and masks all values over 0 as edges. We also choose the edge gain to be $K_{\text{edge}} = 100$. Training our models for a final time

yields the plots in Figure 8.3. Unlike the previous training plots, the MSE and BCE test loss does not increase even to 100 epochs, indicating that our model is not over-fitting our input data. Otherwise, we see that the addition of the MAE term adds a loss value of about 6000 to both MSE and BCE plots.

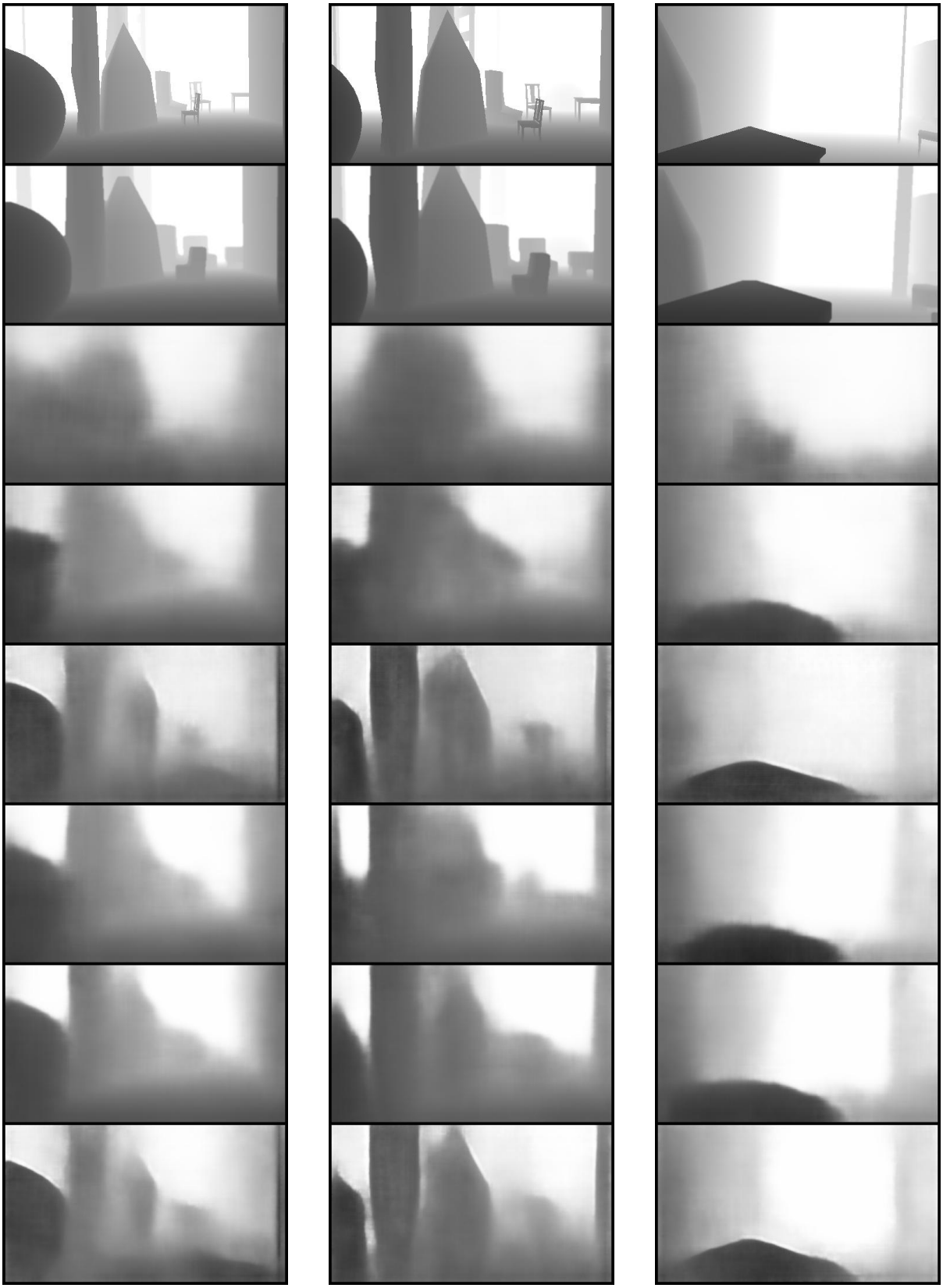
8.2 Testing

So, from the training plots, it is hard to deduce the reconstruction ability of the VAE. This is because we judge this metric quite qualitatively, such that one must instead simply inspect the images either throughout or at the end of the training process. With this, we can examine the reconstruction ability of all models in Figure 8.4.

From the figure, we can note very clear differences in the different loss functions, and also a difference from using MSE compared to BCE as loss functions. We see first that the reconstructions for the vanilla loss functions are much more blur than the depth weight ones, even when we do not add the edge weight. This can be explained by the fact that we consider the 'blurness' as an inability to reconstruct objects – in this case obstacles – in images. However, all objects in the scene have a depth that is much closer than the background, such that if we weight the the loss according to the closeness of pixels, the VAE be motivated to reconstruct obstacles with a higher fidelity.

Moving onwards, we see that by adding edge losses to the depth images, we confirm our expectations that we obtain clearer edges along close obstacles to some degree. This is most prevalent for the MSE reconstructions 6) and 7), where we see that the middle pillar depth is more accurately captured than the non-edge-loss one.

Another big difference is that of the BCE and MSE plots, where we see that the BCE loss performs much better overall in terms of clarity. However, the we can also note BCE plots also assume obstacles are closer than what they actually are, in we look at plots 5) and 8) and observe the colour of the middle grey pillar. Nonetheless, for the purpose of collision avoidance, we chose the last model, the depth weighted BCE with edge loss to serve as our inference network for our model.



(a) Caption **(b)** Caption **(c)** Caption

Figure 8.4: Viewing the some sample reconstructions of our trained models. The downwards order of images are: 1) Input 2) Target 3) Vanilla MSE 4) Vanilla BCE 5) Depth weighted BCE 6) Depth weighted MSE 7) Depth weighted MSE with edge loss 8) Depth weighted BCE with edge loss. Note the flipped order for 5) and 6).

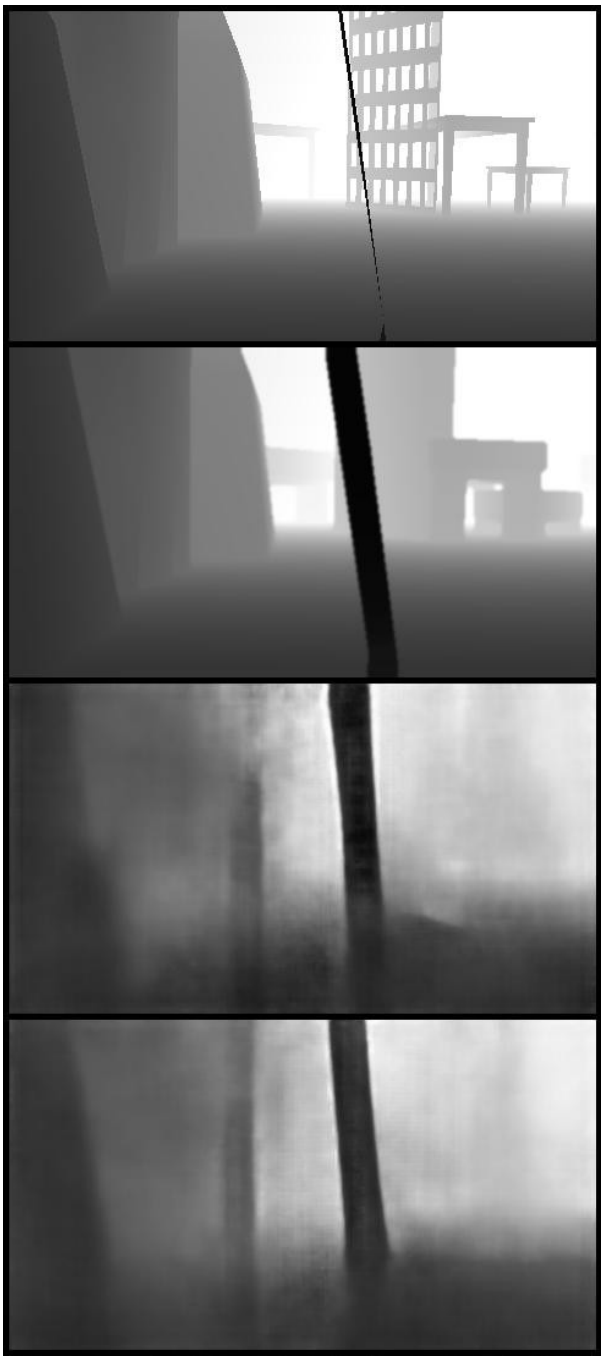


Figure 8.5: Reconstruction of a thin wire observed in our training dataset. We choose the two MSE and BCE models with edge loss, corresponding to the third and fourth reconstruction respectively.

Chapter 9

Discussion

9.1 Learning Collision Avoidance

When designing a complex navigation policy, a combination of multiple factors ultimately shape and facilitate the convergence of the training process. Along the curriculum, we observed several features worth discussion, such as the oscillation of the average return during training, the existence of poor policy configuration spaces and the seemingly bounded timeout rates. In the following sections, we aim to uncover the underlying factors which could give rise to these features, and consider possible improvements.

9.1.1 Oscillations and Low Returns Despite Low Collisions Rates

Two of the most prominent features seen in almost all the figures of Section 7.1 are the oscillations of the average return, and the combination of low returns with low collision rates. Beginning with oscillations, we mentioned that intuitively, these can be accredited to the exploratory nature of a reinforcement learning agent, such that attempting poor actions are a part of the learning process. But, what does this mean *exactly*? We can ask ourselves – why were the oscillations so large, or why did the policy not just continue improving after finding the goal?

These may sound like naive questions, but are in fact a part of large, central themes of deep reinforcement learning, primarily the questions of *exploration* and *stability* during training – to which a multitude of algorithms have been developed to "solve". Taking for example the novel on-policy PPO, some of its benefits include faster training by, "ignoring uninteresting parts of the space" and "faster initial planning" [40], better data-efficiency [47], and also "stronger convergence results" when sampling on-policy [40] and the supposed, "guaranteed monotonic improvement" from trust-region based policy optimisation [48].

To try and explain the oscillations, we have to remember that we have a very complicated state-action space, along with a few loss functions in e.g. actor-critics, that results in a very com-

plex optimisation space. We can provide the common analogy that the policy space is like hills on a grassland, where we wish to perform gradient ascent to reach the highest peak. An example given in Section 7.1.2 was that we observed an early behaviour where the agent had learned to avoid obstacles by passing them always on one side. This example could correspond to a locally optimum solution for that environment – or a hill. However, in progressively cluttered environments, this locally optimal solution does not hold and results in crashes that push the agent away from that locally optimum solution. In a more complicated optimisation space, the hills becomes a mountain range which have sharp maxima due to the high-dimensional optimisation space. So in this situation, if we move away from a locally optimal solution, the agent policy can either diverge if the actions performed are exceedingly poor – resulting in high actor losses and gradient updates that push the policy completely off the mountain – or it could find itself converging to a new locally optimum solution – to the neighbouring mountain. But in between mountain peaks, the observed policy performance would be poor, such that we observe large dips in the average return. In addition, due to accidents and the random nature of our stochastic policy, small deviations that motivate our policy to keep exploring the policy space is inevitable.

However, this theory does not explain how the collision rate stays low in between locally optimal policies. This is a more difficult question, as we cannot simply excuse the good performance to random exploration as we did for the average return. To provide justification to this, we can attempt to explain it through the reward function.

9.1.2 Converging to an Optimal Policy with Sparse Rewards

The only feedback signal an agent receives while searching its complex optimisation space is the notion of the reward. We can think of the reward as the guide for providing gradients in this optimisation space, and the reward function as the one which shapes it. In hindsight, a rather clear aspect of the reward that was not apparent during its design was the problem of *sparse rewards* combined with heavy penalty *shaping*. Sparsity in this context refers to the amount of states in the state distribution that actually provide a reward, while shaping refers to the notion of adding extra reward terms to shape the final behaviour of the quadrotor. In our case, the positive rewards only applied to the states very close to goal, while the added penalties could apply to all states, as seen in Section 5.1.1.

This concept can therefore explain why collision rates were consistently quite low, while "optimal" policies were less common. Since we *shape* the reward function with many penalties, the task of avoiding an obstacle is well-defined meaning that with enough experience, the agent will understand to not crash into obstacles due to its prevalent penalties. In contrast, state-action pairs that should deserve to trigger reward (by passing obstacles safely or to mark progress) do not get rewarded, which can result in the quadrotor having to sample the environment intensively in order to find those rewards to obtain gradients to push it back to a locally optimal

solution space – reflected through the significant, relatively low-frequency oscillations of the reward function.

Though this motivates us to shape the reward function more, it can be argued that ideally we wish to avoid doing so. The main reason is that reward shaping is a adds a layer of human bias on an already very complicated task, which could lead to sub-optimal performance unless those rewards were engineered by some expert in the task. Otherwise, added features could lead to unexpected consequences – as mentioned in Section 5.1.1, "any undesired behaviour that is observed in test-time is often a consequence of a poorly designed reward function". Nonetheless, despite having sparse rewards, we did observe that the agent managed to learn how to solve its training task to a great extent at specific iterations, such as the best policy in for 9 obstacles in Section 7.1.5 which proved to be optimal for its task during evaluation. But then a natural follow-up would be: with such sparsity in the reward function, how did the agent learn how to perform so well across its entire state-space distribution?

A more theoretical explanation follows from the definition of the problem task, and how reinforcement learning agents learn optimal policies. We first note that an optimal policy does not need to learn how to maximise its return from all states, but only the states under the induced trajectory distribution p_π. This is important since PPO is on-policy, such that due to it limited exploration it could perform quite poorly in states uncommon under its current policy $\pi_{\boldsymbol{\theta}}$. This was also observed to some extent in the large environment in Figure 7.18, where the a large proportion of crashes occurred in states which were uncommon. Next, we have to recall that by definition, maximising its return means to maximise the *infinite sum of discounted reward*. This means that even though we have sparse rewards, the expected return for states-action pairs far away from goal are non-zero. In fact, if we take a simple example where the reward function is just +8 if goal is reached, and we assume it takes 160 timesteps to reach goal (the length of our episodes), the expected return for being in its current state should be evaluated to be $8 * 0.99^{160} = 1.6$, where we choose the discount factor $\gamma = 0.99$. In this example, the sum of discounted rewards was simply the last timestep of the whole episode, though for our training setup the reward is much more significant since we allow the reward to be gathered until timeout (on timeout we also bootstrap the critic estimated return to the reward), while the penalties are quite low with the exception of the collision penalty. This means that while PPO is training, even though actions do not produce rewards immediately, the critic network should have an idea of the value of certain states, such that it can calculate the advantage $\widehat{A}_t$ and provide correct gradient feedback to the policy. Therefore, if certain actions incur penalties straight away, but the trajectory ultimately leads to goal, the policy will be updated to increase the probability of doing these actions again (given that the critic did not expect them to reach goal).

Yet, if we continue to larger and more complicated tasks, we can approach a problem where these rewards are very rarely sampled. Thus, if a policy continuously samples a trajectory where

all actions lead to a collision, the critic will evaluate any action at all as bad, since the expected return is less than just remaining still. Therefore, another detail to keep in mind is that the number of timesteps cannot be too low such that the quadrotor has no time to reach goal. If we increase the episode length, the agent will be less "rushed" to reach the goal, and can so take more careful or elaborate routes to obtain the reward. This can also explain why we observed the timeout rates to be higher in the training plots for the 3-obstacle rate compared to the 1-obstacle, as we increased the number of timesteps from 100 to 160. However, the number of timesteps to set should be upper bounded, due the the discounted nature of the return.

To summarise, the expected value of states far away from rewards must be non-zero, such that gradients for actions performed in these states can be evaluated. Then to ensure this, we identify two important factors:

- The number of timesteps between start and goal cannot be too large ($> 400 timesteps$). Otherwise, the expected return for early states would be negative due to the heavily discounted reward and prevalence of penalties in the environment.
- Despite being sparse, rewards cannot be too sparse to the extent they are never experienced. Otherwise, the policy will never be able to motivate any choice of action.

To deal with the first point, we can limit the size of the environment to some maximum size or increase the discount factor to a larger value, such as 0.998 in [49]. As for the second, we can ensure that the agent always has time to reach goal by increasing the episode length some minimum value. However, this is not always necessary if we include a curriculum.

9.1.3 The Role of the Curriculum

In situations where it is desirable to have sparse rewards, we can utilise a curriculum to solve the problem. In this context, curriculum learning enables us to utilise a simple reward function for learning a complicated task by defining goals of appropriate difficulty for the current policy [84]. Particularly, in regards to robotic tasks, searching directly in the policy space is difficulty, because "it deals simultaneously with complex environmental dynamics and a complex policy" [43]. That is also why [64] states that pretraining is necessary for them, or otherwise the robot wanders around and never accumulates reward. This motivates us to use a curriculum, though at the cost where the curriculum must be defined.

As we observed in the navigation evaluation studies, the learned policy could do well in for tasks it had trained for, though this had taken over 14 hours to learn. The overall initial concern is that reinforcement learning is generally sample-intensive, which is why learning a policy from scratch almost always demands the use of a simulator. This was due to the nature of optimisation, where the policy cannot learn how to solve a complicated task unless the policy repeatedly attempts some "lucky" sequences of actions which have high reward, and provides a large gradient to initially push the policy network in the right direction. This was also discussed

in the training stage, where despite having a curriculum, choosing policies sometimes amounted to hoping for lucky combinations of good navigational ability and careful collision avoidance.

Therefore, rather than waiting for a lucky sample trajectory, the curriculum places the policy is an ideal policy space such that the return for states in its state distribution p_π is well defined. So for our task, with only rewards near goal, we essentially solve the problem of sparse rewards when we pretrain the policy to navigate towards goal. Intuitively, sparse rewards are defined as rewards that only influence very few states – but under a waypoint reaching policy, the states near goal are now a large part of the policy's state distribution.

As a result of this, we can see for example in Figure 7.3, we can create a navigation policy that knows how to avoid one obstacle over 80% of times is just 30 minutes. However, learning this as quickly would not possible for end-to-end methods, unless we provided a more descriptive reward function that provided feedback to all states.

9.1.4 Exploration with Parallel Sampling

The underlying idea with the curriculum is that we wish to experience the most common states such that we can learn the best actions for these states quickly. In other words, we compensate for the limited exploratory nature of on-policy PPO by explicitly aiding the *exploration* of the policy. Yet, simply introducing a curriculum to train a generalisable policy is not straightforward since we might artificially limit what it can explore, such that the policy converges to a *strictly local* optimal solution. The benefit of parallel initialisation and sampling is that we can have varied environments, such that we facilitate exploring multiple states simultaneously. This addresses three things: first, to converge to an optimal solution we require very accurate gradients, second, for stability PPO requires that sampled data is from a very recent policy [85], and third, to learn a complex behaviour requires a wide variety of challenges [21].

To solve the first task, obtaining accurate gradients is synonymous with reducing variance in gradient updates. One of the most direct ways of reducing variance is through large batch sizes. Parallel sampling allows us to sample very large batch sizes, while making it computationally efficient to do so. Second, since PPO is based on a proximal policy, our policy gradient is affected by a clipping so to not move too far from some recent policy. This requires that gradient updates are made in small increments, very often to maintain stability – data collected from a very old policy cannot be used to calculate gradients for a new policy [85]. With parallel sampling with Isaac Gym, we avoid this problem since a recent policy is always used to sample data, and through its end-to-end simulation, we avoid asynchronous update schemes such that data is only sampled after the policy has been updated. Finally and most intuitive, by exposing our policy to many environments simultaneously, the learned behaviour can instantly be applied to all those settings. The mentioned benefits are also in line with [86], which give credit to three main reasons for their success with PPO: 1) parallel sampling scheme 2) distributed initialisation

strategy 3) random track curriculum.

9.2 Collisions in an Otherwise Optimal Policy

Now that we have seen some of the reasons for how our agent has learned an optimal policy, we can delve into some possible reasons why certain behaviours were displayed during evaluation. Namely, we will look more closely into the causes of collisions, why our results could not readily be generalised to larger environments and possible improvements.

9.2.1 Collisions by the Goal

Evident in almost all test plots are the collisions that occur when the quadrotor are at goal. These are also the most unintuitive and difficult to explain. The first possible explanation is that due the the very parallel sampling scheme, by pure randomness in the sampled actions of the policy, the agent might accidentally collide with the ground. This can be unlikely since we observed high robustness to noise in the robustness test, and collisions were no-more likely in the noisy case than for the normal cases.

Conversely, it was observed post-testing that many of these 'collisions' occurred while the quadrotor was in the air, which further added to the confusion. However, if we recall from how we detect collisions in the first place, this was either through a contact force measurement or being less than 0.2m from an obstacle. From this we can identify that the quadrotor does not necessarily have to collide with the ground, but only be within 0.2m of it. So then, since the agent has no idea of its height with respect to the global coordinate frame, only with respect to the target, we can see how collisions can occur when the target is initialised to have $z \in [0.5, 1.5]$, as the agent has only a 0.3m margin from collision when the goal is initiated at its lowest.

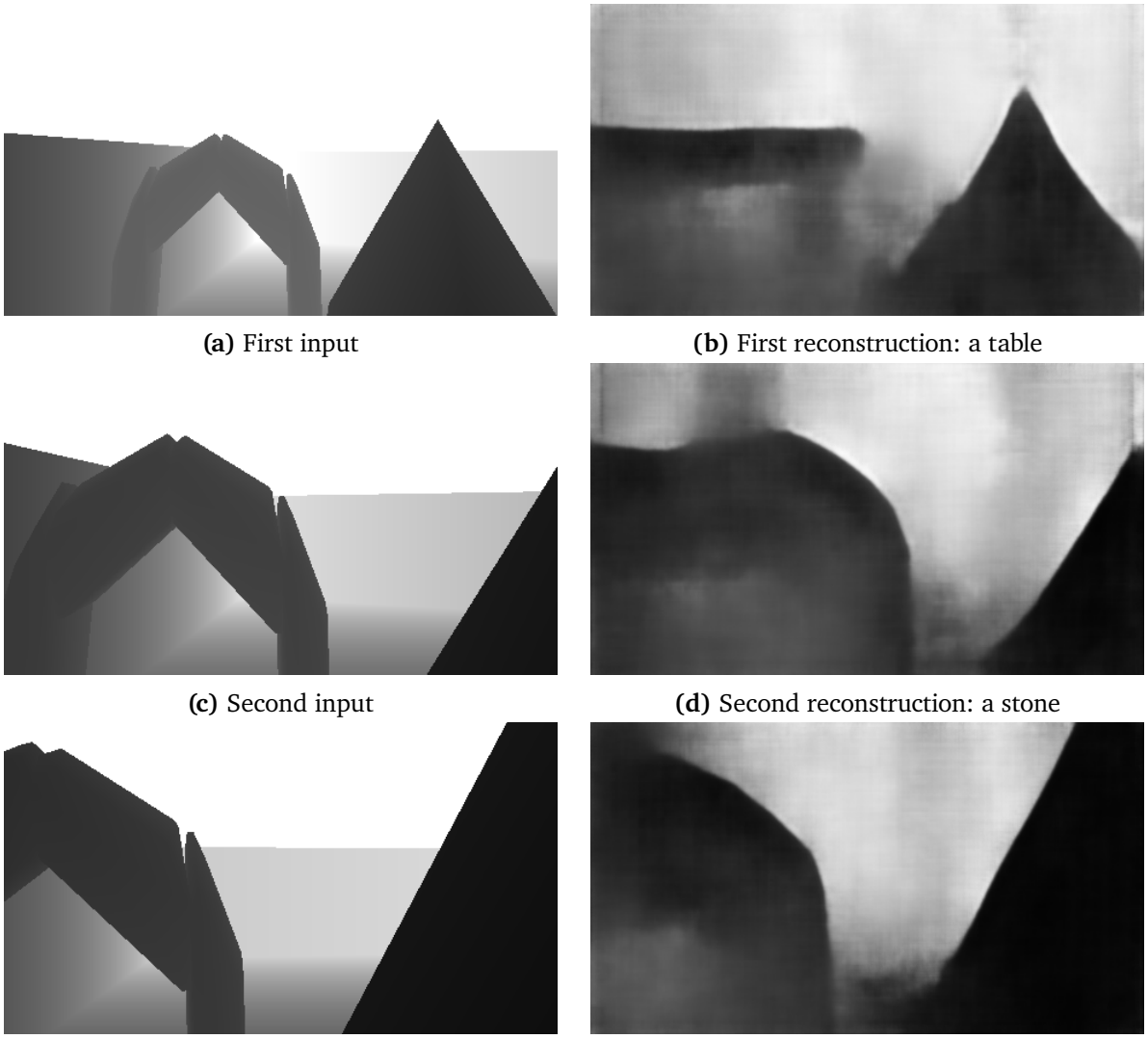
We can see that this explanation does make collisions near goal more plausible due to small margins, but it does not explain why the quadrotor descends so low. For this, we can recall from the evaluation studies that descending was a response to two events: either to reach goal from recently flying over an obstacle, or when reversing when the path ahead is blocked. Following this, we can imagine that when the quadrotor is very high and the goal is initialised at the bottom, this could result in collision, as mentioned in the evaluation. Furthermore, when flying through the environment, the quadrotor position is generally quite high. If we consider Figure 7.15, we can see that all out of bounds events are right by goal suggesting that its position is very high even here. Furthermore, we can justify over-descent since we observe the agent over-shooting the goal, such that it has trajectory very often goes beyond the plotted targets.

Yet, by overshooting the goal, this leads to another unintended consequence, namely going to close to the edge such that it starts to reverse and descend. As already mentioned, this is a an adverse behaviour that the policy has learnt for unknown reasons – most likely just to pass

beneath simple-u obstacles. Nevertheless, we see that if the quadrotor position is quite low when it approaches the edge wall, a collision can occur too. This behaviour was also observed when we (accidentally) ran the depth robustness test with multiplicative Gaussian white noise – since the depth images were set to 0 the quadrotor would reverse in circles until collision with the ground.

So, even though our policy is optimal in the sense of collision avoidance, it is not optimal in all aspects. To fix this problem is fortunately quite straightforward, we can increase the target and quadrotor minimum initialisation position, to be e.g. between 1.0 and 2.0 such as in Isaac Gyms *Ingenuity* example.

9.2.2 Difficulty of Simple-U's



(a) First input **(b)** First reconstruction: a table

(c) Second input **(d)** Second reconstruction: a stone

(e) Third input **(f)** Third reconstruction: a stone

Figure 9.1: Visualising the difficulty to reconstruct simple-U's. This difficulty suggests that the obstacle is not represented properly in the VAE latent space.

Another challenging aspect of the quadrotor performance seemed to be the collision avoidance of the simple-U obstacles. We can also observe, for example in Figure 7.8, that the agent opts to fly around these when it is simpler to fly through them. Intuitively we can accept this since the agent does risk colliding with the sides and the top of the arch when passing through, which is not in line with a conservative policy. However, it other cases, such as in the large environment, this behaviour led to many collisions as the quadrotor instead attempted to pass a narrow opening to the left of the simple-U instead of going through.

To explain the difficulty of this obstacle, we found out during the depth tests that this is most likely due to the VAE not properly being able represent the simple-U in its latent space. From Figure 9.1, we see that due to the filtered nature of the VAE, the arches become coloured in, such that they more closely represent a rounded stone than an arch. So the reason for not properly being able to navigate through these obstacles is not necessarily due to poor motion planning, but rather due to not being able to clearly see it.

9.2.3 Lack of Normalisation in Larger Environments

Initially, the performance of the agent in the large environments were largely unexpected. We explained that this is more likely due to a lack of generalisation, yet a simple change in environment size should be compensated for due to normalisation. Generally, if we wish to use a single architecture for a variety of tasks, we should normalise the input and outputs of the agent [43]. When we analysed the implementation of Isaac Gym's normalising method – the RunningMeanStd class – we realised the once the agent training had been completed, the means and variances of our observation and action spaces are fixed by parameters, which are then loaded in at test time.

Since we did not normalise our observation space in our environment definition, through e.g. dividing the x and y observations by a function of the environment size, this resulted in the quadrotor receiving much higher value observations for its position than normal. Thus, as discussed in Section 9.1.4, by not having the opportunity to explore this state-space leads to inferior performance. This goes in combination with our prior explanation for poor performance in the evaluation, where we reasoned that the the agent was not used to seeing obstacles so early on its state-space.

Thus, in future implementations, we should also normalised the observation and action spaces as a function of their max value so that the performance of the policy is unaffected by changes in the environment at test time.

9.2.4 A Reactionary Policy

Despite good performances, there is still room for improvement. As we saw in the evaluation studies, even with a near perfect score in the known environments, the policy could still crash even in the easy environment, seen in Figure 7.8. Otherwise, we saw that in the hard environment in Figure 7.10, the agent was subject to many pass-by and tight collisions, mainly as a result of an indirect approach and turning into a collision due to "not remembering" that an obstacle is right next to them. Since this is a problem that the agent cannot predict, it can also lead to inferior training performance.

From the evaluation, we observed that the policy could mend this problem itself by producing actions that do not put it too close to obstacles. However, in situations where we cannot decide

on the environment before hand, such as in a random test environment in real-life, how can we prevent this? This leads use to the main improvement that should be done with this approach, which is to included some form for internal memory. This was proposed already quite early on, in works such as [25] and implemented in [65], where we use the RNN hidden state rather than the VAE latent code as a representation of the world. The discussion from [25] summarises this very well, stating "we observed that the deep planner is able to avoid small dead ends if it approaches them from the outside. Once the robot enters a convex dead-end region, it is not capable of freeing itself. In addition to that, the robot's heading sometimes fluctuates before avoiding an obstacle. This issue will be further analyzed and might be solved by using recurrent neural networks with internal memory". These were points were also discussed in the form of corner-situations and the concept of a favourite side for passing obstacles in the evaluation. By introducing a memory state, we also avoid this blindness problem when passing by obstacles and can more precisely pass obstacles in both pass-by scenarios but also tight ones. Moreover, since the agent being able to predict not just one time-step ahead (in a state-action mapping) policy, but also be able to predict its position and actions for future time steps, it should be able to more carefully judge which actions should be taken in the current timestep.

Generally, this approach has been widely adopted for local motion planning [13, 24, 53] such that we can expect to gain an even better performance, particularly for the hard and larger environments. However, a paper which found that this should not be taken for granted as from their tests, only marginal performance was gained in using an LSTM hidden state, where they concluded the most important design choice was to include the use of a latent code from a VAE as a representation.

9.3 Learning a Latent Space for Collision Avoidance

In the VAE results, we saw that despite the training curves being uninformative, the use of different reconstruction errors resulted in very image outputs. We explain some of the consequences and some why they are so different.

9.3.1 MSE versus BCE

Though perhaps less clear than the modifications to the loss functions, is why the MSE and BCE had so different output reconstructions. To explain this, we can recall that the the main difference in the losses is what distribution they assume the input data distribution to be. Both methods do maximum likelihood estimation, but BCE assumes that our data is Bernoulli distributed, while MSE assumes it is Gaussian. What this means is that the since Bernoulli distributed variables are either 0 or 1, our BCE has a bias to pushing pixels a more contrastive difference – black or white. On the other hand, MSE assumes that our data is centered around 0.5, and

that pixel values are less likely to be at the extreme values. This shows why the BCE sometimes imagined that obstacles were closer than they were and had high contrast, while MSE had blur reconstructions, though could on average be more accurate in terms of depth.

9.3.2 Depth Gain and the Edge Loss

Though less apparent in the results is the difference between the depth gain and edge loss models. The main motivations for the edge loss was to curb the adverse effect of over-blurring the depth images. Since we specified that it was important to *at least* detect close obstacles, the VAE was inclined to create very large obstacles, or even *phantom obstacles*. This occurred because the VAE only incurred a high loss if it *did not* detect a close obstacle, but not if it misdetected one. In comparison, the edge loss did improve the resolution (contrast) of the reconstructions, so that they are not over-blurred as a result of filtering. In a sense, it provides an extra loss signal to give the VAE a direction on how to filter depth images implicitly.

9.3.3 Latent Over-Fitting

A consequence of learning how to minimised edge reconstruction error is the possibility of over-fitting. In other words, due to the large weight of the edge loss, the VAE will indirectly learn what objects have what edges. In our training plots, we see that this is not an issue for the edge losses – but we have to remember that the training and test sets both have the same depth data distribution, such that no new obstacles are seen at test time. Therefore, there is a possibility that the VAE has been over-fitting the depth data distribution, such that it generalises poorly to unseen ones, despite having no increase in test loss.

This is also a possible reason why the simple-U's were reconstructed poorly, as they were not part of the input dataset. To mitigate this, we could simply ensure that our simulated dataset has the same distribution of shapes and obstacles that we will have at test time (real environments or otherwise). However, the more correct approach is to simply reduce the possibility of over-fitting by introducing some form or regularisation in the VAE network, such as Monte Carlo dropout done in [23] and [24].

Chapter 10

Conclusions

Learning-based approaches to autonomous navigation in cluttered environments are becoming increasingly popular due to being able to learn flexible, complex behaviours that can solve various challenging tasks without the need to explicitly program them. Due to their ability to represent complex functions through deep architectures, they are able to direct map raw sensor inputs to actions which, combined with the fast parallel computing hardware, allows them to plan and execute actions at a speed at which a model-based pipeline cannot compete. Further, when presented with a novel task with no prior demonstrations and an optimal solution that is hard to formulate, reinforcement learning provides a method to learn this through only the specification of an abstract reward function.

In this thesis, we thus explored the ability of reinforcement learning to solve a collision avoidance task for a quadrotor using only a depth camera. We proposed the use of a two-part CNN-MLP model, where the CNN is the inference network of the VAE, while the MLP serves as the reinforcement learning agent actor-critic. To shape our model for collision avoidance, we then introduced a novel reward function for the reinforcement learning agent and a novel loss function for the VAE. The reward function was shaped with penalties to demotivate risky behaviour, which in hindsight may not have been optimal due to the inherent sparsity of rewards. Similarly, the customised reconstruction error of the VAE enabled us to prioritise collision-relevant features of depth images, though it potentially resulted in over-fitting of the depth data distribution.

Nonetheless, the evaluation studies showed that the overall approach was successful at its task, achieving 98.6% in the hard environment which accounted for tight space. The approach also showed promising robustness to noise in the quadrotor state and actions, where its adverse effects were most present in tight paths, such as in the hard environment. Despite these results, the study also showed that much could be improved, both in the proposed approach and the implementation of our model. Specifically, it was shown that many pass-by collisions occur due to blindness when turning, the agent frequently reversed or descended into collisions, and the agent maintained a bias when navigating past obstacles. It was discussed that to alleviate these

problems, the concept of an internal memory had to be added to the policy, through, for example, the hidden state of an RNN, such that the agent is capable of predicting collisions more than one timesteps ahead, remembering passed obstacles, and finally capable of deciding more fine-tuned actions which account for these. Otherwise, implementation errors included a too small margin between the goal height and ground, and a lack of normalisation in the environment setup.

Moreover, this thesis demonstrated the importance of a meaningful and well-formed latent space, where we saw that the latent code captured the placements of obstacles and their edges. By adding the filtering operation to be implicitly learned in the forward pass of the VAE, we simplify the complexity of the depth images such that the VAE focuses only on rough shapes, and we achieve a layer of safety when navigating close to obstacles. However, we saw that the navigation policy is highly dependent on accurate depth images, such that in the future, we should account for noisy depth images by training the VAE to perform denoising.

Bibliography

[1] J. P. Nitschke, 'A comparative study of deep deterministic policy gradients and proximal policy optimisation for quadrotor guidance,' *Project Thesis, Department of Engineering Cybernetics, NTNU - The Norwegian University of Science and Technology*, Jan. 2022.

[2] Kongsberg Group. 'Argeo has signed the first commercial contract for eelume's snake robot technology.' (Nov. 2021), [Online]. Available: https://www.kongsberg.com/maritime/about-us/news-and-media/news-archive/2021/argeo-chooses-kongsberg-maritime-supported--eelume/.

[3] C. Gehring *et al.*, 'Anymal in the field: Solving industrial inspection of an offshore hvdc platform with a quadrupedal robot,' in Jan. 2021, pp. 247–260, ISBN: 978-981-15-9459-5. DOI: 10.1007/978-981-15-9460-1_18.

[4] Y. Djenouri, J. Hatleskog, J. Hjelmervik, E. Bjorne, T. Utstumo and M. Mobarhan, 'Deep learning based decomposition for visual navigation in industrial platforms,' *Applied Intelligence*, vol. 52, no. 7, pp. 8101–8117, May 2022, ISSN: 1573-7497. DOI: 10.1007/s10489-021-02908-z. [Online]. Available: https://doi.org/10.1007/s10489-021-02908-z.

[5] I. B. Hagen. 'Autoferry in paris?' (Apr. 2022), [Online]. Available: https://www.ntnu.edu/autoferry/news.

[6] D. Montgomery. 'Inside the pentagon's $82 million super bowl of robots.' (Nov. 2021), [Online]. Available: https://www.washingtonpost.com/magazine/2021/11/10/darpa-robot-competition/.

[7] S. Neema. 'Assured autonomy.' (Apr. 2022), [Online]. Available: https://www.darpa.mil/program/assured-autonomy.

[8] D. Ferguson, M. Darms, C. Urmson and S. Kolski, 'Detection, prediction, and avoidance of dynamic obstacles in urban environments,' in *Proceedings of IEEE Intelligent Vehicles Symposium (IV '08)*, Eindhofen: IEEE, Jun. 2008, pp. 1149–1154.

[9] A. Loquercio, E. Kaufmann, R. Ranftl, M. Müller, V. Koltun and D. Scaramuzza, 'Learning high-speed flight in the wild,' *CoRR*, vol. abs/2110.05113, 2021. arXiv: 2110.05113. [Online]. Available: https://arxiv.org/abs/2110.05113.

[10] P. Liljebäck and R. Mills, 'Eelume: A flexible and subsea resident imr vehicle,' in *OCEANS 2017 - Aberdeen*, 2017, pp. 1–4. DOI: 10.1109/OCEANSE.2017.8084826.

[11] T. Dang, F. Mascarich, S. Khattak, C. Papachristos and K. Alexis, 'Graph-based path planning for autonomous robotic exploration in subterranean environments,' in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 3105–3112. DOI: 10.1109/IROS40897.2019.8968151.

[12] A. Bachrach, R. He and N. Roy, 'Autonomous flight in unstructured and unknown indoor environments,' Dec. 2011.

[13] D. Hoeller, L. Wellhausen, F. Farshidian and M. Hutter, 'Learning a state representation and navigation in cluttered and dynamic environments,' *CoRR*, vol. abs/2103.04351, 2021. arXiv: 2103.04351. [Online]. Available: https://arxiv.org/abs/2103.04351.

[14] D. Dugas, J. I. Nieto, R. Siegwart and J. J. Chung, 'Navrep: Unsupervised representations for reinforcement learning of robot navigation in dynamic human environments,' *CoRR*, vol. abs/2012.04406, 2020. arXiv: 2012.04406. [Online]. Available: https://arxiv.org/abs/2012.04406.

[15] R. Reinhart, T. Dang, E. Hand, C. Papachristos and K. Alexis, 'Learning-based path planning for autonomous exploration of subterranean environments,' in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 1215–1221. DOI: 10.1109/ICRA40945.2020.9196662.

[16] R. Mahony, V. Kumar and P. Corke, 'Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor,' *IEEE Robotics Automation Magazine*, vol. 19, no. 3, pp. 20–32, 2012. DOI: 10.1109/MRA.2012.2206474.

[17] J. Kober, J. Bagnell and J. Peters, 'Reinforcement learning in robotics: A survey,' *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, Sep. 2013. DOI: 10.1177/0278364913495721.

[18] T. Fan, P. Long, W. Liu and J. Pan, 'Fully distributed multi-robot collision avoidance via deep reinforcement learning for safe and efficient navigation in complex scenarios,' 2018. DOI: 10.48550/ARXIV.1808.03841. [Online]. Available: https://arxiv.org/abs/1808.03841.

[19] T. Eppenberger, G. Cesari, M. Dymczyk, R. Siegwart and R. Dubé, 'Leveraging stereo-camera data for real-time dynamic obstacle detection and tracking,' *CoRR*, vol. abs/2007.10743, 2020. arXiv: 2007.10743. [Online]. Available: https://arxiv.org/abs/2007.10743.

[20] N. Rudin, D. Hoeller, P. Reist and M. Hutter, 'Learning to walk in minutes using massively parallel deep reinforcement learning,' *CoRR*, vol. abs/2109.11978, 2021. arXiv: 2109.11978. [Online]. Available: https://arxiv.org/abs/2109.11978.

[21] N. Heess *et al.*, 'Emergence of locomotion behaviours in rich environments,' *CoRR*, vol. abs/1707.02286, 2017. arXiv: 1707.02286. [Online]. Available: http://arxiv.org/abs/1707.02286.

[22] F. Sadeghi and S. Levine, 'Cad^{2}rl: Real single-image flight without a single real image,' *CoRR*, vol. abs/1611.04201, 2016. arXiv: 1611.04201. [Online]. Available: http://arxiv.org/abs/1611.04201.

[23] A. Loquercio, A. I. Maqueda, C. R. del-Blanco and D. Scaramuzza, 'Dronet: Learning to fly by driving,' *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 1088–1095, 2018. DOI: 10.1109/LRA.2018.2795643.

[24] H. Nguyen, S. H. Fyhn, P. De Petris and K. Alexis, *Motion primitives-based navigation planning using deep collision prediction*, 2022. DOI: 10.48550/ARXIV.2201.03254. [Online]. Available: https://arxiv.org/abs/2201.03254.

[25] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart and C. Cadena, 'From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots,' in *IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2017, pp. 1527–1533. DOI: 10.1109/ICRA.2017.7989182.

[26] V. Makoviychuk *et al.*, 'Isaac gym: High performance gpu-based physics simulation for robot learning,' *CoRR*, vol. abs/2108.10470, 2021. arXiv: 2108.10470. [Online]. Available: https://arxiv.org/abs/2108.10470.

[27] A. Ng, *Lecture notes in cs294a*, May 2022. [Online]. Available: https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf.

[28] I. J. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[29] Y. Bengio, A. Courville and P. Vincent, 'Representation learning: A review and new perspectives,' *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013. DOI: 10.1109/TPAMI.2013.50.

[30] M. A. Kramer, 'Nonlinear principal component analysis using autoassociative neural networks,' *AIChE Journal*, vol. 37, no. 2, pp. 233–243, 1991. DOI: https://doi.org/10.1002/aic.690370209. eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.690370209. [Online]. Available: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690370209.

[31] A. Sperduti and N. Navarin, *Lecture notes in deep learning*, Apr. 2021.

[32] D. P. Kingma and M. Welling, 'Auto-encoding variational bayes,' 2013. DOI: 10.48550/ARXIV.1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114.

[33] D. J. Rezende, S. Mohamed and D. Wierstra, 'Stochastic backpropagation and approximate inference in deep generative models,' 2014. DOI: 10.48550/ARXIV.1401.4082. [Online]. Available: https://arxiv.org/abs/1401.4082.

[34] K. P. Murphy, *Machine learning : a probabilistic perspective*, ser. Adaptive computation and machine learning series. Cambridge, MA: MIT, 2012. [Online]. Available: https://www.worldcat.org/title/machine-learning-a-probabilistic-perspective/oclc/781277861?referer=br&ht=edition.

[35] M. D. Hoffman, D. M. Blei, C. Wang and J. Paisley, 'Stochastic variational inference,' *Journal of Machine Learning Research*, 2013.

[36] I. Higgins *et al.*, 'Beta-vae: Learning basic visual concepts with a constrained variational framework,' in *ICLR*, 2017.

[37] Y. Pu *et al.*, 'Variational autoencoder for deep learning of images, labels and captions,' 2016. DOI: 10.48550/ARXIV.1609.08976. [Online]. Available: https://arxiv.org/abs/1609.08976.

[38] K. He, X. Zhang, S. Ren and J. Sun, 'Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,' *CoRR*, vol. abs/1502.01852, 2015. arXiv: 1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852.

[39] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, 2016. DOI: 10.48550/ARXIV.1603.07285. [Online]. Available: https://arxiv.org/abs/1603.07285.

[40] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[41] R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*. Princetown University Press, 1962.

[42] C. Watkins and P. Dayan, 'Technical note: Q-learning,' *Machine Learning*, vol. 8, pp. 279–292, May 1992. DOI: 10.1007/BF00992698.

[43] T. P. Lillicrap *et al.*, 'Continuous control with deep reinforcement learning,' 2019. arXiv: 1509.02971 [cs.LG].

[44] J. A. Bagnell and J. G. Schneider, 'Covariant policy search,' in *IJCAI*, 2003, pp. 1019–1024. [Online]. Available: https://www.ijcai.org/Proceedings/03/Papers/146.pdf.

[45] V. Mnih *et al.*, 'Human-level control through deep reinforcement learning,' *Nature*, vol. 518, pp. 529–33, Feb. 2015. DOI: 10.1038/nature14236.

[46] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, 'Deterministic policy gradient algorithms,' *31st International Conference on Machine Learning, ICML 2014*, vol. 1, Jun. 2014.

[47] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, 'Proximal policy optimization algorithms,' 2017. arXiv: 1707.06347 [cs.LG].

[48] J. Schulman, S. Levine, P. Moritz, M. I. Jordan and P. Abbeel, 'Trust region policy optimization,' 2017. arXiv: 1502.05477 [cs.LG].

[49] X. B. Peng, Z. Ma, P. Abbeel, S. Levine and A. Kanazawa, 'Amp: Adversarial motion priors for stylized physics-based character control,' *ACM Trans. Graph.*, vol. 40, no. 4, Jul. 2021. DOI: 10.1145/3450626.3459670. [Online]. Available: http://doi.acm.org/10.1145/3450626.3459670.

[50] K. He, X. Zhang, S. Ren and J. Sun, 'Deep residual learning for image recognition,' 2015. DOI: 10.48550/ARXIV.1512.03385. [Online]. Available: https://arxiv.org/abs/1512.03385.

[51] E. Kaufmann, A. Loquercio, R. Ranftl, A. Dosovitskiy, V. Koltun and D. Scaramuzza, 'Deep drone racing: Learning agile flight in dynamic environments,' *CoRR*, vol. abs/1806.08548, 2018. arXiv: 1806.08548. [Online]. Available: http://arxiv.org/abs/1806.08548.

[52] D. Gandhi, L. Pinto and A. Gupta, 'Learning to fly by crashing,' *CoRR*, vol. abs/1704.05588, 2017. arXiv: 1704.05588. [Online]. Available: http://arxiv.org/abs/1704.05588.

[53] G. Kahn, P. Abbeel and S. Levine, 'BADGR: an autonomous self-supervised learning-based navigation system,' *CoRR*, vol. abs/2002.05700, 2020. arXiv: 2002.05700. [Online]. Available: https://arxiv.org/abs/2002.05700.

[54] S. Hochreiter and J. Schmidhuber, 'Long short-term memory,' *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[55] A. Nagabandi, K. Konolige, S. Levine and V. Kumar, 'Deep dynamics models for learning dexterous manipulation,' *CoRR*, vol. abs/1909.11652, 2019. arXiv: 1909.11652. [Online]. Available: http://arxiv.org/abs/1909.11652.

[56] A. Attia and S. Dayan, 'Global overview of imitation learning,' 2018. DOI: 10.48550/ARXIV.1801.06503. [Online]. Available: https://arxiv.org/abs/1801.06503.

[57] S. Ross *et al.*, 'Learning monocular reactive UAV control in cluttered natural environments,' *CoRR*, vol. abs/1211.1690, 2012. DOI: 10.48550/arXiv.1211.1690.

[58] S. Ross, G. J. Gordon and J. A. Bagnell, 'No-regret reductions for imitation learning and structured prediction,' *CoRR*, vol. abs/1011.0686, 2010. DOI: 10.48550/arXiv.1211.1690.

[59] J. Michels, A. Saxena and A. Y. Ng, 'High speed obstacle avoidance using monocular vision and reinforcement learning,' in *Proceedings of the 22nd International Conference on Machine Learning*, ser. ICML '05, Bonn, Germany: Association for Computing Machinery, 2005, pp. 593–600, ISBN: 1595931805. DOI: 10.1145/1102351.1102426. [Online]. Available: https://doi.org/10.1145/1102351.1102426.

[60] L. Xie, S. Wang, A. Markham and N. Trigoni, 'Towards monocular vision based obstacle avoidance through deep reinforcement learning,' 2017. DOI: 10.48550/ARXIV.1706.09829. [Online]. Available: https://arxiv.org/abs/1706.09829.

[61] K. Simonyan and A. Zisserman, 'Very deep convolutional networks for large-scale image recognition,' 2014. DOI: 10.48550/ARXIV.1409.1556. [Online]. Available: https://arxiv.org/abs/1409.1556.

[62] L. Tai, G. Paolo and M. Liu, 'Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation,' 2017. DOI: 10.48550/ARXIV.1703.00420. [Online]. Available: https://arxiv.org/abs/1703.00420.

[63] T. Lesort, N. Díaz-Rodríguez, J.-F. Goudou and D. Filliat, 'State representation learning for control: An overview,' *Neural Networks*, vol. 108, pp. 379–392, Dec. 2018. DOI: 10.1016/j.neunet.2018.07.006. [Online]. Available: https://doi.org/10.1016%5C%2Fj.neunet.2018.07.006.

[64] M. Everett, Y. F. Chen and J. P. How, 'Motion planning among dynamic, decision-making agents with deep reinforcement learning,' 2018. DOI: 10.48550/ARXIV.1805.01956. [Online]. Available: https://arxiv.org/abs/1805.01956.

[65] D. Ha and J. Schmidhuber, 'Recurrent world models facilitate policy evolution,' 2018. DOI: 10.48550/ARXIV.1809.01999. [Online]. Available: https://arxiv.org/abs/1809.01999.

[66] G. Brockman *et al.*, *Openai gym*, 2016. arXiv: 1606.01540 [cs.LG].

[67] A. X. Lee, A. Nagabandi, P. Abbeel and S. Levine, 'Stochastic latent actor-critic: Deep reinforcement learning with a latent variable model,' 2019. DOI: 10.48550/ARXIV.1907.00953. [Online]. Available: https://arxiv.org/abs/1907.00953.

[68] A. Vaswani *et al.*, *Attention is all you need*, 2017. DOI: 10.48550/ARXIV.1706.03762. [Online]. Available: https://arxiv.org/abs/1706.03762.

[69] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. 2021, ISBN: 978-1-119-57505-4.

[70] M. Gaertner, M. Bjelonic, F. Farshidian and M. Hutter, 'Collision-free mpc for legged robots in static and dynamic scenes,' in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021, pp. 8266–8272. DOI: 10.1109/ICRA48506.2021.9561326.

[71] OpenAI *et al.*, 'Learning dexterous in-hand manipulation,' *CoRR*, 2018. [Online]. Available: http://arxiv.org/abs/1808.00177.

[72] S. Bubeck and M. Sellke, 'A universal law of robustness via isoperimetry,' in *NeurIPS 2021*, Dec. 2021. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-universal-law-of-robustness-via-isoperimetry/.

[73] D.-A. Clevert, T. Unterthiner and S. Hochreiter, 'Fast and accurate deep network learning by exponential linear units (elus),' 2015. DOI: 10.48550/ARXIV.1511.07289. [Online]. Available: https://arxiv.org/abs/1511.07289.

[74] V. Nair and G. E. Hinton, 'Rectified linear units improve restricted boltzmann machines,' in *ICML*, 2010.

[75] R. Bonatti, R. Madaan, V. Vineet, S. Scherer and A. Kapoor, 'Learning visuomotor policies for aerial navigation using cross-modal representations,' *arXiv preprint arXiv:1909.06993*, 2020. DOI: 10.48550/ARXIV.1909.06993.

[76] J. Ku, A. Harakeh and S. L. Waslander, *In defense of classical image processing: Fast depth completion on the cpu*, 2018. DOI: 10.48550/ARXIV.1802.00036. [Online]. Available: https://arxiv.org/abs/1802.00036.

[77] J. Canny, 'A computational approach to edge detection,' *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, 1986. DOI: 10.1.1.420.3300.

[78] A. Odena, V. Dumoulin and C. Olah, 'Deconvolution and checkerboard artifacts,' *Distill*, 2016. DOI: 10.23915/distill.00003. [Online]. Available: http://distill.pub/2016/deconv-checkerboard.

[79] D. Makoviichuk and V. Makoviychuk, *Rl-games: A high-performance framework for reinforcement learning*, https://github.com/Denys88/rl_games, May 2022.

[80] F. Furrer, M. Burri, M. Achtelik and R. Siegwart, 'Robot operating system (ros): The complete reference (volume 1),' in A. Koubaa, Ed. Cham: Springer International Publishing, 2016, ch. RotorS—A Modular Gazebo MAV Simulator Framework, pp. 595–625, ISBN: 978-3-319-26054-9. DOI: 10.1007/978-3-319-26054-9_23. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26054-9_23.

[81] N. Koenig and A. Howard, 'Design and use paradigms for gazebo, an open-source multi-robot simulator,' in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[82] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus and N. Dormann, 'Stable-baselines3: Reliable reinforcement learning implementations,' *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[83] D. P. Kingma and J. Ba, 'Adam: A method for stochastic optimization,' 2014. DOI: 10.48550/ARXIV.1412.6980. [Online]. Available: https://arxiv.org/abs/1412.6980.

[84] C. Florensa, D. Held, X. Geng and P. Abbeel, 'Automatic goal generation for reinforcement learning agents,' in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Jul. 2018, pp. 1515–1528. [Online]. Available: https://proceedings.mlr.press/v80/florensa18a.html.

[85] J. Hilton, K. Cobbe and J. Schulman, 'Batch size-invariance for policy optimization,' 2021. arXiv: 2110.00641 [cs.LG].

[86] Y. Song, M. Steinweg, E. Kaufmann and D. Scaramuzza, 'Autonomous drone racing with deep reinforcement learning,' 2021. arXiv: 2103.08624 [cs.RO].

Appendix A

Algorithms

A.1 One-step Actor-Critic

Algorithm 1 One-step Actor-Critic, from [40]

Input: a differentiable policy parameterisation $\pi(a \mid s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterisation $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\boldsymbol{w}} > 0$ (learning rates)
Initialize policy parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^{d}$
for each episode **do**
    Initialize S (first state of episode)
    I $\leftarrow 1$
    while S is not terminal **do**
        A $\sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action A, observe S', R
        $\delta \leftarrow R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S, \boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A \mid S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
    end while
end for

This implementation is taken from the Sutton & Barto's book [40].

A.2 Proximal Policy Optimization

Algorithm 2 PPO Actor-Critic style, from [47]

for iteration = 1, 2, ... **do**
  for actor = 1, 2, ..., N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  Optimize surrogate L with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

This implementation is acquired from the original paper [47]. Note that in our implementation we have $N = 512$ actors.

Algorithm 3 PPO-Clip, from Spinning Up, OpenAI

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for k = 0, 1, 2, ... **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute the return $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ based on the current value function V_{ϕ_k}
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via a stochastic gradient ascent algorithm with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some stochastic gradient descent algorithm.
end for

Algorithm 3 is acquired from OpenAI's Spinning Up documentation:
https://spinningup.openai.com/en/latest/algorithms/ppo.html

A.3 Auto-Encoding Variational Bayes

Algorithm 4 Minibatch version of Auto-Encoding Variational Bayes (AEVB) algorithm, from [32]

$\boldsymbol{\theta}, \boldsymbol{\phi} \leftarrow$ Initialise parameters
repeat
  $\boldsymbol{X}^M \leftarrow$ Random minibatch of M datapoints (drawn from the full dataset)
  $\boldsymbol{\epsilon} \leftarrow$ random samples from noise distribution $p(\boldsymbol{\epsilon})$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta},\boldsymbol{\phi}} \mathcal{L}^M(\boldsymbol{\theta}, \boldsymbol{\phi}; \boldsymbol{X}^M, \boldsymbol{\epsilon})$ calculate gradients of minibatch estimator (2.10)
  $\boldsymbol{\theta}, \boldsymbol{\phi} \leftarrow \boldsymbol{g}$ Update parameters using gradients $\boldsymbol{g}$
until convergence of parameters

This implementation is acquired from the original paper [32]. We chose a Gaussian distribution $\mathcal{N}(\epsilon; 0, 1)$ for the noise distribution $p(\boldsymbol{\epsilon})$, and updated our parameters using Adam [83].

Appendix B

VAE

The parameters and output shapes of each layer in the VAE are presented in detail. Their format follows the PyTorch convention, where for example, `[-1, 32, 135, 240]` represents the batch dimension, number of filters and dimension of the feature map. The outputs are generated from the *torchsummary* package at `https://github.com/sksq96/pytorch-summary`.

B.1 VAE Encoder

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 135, 240]             832
            Conv2d-2          [-1, 32, 68, 120]          25,632
       BatchNorm2d-3          [-1, 32, 68, 120]              64
            Conv2d-4           [-1, 32, 34, 60]           9,248
       BatchNorm2d-5           [-1, 32, 34, 60]              64
            Conv2d-6           [-1, 32, 34, 60]           9,248
            Conv2d-7           [-1, 32, 34, 60]           1,056
       BatchNorm2d-8           [-1, 32, 34, 60]              64
            Conv2d-9           [-1, 64, 17, 30]          18,496
      BatchNorm2d-10           [-1, 64, 17, 30]             128
           Conv2d-11           [-1, 64, 17, 30]          36,928
           Conv2d-12           [-1, 64, 17, 30]           2,112
      BatchNorm2d-13           [-1, 64, 17, 30]             128
           Conv2d-14           [-1, 128, 9, 15]          73,856
      BatchNorm2d-15           [-1, 128, 9, 15]             256
           Conv2d-16           [-1, 128, 9, 15]         147,584
```

```
           Conv2d-17          [-1, 128, 9, 15]           8,320
           Linear-18                [-1, 128]       2,211,968
           Linear-19                [-1, 128]          16,512
================================================================
Total params: 2,562,496
Trainable params: 2,562,496
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.49
Forward/backward pass size (MB): 16.16
Params size (MB): 9.78
Estimated Total Size (MB): 26.43
----------------------------------------------------------------
```

B.2 VAE Decoder

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 128]           8,320
            Linear-2             [-1, 1, 17280]       2,229,120
   ConvTranspose2d-3           [-1, 128, 9, 15]         147,584
   ConvTranspose2d-4           [-1, 64, 17, 30]         204,864
   ConvTranspose2d-5           [-1, 64, 34, 60]         147,520
   ConvTranspose2d-6          [-1, 32, 68, 120]          73,760
   ConvTranspose2d-7         [-1, 32, 135, 240]          25,632
   ConvTranspose2d-8         [-1, 16, 270, 480]          18,448
   ConvTranspose2d-9          [-1, 1, 270, 480]             401
================================================================
Total params: 2,855,649
Trainable params: 2,855,649
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 28.22
Params size (MB): 10.89
Estimated Total Size (MB): 39.11
----------------------------------------------------------------
```

B.3 VAE

Note that the two Lambda layers simply perform a middle split of the last layer of the encoder according the the dimension of $\boldsymbol{z}_t$. This is to recover the mean and covariance of the parametrised approximate posterior $q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{d})$, as shown in Figure 5.7.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 135, 240]             832
            Conv2d-2          [-1, 32, 68, 120]          25,632
       BatchNorm2d-3          [-1, 32, 68, 120]              64
            Conv2d-4           [-1, 32, 34, 60]           9,248
       BatchNorm2d-5           [-1, 32, 34, 60]              64
            Conv2d-6           [-1, 32, 34, 60]           9,248
            Conv2d-7           [-1, 32, 34, 60]           1,056
       BatchNorm2d-8           [-1, 32, 34, 60]              64
            Conv2d-9           [-1, 64, 17, 30]          18,496
      BatchNorm2d-10           [-1, 64, 17, 30]             128
           Conv2d-11           [-1, 64, 17, 30]          36,928
           Conv2d-12           [-1, 64, 17, 30]           2,112
      BatchNorm2d-13           [-1, 64, 17, 30]             128
           Conv2d-14           [-1, 128, 9, 15]          73,856
      BatchNorm2d-15           [-1, 128, 9, 15]             256
           Conv2d-16           [-1, 128, 9, 15]         147,584
           Conv2d-17           [-1, 128, 9, 15]           8,320
           Linear-18                  [-1, 128]       2,211,968
           Linear-19                  [-1, 128]          16,512
           Dronet-20                  [-1, 128]               0
           Lambda-21                   [-1, 64]               0
           Lambda-22                   [-1, 64]               0
           Linear-23                  [-1, 128]           8,320
           Linear-24                [-1, 17280]       2,229,120
  ConvTranspose2d-25           [-1, 128, 9, 15]         147,584
  ConvTranspose2d-26           [-1, 64, 17, 30]         204,864
  ConvTranspose2d-27           [-1, 64, 34, 60]         147,520
  ConvTranspose2d-28          [-1, 32, 68, 120]          73,760
  ConvTranspose2d-29         [-1, 32, 135, 240]          25,632
  ConvTranspose2d-30         [-1, 16, 270, 480]          18,448
```

```
  ConvTranspose2d-31          [-1, 1, 270, 480]             401
       ImgDecoder-32          [-1, 1, 270, 480]               0
================================================================
Total params: 5,418,145
Trainable params: 5,418,145
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.49
Forward/backward pass size (MB): 45.37
Params size (MB): 20.67
Estimated Total Size (MB): 66.53
----------------------------------------------------------------
```

Appendix C

Training Plots

C.1 Large Environment Policy

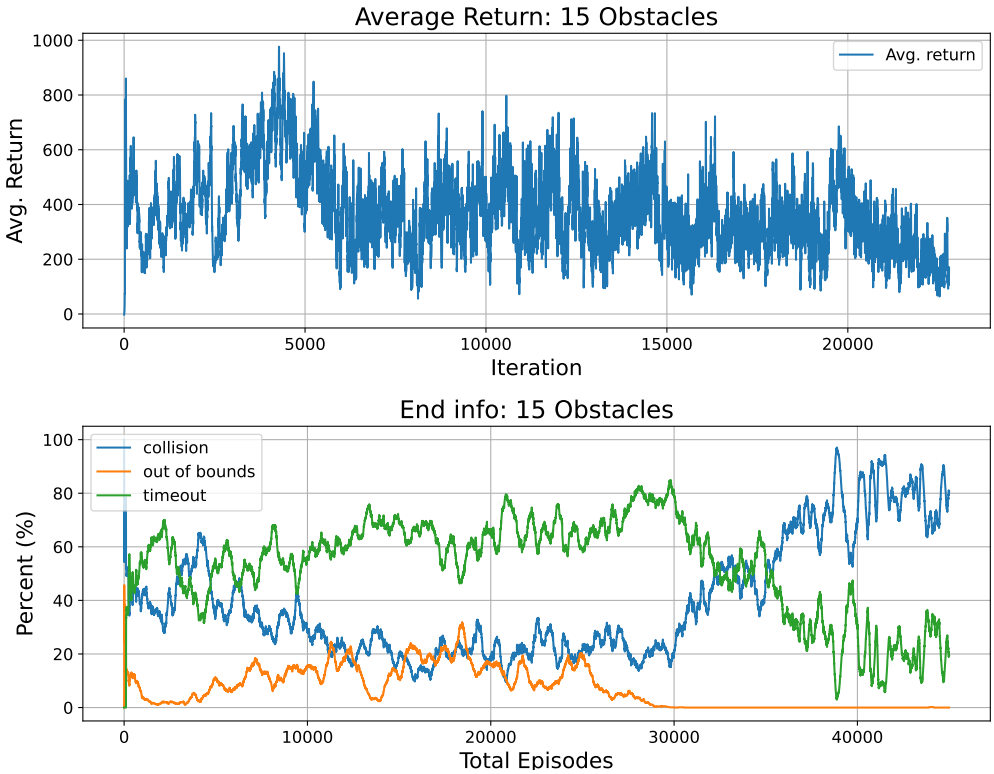


Figure C.1: Training the navigation policy with 15 obstacles in an environments of size $[24, 12]$. The best model is selected at 4410 iterations.

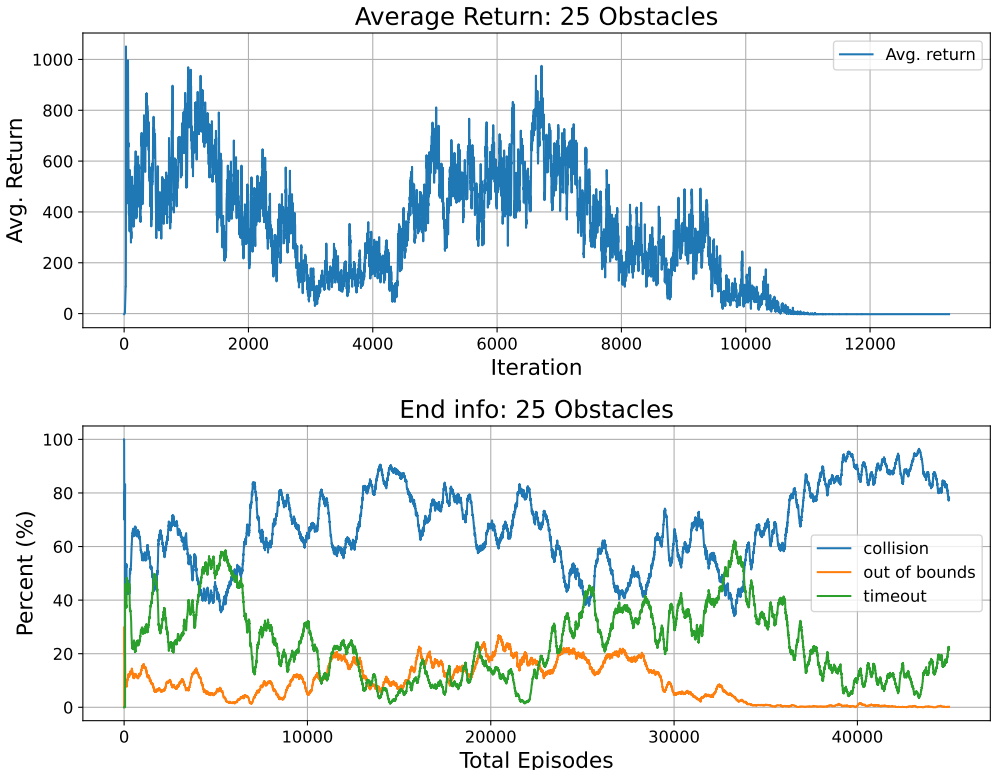


Figure C.2: Training the navigation policy with 25 obstacles in an environment of size $[30, 15]$. The best models were selected at 6550 iterations.

Appendix D

Environments

D.1 Known Environments with Varied Clutter

Each of the following environments have a dimension of [20, 10].

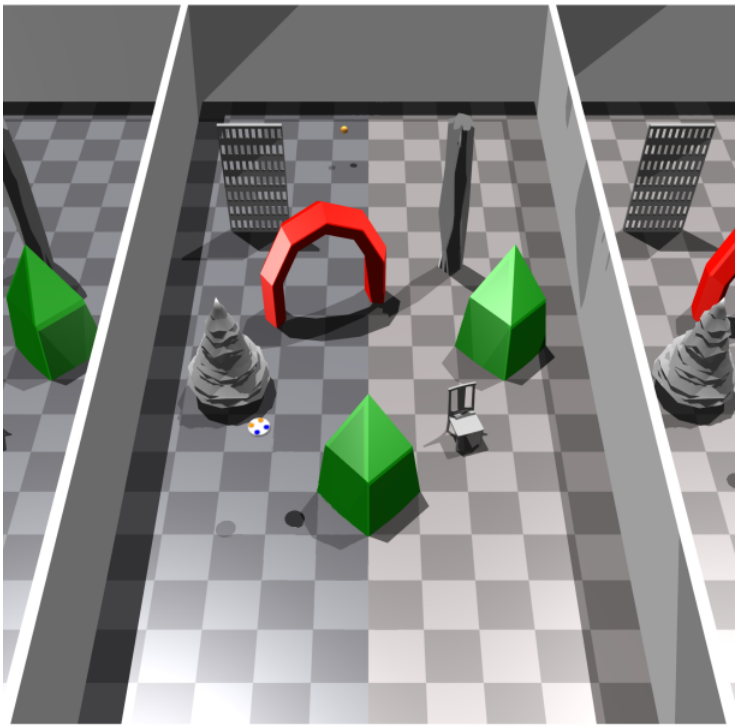


Figure D.1: The easy environment, with 7 obstacles.

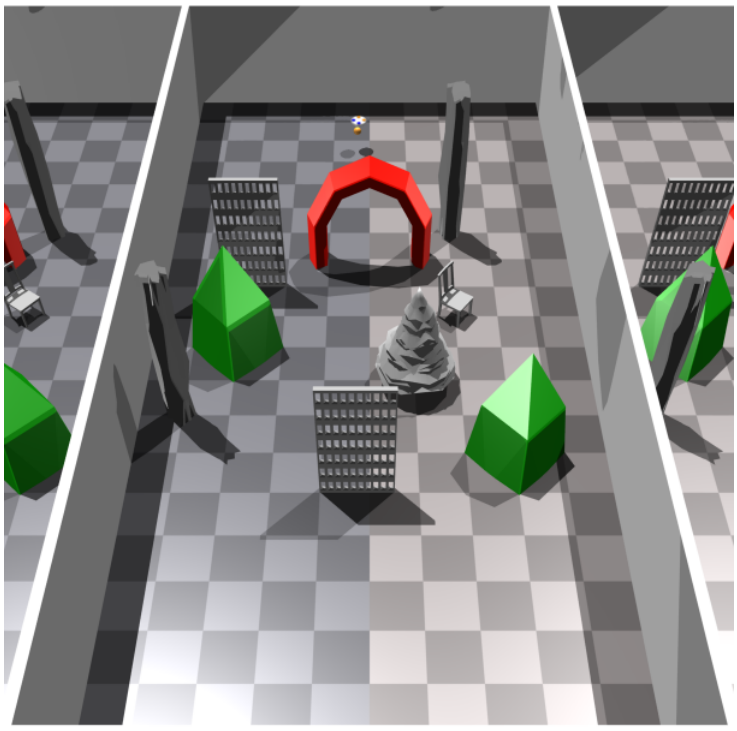


Figure D.2: The medium environment, with 9 obstacles.

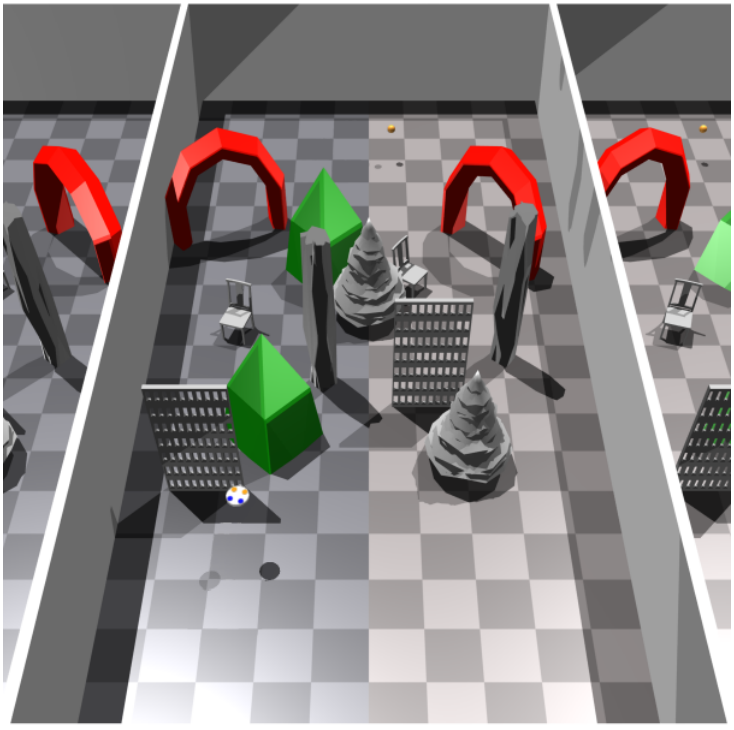


Figure D.3: The hard environment, with 12 obstacles.

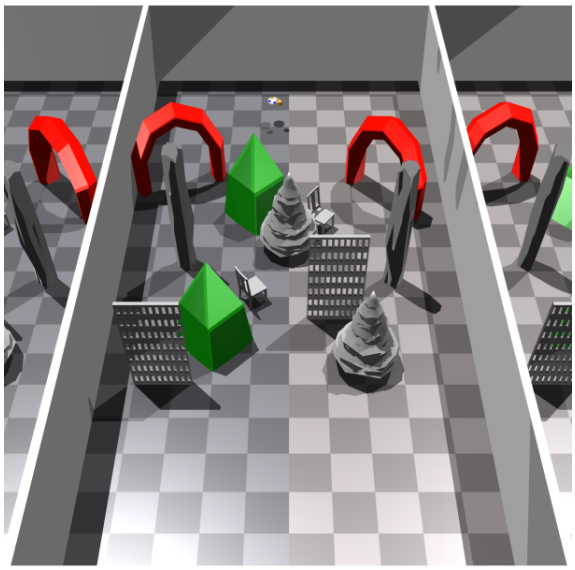


Figure D.4: The hard-swapped environment, with 12 obstacles. The chair and simple stone in the center are swapped to not artificially block the agent.

D.2 Large Environment

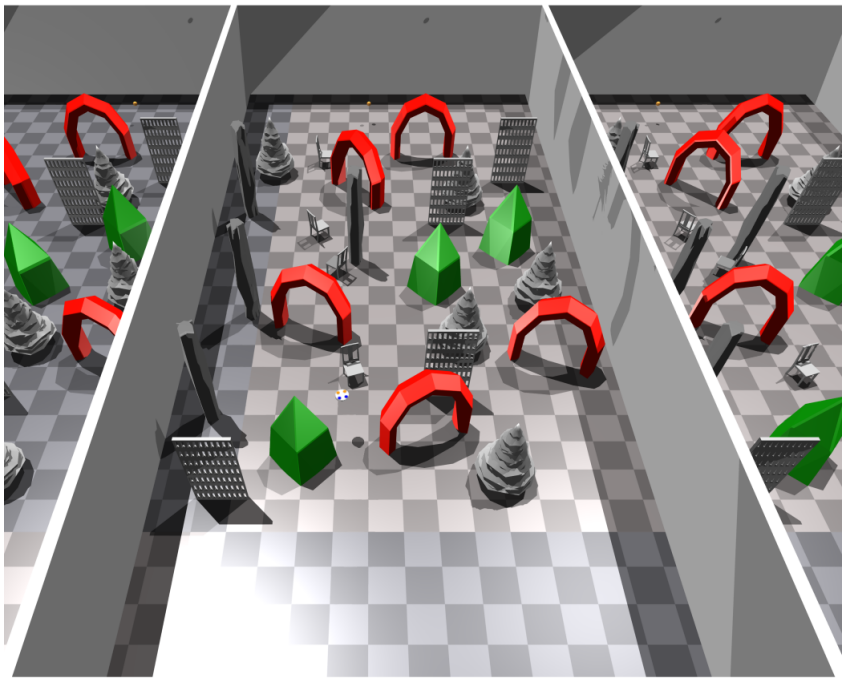
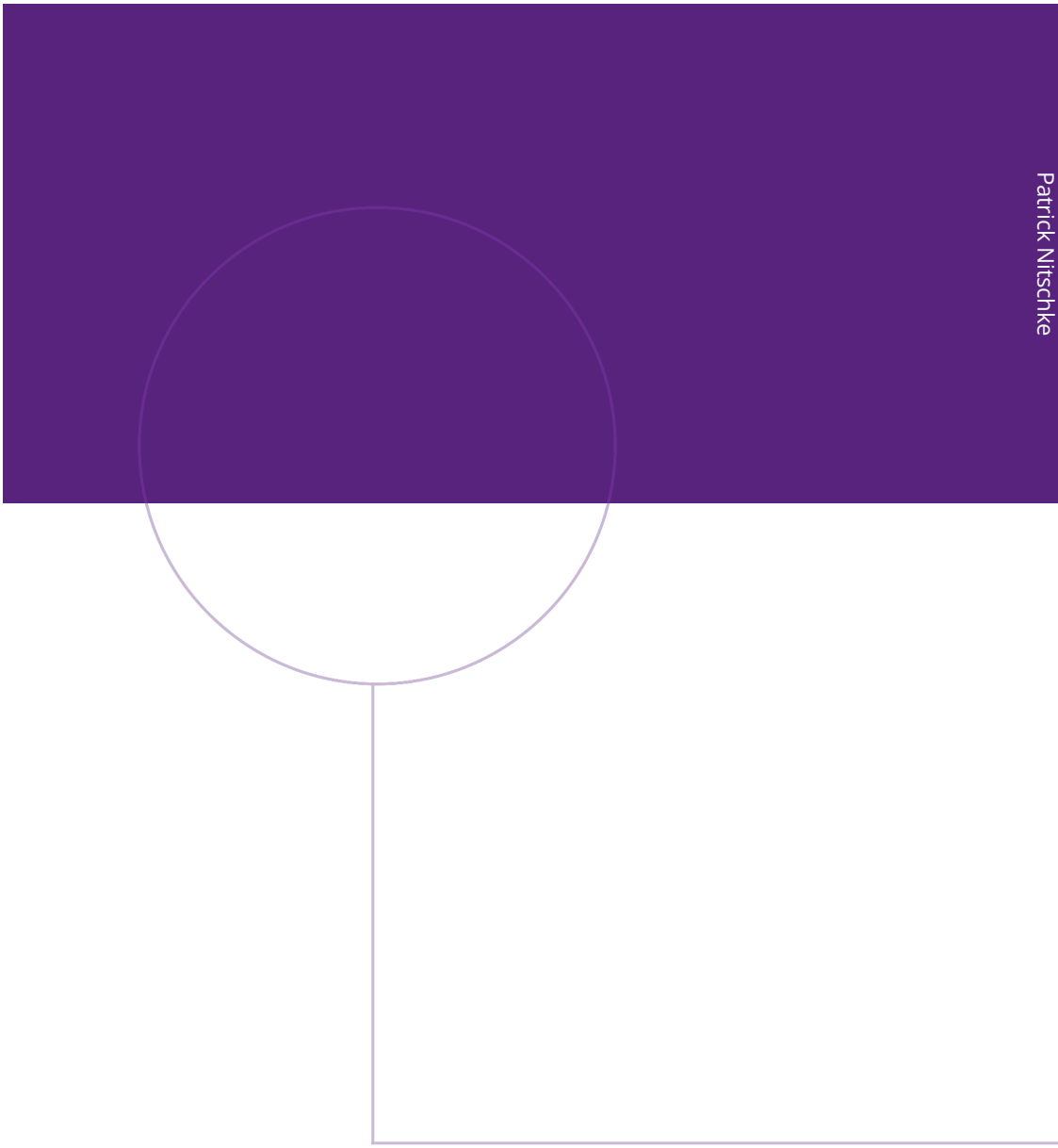


Figure D.5: The large environment, with a dimension of [30, 15] and 25 obstacles.



Patrick Nitschke

NTNU
Kunnskap for en bedre verden