Visual Active Tracking in Simulation with Task-Relevant Features and Deep Reinforcement Learning

Kirsten Nicole Crane

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

June 2024

I would like to dedicate this thesis to my mum, who has asked me about it, supported me, worried for me, and celebrated with me, from the very first day to the very last.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Kirsten Nicole Crane
June 2024

Acknowledgements

Thank you to my supervisors Dr. Stephen McGough, Prof. Nick Wright, and Prof. Per Berggren for their continued support and expert advice throughout my PhD.

Thank you to Prof. Paul Watson for the opportunity to pursue a PhD through the Cloud Computing for Big Data CDT, and to Dr. Savas Parastatidis for his generous alumni donations which funded my preceding Masters degree.

Thank you to Dr. Matthew Sharpe for all of the fieldwork efforts and for answering any questions that required domain knowledge.

Thank you to Tawn Kramer for developing and open sourcing SD Sandbox and gym_donkeycar. Thank you also to Antonin Raffin and the rest of the Stable Baselines core developers team.

Thank you to Jennifer Wood and Andrew Turnbull for your support with University administration, international conference trips, and the organisation of CDT activities, both social and professional. A special thank you to Jennifer Wood for the emotional support she has provided as I have navigated some difficult personal circumstances.

An enormous thank you to Georgia Atkinson and Dr. Cameron Trotter, with whom I have had discussions more or less every day for the entirety of this project. You have been beyond generous with your time and support.

Finally, thank you to my parents, Nigel and Elaine Crane, my siblings, my partner, my dog, and my extended family; especially my uncle David Crane who recommended the PhD programme. I would not have achieved what I have without you all.

This work was supported by the Engineering and Physical Sciences Research Council
Centre for Doctoral Training in Cloud Computing for Big Data
[EP/L015358/1]

Abstract

Using an autonomous underwater vehicle to film marine animals such as dolphins in their natural habitat can greatly aid monitoring, health assessment and animal behaviour research. Having a vehicle autonomously follow and orient toward a species of interest, without the need for tagging, presents a challenging visual active tracking (VAT) problem using image data from the onboard camera. This thesis investigates model-free deep reinforcement learning (DRL) algorithm Soft Actor Critic (SAC) as a potential solution. The utility of this approach is demonstrated in simulation. A follow-up robotics project would then need to look at integrating the simulation-trained control policy with the real vehicle. DRL was selected given that it can support accurate, real-time tracking without needing to model the complexities of a marine environment.

In the VAT literature, research can be divided into end-to-end and task-separated solutions, based on whether or not the state estimation and control sub-tasks are jointly optimised. The benefit of joint optimisation is that state estimation can respond to control performance, and the control can adapt to imperfect state estimation. The challenge is that this requires a network large enough for learning rich representations, whilst also needing to limit the number of network parameters for the difficult credit assignment problem faced by the DRL agent. This thesis explores an approach to VAT which is end-to-end but alleviates some of the burden by learning the majority of perceptual skills prior to agent training, with a separate model - a variational autoencoder (VAE). Furthermore, the task-relevance of these perceptual skills is ensured through the use of a multi-part loss function fed by three auxiliary tasks of target state prediction. This approach to a constrained VAE was presented by Bonatti et al. (2020) in the aerial navigation space, upstream of imitation learning. This thesis extends the approach to DRL and VAT, with a new framework called T2FO (tracking with task-relevant feature observations). T2FO achieves mean episodic return of 2,057 from a possible 3,000, across 100 inference runs of the trained policy. The framework outperforms three baseline SAC policies trained with raw image observations (1,049), unconstrained VAE features (1,198) and target state predictions from the auxiliary networks (1,987).

Neither agent training nor VAE training were possible without first developing a custom environment for the custom problem. This thesis additionally presents three environments developed using the commercial game engine Unity and Open AI's widely used library Gym: a toy environment CubeTrack, a car environment DonkeyTrack, and an application-focused underwater environment SWiMM DEEPeR. For supplementary videos see `https://www.youtube.com/channel/UCA4fgSfe2IctRv5N-Gr0OrQ`.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Conscious of it or not, humans have fantastically good vision-based motor skills, referred to as visuomotor-control. Yet even the simplest of tasks, such as reaching, grasping and object manipulation, can pose a big challenge to machines (Levine et al., 2016a). Of course machines can perform basic motor tasks without visual perception or other human-like cognitive capabilities. These non-intelligent machines have become widespread in industries such as manufacturing, where the environment is highly structured and the tasks are low-skill. However, in a new age of automation driven by artificial intelligence (AI), the focus is on building robots with the cognitive skills to tackle complex problems in more realistic, noisy and unpredictable environments. AI techniques and sub-disciplines can be grouped under two main strands (Samoili et al., 2020). The first rule-based strand includes knowledge representation, reasoning, and planning. In the second strand, where cognitive skills are *learnt* and not programmed, AI branches into the sub-disciplines of computer vision, natural lanuage processing (NLP) and machine learning (ML). Across all of these sub-disciplines, the use of deep neural networks provides the ability to learn from raw, unstructured data – referred to as deep learning (DL).

Solving visual tasks with deep neural networks can be traced as far back as 1989 with LeNet, widely considered as the first convolutional neural network (LeCun et al., 1989). However, it was not until well after the millennium that faster GPUs, the arrival of big data, and solutions to the vanishing gradient problem pushed deep learning into a global boom. In 2009, Fei-Fei Li of Stanford University launched ImageNet (Russakovsky et al., 2015), a free database of more than 14 million labelled images. This provided a real catalyst for the computer vision community, leading to milestone developments such as AlexNet (Krizhevsky et al., 2012), ZF Net (Zeiler and Fergus, 2014) and VGG Net (Simonyan and Zisserman, 2014). Since then, classic computer vision tasks such as classification, localisation, detection, and segmentation

have progressed enormously. For example, at the time of writing, top-1 accuracy (how often the model assigns the highest probability to the correct class) on ImageNet classification is as high as 91% (Yu et al., 2022). Visuomotor-control, on the other hand, remains very much an open problem, especially in the context of robotics and embodied AI.

When it comes to challenging visuomotor problems such as autonomous driving, navigation and visual active tracking, the incentive for success is huge. In transportation, autonomous cars are expected to reduce traffic accidents, improve traffic efficiency, and reduce fuel costs (Waldrop et al., 2015). Autonomous mobile robots can be deployed as service robots in homes (Portugal et al., 2015) or in medical facilities (Fragapane et al., 2020), and in fact they were utilised at the height of the Covid-19 pandemic (Cardona et al., 2020). Unmanned aerial vehicles (UAVs), or drones, can be used for anything from power line inspections (Iversen et al., 2021) to providing counter measures for malicious drone activity (Çetin et al., 2020). In the growing commercial drone market, deep learning could improve autonomous features such as the currently GPS-reliant 'follow-me' technology sought after by hobbyist cinematographers (Do and Ahn, 2018). In defence, as militaries around the world compete for the portions of the electromagnetic spectrum used to control unmanned vehicles and combat systems, progress in AI-driven automation is not just a nice to have – it is a high-stakes race (Swett et al., 2021).

Cars and drones may be the better known of the autonomous vehicles, but the applications for autonomous underwater vehicles (AUVs) are enormous, from inspecting marine infrastructures for the energy industry, to detecting naval mines for the defence industry (Inzartsev, 2009). They are also a valuable asset for research. Dangers such as high water pressure and fluctuating currents complicate and prevent ocean exploration, but AUVs can extend our reach. They are smaller, lighter and more portable in comparison to submarines, and yet they can be robust enough to handle extreme environments such as the deep ocean or polar regions. This benefits a broad range of scientific fields, including medicine, geology, ecology and meteorology. Amidst a backdrop of climate change, biodiversity loss and depleting land resources, the adoption of AUVs and other technology in areas like exploration and conservation may well be crucial (Liu et al., 2022). The next section details the very specific AUV application this thesis considers, as well as the impact of this application and a high-level research question.

1.1 Impact Areas and Research Question

This thesis focuses on the application of AUVs to the scientific field of marine biology, specifically marine mammal research, such as population monitoring, sound production, and the assessment of bycatch and other anthropogenic threats. Population monitoring often involves conducting longitudinal research on the species of interest, producing a catalogue of individuals sighted during dedicated surveys. Monitoring cetaceans (whales and dolphins) is particularly important because they are natural sentinels (barometers) of ecosystem health (Bossart, 2011). They feed at a high trophic level, have a long life span, and their thick layer of blubber can store anthropogenic toxins. This makes them useful indicators of changes to ocean temperature, salinity, currents, production hotspots, food webs, contaminant levels, and disease pathways (Moore, 2008).

Conventional data collection methods tend to be vessel-based. For example, above water photographs are taken of the dorsal fin as the animal breaches the waterline. These are then used in a process called photo-identification – cross-referencing incoming data with a catalogue, for the purpose of identifying individuals from unique features such as dorsal fin shape, notches, scarring, and pigmentation (Würsig and Würsig, 1977). Similarly, the behaviour of the animals during any given sighting is determined by surface activity, using broad categories such as resting, foraging, travelling or socialising. These methods have some obvious limitations when studying members of a species that spends the majority of their time underwater, especially when they surface so fleetingly and with lots of spray.

The best scientific insights and conservation outcomes require varied data, including aerial photography, underwater photography, and underwater acoustic recordings collected in unison with underwater video footage. For example, underwater video stills can support external health assessment, such as the identification of a skin disorder amongst the monitored population (VanBressem et al., 2018). The same underwater imagery would also benefit photo-identification, given that more of the animal would be visible and therefore any unique markers on the head, flanks, fluke or pectoral fins could also be incorporated. However, where underwater data would likely contribute most is behavioural study. Underwater video data would allow researchers to observe a much broader range of behaviours as well as individual differences, providing insights into the nature of social interactions, for example. The opportunity to link these observations with time-matched audio recordings would also provide exciting new data for studying vocalisations.

The footage that supported the VanBressem et al. study was captured by a human diver. Diving with these animals requires a licence, good sea conditions, daylight hours, and a nearby support vessel. An AUV on the other hand is a bit more robust to deployment conditions, and can travel faster, deeper and with more agility. As such, there has been an increasing use of AUVs in marine mammal research (Dutton et al., 2019; Nelms et al., 2021). There is also the potential for independent missions, or at least the ability to stray further from a supporting research vessel, avoiding any compounding influence this might have on animal behaviour (Guerra et al., 2014; Heiler et al., 2016). A prime example of the utility of AUVs is the 'SharkCam' study Kukulya et al. (2015) on white sharks near Guadalupe Island, Mexico. In this and similar follow-up studies (Dodge et al., 2018; Kukulya et al., 2016; Packard et al., 2013), the AUV was fitted with an Ultra Short BaseLine receiver, used to interrogate a 38cm transponder tag attached to the dorsal fin of the shark. The round trip travel time of the response provided the shark's range, whilst beam-forming of the response provided a bearing, and a second, time-delayed response provided a depth. Together, this 3D location was fed into a (non-disclosed) algorithm, using past behavioural data to forward predict the shark's location and therefore target location for the vehicle to move to. Following this methodology, footage captured at 100m provided the very first observations of deep water attacks, and detailed behavioural data that the authors claim would be impossible to collect with conventional techniques.

Whilst the study was a success in terms of the observational outcomes, the autonomous active tracking was sub-optimal. For example, Kukulya et al. note the poor performance of the prediction algorithm due to limited training data and the shark's dynamic movements. They describe regular tracking failures such as 'fly-bys' (motoring past the animal) and needing to manually adjust the vehicle's speed. Furthermore, the methodology relies on first being able to find and successfully tag the shark. Although the use of animal-borne electronic devices is common practice in marine science, it is challenging and time consuming, requiring expertise within the team, an appropriate tag (sometimes species specific), and lots of considerations with respect to the safety and welfare of both the animal and the crew. Methods of deployment and best practice guidelines continue to improve, however the process is not without hazards (Andrews et al., 2019; Horning et al., 2019, 2017). This prompts the question:

Can computer vision and deep learning facilitate a better performing, tagless method of AUV control for collecting data on free-roaming marine megafauna in the wild?

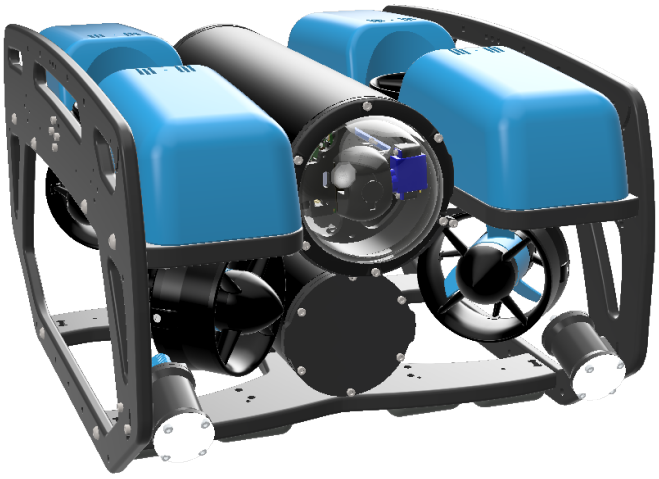


Fig. 1.1 The BlueROV2 from Blue Robotics.

The next section translates this high-level research question into a problem specification and set of aims, firstly in terms of the big picture, and then in terms of the specific problem addressed by this thesis.

1.2 Aims and Contributions

The ultimate aim is to send out an AUV on an independent mission to survey a given area and collect high quality data if and when a target species is encountered. The craft recommended by collaborators in the School of Engineering is pictured in Figure 1.1 - the BlueROV2 from Blue Robotics[1]. As the most affordable, high-performance remote operated vehicle (ROV) on the market, it is a popular choice, with many accessories and expansion options. More importantly, it uses open source software with thorough documentation, facilitating development and modifications. Like most ROVs, the BlueROV2 is designed to be connected to a long umbilical tether used for sending sensory information and receiving commands from a human pilot. As long as 300m, the tether connects with a laptop, which then connects with a gamepad controller. Although the vehicle can (and will) be used in this way, the long tether can be hard to manage, is costly, and can limit the range and freedom of movement of the vehicle (Wynn et al., 2014).

[1] https://bluerobotics.com/store/rov/bluerov2/

The goal, therefore, is to cut the cord between human and robot and introduce autonomous control. In recent times, AUVs have become commercially available but are intended for large corporations and cost anything from tens of thousands to millions of pounds (Fowler et al., 2016). Modifying the BlueROV2 is therefore more cost-effective, but this will require in-house development of autonomous control algorithms which we seek to provide through deep learning. AUVs have varying levels of autonomy and, in the case of commercial AUVs, an autonomous operational mode is typically nothing more than preplanned route following. Here, the requirement is for visual active tracking of a dynamic, moving target – a two-part problem of state estimation and control, demanding high processing speeds and adaptive sequential decision making. This specification motivates the need to use deep reinforcement learning (DRL) – a machine learning paradigm (detailed in Chapter 2) which has had considerable success at directly mapping pixels to control; so called end-to-end control (e.g. Mnih et al. (2015), Lillicrap et al. (2016)). Although DRL could be used with non-visual observations, the standard configuration of the BlueROV2 ships with a monocular RGB camera, and therefore image data presents the most cost-effective solution.

Figure 1.2 provides a high level overview of the proposed solution. Without a tether, the modified BlueROV2 requires an alternative method of communication. Since radio waves do not propagate well through water, most AUVs use acoustic waves, however this method suffers from small bandwidths, low data rates, high latency, limited range, frequent data loss and is susceptible to security threats (Liu et al., 2022). This is not well suited to real-time decision making. Instead, the trained deep neural networks and the scripts that orchestrate model inference will sit on the BlueROV2's existing onboard Raspberry Pi. Some form of land or vessel communication will still be required for purposes such as human-in-the-loop, mission evaluation, and system recovery. These much less frequent interactions will need to suffer the higher latency, but the autonomous operational mode will benefit from the very low latency of an onboard solution. The DRL algorithm further benefits from not needing to model the complex and highly non-linear hydrodynamics of the vehicle. Since BlueROV2 is designed to receive human operator commands, the vehicle has sophisticated autopilot software (ArduSub [2] and Ardupilot [3]), responsible for translating inputs from a gamepad into signals for the individual thrusters.

To develop this proposed solution to the point of deployment, with all of the necessary safety features and capabilities, will undoubtedly require a multi-part solution and a huge

[2] https://www.ardusub.com/

[3] https://ardupilot.org/

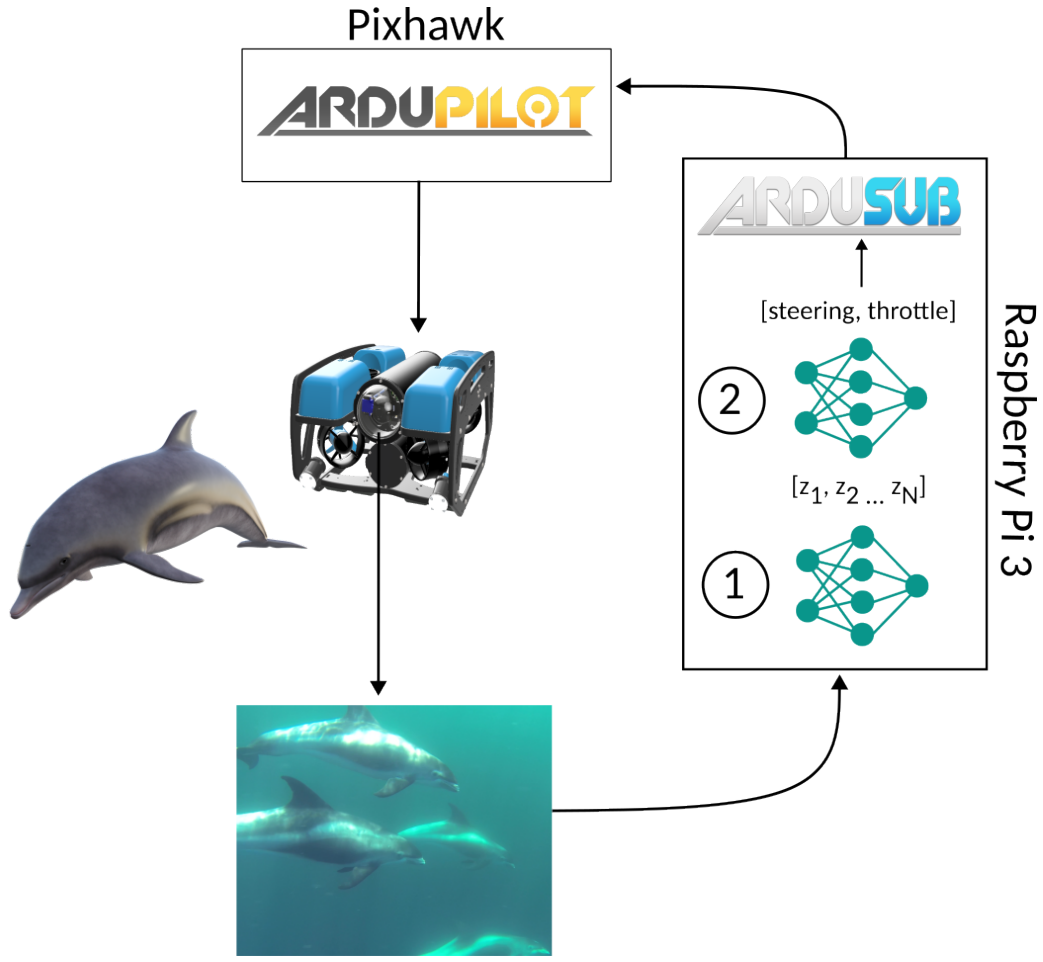


Fig. 1.2 High level overview of the proposed solution: camera frames are sent to the onboard Raspberry Pi, where 1) an autoencoder maps the raw image to a 1D feature vector, and 2) a policy network maps the feature vector to a throttle and steering value to be sent forward to the existing autopilot software.

number of considerations. For a species that travels in pods, a multi-object tracking algorithm is required. For a species that might interpret tailgating as predatory, the active tracking algorithm will need to position itself intelligently. In all cases, the algorithm will need to incorporate collision avoidance and battle against tides and other hydrodynamic forces. The enormity of this challenge is going to require a project that spans multiple PhD projects, with collaboration across the School of Computing, School of Engineering, and School of Natural and Environmental Sciences.

For this computer science doctorate project, the focus is on the **underlying computer science problem of DRL for visual active tracking**, and represents initial forays into the problem. As is extremely common in the DRL field, the project uses a simulation for training agents.

A simulation representing this very niche problem was not available off-the-shelf, therefore it was necessary to tackle both agent training and environment development. The agent is trained in simulation as a precursor to using the agent to control the real vehicle, with or without additional real-world training. However, this process of sim-to-real transfer falls outside the scope of this thesis and would be better suited to a robotics project. Instead, the robustness of the simulation-trained models to the sim-to-real gap is tested with a proof of concept study toward the end of Chapter 4.

Finally, given the difficulty of vision-based DRL, control has been limited to two dimensions – forward and backward on the z-plane (surge, or thrust) and rotation around the vertical axis (yaw, or steering). The problem does not consider heave (depth), pitch or roll. The active tracking problem has also been reduced to distractor-free, single-object tracking. Distractor-free means that the learning environment provides an empty scene other than the moving target object. This restricts collision avoidance to collisions with the target itself, and allows the solution to be class-agnostic i.e. the only required classification capability is between object and background, not between object and object.

The main contributions are as follows:

1. **T2FO**: A framework for the training of visual active tracking. The acronym stands for 'tracking with task-relevant feature observations', describing a framework whereby a pre-trained encoder network takes a raw image observation of the DRL environment, encodes it to a one-dimensional feature vector, and then passes this feature observation as input to a DRL policy network. Specifically, the pre-trained encoder is a variational autoencoder, but one trained with the downstream control task in mind, using auxiliary tasks of target state prediction in order to cosntrain and disentangle what is learnt in a subset of the features, ensuring their task relevance. This solution is novel to visual active tracking, and outperforms the same policy network trained with raw image observation or standard, unconstrained features.

2. **CubeTrack:** A new, open source, Unity learning environment for the DRL researcher. This toy problem of two cubes moving around an enclosed platform follows the design of environments within the suite provided by the Unity ML-Agents Toolkit (Unity-Technologies, 2017b). At the time of writing, this environment suite did not include an active tracking problem, and so CubeTrack addresses that gap. The simple graphics, high-contrast colours, discrete action space, and option to use ground truth object properties as a vector observation, make this environment a perfect test environment or entry-level environment for active tracking research.

3. **DonkeyTrack:** A modified version of the open source, Unity learning environment 'gym-donkeycar' (Kramer, 2018). The original environment is intended for the problem of track racing whereas DonkeyTrack repurposes the environment for active tracking. In comparison to CubeTrack, DonkeyTrack is a more realistic environment, both in terms of visual and physical fidelity. It also progresses from discrete actions to continuous control. DonkeyTrack was a necessary stand-in environment whilst a collaboration was formed with the Games Engineering department to develop a custom underwater environment.

4. **SWiMM DEEPeR:** A new, open source, *underwater* Unity learning environment, purpose built for the conservation application described. The simulation includes a pseudo-realistic open ocean scene, animated dolphin, and simulated BlueROV2 vehicle. A real focus is placed on the environment being user-friendly and data-driven, making as many aspects as possible configurable without needing to rebuild the Unity executable. This custom-built application is essential to ongoing work on *this* project, but also benefits both the underwater robotics and DRL communities. To the best of the author's knowledge, it is the first AUV simulation in the underwater robotics community to provide a conservation angle. In the DRL community, it adds to a limited number of pseudo-realistic environments for learning greater than two dimensions of continuous control.

1.3 Thesis Structure

Chapter 2 This initial chapter entitled 'Background' lays the theoretical foundations on which the rest of the thesis is built. Background in terms of related research is reserved for Chapters 3, 4 and 5, such that a dedicated literature review is presented in proximity to the work to which it corresponds. The first section provides a definition for deep learning and then introduces neural networks, objective functions, gradient based learning and backpropagation. Next are two model types central to the proposed solution – the convolutional neural network and the (variational) autoencoder. Following this is a broad introduction to model-free reinforcement learning, before focusing in on model-free *deep* reinforcement learning and the core algorithms in this space.

Chapter 3 This is the first of the core research chapters and the equivalent of a data chapter. The introduction to this chapter explains why simulation-based training is necessary, the involved software stack, and why there is a growing trend toward the use of commercial game engines. The related work section reviews relevant platforms,

toolkits and environments, starting with general purpose toy environments, before looking at higher fidelity car, drone, and finally, AUV simulations. The main body of the chapter then presents all three custom environments developed in the course of this project: CubeTrack, DonkeyTrack, and SWiMM DEEPeR.

Chapter 4 This second core research chapter introduces the idea of decoupling representation learning from policy learning (learning control), both to reduce training times and to improve the robustness of sim-to-real transfer. The related work section reviews examples of just that, first in terms of standard representation learning and then in terms of state representation learning. The work presented in this chapter is a reproduction of the 'Cross-Modal Variational Autoencoder' presented by Bonatti et al. (2020), but using the DonkeyTrack environment as opposed to AirSim. A section is dedicated to summarising the approach proposed by this paper, before detailing the data collection and models in the methodology section, including an initial 'proof of concept' experiment into sim-to-real transfer. The results section reproduces the same figures as the original paper to provide a direct comparison, providing evidence of the method's reproducibility and successful transfer to new simulation environments. An ablation study compares the results to an unconstrained variational autoencoder with the same architecture, and a variational autoencoder without the residual blocks provided by the ResNet architecture. Results are also provided for the sim-to-real transfer experiment, assessing the similarity of feature vectors encoded from simulated images to feature vectors encoded from real world counterpart images.

Chapter 5 This third and final research chapter focuses on policy learning. The introduction goes in to more detail on visual active tracking as a visuomotor-control problem, and how it differs from the better known problem of visual object tracking. The related work section reviews notable solutions to similar problems, comparing the task-separated approach to visuomotor-control versus end-to-end. The methodology section details the overall framework for policy training, reward functions and episode termination criteria, and the Stable Baselines implementation of the Soft Actor Critic algorithm introduced in Chapter 2. The results section provides training graphs of episodic return, inference results and videos of trained agent behaviour, across all three environments, but with greater focus on the DonkeyTrack environment where the majority of tuning and experimentation was carried out. The influence of various hyperparameters and reward functions is reported on, with a note on responsible reporting. An experiment is dedicated to comparing observation spaces (raw pixels versus feature vectors, versus

state predictions) and to comparing feature observations learnt under the cross-modal approach as opposed to an unconstrained approach.

Chapter 6 This chapter summarises the thesis and provides a critical evaluation against the thesis aims. The future work section describes several interesting directions for future research, building upon the work undertaken in this thesis.

1.4 Publications

The work outlined in this thesis has led to the publication of the following peer-reviewed paper:

Appleby et al. (2023) Appleby, Sam, Crane, Kirsten, Bergami, Giacomo and McGough, A. Stephen, 2023. SWiMM DEEPeR: A Simulated Underwater Environment for Tracking Marine Mammals Using Deep Reinforcement Learning and BlueROV2.

Specifically, this paper relates to the development of the novel underwater environment SWiMM DEEPeR, including the Unity simulation, Python application, and network communication between the two. It therefore mostly relates to Chapter 3, but provides results for autoencoder training (Chapter 4) and policy training (Chapter 5). This paper was accepted to IEEE Conference on Games 2023 for a fifteen minute talk. Post conference, an invitation was received to contribute an extended version to the IEEE Transactions on Games journal, running a special issue to highlight the best works from the conference. Equal contribution was provided by the first and second author, but given the focus of the conference it was felt that it was more appropriate for Samuel Appleby, the games engineering student, to be first author. Samuel was responsible for developing the Unity simulation, I was responsible for the Python application and for the requirements of the Unity simulation, and we collaborated together on the network communication.

The following publication was also produced during this thesis. Born from the same working collaboration, it shares the same motivation (AI for conservation), but is not directly related to the contents of this thesis.

Trotter et al. (2020) Trotter, Cameron, Atkinson, Georgia, Sharpe, Matt, Richardson, Kirsten McGough, A. Stephen, Wright, Nick, Burville, Ben and Berggren, Per, 2020. The Northumberland Dolphin Dataset: A Multimedia Individual Cetacean Dataset for Fine-Grained Categorisation. In *The 6th Workshop on Fine-Grained Visual Categorization, CVPR 2019*. Available: doi.org/10.48550/arXiv.1908.02669.

Specifically, this paper presents a novel computer vision image dataset of two different dolphin species, annotated for both coarse and fine-grained instance segmentation and categorisation. Although this dataset does not feed any of the models discussed in this thesis, a crew of at least two (but ideally four) people were required to collect this data, conducting North Sea surveys in a research vessel as many days as the weather permitted during the summer months each year. Therefore, the more people from the lab that contributed, the more feasible this was. The resulting dataset has been well received; it was used by Meta AI Research to evaluate their 'Segment Anything Model' (Kirillov et al., 2023) and was included in a publicly hosted Kaggle competition [4].

[4]https://www.kaggle.com/c/happy-whale-and-dolphin/

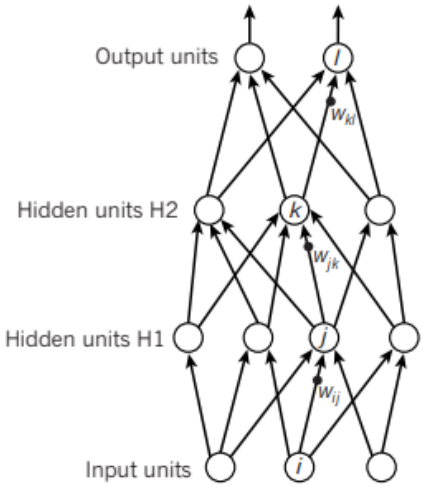
Chapter 2

Background

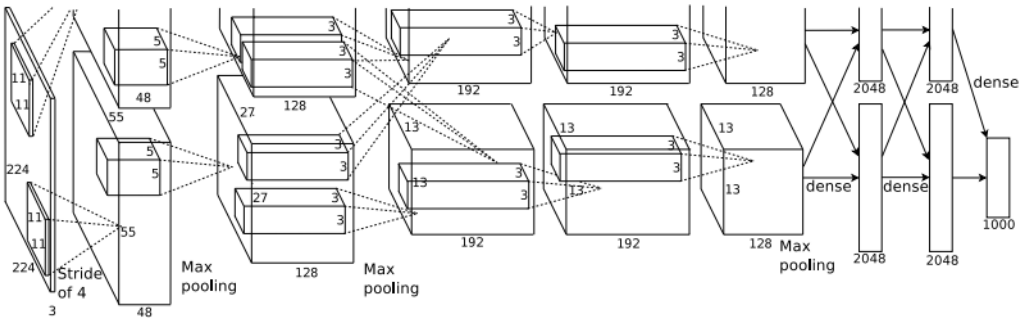
2.1 Deep Learning

Machine learning describes the ability of computer systems to learn in the absence of rules and explicit instruction, using statistical models to draw inferences from patterns in data. The term 'deep learning' indicates the use of an artificial neural network (ANN) with two or more processing layers, affording greater complexity of learnt representations. In a deep neural network (DNN), each layer builds on the next, representing the input data at a greater and greater level of abstraction until, at the final layer, the network outputs a prediction (Goodfellow et al., 2016, Chapter 6). This process of forward propagation, giving rise to the name feed-forward network, is what allows for DNNs to learn directly from raw, unstructured data, unlike classical machine learning which requires a method of feature extraction (see Khalid et al. (2014) for examples). *Learning* features as opposed to specifying features can be extremely powerful and removes the need for any domain knowledge.

Introductions to deep learning commonly illustrate a DNN with a simple, directed, acyclical graph, such as the one in Figure 2.1a. In research papers, network visualisations look more like Figure 2.1b, representing layers as blocks to avoid drawing hundreds of nodes and edges. The nodes of a neural network are called neurons, named after the specialised information processing units of the brain. Biological neurons receive multiple incoming signals through structures called dendrites that branch from the cell body. The cell's output travels down a fiber called the axon as a sequence of electrical pulses called action potentials. At the synapse (axon end), the signal is transmitted to the dendrites of other neurons across a small gap (cleft), typically with a chemical signalling process involving neurotransmitters (Sutton and Barto, 2018, Chapter 15). Inspired by this, artificial neurons in hidden layers (any layer between the 'visible' input and output layers) receive multiple inputs from neurons in



(a) Simplistic diagram of a neural network, from LeCun et al. (2015)



(b) A diagram of the AlexNet architecture from Krizhevsky et al. (2012)

Fig. 2.1 Example diagrams of artificial neural networks.

the previous layer and pass a single output to neurons in the next layer. If each neuron is connected to *all* of the neurons in the previous layer, it is referred to as a fully-connected layer.

At the level of a single node, inputs are combined with a weighted sum, followed by the addition of a vector called the bias (b), followed by a simple non-linear transformation such as the rectified linear unit (ReLU) transformation – $\max(0,z)$ (Nair and Hinton, 2010). Taking from the idea of biological neurons 'firing' or 'activating', the resulting scalar value is called the activation and the non-linear function applied to the weighted sum, the activation function. The same vector-to-scalar function is performed in parallel by all neurons in the layer, and the scalar outputs are combined to provide the vector input to neurons in the next layer. Therefore, the layer as a whole provides a vector-to-vector function and the network as a whole can be viewed as a nested function or function chain. The idea is that the resulting complex function approximates a function $y = f(x)$ which maps input x to desired output y

(Goodfellow et al., 2016, Chapter 6).

There are lots of factors that affect this mapping, including the number of layers, the number of neurons in each layer, and the non-linear activation functions that are used, to name just a few. Many of these are design choices for the programmer and will be task-specific, although there are popular architectures that seem to do well across a range of tasks, and a handful of activation functions which are almost always selected from in practice (see Sharma et al. (2017) for examples). Introducing these non-linearities throughout the computational graph allows the machine to learn complex representations with non-linear decision boundaries. The concept of learning here refers to an iterative tuning process done at the level of the weighted sum. The vectors of weights assigned to the neurons are collectively referred to as the network's parameters, often denoted θ. Typically, these are randomly initialised and then adjusted a small amount with each network update (with the amount and cadence specified by hyperparameters). A common methodology for network updates is gradient-based learning, discussed in section 2.1.2 following an introduction to objective functions.

2.1.1 Objective functions

A key concept in deep learning is that of an objective function $J(\theta)$. Without some means of measuring the quality of the model output, the machine cannot be expected to optimise the model parameters. A very common expression of 'quality' is some measure of error. To provide an example, if the task is to predict house prices and the *true* price is known (regression problem with a supervised learning approach), then the error can be measured as the difference between the prediction and the ground truth with a function such as

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i is the ground truth value, $\hat{y}_i$ is the predicted value, and N is the number of samples. This particular objective function is called the mean squared error (MSE). When an objective function measures something that ought to be minimised, it is commonly referred to as a loss function or cost function (see Wang et al. (2022) for a summary and analysis of 31 classical loss functions, including hinge loss, logarithmic loss and Softmax cross entropy loss). The unsupervised and reinforced learning paradigms also use objective functions. For example, unsupervised training of a generative model will likely involve some measure of divergence - the difference between the distribution of our input data (e.g. a photograph) and

the distribution of our generated data (e.g. the photograph styled as a Monet painting).

One unifying way to talk about objective functions is to formulate the problem as a density estimation problem, and one solution to density estimation is Maximum Likelihood Estimation (MLE). Typically, MLE is described as finding $P(X)$ (the probability distribution of X) that best explains the observed data. This is done by maximising $L(X)$, the likelihood of observing data X. This same approach can be applied to data where we have input and output variables, changing $P(X)$ to $\mathrm{P}(Y|X)$ (the probability of output Y given input X) and changing $L(X)$ to $L(Y|X)$. Both the probability and likelihood are written as $P(Y|X;\theta)$, since they are also conditional on the model parameters. Note that a semicolon is used since θ is an unknown parameter and not a random variable. The capitalisation of X and Y indicates a set of multiple individual samples. Given a training dataset with N samples of input data X and output data Y, $P(Y|X;\theta)$ is actually the joint probability distribution $\prod_{i=1}^{N} P(Y_i|X_i;\theta)$. Multiplying many small probabilities together can be numerically unstable and so the probability is expressed as a log-probability and the product restated as a sum according to the log rule $log_b(mn) = log_b(m) + log_b(n)$. The likelihood therefore becomes a log likelihood, and is typically multiplied by -1, such that it is something to minimise and not maximise. The resulting *negative* log-likelihood is also called the cross-entropy between the training data and the model distribution. Cost functions like MSE can be derived from here by expanding the equation and dropping any terms that do not depend on θ (Goodfellow et al., 2016, Chapter 6).

2.1.2 Gradient-based learning

For any given set of model parameters, the model can be evaluated according to the objective function. As discussed, the objective function produces some measure of model output quality, such as loss, as an average over input samples. If one were to map all possible parameter choices to this quality measure, the result would be a very highly dimensional plane given that the model has many parameters (see Li et al. (2018b) for visualisations referred to as loss landscapes). The aim of model optimisation is to find the global minimum (deepest trough) on this plane since, at this point, the parameter values produce the smallest loss. With each model update, the computer system computes the gradient of the current position on the landscape (i.e. the gradient of the objective function using the outputs produced with the current model parameters). Parameters are then adjusted a small amount in the direction of negative gradient (if the quality measure is something like cost that ought to be minimised). Obviously, if the quality measure is instead something to maximise, then parameters are updated in the direction of positive gradient. If the high dimensional plane were convex, this

would represent a small step toward the global minimum. However, since it is non-convex, the update might represent a step toward a local minimum.

In one dimension, the gradient of a function $f(x)$ is the derivative of the function i.e. $f'(x)$ or $\frac{\delta f(x)}{\delta x}$. In multi-dimensions, the gradient is the vector of partial derivatives along each dimension i.e. $(\frac{\delta f(x)}{\delta x_1}, \frac{\delta f(x)}{\delta x_2}, ... \frac{\delta f(x)}{\delta x_n})$ or $\nabla_x f(x)$. Applying this to deep learning, the parameters of a network can be updated each iteration with

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$$

where J is something of the form $\frac{1}{N}\sum_{i=1}^{N} \mathscr{L}(\hat{y}_i, y_i)$, like the MSE equation given in 2.1.1. The ∇_θ denotes we are taking the gradient with respect to (w.r.t) the network parameters, and the α multiplier is the learning rate. There will be much said on learning rates in later sections, but for now it is a value less than one which limits the size of the update step.

A problem with this update rule is the computational expense of averaging over N when N is very large, as it needs to be when applying deep learning. Rather than evaluate the model against all samples before taking an update step, it is possible to evaluate the model against a subset of samples (a mini-batch), providing an estimate of the overall loss. This approach is called mini-batch Stochastic Gradient Descent (SGD) based on the method of stochastic approximation first described by Robbins and Monro (1951). Evaluating the model against a subset of data provides a noisy gradient estimate and as such the 'path' to the optimal solution will be less direct (hence the term stochastic). Whilst this results in a longer convergence time, it reduces computational time and memory requirements. As such, SGD and its improved variants are widely used and, when compared with other gradient-based algorithms, provide the best generalisation performance (Bottou and Bousquet, 2007).

2.1.3 Backpropagation

Even if the averaging were removed entirely and the objective function were fed just one sample, computing the gradient of the objective function w.r.t the network parameters is non-trivial when the network may have millions of trainable parameters. In a fully-connected layer, each node in the layer has a number of parameters equal to the number of nodes in the previous layer. Therefore, the number of parameters for the layer is the number of nodes in the previous layer multiplied by the number of nodes in the layer, plus the number of nodes in the layer (since each node barring the bias is also connected to the bias of the previous

layer). When the network is 152 layers deep, as it is in a variant of ResNet (He et al., 2016), the number of partial derivatives to compute becomes very large. Backpropagation, short for 'backward propagation of errors', provides an efficient method for computing the gradient analytically as opposed to numerically, using the chain rule

$$\nabla f(g(x)) = g'(f(x)) \cdot f'(x)$$

i.e. how to find the derivative of a composite function. In the opening of Section 2.1, the network was described as a computational graph – a chain of functions for the input to pass through in what is called the forward pass. The objective function can be thought of as just one more function at the end of this function chain, and then backpropagation is essentially a *backward* pass through the function chain to *recursively* compute the gradient w.r.t every variable in the computational graph. Moving back through the network layers, information (i.e. partial derivatives) is passed back, reducing the amount of computation required at the next layer. Backpropagation was first proposed by Linnainmaa in their Master's thesis in 1970, but it was not until 1986 that Rumelhart et al. demonstrated the advantage of using backpropagation when working with computational graphs.

2.1.4 Convolutional neural networks

The standard form of feed-forward network discussed thus far is often referred to as a Multi-layer Perceptron. In the computer vision context, the input data is image data, represented as a $h \times w \times d$ (or $d \times w \times h$) matrix of pixel values, where h is the image height, w is the image width, and d is the number of colour channels (i.e. 3 for RGB images). Of course, the matrix can be flattened to give a $1 \times hwd$ vector, which can then be passed through an MLP like any other data. However, treating the pixel values as a 1D vector means that crucial spatial information is lost. To this end, the convolutional neural network (CNN) was introduced – the idea appearing in the late 1980s (Fukushima, 1988) and the first practical application being seen in the late 1990s (LeCun et al., 2015) in the context of digit recognition using the MNIST dataset (Deng, 2012).

Recall how, in an MLP, the incoming values to each neuron are combined in a weighted sum to give a single value called the activation. This weighted sum is a dot product between the input vector and the weight vector. In CNNs, weights are stored in a matrix called a kernel or filter, with the same depth as the input data and often something like 3×3 or 5×5 width and height. Starting (typically) in the top left corner, the kernel is overlaid with the image and a dot product taken. Using a sliding window method, the kernel is then moved right by n pixels

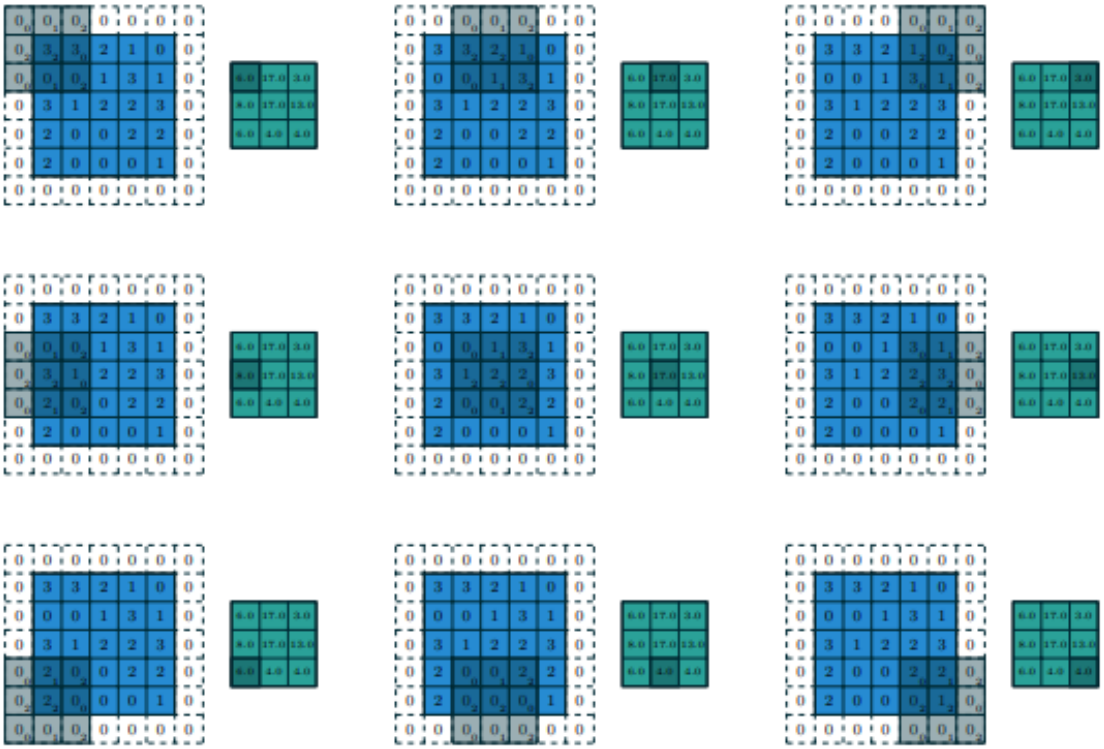


Fig. 2.2 Visual aid for understanding convolutions, from Dumoulin and Visin (2016). The blue squares represent the image matrix (surrounded with padding), the grey squares represent the kernel, and the green squares represent the feature map.

(called the stride) and the dot product taken again. Depending on the image-to-kernel width ratio and the stride, the kernel may overhang the image when it reaches the edge. Rather than not take a dot product here, the image can be padded i.e. the border of the image surrounded with additional values. The three common padding type are fixed (filling the outer padded region with a fixed value, usually zero), reflection (reflecting the input image pixels about the border), and replication (copying the nearest border pixel). Each dot product is stored in the corresponding cell of the output matrix called the activation map or feature map. In the case where depth is greater than one, as with RGB images, the kernel will have the same number of dimensions and the channels are summed element-wise along with the bias. Figure 2.2 provides a visual aid.

The kernel size K, stride S and padding P determine the size of the feature map according to the formula $[(I-F+2*P)/S]+1$, where I is the size of the input image. Using a small kernel not only reduces computation (the sparse weights advantage), but encourages a form of dimensionality reduction. Thinking about this in quite a literal sense, a raw image might

be thousands of pixels tall and wide, but the number of pixels containing edges will only be a small subset. Interestingly, the convolutional layers of a CNN act in much the same way as the visual pathways of the brain, representing raw data with increasing levels of complexity, from edges to shapes to objects. In addition, different kernels can be applied over the same input, producing multiple feature maps and allowing the model to distil different features at the same level, for example, image brightness as well as edges. As before with MLPs, features like these are *learnt* and not specified.

Other than convolutional layers, CNNs also utilise pooling layers to down sample the data, reducing sensitivity to the position of features. Pooling layers apply the same sliding window approach as convolutional layers, except rather than learn the operation (i.e. the coefficients of the weighted sum) the operation is specified. For example, taking the maximum of all values in the window (max pooling) or the mean (average pooling). As features become more and more abstract, spatial dependencies dissolve and the feature map can be flattened and passed through a fully-connected layer. In a classification task, for example, the final layer will be a fully-connected layer with the same number of neurons as classes. Use of the Softmax activation function produces values in the range [0, 1] which sum to one, allowing the model to output the probability of the input belonging to the class for each class.

There are additional implementation details and additional concepts such as equivariant representations that are beyond the scope of this chapter. The material covered here provides enough background on deep learning to provide a foundation for the deep reinforcement learning algorithms and deep generative models presented as solutions. The reader is pointed to the Stanford University lecture collection 'Convolutional Neural Networks for Visual Recognition' (2017) for a comprehensive and detailed explanation of backpropagation, objective functions, optimisation, CNNs, and lots of additional content such as regularisation methods, gradient descent algorithms other than vanilla SGD, and another family of networks called the *recurrent* neural network.

2.2 Feature Learning

Convolutional neural networks appear wherever there is image data involved, providing the underlying architecture for many models, across a wide range of tasks. As discussed, convolutional layers and pooling layers can be used to learn a compressed representation of the data. In the classification example presented above, the utility of the compressed representation (features) is reflected in the prediction error. If the features do not capture

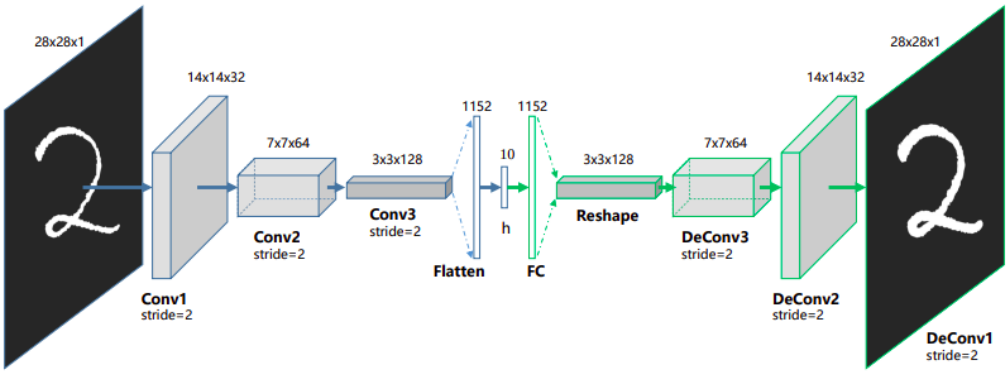


Fig. 2.3 An example convolutional autoencoder architecture, from Guo et al. (2017).

meaningful attributes of the input data then the model will not be able to perform the task and the discrepancy between output and ground truth will be large. Learnt features can also be evaluated in an unsupervised manner, in the absence of any prediction task or labels. If the task for the model is to simply decompress the compressed representation and recover the input data, then the model is incentivised to work toward a compression which is as close to lossless as possible. This method of DNN-based representation learning, with no other objective but to learn good features, can be achieved with a model called an autoencoder.

2.2.1 Autoencoders

An autoencoder is a special use case of a feed-forward network whereby the dimensionality of the output matches the input, but the input has to pass through a 'bottle neck' of hidden layers which reduce the dimensionality of the data down to a low-dimensional vector of features called the latent vector. In the case of image data, the feed-forward network is a CNN. Convolutional layers, like the ones discussed in 2.1.4, down sample the data, whilst transposed convolutional layers (sometimes wrongly referred to as deconvolutional layers) up sample the data in an attempt to recover the input. Often autoencoders are discussed as if they were two models – the down sampling layers between the input and the latent vector being the 'encoder' and the up sampling layers between the latent vector and the output being the 'decoder'. Together they can be depicted with an hourglass shaped architecture, like the one shown in Figure 2.3. Note that if the encoder and decoder had only one layer and the non-linear activation functions were removed, the autoencoder would fulfil the same role as principal component analysis (PCA), although without the orthogonality constraints.
Given that the task is to reconstruct the input, the optimisation problem can be formulated as if it were a supervised learning problem, using the reconstruction as the prediction $\hat{x}$ and

the input data x as its own ground truth label. For this reason, a more accurate description of the optimisation process would be *self*-supervised. The reconstruction error $\mathscr{L}(x,\hat{x})$ can then be something like MSE using the pixel-wise differences, and the algorithm can utilise SGD to find the minimum. Once trained, the decoder can be discarded and the trained encoder maintained for use as a feature extractor. Trained encoders are often useful in the preprocessing stage of a wider framework, taking raw data as input and outputting a low-dimensional latent vector to serve as input to another model.

2.2.2 Variational Autoencoder

In a revised form called a variational autoencoder (VAE), autoencoders have also made a big contribution to the generative models family of DNNs. After training, rather than discard the decoder and utilise the encoder, the reverse can be done, discarding the encoder and utilising the decoder for content generation, much like the generator model in a generative adversarial network (GAN). The idea is that, if we simply sample a latent vector from the latent space rather than map an input image to the latent space, then the decoder will produce an image that belongs to the same space as the input data but is 'new' as opposed to a reconstruction. In practice, decoding a randomly sampled latent vector will likely produce a meaningless output unless the latent space is well organised. Not only should all points in the latent space produce meaningful decodings (completeness), the learnt space should also have continuity, i.e. two points close together should produce similar content once decoded. Since the organisation of the latent space depends on both the distribution of the data in the original space, the dimensionality of the latent space (hyperparameter), *and* the architecture of the encoder, it is easier to control the adoption of good latent space properties with explicit regularisation.

In a vanilla autoencoder, the encoder outputs a vector of n feature values. In a VAE, the $1 \times n$ encoder output is doubled to $1 \times 2n$, with the first n values representing means μ and the second n values representing standard deviations σ of n univariate normal distributions (or together, the covariance matrix of one multivariate normal distribution). The feature value for each dimension of the latent space is then sampled from the corresponding univariate distribution for that dimension. As such, if x is our image sample, y is our encoder output and z is our latent vector, then

$$(z_1,...,z_n) \sim \mathbb{N}_n(\mu,\Sigma)$$
$$where\ \ \mu = (y_1,...,y_n), \mathrm{Cov}[z_i,z_j] = 0$$
$$and\ \ \mathrm{Cov}[z_i,z_i] = \sigma_i^2 = y_{i+n}^2\ \ for\ \ i,j\ \ in\ \ 1,...,n$$

The change from feature values to feature distributions allows for a second term in the VAE objective function, producing

$$\mathscr{L}(x,\hat{x}) = ||x-\hat{x}||^2 + \beta KL[\mathbb{N}(\mu_x,\Sigma_x),\mathbb{N}(0,I)]$$

where x is the input image, $\hat{x}$ is the image reconstruction, KL is short for Kullback-Leibler divergence, and β is a hyperparameter which determines the weighting applied to the second objective. Specifically, the added regularisation term is the Kullback-Leibler (KL) divergence between the multivariate Gaussian described by the encoder output and a standard multivariate Gaussian with mean zero and covariance matrix I (identity matrix). The KL divergence between two probability distributions can be thought of as a distance metric, and so the model is incentivised to make the encoded distributions 'close to' a standard multivariate Gaussian (i.e. to produce a mean close to zero and a covariance matrix close to I). Distance is not entirely the correct term since the metric is not commutative. When comparing two distributions $P(x)$ and $Q(x)$, KL divergence represents the average predictive power of a sample from $P(x)$ when trying to distinguish $P(x)$ from $Q(x)$. The higher this value, the more dissimilar the two functions i.e. the greater the predictive power of an average individual sample, the stronger evidence it provides that $Q(x)$ is not $P(x)$. This concept of predictive power is why the KL divergence is sometimes referred to as the relative entropy.

Penalising distance from a common target distribution encourages encodings to overlap, helping toward completeness and continuity. However, as is the case with regularisation, by forcibly preventing the encoder from overfitting, reconstruction loss is increased. The two objectives create a trade-off, and therefore the multiplier β is included as a hyperparameter to control the weighting of the second term. This fairly intuitive objective function can also be derived using Bayes theorem and the statistical method of variational inference, hence the name variational autoencoder. This derivation is well beyond the scope of this chapter but the interested reader is referred to Kingma and Welling (2019) and other works by these founding authors for a theoretical deep dive, or to Zhai et al. (2018) for derivations of each of the variants of the autoencoder across its lineage.

2.3 Reinforcement Learning

Reinforcement learning is one of three paradigms in machine learning, sitting somewhere in the middle of the supervised and unsupervised paradigms. In supervised learning, data is labelled as input-output pairs, providing a ground truth for any given input sample. Conversely, in unsupervised learning the data is unlabelled and the model is left to find patterns and hidden structure in the data. In reinforcement learning (RL), the learning process could be described as semi-supervised, with the algorithm receiving partial feedback in the form of a numeric reward signal. Both positive and negative values can be issued, with a value further toward positive infinity signalling that a behaviour, or a situation, is considered 'better' than if a value further toward negative infinity had been returned. Borrowing from B.F. Skinner's operant conditioning methodology in Psychology (Skinner, 1988), the idea is that the decision or behaviour will be strengthened or suppressed in accordance with this feedback. Informally, this makes RL somewhat akin to teaching a dog to sit, or to a parent using a gold star chart to discourage tantrums and encourage chores.

Perhaps attributable to this parallel with animalistic learning, the language of RL is very much personified, referring to the model or algorithm as an agent, actively exploring and interacting with an environment. At each time step, the agent observes the state of the environment and chooses an action based on this observation. The action is then implemented, pushing the environment into a next state according to a state transition probability matrix unknown to the agent. The next state is observed by the agent and the cycle continues (see figure 2.4). The agent's interaction with the environment at each time step (state, action, next state) is passed through the reward function and the returned value becomes a sort of label for the data trace and the basis for any learning. As such, the reward function is an important design feature and an expression of what is considered good, bad, better or worse. It can be as dense or as sparse as the designer deems suitable, with anything from an informative value at every time step to a reward of zero for every state but the the terminal state, for example, if the agent were to win in a game of chess. In this way, RL provides a framework for training a machine to perform a sequential decision making task without necessarily knowing, or wanting to expose, what the 'correct' or best decision is at any given moment. Chess is a perfect example here – not even a chess grand master would know definitively how to label every single board configuration with the ground truth best move, but most people would be able to congratulate the winner.

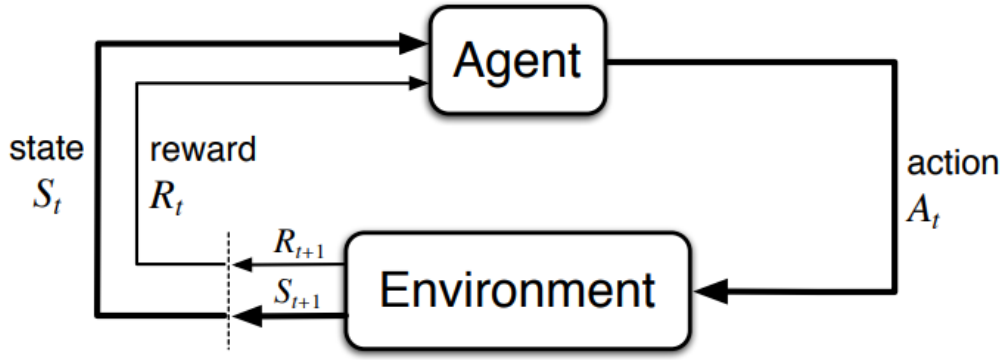


Fig. 2.4 The agent-environment interaction cycle, from Sutton and Barto (2018).

2.3.1 Terminology

The reinforcement learning paradigm introduces a lot of new terms. Before describing any RL algorithms, or concepts behind the alogirthms, it is worthwhile providing clear definitions for the most important terms.

State (s) A complete description of the environment at a given time step. If a task has a terminal state it is considered episodic. If there is no terminal state it is considered continuous. All RL work in this thesis is in the context of episodic tasks. The full set of possible states in an environment is called the state space, expressed S. The state at time step t is expressed s_t.

Observation (o) What information the agent receives on the state of the environment. If any information is omitted the observation is considered a partial observation.

Note: Many papers and textbooks use s where technically it ought to be o. Following this norm, s will be used throughout notation, but we acknowledge that there is a distinction here. Also note that the next state is expressed interchangeably as s_{t+1} and s'.

Action (a) The decision selected by the agent. The full set of actions available in the environment is called the action space, expressed A. There are two types of action space: continuous (real-valued vectors) and discrete (finite number of options). The full set of actions available in a particular state is expressed $A(s)$, the action taken at time step t is expressed a_t, and the next action is expressed interchangably as a_{t+1} or a'.

Reward (r) The numeric token issued by the programmed reward function R, taking state and action as arguments. The reward function can be thought of as part of the environment,

with rewards being issued to the agent following an action (as seen in Figure 2.4). Note that this immediate reward is expressed interchangeable as r_t, given s_{t-1} and a_{t-1}, or r_{t+1}, given s_t and a_t. This is simply a notation choice as the two are logically the same. The r_{t+1} notation will be used throughout this thesis.

Policy (π) A table or function mapping observations to actions i.e. the agent's rulebook or strategy. A policy can be deterministic (a direct mapping from observation to action, sometimes denoted with μ instead of π) or stochastic (observations map to an action distribution, in the case of continuous action spaces, or to action probabilities in the case of discrete). Further details on this and action sampling are reserved for section 2.4 since there are differences in implementation across algorithms.

Trajectory (τ) A sequence of observed states and taken actions, sometimes called environment rollouts.

Return ($R(\tau)$) Total cumulative reward for a trajectory. The return can be described as having a finite horizon, i.e. τ has a finite number of time steps T, or it can be described as having an infinite horizon, i.e. τ is infinitely long. Additionally, return can be undiscounted (rewards simply summed) or discounted (reward at each time step is first multiplied by a discount factor γ which decays throughout the time series). Discounted return means that a reward received k time steps in the future is worth γ^{k-1} times less than an immediate reward. Using a discount factor not only encapsulates the idea that reward now is preferable to reward received later (because of the increasing uncertainty around future events), it also makes infinite horizons mathematically tractable. In the context of episodic tasks, like the ones discussed in this thesis, return has a finite horizon and discounting is not essential, however, RL literature and formalism tend to use a unified notation. It is straightforward to apply **infinite-horizon discounted return** to both continuous and episodic tasks by making the terminal state in episodic tasks a special absorbing state which transitions to itself in a continuous loop but with reward zero.

Note: The term 'return' and the term 'reward-to-go' are often used interchangeably but they are not the same. Strictly, return $R(\tau)$ is the sum of rewards across the entire trajectory, whereas reward-to-go G_t is the sum of rewards from the current time step onward, i.e. from r_{t+1} onward. Mostly, we are interested in the latter. In the case where 'return' is used to mean sum of rewards from the current point onward, the assumption is that s_t is being treat as s_0 of the trajectory.

Policy-based vs value-based In policy-based RL (often called policy optimisation), the policy is represented explicitly as $\pi(a|s)$ and is optimised directly. In value-based methods (often called Q-learning), the policy is represented implicitly in an approximation of the optimal action value function Q^*. Here, actions are selected by maximising over value estimates. This of course is still policy optimisation albeit indirect and is inherently less stable for being indirect. It does however allow for learning to take place off-policy.

Off-policy vs on-policy Off-policy means updates can utilise data collected at any point and with any policy or policy version. This data re-use makes off-policy learning more sample efficient than on-policy learning. With on-policy learning, incoming data is used and then thrown away, meaning all updates are based on data collected by the most up-to-date version of the policy.

2.3.2 Formalism

Formally, the environment in RL is a Markov Decision Process (MDP) described by the tuple $< S,A,R,P,p_0 >$, where S is the state space, A is the action space, R is the reward function, P is the transition probability function and p_0 is the starting state distribution. This process describing the environment satisfies the Markov property – the future is independent of the past given the present. What this means is that transitions are conditioned only on the most recent state and action, since the most recent state encapsulates any history. With this property in mind, below are some key formulas that will help with understanding when discussing algorithms in greater detail. In the case of deterministic policies,

$$a_t = \mu(s_t)$$

whereas in the case of stochastic policies,

$$a_t \sim \pi(.|s_t)$$

The probability of a trajectory with T time steps is

$$P(\tau|\pi) = \rho_0(s_0)\prod_{t=0}^{T} P(s_{t+1}|s_t,a_t)\pi(a_t|s_t)$$

that is, the probability of the initial state multiplied by the product of probabilities for each state, action, next state tuple in the trajectory. In turn, this is the probability of taking the action multiplied by the state transition probability.

Expected return is then

$$\mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] = \int_{\tau} P(\tau)R(\tau)$$
$$where\ R(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Finally, an explanation of the log-derivative trick is given in preparation for later derivations. Consider X is a function $g(x)$ such as $\pi(a|s)$ or $P(\tau|\pi)$, making $logX$ the composite function $f(g(x))$. The chain rule specifies that $\nabla f(g(x)) = g'(f(x)) \cdot f'(x)$, and since the derivative of $logx$ with respect to x is $1/x$, $f'(g(x))$ here becomes $1/X$. If we leave $g'(x)$ written as ∇X, $\nabla logX = \nabla X/X$. By multiplying $\nabla logX$ by X we are saying that X multiplied by $\nabla X/X$ is equal to ∇X, which is true given that we are multiplying and dividing by the same term.

The result is

$$\nabla X = X\nabla logX.$$

2.3.3 Value functions

The next crucial concept in RL is that of assigning value to states and actions. Intuitively, 'value' indicates to the agent how good the state or action is, given the agent's goal. Mathematically, the value of a state $v^{\pi}(s)$, for example, is the expected return for a trajectory starting in state s and following policy π thereafter. In value-based RL, the agent is tasked with estimating the value of states it has visited, and actions it has taken in visited states, and can improve upon these estimates with increasing experience. Estimates are 'held' in a value function, e.g. the state value function $V(s)$. Value functions are so integral to RL, it is worth covering not only the equations but some of their properties and the relationships between them.

The state value function can be expressed

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\pi}[G_t|s_t = s]$$

and the optimal value function holds the value of each state under the optimal policy π^* i.e. the policy which maximises the RHS

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

An important property of the value function is that it obeys a special self-consistency equation called the Bellman equation. Put simply, the value function is recursive in nature, where the value of any state is equal to the reward of being there plus the value of where you land next. Mathematically, the value of state s_t is equal to the immediate reward r_{t+1} plus the value of the next state s_{t+1}, where the immediate reward is governed by $R(s_t, a_t)$ and a_t is governed by $\pi(a_t|s_t)$. Below is a derivation that encapsulates all of this.

$$\begin{aligned}
V^{\pi}(s) &= \mathop{\mathbb{E}}_{\pi}[G_t|s_t = s] && (2.1)\\
&= \mathop{\mathbb{E}}_{\pi}[r_{t+1} + \gamma G_{t+1}|s_t = s] && (2.2)\\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathop{\mathbb{E}}_{\pi}[G_{t+1}|s_{t+1} = s']] && (2.3)\\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)[r + \gamma V^{\pi}(s')] && (2.4)\\
\therefore V^*(s) &= \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma V^*(s')] && (2.5)
\end{aligned}$$

The first step (2.2) is to pull the r_{t+1} term out from G_t, which is the sum of all reward values from $t+1$ onward. The next step (2.3) is to rewrite the expectation of r_{t+1} as $\sum_{r \in R} r p(r|s)$, a weighted average of its possible numeric outcomes, where the weights are the probabilities of the outcome occurring. However, the $p(r|s)$ distribution is a marginal distribution that also contains the variables a and s', where $p(r|s) = \sum_{s' \in S} \sum_{a \in A} p(s', a, r|s) = \sum_{s' \in S} \sum_{a \in A} p(a|s) p(s', r|s)$. Note that in 2.3 (in line withSutton and Barto (2018)) the set notation has been dropped from the sum subscript for readability and $p(a|s)$ is written as $\pi(a|s)$. In the next step (2.4), since the value of s is equal to the expectation of G_t (2.1), the expectation of G_{t+1} can be replaced with the value of s'. Finally, if $V*$ holds the value of each state under the optimal policy, then it is not necessary to consider the full set of $a \in A$

when expanding the expectation of r_{t+1}, since the optimal policy will always select the action with the highest value.

Very similar to the value function is the action value function or state-action value function, denoted Q. The subtle difference is that $Q(s,a)$ takes a state-action *pair* and gives the expected return for a trajectory starting in state s, taking action a (which may or may not have came from the policy) and *then* following policy π thereafter.

The action value function can be expressed

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\pi}[G_t|s_t = s, a_t = a]$$

and the optimal action value function is

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

Similarly, the derivation for the Bellman equation and Bellman optimality equation is

$$\begin{aligned}
Q^{\pi}(s,a) &= \mathop{\mathbb{E}}_{\pi}[G_t|s_t = s, a_t = a] \\
&= \mathop{\mathbb{E}}_{\pi}[r_{t+1} + \gamma G_{t+1}|s_t = s, a_t = a] \\
&= \sum_{s'}\sum_{r} p(s', r|s, a)[r + \gamma \mathop{\mathbb{E}}_{\pi}[G_{t+1}|s_{t+1} = s']] \\
&= \sum_{s',r} p(s', r|s, a)[r + \gamma V^{\pi}(s')] \\
\therefore Q^{*}(s,a) &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} Q^{*}(s', a')]
\end{aligned}$$

The steps here are the same as the derivation of V except a is known and so there is no sum over $a \in A$. There are relationships between V and Q which are useful to note, e.g.

$$
\begin{aligned}
V^{\pi}(s) &= \mathop{\mathbb{E}}_{a\sim\pi}[Q^{\pi}(s,a)] \\
V^{*}(s) &= \max_{a} Q^{*}(s,a) \\
Q^{*}(s,a) &= \mathbb{E}[r_{t+1} + \gamma V^{*}(s_{t+1}) | s_t = s, a_t = a]
\end{aligned} \tag{2.6}
$$

Finally, there is a combination of V and Q referred to as the advantage function and denoted A. Intuitively, the advantage function gives a sense of how much better an action is compared to the other choices for a given situation on average. It is therefore the relative value of a state-action pair as opposed to absolute value. Mathematically, it is expressed as

$$
A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.7}
$$

2.3.4 Monte Carlo and Temporal Difference learning

Bootstrapping is the method of updating an estimate using a previous estimate, and it is very widely used in RL and DRL algorithms. Therefore, to provide some reassurance on the validity of this method, the concept is explained here, but in the context of shallow, tabular RL for greater clarity. Specifically, this section provides a brief introduction to the Monte Carlo method and the Temporal Difference learning method TD(0). The difference between these two families stems from how the algorithm computes the expectation of reward-to-go on which value estimates are based. Importantly, the two are opposite ends of a scale describing the degree to which they bootstrap. As mentioned, the assumption here is that all tasks being learnt are episodic i.e. at some point the agent fails or succeeds in some way, wins or loses the game, violates a condition, or the session times out.

A fairly obvious way of computing an expectation is to use the mean. If the agent interacts with the environment until the end of an episode, makes a record of the reward-to-go for each state (or state-action pair), and repeats N times, the mean of these samples will tend toward true V (or true Q) as N tends toward infinity. Of course, it is possible to calculate this mean incrementally rather than wait until N episodes have been played out. To do so, the current mean (and therefore most up-to-date estimate of V or Q) is subtracted from the new sample of G_t. The result is divided by N and then appended to (i.e. summed with) the current estimate. The multiplication by $1/N$ updates the currently held mean to the new mean, having added the new sample to the population. It is also possible to replace $1/N$ with

a step size multiplier – a value less than one often denoted by α and interchangeably referred to as the learning rate. What the step size multiplier does is to move the current estimate *some way* toward the new mean. Following this logic, the update rule for an estimate such as $V^{\pi}(s_t)$ is

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha[G_t - V^{\pi}(s_t)]$$

The update rule for an estimate of Q would look the same, substituting $V^{\pi}(s_t)$ for $Q^{\pi}(s_t, a_t)$. In fact, whilst the terms may change, this structure of **updating** an estimate a **little** bit in the **direction of the error** will crop up time and time again. Updating an estimate incrementally like this facilitates a framework for learning called general policy iteration (GPI) – an iterative two-stage process that alternates between evaluation (using data to refine value estimates) and policy improvement (using information on state and state-action pair values to adjust the policy). "Each creates a moving target for the other, but together they cause both policy and value function to approach optimality" Sutton and Barto (2018).

Albeit incremental, estimate updates in Monte Carlo learning are reserved until the end of an episode. TD(0) takes the incremental approach to the extreme, updating the estimate for V or Q after each time step as opposed to each episode. Rather than updating the estimate with experienced reward-to-go (G_t), TD(0) updates the estimate for V or Q with an *estimate* of the reward-to-go. An estimate of reward-to-go is exactly what V and Q provide, and this is where the bootstrapping comes in – updating an estimate with a previous estimate. The update rule is as follows

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha[r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)] \tag{2.8}$$

The format is the same as before except G_t is replaced with $r_t + \gamma V^{\pi}(s_{t+1})$, what is called the TD target. Again, any terms involving some form of $V(s)$ can be replaced with $Q(s,a)$ to get the update rule for Q. Note the connection to the Bellman equation. An expansion of G_t is $r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...$, and since V and Q are recursive, everything following r_{t+1} can be substituted with the value of the next state or state-action pair. That is, G_t and $r_{t+1} + \gamma V^{\pi}(s_{t+1})$ are in fact equivelant when V is true V. The difference, of course, is that V in the above update rule is only an estimate of V and yet, over time, the highly bootstrapped estimate is still able to converge, since the TD target is one step of experience more informed than the previous estimate. There is a trade-off here between bias and variance. G_t is an

unbiased estimate of value whereas the TD target is biased. However, TD is lower variance because G_t is based on the actual trajectory of the agent, which includes noise all the way through, whereas the TD target only incurs noise over one step. The lower variance means TD is more efficient, however the bootstrapping means TD is more sensitive to initial values and, in the case of function approximators, is not guaranteed to converge.

These update rules are just a starting point for shallow RL. The method of TD(0) which involves one step of experience prior to update expands to n-steps, filling the gap between TD(0) and the Monte Carlo method. Should the reader be interested, there are additional methods such as TD(λ), additional concepts such as eligibility traces and importance sampling, and named algorithms such as SARSA and Q-learning. Given that the work presented in this thesis uses *deep* reinforcement learning, details are reserved for the next section on DRL algorithms.

The transition from RL to DRL revolves around using function approximation in place of lookup tables for the policy and value function(s). Specifically, it involves using parameterised DNNs, denoting the network parameters (connecting weights) typically with θ or ϕ. As such, $V^{\pi}(s)$ becomes $V^{\pi}_{\theta}(s)$ or $V^{\pi_\theta}(s)$ or, with a slight abuse of notation, the π is dropped to give $V_\theta(s)$. Using functions in place of tables becomes important for complex tasks where the state space is too large to hold in memory. Functions also carry the benefit of interpolation, providing values for unvisited states.

The material covered in this section summarises the seminal textbook *Reinforcement Learning: An Introduction* by Sutton and Barto (2018). Learning about this material was also greatly aided by the video lecture series 'RL Course by David Silver' (2015), available on YouTube. Silver is a professor at City University London and principal researcher at DeepMind, leading the AlphaGo, AlphaZero and AlphaStar projects famously known for being the first programs to beat the world champions of Go, Chess, and StarCraft.

2.4 Deep RL Algorithms

In tabular RL, value functions are lookup tables and estimates are updated directly. In DRL, value functions are neural networks and estimates are updated indirectly by updating network parameters. Figure 2.5 provides a taxonomy illustration for some of the most seminal DRL algorithms. At a high level, DRL algorithms can be categorised as either model-based or model-free. If an agent has access to a model of its environment and is therefore privy to state

transitions and reward allocation, decision making can be based on forward planning. Rather than take a particular action and observe the consequences, the agent can consult the model and consider what would happen for a range of possible choices. This process of simulating environment rollouts is referred to as introspection and is clearly more sample efficient (fewer environment interactions required) than the trial and error approach of model-free methods. However, ground truth models of the environment are very rarely available and building a model of the environment requires experience, circling back to the necessity to collect samples. Model-free methods circumvent the need to learn a model of the environment and instead focus directly on learning what behaviour maximises return. They are also easier to implement and tune and have received the most attention in the field. Given this, model-free DRL was selected for the work in this thesis, and will be the focus of all algorithm discussions from hereafter.

Learning what behaviour maximises return, that is, closing the gap between an agent's current policy and the optimal policy for a given environment, can largely be achieved in one of two ways. In policy optimisation methods, the policy is represented explicitly as the function $\pi_\theta(a|s)$ and is updated toward the optimal policy by gradient ascent on a measure of policy performance $J(\pi_\theta)$. In Q-learning methods, the policy is represented implicitly in an approximation of the optimal action value function Q^*, written $Q_\theta(s,a)$. Here, actions are selected by maximising over value estimates and estimates are improved by gradient descent on a measure of error between 'true' Q^* and the approximation. Although Q-learning and policy optimisation form two categories of algorithm, they are not all that distinct, with significant crossover in implementation (for example, using an approximator of Q within $J(\pi_\theta)$). Hybrid algorithms have also emerged in order to best exploit the various trade-offs. One example of this is Soft Actor Critic, the algorithm selected in this thesis for reasons to be discussed in the dedicated Soft Actor Critic (SAC) section below.

Before detailing SAC, a handful of earlier algorithms are discussed. This is done in order to build up to a more accessible explanation of SAC, as well as providing a more detailed compare and contrast. Vanilla Policy Gradient and Deep Q-Network are presented as the most prominent examples of policy optimisation and Q-learning respectively. Deep Deterministic Policy Gradient (DDPG) is then presented as the first algorithm to appear in the hybrid space. Finally, Twin Delayed Deep Deterministic Policy Gradient (TD3) and SAC are discussed as two alternative improvements on DDPG.

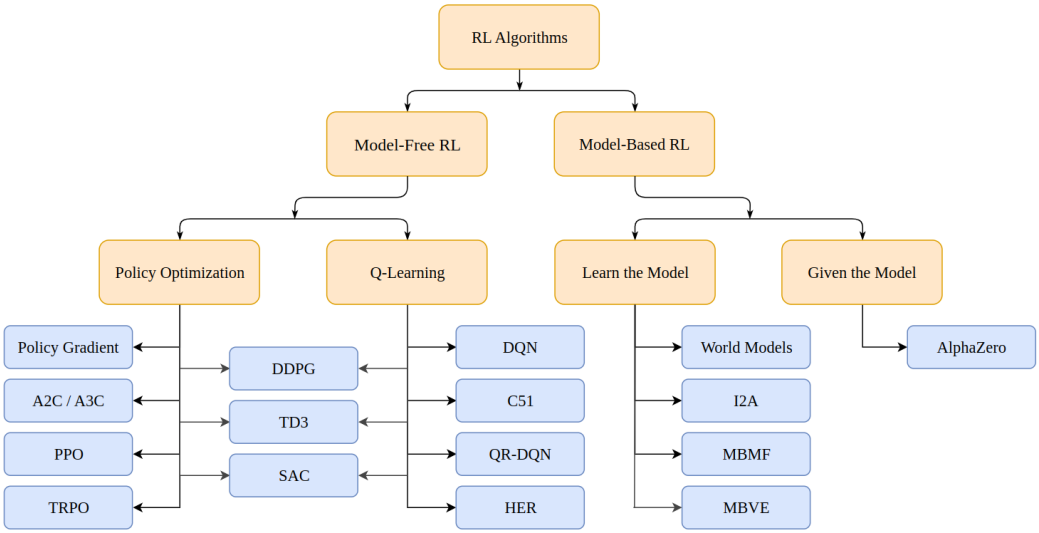


Fig. 2.5 Taxonomy of seminal DRL algorithms, from 'Spinning Up in Deep RL' (2018).

2.4.1 Deep Q-Network (DQN)

Q-learning methods like Deep Q-Networks (DQN) approximate the optimal action value function Q^*, that is, the maximum return one could expect from the given state-action pairing across all policy options (or the state-action value when acting according to the optimal policy π^*). If ground truth values were available, then learning the parameterised approximation Q_θ could be approached with a least squares algorithm, minimising the mean squared error between the approximation and ground truth. Since RL is not supervised and ground truth Q values are not available, DQN replaces true Q^* with a 'target' (y_i in equation 2.9) i.e. estimate of the ground truth. This approach to the least squares algorithm mirrors the approach introduced in Section 2.3.4 on Monte Carlo and Temporal Difference learning. As before, the value estimate can take many forms, including G_t or a bootstrapped TD target. The TD target introduced in equation 2.8 was with respect to estimating V. Since DQN estimates Q^*, the TD target is $r + \gamma max_{a'} Q_\theta(s', a')$, matching the Bellman optimality equation for Q (equation 2.6).

In order to find the minimum of this objective $\mathscr{L}_i(\theta_i)$, the gradient is taken, as shown in equation 2.10. Had the y_i term been the oracle Q^* or the target G_t, then it would be clear why the gradient of this term is not taken, since it does not depend on the parameters θ. Although the target $r + \gamma max_{a'} Q_\theta(s', a')$ does depend on θ, in practice the gradient of this term is still not taken, so as not to reverse the flow of time (the mathematics becomes problematic when trying to update an estimate of what is going to happen toward what has already happened). The expectation is removed by sampling the gradient at each time step and descending the

gradient a small amount each time i.e. SGD. As shown in the update rule (equation 2.11), at each time step the parameters are adjusted by an amount proportional to the learning rate α, in the direction of negative gradient. In principle this method should converge on or be close to the global minimum, meaning that the approximator Q_θ has been fit to the true optimal action value function Q^*. The policy is then implicit in Q, i.e. it is the greedy strategy $a = \underset{a}{max} Q_\theta(s,a)$.

$$\mathscr{L}_i(\theta_i) = \mathop{\mathbb{E}}_{s,a,r,s' \sim U(D)} [(y_i - Q_{\theta_i}(s,a))^2] \tag{2.9}$$

$$\nabla_{\theta_i} \mathscr{L}_i(\theta_i) = \mathop{\mathbb{E}}_{s,a,r,s' \sim U(D)} [(r + \gamma \underset{a'}{max}\, Q_{\bar{\theta}}(s',a') - Q_{\theta_i}(s,a) \nabla_{\theta_i} Q_{\theta_i}(s,a)] \tag{2.10}$$

$$\theta_{i+1} = \theta_i + \alpha \nabla_{\theta_i} \mathscr{L}_i(\theta_i) \tag{2.11}$$

The implementation of DQN presented in 2013 by Mnih et al. (and presented again in Nature in 2015 by Mnih et al.) includes two design choices that have a stabilising effect on learning:

1. Experience replay – The agent's experience at each time step (s_t, a_t, r_t, s_{t+1}) is stored in a dataset D called the replay memory. When applying the update rule, a minibatch of these traces is sampled at random from D (hence the $s,a,r,s' \sim U(D)$ in equation 2.9 and equation 2.10, where U stands for uniform). This has a stabilising effect because randomising the samples used in an update decorrelates the data, reducing the variance of the update.

2. Target network – Updating Q_θ toward values generated with the same iteratively updated Q_θ creates somewhat of a moving target situation. To reduce correlations between the approximator and the target, a separate Q-network is used for the generation of the target value (target network), the parameters of which, $\bar{\theta}$, are only updated periodically, by cloning the iteratively updated Q-network every C steps. Freezing the parameters of the target network for a chosen number of steps helps with stability, making divergence and oscillations less likely.

Using experience replay makes DQN an off-policy algorithm because the update uses data from D which may have been collected some time ago according to different parameter values. Off-policy learning works in this setting because Q^* should satisfy the Bellman equation for *any* possible transition and so the circumstances under which the transition was sampled are not important when trying to approximate Q^* with mean-squared Bellman error (MSBE) minimisation. Off-policy learning has the benefit of being more data efficient, since transitions can be used for updates again and again. An additional benefit is that, with

on-policy learning, the policy is always updated according to data collected by the most current version of the policy, introducing the possibility of a downward spiral (a bad policy leading to poor data, leading to an even worse policy). Learning can oscillate, blow up (diverge instead of converge) or get stuck in a poor local minimum. The uniform sampling used in experience replay can help mitigate this. Collecting data off-policy also provides an opportunity to bake in a degree of exploration to the data collecting policy (behavioural policy) whilst keeping the target policy (the policy being optimised for use at inference time) fully exploitative. Both policies can be implicit in the same Q-function. At inference time the policy is said to be greedy, with actions always selected using a max operator over state-action values. At training time the policy is said to be ε-greedy, with actions selected in that same greedy manner with probability $1-\varepsilon$, but otherwise selected at random. Selecting actions at random a proportion of the time helps to expand knowledge and map uncharted territory, potentially leading to the discovery of higher value action selections than the current best strategy.

DQN is the only example from the Q-learning section of the taxonomy discussed in detail. The next algorithms presented fall under the policy optimisation branch, starting with the vanilla form and adding various extensions throughout the algorithms that follow. In DQN, the selection of a using the max operator is computationally expensive, making it undesirable for use with large or continuous action spaces, as well as limiting the algorithm to the generation of deterministic policies. Policy optimisation algorithms, on the other hand, are capable of learning stochastic policies given that they directly model the policy with action probabilities. Also, with DQN, a small change in the value estimate can affect whether or not the action is selected and this discontinuity can be an obstacle to convergence. With policy optimisation, changes to the policy are much smoother, with only small changes to the policy parameters and additional constructs such as trust regions. This being said, value-based methods like DQN, in the right setting, are very efficient, with the max operator being a very aggressive way of pushing the policy toward an optimal policy.

2.4.2 Vanilla Policy Gradient (VPG)

Policy optimisation methods like Vanilla Policy Gradient (VPG) parameterise the policy function so that it can be updated directly as opposed to being updated as an outcome of updating a value function approximator. Whereas with value function approximators the objective function was a loss function to be minimised, with policy optimisation the objective function is a measure of performance and therefore something to maximise. This performance-measuring objective function $J(\pi_\theta)$ is simply the expected return $R(\tau)$, experi-

enced as a result of following the policy. The gradient, $\nabla_\theta J(\pi_\theta)$, is referred to as the **policy gradient**, hence the term policy gradient methods.

The derivation for the policy gradient is a useful derivation to walk through given that it forms the basis of so many algorithms. If we consider a sampled trajectory τ as a random variable X with a continuous distribution and a probability density function $P(\tau|\theta)$, then the expectation of $R(\tau)$ can be expanded by integrating $g(x)$ (the reward function R) multiplied by $f(x)$ (the probability density function P) for all values of X. The gradient of our performance objective $J(\pi_\theta)$ is therefore equivalent to the gradient of $\mathbb{E}[R(\tau)]$ and therefore equivalent to the gradient of $\int_\tau P(\tau|\theta)R(\tau)$. The gradient can be brought under the integral since integration is with respect to τ and the gradient is with respect to θ.

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\ &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau)\end{aligned}$$

Using the log-derivative trick described in Section 2.3.2, the gradient of the trajectory probability multiplied by the return can be transformed into the trajectory probability multiplied by the gradient of the **log** of the trajectory probability, multiplied by the return. The integral and the trajectory probability multiplier can then be taken back out, returning the expression to expectation form.

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\ &= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\ &= \int_\tau P(\tau|\theta)\nabla_\theta log P(\tau|\theta)R(\tau) \\ &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta log P(\tau|\theta)R(\tau)]\end{aligned}$$

If we sum over timesteps, $P(\tau|\theta)$ can be substituted for $\pi_\theta(a_t|s_t)$, the joint probability of choosing action a given state s and parameters θ. This is because, if we refer back to the definition for $P(\tau|\pi)$ in the formalism section (Section 2.3.2), $\pi(a_t|s_t)$ is the only term which

depends on θ, and the product can be replaced with a sum given that the log is taken. Finally, the expectation can be estimated with a sample mean, by summing over trajectories in a set of trajectories D and dividing by $|D|$.

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] \\
&= \nabla_\theta \int_\tau P(\tau|\theta)R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta)R(\tau) \\
&= \int_\tau P(\tau|\theta)\nabla_\theta log P(\tau|\theta)R(\tau) \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\nabla_\theta log P(\tau|\theta)R(\tau)] \\
\therefore \nabla_\theta J(\pi_\theta) &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)R(\tau)] \qquad (2.12) \\
\therefore \hat{g} &= 1/|D| \sum_{\tau\in D}\sum_{t=0}^{T} \nabla_\theta log\pi_\theta(a_t|s_t)R(\tau) \qquad (2.13)
\end{aligned}$$

The result is an expression that can be numerically computed from a finite number of environment interactions. The $log\pi_\theta(a|s)$ term seen in equation 2.12 and equation 2.13 is sometimes referred to as the 'log-prob', with the gradient of this term referred to as the 'grad-log-prob' or the 'score function'. The grad-log-prob was introduced in Section 2.1.1, when discussing the objective function in the context of density estimation. To recap, the score function describes the direction in which to change parameters in order to find the maximum likelihood estimator, that is, the parameterised model $P_\theta(X)$ that most resembles the true probability distribution $P(X)$ and so maximises the likelihood of data X being generated. As a concave function, the parameter values that give the maximum likelihood estimator are the θ values associated with the apex of this function, where the gradient is zero, meaning there is no way to make data X more likely.

In supervised learning, data X is a fixed data distribution providing a static target for optimisation, but in reinforcement learning there is no fixed dataset. In policy optimisation, $P_\theta(X)$ is the policy approximation, $P(X)$ is the optimal policy, and data X is the conditional distribution of optimal actions generated by following the optimal policy. Not only is X unknown, but the data being generated by the approximator π_θ is dependent on the parameters being optimised, making it quite unlike a score function in the traditional sense. Even so, **when evaluated at the current parameters, with data generated by the current parameters**

(hence the need for on-policy learning), $\nabla_\theta log\pi_\theta(a|s)$ can instruct which direction to update the parameters to increase the likelihood of that same data being observed again i.e. that same action being selected again. Put simply, it tells us how to get more of something. By taking a step in the direction of this gradient multiplied by the return, the probability of actions resulting in positive return is increased whilst the probability of actions resulting in negative return is decreased, relative to the return magnitude.

This is intuitively pleasing although by using the multiplier $R(\tau)$ (the sum of rewards obtained across the full trajectory thus far), actions a_t are being evaluated based on rewards obtained prior to time t. What would make more sense is to evaluate the decision based on rewards obtained after the decision is taken, i.e. the reward-to-go G_t. VPG, using G_t as the multiplier, is better known as the 'REINFORCE' algorithm, proposed by Williams in 1992. In practice this algorithm is rarely used because, as with all Monte-Carlo methods, it is low bias but high variance. Policy evaluation based on rewards actually obtained avoids any bias given that no estimation is involved, but since rewards obtained in one trajectory could vary wildly from the next, it introduces a lot of noise into the policy gradient estimate that scales with trajectory length, slowing convergence.

The variance can be reduced by subtracting some form of 'baseline' value. A common choice for b is the state value function, or, since the true state value function is unknown, an approximation $V_\phi(s_t)$. Mathematically this is valid because any function b that only depends on state s can be added or subtracted from the expression for the policy gradient without changing it in expectation, since $\mathbb{E}_{a_t\sim\pi_\theta}[\nabla_\theta log\pi_\theta(a_t|s_t)b(s_t)] = 0$. Subtracting $V_\phi(s_t)$ from G_t alleviates some of the noise in G_t by reducing it to a measure of return *relative* to the return expressed by V, which is an average measure of return. Clearly, this relative value will fluctuate less than an absolute value of return and therefore variance is reduced, making it a more conservative and stable approach.

As has already been seen, another way to reduce variance is to *replace* G_t with a lower variance estimate of return, perhaps $Q(s,a)$. If both approaches are used, i.e. the baseline $V(s)$ is subtracted from the return estimate $Q(s,a)$, then the score function becomes an estimate of the advantage A, described in equation 2.7. Updating the policy in favour of decisions that panned out better than expected, away from decisions that panned out worse than expected, and remaining neutral to decisions that cultivated an expected level of return is, again, intuitively pleasing. In a method called general advantage estimation (GAE) proposed by Schulman et al. (2015), G_t is replaced with the TD target $r_{t+1} + \gamma V(s_{t+1})$, before subtract-

ing the baseline $V(s_t)$. This version of advantage estimation reduces the number of required networks from three to two, requiring an approximation of V but not Q. Any algorithm that combines policy gradient estimation with value function approximation (whether it be V, Q or both) belongs to a family of algorithms called 'actor-critic' algorithms, discussed as early as 1999 by Sutton et al.. The policy network is referred to as the 'actor', updating the policy parameters θ in the direction stipulated by the 'critic' (some value function approximation), parameterised by ϕ and updated concurrently.

To conclude, the update rule for VPG is

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_{\theta_k}) \tag{2.14}$$
$$where\, \nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} [\sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t)]$$

with the term $A^{\pi_\theta}(s_t, a_t)$ representing just one example of the score function which is, in practice, implemented in lots of different ways.

2.4.3 Deep Deterministic Policy Gradient (DDPG)

Policy optimisation algorithms like VPG are sample inefficient since they can only learn from fresh samples collected with the current policy; there is no data re-use. Returning to the idea of off-policy learning where data re-use is possible, recall how in DQN the policy is implicit in Q by selecting the action with the greatest state-action value, i.e. $arg\max_a Q(s,a)$. Therefore, by learning Q^*, we get a^* for free. This approach is fine for small discrete action spaces, but when working with large or continuous action spaces the max operator becomes problematic.

DDPG , (Lillicrap et al., 2016) was the first algorithm to appear in the hybrid section of the taxonomy graph, combining DQN with the policy gradient algorithm to produce a version of DQN that avoids the max operator and therefore is suitable for the large and continuous action spaces which so frequently come up in research areas such as robotics. Like policy gradient methods, DDPG uses a neural network to explicitly model the policy. However, rather than learn stochastic policy $\pi(.|s)$, DDPG learns the deterministic policy $a = \mu(s)$. Just as the objective in VPG and variants was to learn π^*, the objective is to learn optimal policy μ^*. Since the policy is deterministic, in the case where μ is in fact optimal, then the action selected by μ ought to be the action with the highest value. As such, DDPG

circumvents the expensive max operator by approximating it with $max_a Q(s,a) \approx Q(s,\mu(s))$, transforming the MSBE function from DQN

$$\mathscr{L}(\phi) = \underset{s,a,r,s' \sim U(D)}{\mathbb{E}} [(Q_\phi(s,a) - (r + \gamma \max_{a'} Q_\phi(s',a')))^2]$$

into

$$\mathscr{L}(\phi) = \underset{s,a,r,s' \sim U(D)}{\mathbb{E}} [(Q_\phi(s,a) - (r + \gamma Q_\phi(s',\mu_\theta(s'))))^2]. \tag{2.15}$$

In both cases, the target is telling us the same thing – it is an estimate of the return we expect to see following this state-action pairing by looking at the immediate reward plus the best possible return we can expect given the next state. The difference is that, before, we were arriving at the best possible return by looking at the values attached to next state s' for every action in the action space and selecting the one with the maximum value. However, because we are concurrently learning an explicit policy that is deterministic and (hopefully) converging on the optimal policy, then we need only look at the value attached to the action that this policy selects for. That is, $\mu_\theta(s)$ ought to give the action that maximises $Q_\phi(s,a)$, and so using it as input to the Q-function is approximately equivalent to using the max operator on the Q-function outputs.

In 2014, the team at DeepMind demonstrated that not only was it possible to learn $\mu_\theta(s)$ with the policy gradient algorithm, but that estimating the gradient of a deterministic policy is in fact more efficient than estimating the gradient of a stochastic policy (Silver et al., 2014). Recall that, with policy gradient methods, the objective function $J(\pi_\theta)$ is a measure of performance and can be expressed as the expected return from a trajectory experienced following the policy under evaluation. This expectation $\mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]$ was expanded to $\int_\tau P(\tau|\theta)R(\tau)$. Recall also the discussion on how it made more sense to use the expected return from the present onward (G_t) and that this could be estimated with $Q^\pi(s,a)$, therefore $\int_\tau P(\tau|\theta)R(\tau)$ becomes $\int_S p^\pi(s) \int_A \pi(a|s) Q^\pi(s,a)$. When the policy is not the stochastic policy π but the deterministic policy μ, the probability of the action a can be dropped and $J(\mu_\theta)$ becomes $\int_S p^\mu(s) Q^\mu(s,\mu_\theta(s))$. Therefore, the policy gradient in the deterministic setting is equivalent to the expected gradient of the state-action value function. Estimating the gradient in this setting only involves integrating over the state space and not the state and action space, hence

the improved efficiency.

The chain rule $\nabla f(g(x)) = g'(f(x)) \cdot f'(x)$ means that the gradient of $Q^{\mu}(s, \mu_{\theta}(s))$ with respect to θ becomes the gradient of the Q-function with respect to a multiplied by the gradient of the policy with respect to θ. This limits DDPG to use with continuous action spaces, so that $Q(s,a)$ can be presumed to be differentiable with respect to a. The resulting policy gradient (2.17) and update rule (2.16) is

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} J(\mu_{\theta_k}) \tag{2.16}$$

$$\begin{aligned} where\, \nabla_{\theta} J(\mu_{\theta}) &= \int_S p^{\mu}(s) \nabla_a Q^{\mu}(s, \mu_{\theta}(s)) \nabla_{\theta} \mu_{\theta}(s)\, ds \\ &= \mathop{\mathbb{E}}_{s \sim p^{\mu}} [\nabla_a Q^{\mu}(s,a) \nabla_{\theta} \mu_{\theta}(s)|_{a=\mu_{\theta}(s)}] \end{aligned} \tag{2.17}$$

As with the move from Q-Learning to DQN, the 2016 *deep* implementation of Silver et al.'s deterministic policy gradient algorithm applies the same stabilising methods as DQN, i.e. a replay buffer to decorrelate the data and a second network $Q_{\bar{\phi}}(s, \mu(s'))$ (target network) to address the moving target problem. As in DQN, the parameters of the target Q-network 'lag behind' the parameters of the primary Q-network. In DQN this was approached in quite a literal way by only updating the target Q-network periodically. In DDPG, the parameters of the target Q-network are updated each time the main network is updated, but with a soft update of the form $\bar{\phi} \leftarrow \rho \bar{\phi} + (1-\rho)\phi$, where ρ is a hyperparameter between 0 and 1 (usually close to 1). This method is called polyak averaging and causes the target Q-network to change more slowly despite being updated at the same rate. The policy used in the target value calculation is *also* an additional network, referred to as the target policy (notation $\mu_{\bar{\theta}}(s)$)) and updated in the same way with polyak averaging. Finally, DDPG introduces batch normalisation to address the problem of finding hyperparameter values that generalise to features with different physical units (and potentially different environments). Across the samples in a minibatch, each dimension is normalised to have unit mean and variance, and a running average of the mean and variance is taken to use for normalisation at test time.

As with DQN, the algorithm is off-policy, meaning that exploration can be encouraged in the behavioural policy. This is of increased importance when the policy is deterministic because the variety of experience is reduced. In DDPG, rather than the behavioural policy including random action selections a proportion of the time, the behavioural policy includes noise (a time correlated noise process called Ornstein-Uhlenbeck noise (Uhlenbeck and Ornstein,

1930)) at every time step. In essence, this is just making the deterministic policy more stochastic. In fact, even before the injection of noise, a deterministic policy is just a special case of the stochastic policy, where the probability distribution contains only one non-zero value over one action. As discussed in the DQN section, off-policy learning is appropriate for MSBE minimisation given Q^* should satisfy the Bellman equation for any transition. As discussed in the VPG section, policy gradient methods ordinarily require on-policy learning. It is possible to estimate the policy gradient with off-policy data if an importance sampling ratio is used (weighting the update with the probability ratio between the target policy and behavioural policy), however because the deterministic policy gradient removes the integral over actions, taking into account differences in action probabilities is not necessary.

Like any Q-learning algorithm, DDPG can suffer from overestimated Q-values propagating through the training iterations, causing policy collapse. The next algorithm is a variant of DDPG with alterations to mitigate these difficulties.

2.4.4 Twin Delayed Deep Deterministic Policy Gradient (TD3)

TD3 (Fujimoto et al., 2018) applies three alterations to DDPG. The first is a trick brought over from a variation of DQN called Double DQN (Van Hasselt et al., 2016) to address the issue of overestimated value. Double Q-learning uses two Q-networks, one to fulfil the usual role of policy evaluation and one specifically for use with the max operator when it comes to action selection, decoupling the implicit policy from the value function. TD3 also uses two Q-networks, but tackles overestimation head on by using the lesser of the two values in *both* MSBE functions, as a form of clipping. Underestimation bias is still problematic but not as problematic as overestimation bias, since underestimation does not propagate via TD updates (Bawa and Ramos, 2021).

The second alteration is to delay policy updates. TD3 implements polyak averaging when updating the target policy and target Q-network, as in DDPG, to address the moving target problem, but it also updates the policy networks less frequently than the Q-networks. The paper suggests every other Q-network update is sufficient, but the idea is to delay policy updates until the Q-value error is small enough to reliably inform the policy, in an attempt to avoid the downward spiral situation where bad estimates reduce policy performance, leading to poor data, leading to even worse estimates, and so on.

The third and final alteration is a slight adjustment to the way that DDPG adds noise to actions. Rather than use time-correlated Ornstein-Uhlenbeck noise, TD3 used plain and

simple Gaussian noise, since it is easier to implement and the authors claim the former provides no performance benefit. Noise ξ is sampled from $N(0, \sigma)$, clipped between an upper and lower limit, and added to each dimension of the action. In the DDPG section, adding noise to actions was described as a mechanism for baking a degree of exploration into the policy. In the TD3 literature, adding noise is commonly described as policy smoothing, so-called because if the Q-network overestimates the value of an action, producing a big spike in the function, then some of the value ascribed to that action can be drawn out and ascribed elsewhere (smoothing the spike) by performing an action that is similar but not the same (the action plus noise). Preventing the policy from being too exploitative like this is the same thing as encouraging the policy to explore.

2.4.5 Soft Actor Critic (SAC)

Whilst TD3 decorates DDPG with tricks, SAC (Haarnoja et al., 2018) introduces a major change to the objective function. The notion of an exploration-exploitation trade-off has been a recurring theme, and whereas other algorithms discussed so far have learnt a fully exploitative policy and then injected exploration, either with noise or with proportional random sampling, SAC instead adds a measure of exploration into the objective function, so that the policy *learns* to explore as well as exploit. By stepping away from exploitation-led optimisation, the model is more likely to capture any near-optimal strategies that exist, allocating near-equal probability to actions that are nearly as good. The authors argue this makes SAC more stable than DDPG with its notorious brittle convergence properties and sensitivity to hyperparameters.

The objective function in SAC is a two-part objective, maximising the policy's entropy as well as expected return. Intuitively, entropy can be described as the level of surprise or uncertainty for a distribution $P(X)$. The concept of entropy actually originates from information theory and the work of Shannon in the 1940s. The information entropy, or Shannon entropy, of a random variable is the amount of storage (e.g. number of bits) required to store the variable. More specifically, it is the absolute minimum amount of storage required to succinctly capture the *information* of a variable, where amount of information is not only dictated by the amount of different values a variable might take on (the raw data), but the process by which the variable takes on different values. How the number of bits could possibly translate as uncertainty is not obvious, but Vajapeyam (2014) provides a good analogy, pointing out that the amount of information in an email is not simply the number of words, or the number of possible words in the language the email is written in. If an email provides details on an event that happens every week at the same time and place then it is

not informative or 'surprising' at all. Surprise in the context of probability distributions is inversely related to bias, for example, the outcome of a weighted coin which always lands on heads is not going to be surprising at all, whereas a fair coin with uniform probability yields maximum uncertainty and therefore maximum entropy. Hence, with the objective to maximise entropy as well as return, the algorithm is encouraged to produce a smoother policy.

In SAC, the optimal policy π^* is considered as

$$\pi^* = arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty} \gamma^t (R(s_t, a_t, s_{t+1}) + \lambda H(\pi_\theta(.|s_t)))]$$

where H is the entropy measure and λ, called the entropy temperature, is a hyperparameter which controls the weighting placed on the entropy component of the objective. Note that SAC learns a stochastic policy not a deterministic policy, which makes sense in the context of an entropy maximising objective (a deterministic policy would definitely not meet the 'be surprising' brief).

As in TD3, the SAC algorithm learns two Q-functions. As before, each function is represented by two networks, a primary network and a target network updated more slowly with polyak averaging. Q-network parameters are updated with MSBE minimisation, regressing to a shared target which uses the smaller of the two Q-values (clipped double-Q trick). The difference between the MSBE loss function in DDPG and TD3 (equation 2.15) and the MSBE loss function used in SAC is a change to the TD target $r + \gamma Q(s', \mu(s'))$. The change stems from a new formulation of the Bellman equations with an appended entropy term. For example,

$$Q^\pi(s,a) = \underset{\pi}{\mathbb{E}}[R(s,a,s') + \gamma Q^\pi(s',a')]$$

becomes

$$Q^\pi(s,a) = \underset{\pi}{\mathbb{E}}[R(s,a,s') + \gamma(Q^\pi(s',a') + \lambda H(\pi(.|s')))]$$
$$where\ \ H(P) = \underset{x\sim P}{\mathbb{E}}[-logP(x)]$$

leading to a modified TD target

$$Q^{\pi}(s,a) \approx r_{t+1} + \gamma(Q^{\pi}(s',\tilde{a}') - \lambda log\pi(\tilde{a}'|s')), \qquad \tilde{a}' \sim \pi(.|s')$$

Note the substituting of $\mu(s)$ with $\pi(.|s)$. Note also a subtle bar notation on the next action term a'. This notation is there to stress that next action a' is sampled from the current policy and not the replay memory D, as is the case for r and s'.

In terms of policy optimisation, SAC maximises the objective function

$$J(\pi_{\theta}) = \mathop{\mathbb{E}}_{a_t \sim \pi}[Q_{\phi}(s_t,a_t) - \lambda log\pi_{\theta}(a_t|s_t)]$$

derived from a minimisation of the expected KL divergence

$$J(\pi_{\theta}) = \mathop{\mathbb{E}}_{s_t \sim D}[D_{KL}(\pi_{\theta}(.|s_t)||\frac{exp(Q_{\phi}(s_t,.))}{Z_{\phi}(s_t)})]$$

(see the original paper from Haarnoja et al. (2018), specifically appendix B.2, or the Lilian Weng resource 'Policy Gradient Algorithms' (2018) for full derivations). Ignoring the $log\pi_{\theta}(a_t|s_t)$ entropy term, this is the same approach taken in DDPG, where $\mathbb{E}_{\tau \sim \pi}[R(\tau)]$ is expressed as $\int_S p^{\pi}(s) \int_A \pi(a|s) Q^{\pi}(s,a)$, which in turn is expressed as $\mathbb{E}_{s \sim p^{\mu}}[Q^{\mu}(s,\mu(s))]$, except this time the stochastic policy prevents the dropping of the integration over actions. This introduces a pain point, as this distribution depends on the policy parameters. SAC therefore uses a reparameterisation trick whereby, rather than sampling actions from $\pi_{\theta}(.|s)$, it instead samples noise vector ξ from some fixed distribution, such as a spherical or standard Gaussian, and then represents π_{θ} implicitly in a neural network transformation

$$\tilde{a} = f_{\theta}(\xi;s)$$

such as

$$\tilde{a}(s,\xi) = tanh(\mu_{\theta}(s) + \sigma_{\theta}(s) \odot \xi), \qquad \xi \sim \mathbb{N}(0,I)$$

The *tanh* acts as a squashing function, bounding the actions to a finite range. The fact that the standard deviations are a network output and not a state-independent parameter vector, as

in the other policy optimisation algorithms, means they depend on state in a complex way. The final objective is then

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\xi \sim \mathbb{N}}[Q^{\pi_\theta}(s, \tilde{a}_\theta(s,\xi)) - \lambda log \pi_\theta(\tilde{a}_\theta(s,\xi)|s)]$$

2.5 Summary

This chapter covers the key terms, concepts and methods required to understand the novel work presented in this thesis. The proposed solution is a pipeline of two models: a variational autoencoder and a policy network trained with DRL algorithm Soft Actor Critic. Both are deep learning models and the former receives image data as input, hence why an overview of deep learning and computer vision is provided, before detailing the VAE model type. Then, rather than jump straight to presenting SAC, the chapter builds up to it, starting with RL in general (core concepts, terminology and formalism), before covering value functions, bootstrapping, and some of the most important algorithms which preceded it. This is because the mathematics is much easier to understand starting with the simplest case of policy-based (VPG) and value-based (DQN) learning, and then layering on techniques and changes with each new algorithm that was published. Note that, in addition to the seminal papers which introduce these algorithms, derivations and explanations provided in Section 2.4 have been guided by the educational resource 'Spinning Up in Deep RL' (2018) produced by OpenAI, and the detailed blog 'Policy Gradient Algorithms' (2018) from Lilian Weng's website Lil'Log. 5.

All of the presented algorithms address the challenge of balancing exploration and exploitation. However, it was explained how SAC is unique in the way it approaches this challenge, incorporating exploration directly into the objective function with an entropy maximisation term as opposed to learning a fully exploitative policy and then injecting stochasticity later. This form of policy smoothing prevents the policy from collapsing into always selecting the same action, and allows the policy to capture multiple modes of near-optimal behaviour. This, along with the double Q-learning trick, makes SAC more stable, as well as the sample efficiency gained from off-policy learning.

Chapter 3

Game Engines as a Platform for Simulated Learning Environments

3.1 Introduction

Deep reinforcement learning is notoriously sample inefficient and therefore data hungry, a problem exacerbated by a large state space, as is the case in vision-based DRL, or large action space, as is the case in complex control problems. Data hungry models are problematic in real-world application research since real-world data is often expensive, difficult and time-consuming to collect. The expense and difficulties of real-world experimentation are particularly impeding in the context of underwater robotics, requiring tanks, pools or trials in open water. Training, testing or collecting data in open environments like the sea requires a lot of resources and logistics. With the researcher at the surface on board the research vessel and the robot operating underwater, even just assessing performance can become complicated. Also, days and weeks could pass without a sighting of the animals, and conducting surveys is highly weather dependent.

Beyond these challenges, model-free DRL for robotic control introduces a much bigger complication. In this experiential learning paradigm, data is collected through agent-environment interactions, meaning the agent is inexperienced and prone to error in the early phases of training. These errors could be costly, causing damage to the physical system (see Koryakovskiy et al. (2017) as an example). They could also be dangerous, especially in contexts outside of a lab (a concern being actively addressed by research giants like OpenAI (Ray et al., 2019)). When it comes to training a vehicle to follow a wild animal, allowing unrestricted experiential learning in the real world environment is just not an option, posing too great

a risk to the animal. The solution that many researchers have turned to, particularly at the intersection of reinforcement learning and robotics, is simulators.

In a sim-to-real approach, the robot is trained in a synthetic environment and then deployed in a real environment. Training in simulation is cheaper, safer, easier and more reproducible, due to the higher degree of control. Simulation-based training is also a lot more scalable, with the ability to train a large number of robots in parallel, and offers small practical benefits, such as instant resets when training episodically. Crucially, simulations afford automated data labelling. The most obvious beneficiaries of this are approaches like tracking by detection, whereby bounding box annotation requires intensive human effort. Accuracy is as much a benefit as time here, since human annotation is prone to error and fatigue, even with crowdsourcing approaches such as Amazon Mechanical Turk (Crowston, 2012). However, supervised methods are not the only beneficiary. In RL, we can think of the reward value as the data label. When it comes to complex, fine-grained control tasks, the agent is likely going to require a dense and informative reward signal, i.e. a label per time step, and a label involving ground truth measurements, such as target distance in the case of visual active tracking (VAT). In the real world, calculating ground truth object distances would require specialist equipment (e.g. stereo cameras), whereas in the virtual world, there is direct access to state variables such as positions, orientations and speeds. Since the reward signal is only required at training time, this is a big win for sim-to-real DRL.

With these advantages come some problems. The first is a problem known as the 'sim-to-real gap' – a term used to describe the disparity between the simulated and real world, both in the visual domain and the dynamics domain. The second is the problem of sourcing or developing a suitable simulation. In some ways, this is analogous to the problem of sourcing a suitably large, suitably labelled dataset for supervised learning. However, when it comes to 3D learning environments, the availability of open source resources is less and the skills required for custom development are greater. The terms environment and simulation tend to get used interchangeably, but it is important to understand that an 'environment' is typically an entire software stack. Throughout the rest of this thesis, the term simulation will be reserved for the application providing the graphics (and perhaps physics). This can be thought of as the environment's front end – the place where environment interactions are actually played out i.e. action decisions are implemented and data is collected. A simulation alone does not provide a learning environment, however. It is also necessary to define the MDP (see Section 2.3.2) governing the agent's learning of a particular problem. American artificial intelligence company OpenAI provide the open source Python library Gym (Brockman et al.,

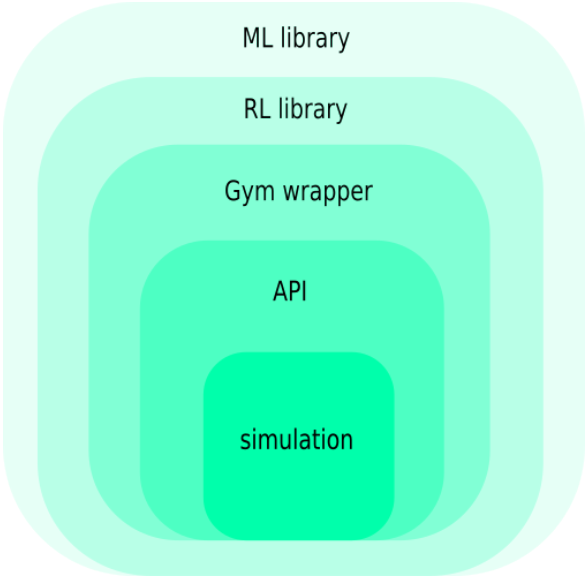


Fig. 3.1 Illustration of the software stack required by simulation-based agent training.

2016) as a standard API for expressing an MDP. This can be thought of as the environment's back end, and will be referred to as the 'Gym environment' or 'Gym wrapper'.

Figure 3.1 provides an illustration of a typical software stack. An ML library like TensorFlow (Abadi et al., 2015) is used by a DRL library like Stable Baselines (Hill et al., 2018a) to implement a DRL algorithm. Most if not all DRL libraries integrate with Gym (sometimes exclusively), and so the interface between the algorithm and the environment can be seamlessly taken care of just by providing the environment with a Gym wrapper (i.e. extending the Gym class `Env`). The difficulty lies in developing the simulation itself, and the communication between the simulation and the Gym environment. This API is responsible for sending simulation data (observations) and retrieving model outputs (actions), and is therefore critical to the cyclical agent-environment interactions at the heart of RL. Not only does this communication need to be fast enough to support real-time simulation, it needs to be robust enough to ensure that the generated data traces `[obs, action, next obs, reward, done]` are correctly paired.

Developing this full stack from the ground up requires time, effort, and a diverse skill set. As such, a large amount of DRL research utilises a small number of existing open source environments, similar to the pervasiveness of ImageNet (Deng et al., 2009) in supervised computer vision research. These environments provide a useful benchmark when proposing new algorithms or optimisations. However, in application-focused research, a custom prob-

lem is likely to require a custom environment.

This need has fed into a growing trend toward the use of commercial game engines. Commercial, cross-platform game engines such as Unity (Haas, 2014) and Unreal Engine (Sanders, 2016) bring powerful graphics rendering and sophisticated physics under one roof, along with intuitive development tools, extensive documentation, and a large, active community. Their GUI-driven, real-time editors make 3D creation efficient and accessible to new developers, with mechanisms like drag and drop. Both have a dedicated 'asset store', with a large collection of models and plugins available for purchase. These assets, many of which are free, can be easily imported into a wider project, accelerating development further. Primarily used to develop games, these platforms have been massively extended over the years through public demand, to the point of providing realistic and physically accurate environments suitable for research.

This chapter details three novel, Unity-hosted, 3D learning environments developed during the course of this project. All three support visual active tracking, but only the last environment, SWiMM DEEPeR, focusses on the intended deployment context. Although the Unity Editor is designed to be accessible to new developers, simulating water is particularly difficult, and creating a pseudo-realistic underwater open ocean environment complete with controllable AUV and animated marine mammals requires experience. A collaboration was formed with the School of Computing's Game Engineering group to outsource development. Forming this collaboration was a slow process, due in part to the pandemic. Therefore, two other environments were developed to progress development of the Gym environment and DRL algorithm. The first was CubeTrack, representing a simplified version of the VAT problem (discrete action space, simple physics, simple graphics, and a small, closed environment), ideal for environment testing and the early stages of reward engineering. For this environment, the communication API was handled by the ML-Agents toolkit (Juliani et al., 2018) described in the next section, and the simulation was built from the ground up, utilising ML-Agents' prefabs. The second was DonkeyTrack, a modification of an existing Gym wrapped Unity simulation, with custom communication framework. This open world environment allowed for experimentation within a more realistic continuous control setting whilst work on SWiMM DEEPeR progressed.

3.2 Related Work

3.2.1 Toy environments

Research in DRL relies heavily on a limited selection of off-the-shelf, open source toy environments. Prominent examples include the suite of toy environments provided by OpenAI Gym, the 'DeepMind Control Suite' (Tassa et al., 2018), and the Unity Machine Learning Agents Toolkit (ML-Agents) (Juliani et al., 2018). The simplicity of these environments, good documentation, and sharing of results/hyperparameters makes them a good starting point for students and researchers entering the field. They are also an extremely valuable resource for RL research and development (R&D). When algorithms are developed or modified, it is important to benchmark against results generated in the same environment, and therefore their pervasiveness is beneficial. However, they lack variety and they lack realism, providing little use for application-focused research, especially where there is the intention to transfer the learnt policy to a real setting.

This is especially true of OpenAI Gym, which provides mostly 2D environments. Gym is both an API (via the Python package `gym`) and a collection of open source environments. The environments are organised into collections, with increasing problem difficulty and simulation complexity. For example, the 'classic control' collection could be considered the entry-level collection in terms of difficulty. These computerised renditions of classic problems from control theory (e.g. the inverted pendulum swingup problem) require only PyGame (Kelly, 2016) for 2D graphics rendering. The collection 'Box2D' offers a step up in difficulty, including games like 'Lunar Lander', 'Car Racing' and 'Bipedal Walker'. These environments use PyGame for rendering but also use Box2D (Parberry, 2017) for 2D physics simulation. There is then a collection of Atari games, simulated using Stella (1996), a multi-platform Atari 2600 emulator, and the Arcade Learning Environment (ALE), a simple framework which separates the details of emulation from agent design (Bellemare et al., 2013). The last and perhaps most challenging is the the MuJoCo collection. MuJoCo (Todorov et al., 2012) (multi-joint dynamics with contact) is a physics engine developed by Emo Todorov for the robotics startup Roboti, and later acquired by DeepMind.

The DeepMind Control Suite extends this Gym MuJoCo collection. These environments are all 3D, there is greater representation of real-world problems (e.g. a simplified soccer-like problem), and there are collections specific to niche research areas, such as language (Hermann et al. (2017)), meta-learning (Wang et al. (2021)), and AI safety (Leike et al. (2017)). However, these environments are still very abstract toy environments, they are

not realistic or even pseudo-realistic. The ML-Agents Toolkit provides a similar offering. Like Gym, ML-Agents is both an environment collection and an API. Although all of the environments are 3D and although ML-Agents utilises a commercial game engine, they are still very much toy problems, ranging in difficulty from tasks like block pushing to multi-agent games like soccer. Across all tasks, the graphics are cartoon-like and the scene is a contained, floating platform. The code and official documentation can be accessed via the public GitHub repository (Unity-Technologies, 2017b).

3.2.2 Car and drone environments

Moving into the space of pseudo-realistic, game-engine-based environments, there are several prominent examples looking at vehicle control. For example, in 2017 Microsoft Research released AirSim as a simulation platform for autonomous driving research (Shah et al., 2018). Built on Unreal Engine 4 (but with an experimental Unity release), AirSim provides pseudo-realistic urban scenes containing roads, city blocks, power lines and trees. The platform supports both manual control and programmatic control, with an API accessible via a variety of languages. This API is available as an independent library, such that it can be deployed on a vehicle's onboard computing device. The primary focus of AirSim is aerial autonomy, although the platform does provide car simulations also. Other well-known car simulations include CARLA from Intel (Dosovitskiy et al., 2017) and TORCS (The Open Racing Car Simulator) from Espié and Guionneau (2016). CARLA is built on Unreal Engine 4, and has been developed from the ground up to support autonomous driving research, incorporating realistic and varied scenarios such as traffic, pedestrians, weather etc. TORCS on the other hand is focused on track racing, and is as much a game as it is a research platform. The popular platform has cultivated a large community, hosting competitions for researchers and hobbyists to submit their AI controllers. Similar to this is the Donkey Car community, with competitions running both in the simulated environment and in real environments with AI-controlled, modified toy cars. This Unity simulation (SDSandbox, 2017) and Gym wrapper (`gym-donkeycar`, 2018) from Tawn Kramer is the basis for DonkeyTrack detailed in Section 3.4. All of these platforms offer sophisticated physics and, in the case of AirSim and CARLA, extremely high visual fidelity. However, they do not provide the appropriate problem (VAT) or the appropriate context (AUV control in an open ocean environment).

3.2.3 AUV simulations

A focused search was conducted for existing AUV simulations. Surveys from Matsebe et al. and Cook et al. provide a good starting point, however these surveys are now dated (published

in 2008 and 2014 respectively) and are not entirely relevant; a lot of the reviewed simulations are focused on ROV pilot training, for example, and therefore do not offer programmatic control. Below are four projects that have been identified as prominent examples of AUV simulators offering programmatic control. Table 3.2 provides a summary of the key features to make the contributions of SWiMM DEEPeR clear. Firstly, the simulations in the first two columns do not utilise game engines whereas the third and fourth do, similar to SWiMM DEEPeR. The former two are also examples of projects that sit firmly in the robotics research space. They are geared toward use cases like mission planning, mission re-enactment, system testing and vehicle design prototyping. Whilst they offer programmatic control, the focus is more toward conventional control strategies, and DRL is not natively supported, or at least not in a way that is straightforward. There is also a strong focus on accurate hardware simulation, whereas, in the simulations presented in this chapter, fidelity is less important. Instead, computational expense is minimised in order to provide the speed and efficiency required for training data-hungry algorithms within a feasible time frame and computational budget. Based on this, SWiMM DEEPeR, like HoloOcean, chooses not to integrate with Robot Operating System (ROS) (Koubâa et al., 2017) – a popular robotics middleware suite which serves as a distributed control centre, allowing different nodes (Python and C++ packages) to communicate through 'topics' within and across machines, via publishing and subscribing mechanisms.

UUV Simulator

Unmanned Underwater Vehicle (UUV) Simulator from Manhães et al. (2016) extends Gazebo (Koenig and Howard, 2004), a leading simulation platform in the robotics field. It does so through the addition of custom underwater modules not otherwise provided, including hydrostatic and hydrodynamic effects, thrusters, fins, and typical underwater sensors. The Gazebo platform exploits the OGRE graphics rendering engine, four dynamic physics engines (ODE, Bullet, Simbody and DART), and provides integration with the middleware software ROS. Both Gazebo and ROS are well-documented, well-maintained, and their use allows for accurate physical behaviour. However, UUV Simulator itself does not look to be actively maintained, and provides no integrated DRL support. Gazebo and ROS are also heavy project dependencies, making installation and usage more cumbersome (Gazebo's Windows installation has 18 steps, for example). That being said, there are some pretty recent and pretty relevant examples of DRL research using UUV Simulator. In 2020, Zhang et al. used UUV Simulator to train a DRL controller to perform path following (both straight and sinusoidal), experimenting with different combinations of environmental and human-allocated rewards. In 2022, Grando et al. used UUV Simulator to train a DRL controller to perform mapless

	UUV Simulator	UWSim	URSim	HoloOcean	SWiMM DEEPeR
Game Engine	✗	✗	✓	✓	✓
Graphics	OpenGL	OSG (osgOcean)	DirectX + Metal + OpenGL + Vulkan	DirectX 11 + 12	DirectX + Metal + OpenGL + Vulkan
Physics	ODE + Bullet + Simbody + DART	Bullet (osgBullet)	PhysX	PhysX	PhysX
Communication	TCP/IP	TCP/IP	TCP/IP	Shared Memory	TCP/IP
ROS Integration	✓	✓	✓	✗	✗
Low-dependency	✗	✗	✗	✓	✓
Action Inference	✗	✗	✗	✗	✓
DRL Integration	✗	✗	✗	✓	✓
Configurable	✗	✓	✓	✓	✓
Dynamic Target	✗	✗	✗	✗	✓

Fig. 3.2 Comparison table between SWiMM DEEPeR and existing AUV simulations. The first three columns are examples of simulations geared more toward robotics and therefore value fidelity, whereas SWiMM DEEPeR and HoloOcean aim to be lightweight, fast, minimally computationally expensive and DRL integrated. SWiMM DEEPeR additionally supports VAT with the introduction of dynamic targets.

navigation and obstacle avoidance, with a hybrid vehicle capable of transitioning from air to water and water to air. In both cases, the simulated environment is a pool.

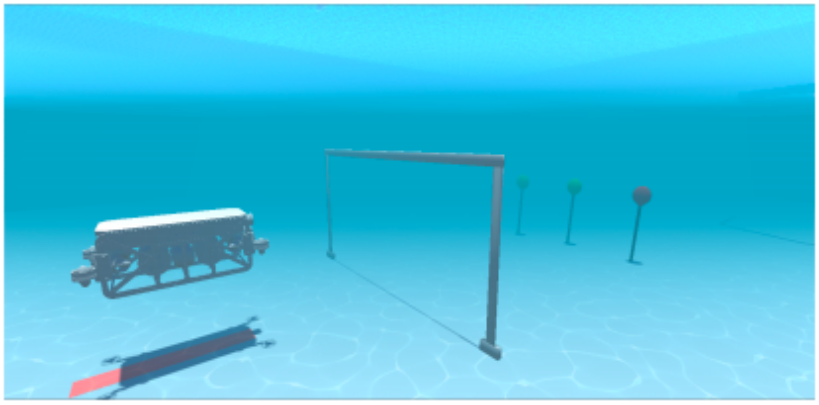
UWSim

Underwater Simulator (UWSim) from Pérez Soler et al. (2013) is a custom built solution utilising Open Scene Graph (OSG) (OpenSceneGraph, 1999) and osgOcean (Bale, 2008). OSG is a cross-platform graphics API which uses C++ and OpenGL, whilst osgOcean is a wrapper library specifically for creating realistic underwater rendering in OpenGL (for

example, ‘god rays’ or surface glare). To achieve physical accuracy, the wrapper osgBullet (Martz, 2011) provides contact physics for OSG. UWSim provides underwater vehicles, surface vessels, robotic manipulators, customisable widgets, and a network interface to ROS, allowing for hardware-in-the-loop simulation. Although UWSim provides a suitable underwater simulation, there is no integrated DRL support. Furthermore, it is designed as a kinematic simulator, meaning that the vehicle is ‘controlled’ by updating the object’s position and rotation. Dynamic simulation (responding to forces) in UWSim requires an external module coded in Matlab, limited to single-body vehicle control. Setting up a new simulation can also be laborious, with XML description files instead of a user interface, and a lack of documentation. Kermorgant (2014) provide an example of pairing UWSim with Gazebo in order to benefit from Gazebo’s user interface and dynamic simulation. However, the integration requires rather complex file synchronisation, scene initialisation still requires monolithic XML description, and the project repository (*freefloating-gazebo*) is now archived. Despite this, it is still being used by other researchers to this day (e.g. Mehta et al. (2021)).

URSim

The project URSim from Katara et al. (2019) is an example of another Unity-hosted solution. This engine provides support for several graphics APIs (DirectX, Metal, OpenGL, Vulkan), calculating and using whichever is the most suitable at runtime. It also provides realistic physics via Nvidia’s PhysX (NVIDIAGameWorks, 2015). Whilst URSim exploits the same game engine as the simulations presented in this chapter, the similarities end there. URSim focusses mostly on mission planning and conventional control methods, integrating with ROS via the plugin ROSBridge (Crick et al., 2017). The paper hints at ‘tracking functionality’, but then makes no further reference to object tracking or neural-network-based control. It may be possible to train DRL agents in URSim using the ROS package `openai_ros`, although this makes reference to Gazebo and therefore will likely not work out of the box with a project utilising Unity and ROSBridge. It is also a fairly complicated way in which to pair Unity with Gym, introducing an unnecessary heavy dependency when there is not the need to make use of the many sensor, actuator and controller components offered by ROS. If the only role of Unity is to generate data, then direct communication with Python, as is done in the next example, makes a lot more sense. URSim is also limited to a pool environment (see Figure 3.3), although in 2021 Osa and Orukpe extended URSim to provide a collection of ocean-themed sandbox environments.



(a) Image of URSim from Katara et al. (2019).

(b) Image of HoloOcean from Potokar et al. (2022b).

Fig. 3.3 Examples of existing game-engine-based AUV simulations.

HoloOcean

HoloOcean from Potokar et al. (2022a) exploits Unreal Engine 4, providing a graphics renderer DirectX11/12 API and PhysX. HoloOcean is a fork of HoloDeck (Greaves et al., 2018) – a Python API with a Gym-like interface – for supporting game-engine-based reinforcement learning. HoloOcean then augments HoloDeck with a game binary, offering a pseudo-realistic ocean environment complete with accurate underwater dynamics, a realistic imaging sonar implementation, and other underwater sensor models. As such, HoloOcean is much more lightweight than the solutions discussed thus far. It has no reliance on ROS; all communication is via the Python interface, and so installation is quick and straightforward. HoloOcean is very flexible and customisable, with good documentation. Whilst graphical

customisation allows for a wider range of computational systems, there is still the requirement for a 'competent GPU', whereas the simulations presented in this chapter have much lower computational demand.

Connection to this thesis

The HoloOcean environment is the most comparable to the application-focused environment presented in this thesis, SWiMM DEEPeR. There are however some differences. The former simulates an in-house rover (see Figure 3.3), whereas the latter simulates the BlueROV2 from Blue Robotics – an affordable and widely used commercial vehicle. The reason for this was to better support open source science and collaborative research. The two simulations are also focused on slightly different problem spaces. All of the scenes provided by HoloOcean contain static marine structures such as shipwrecks, piers, dams and pipes, suitable for training navigation policies. SWiMM DEEPeR on the other hand contains animated marine mammal models, providing a simulation in a new application space (conservation) as well as a new problem space (visual active tracking). Had HoloOcean been published earlier, SWiMM DEEPeR may well have been developed as an extension, however the two were developed in parallel. HoloOcean is certainly a bigger and more advanced project, yet SWiMM DEEPeR offers what is needed for the custom problem that has been posed. The benefit of developing SWiMM DEEPeR from the ground up is the complete control over (and understanding of) the implementation, particularly with respect to the communication framework. The communication framework implemented in HoloDeck is complex and unfamiliar, taking a shared buffer approach involving semaphores, whereas SWiMM DEEPeR uses the much more widely used TCP protocol.

TCP is a connection-oriented protocol, establishing and maintaining a connection until the two involved applications have finished exchanging messages. The protocol determines how to segment data into packets and then sends these packets to (and receives packets from) the network layer. TCP is a particularly strict protocol that aims for error-free data transmission. It requires an acknowledgement that packets have arrived, and handles re-transmission of dropped or corrupt packets. Together with the Internet Protocol (IP), it provides the basic rules that define the internet.

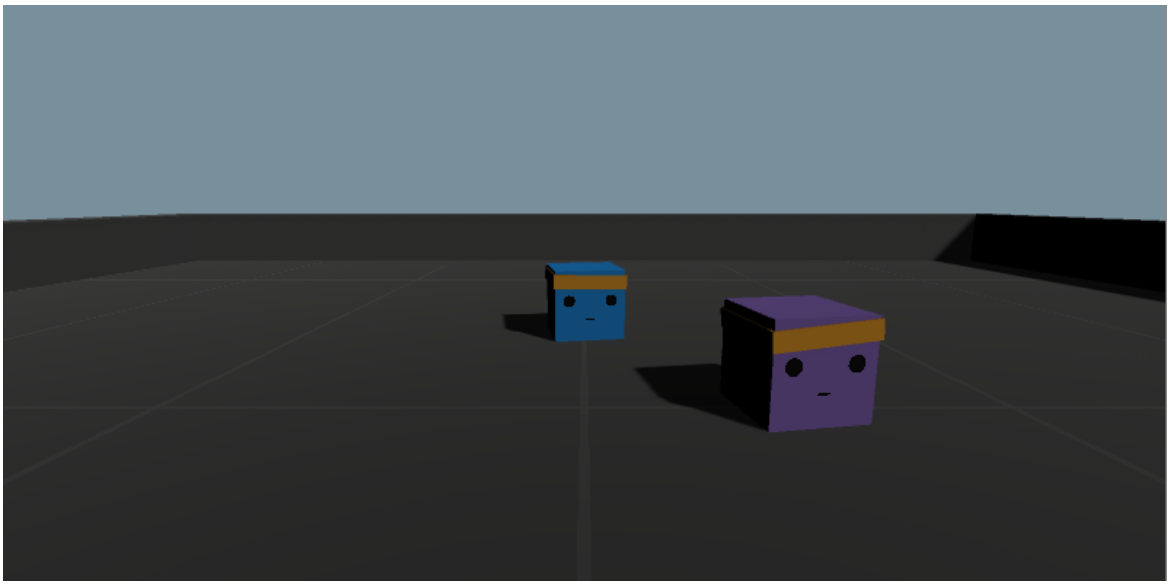


Fig. 3.4 Screenshot of CubeTrack in game view.

3.3 CubeTrack

The environment presented in this section is a novel environment developed from scratch for the purpose of this doctorate research. The code for CubeTrack is available at [1].

3.3.1 Simulator

As with all three novel environments presented in this chapter, the environment CubeTrack is powered by the Unity game engine. The specific version of Unity used for developing CubeTrack was version 2019.4.4f1. CubeTrack is an example of building a game-engine simulation from the ground up *without* outsourcing. As such, the simulation is extremely simple. The Unity scene itself has three main components: a dark grey, flat, rectangular platform, a purple cube (the moving target), and a blue cube (the RL agent). The platform is enclosed by walls to prevent either cube from falling off the floating platform into the abyss of the simulated world. The two cubes are prefabs from the open source ML-Agents repository (Unity-Technologies, 2017b). The world camera and directional light make up the other objects in the Unity Editor Hierarchy window. Both cubes cast and receive shadows, as do the walls of the platform. Figure 3.4 provides a visual of the rendered game.

[1] https://github.com/kncrane/CubeTrack

Target

The target cube constitutes what is called a non-player character (NPC) in games development, that is, any game object not controlled by a player, otherwise referred to as an AI character. This terminology is somewhat confusing in this context, since the AI character is the object *not* being controlled by a machine learning algorithm, and the 'player' character is *not* in fact being controlled by a human. For clarity, AI is referring to the wider definition of AI, which includes rule-based systems, and the *true* AI, the RL agent, is the player in this context. In CubeTrack, the target cube NPC is implemented as a 'NavMesh Agent' (Unity-Documentation, 2021). Adding a NavMesh Agent component to a game object a) helps to create characters which avoid moving obstacles, including other NPCs, and b) helps with pathfinding and movement within a restricted area called a NavMesh. The process of creating a NavMesh data structure from the scene geometry is called 'baking'. Once baked, the mesh describes the walkable surfaces of the game. From there, scripting the NPC is as simple as setting the desired destination point – everything else is taken care of by Unity. In CubeTrack, this destination point is randomly selected from 13 waypoints evenly spaced around the platform (see green dots in Figure 3.5). The waypoints are in fact small cylinder objects, with their mesh renderer turned off and their capsule collider turned off. They are each assigned the same tag, named 'RandomPoint'. The resulting behaviour is that the target cube will travel in a straight line to a random location on the platform, before pivoting and heading off in a new direction. The speed of the target cube remains constant, set to 4 after some experimentation. At the start of each episode, the target spawns in a random location on the platform, requiring a degree of search before track. The script handling target movement can be found in the CubeTrack repository [1] at [2].

Agent

The agent cube has a 'Rigidbody' component, meaning it is subject to gravity, will react to collisions, and will respond to forces applied by the scripting API (Unity-Documentation, 2021). The mass field of the Rigidbody component was set to 10 kilograms and the 'drag' (friction against the ground in this case) was set to 0.5 (how much velocity is lost per tick). The choice of value in both cases alters the way in which the game object responds to applied forces, therefore it was possible to experiment with different values and then observe the result in a demo run of the simulation. Values were chosen that gave sensible visual results, for example, the cube does not travel for too long before coming to a natural stop. The agent cube also has an additional child object – a camera pointing out from the front face of the

[2] Assets/Scripts/TargetMovement.cs

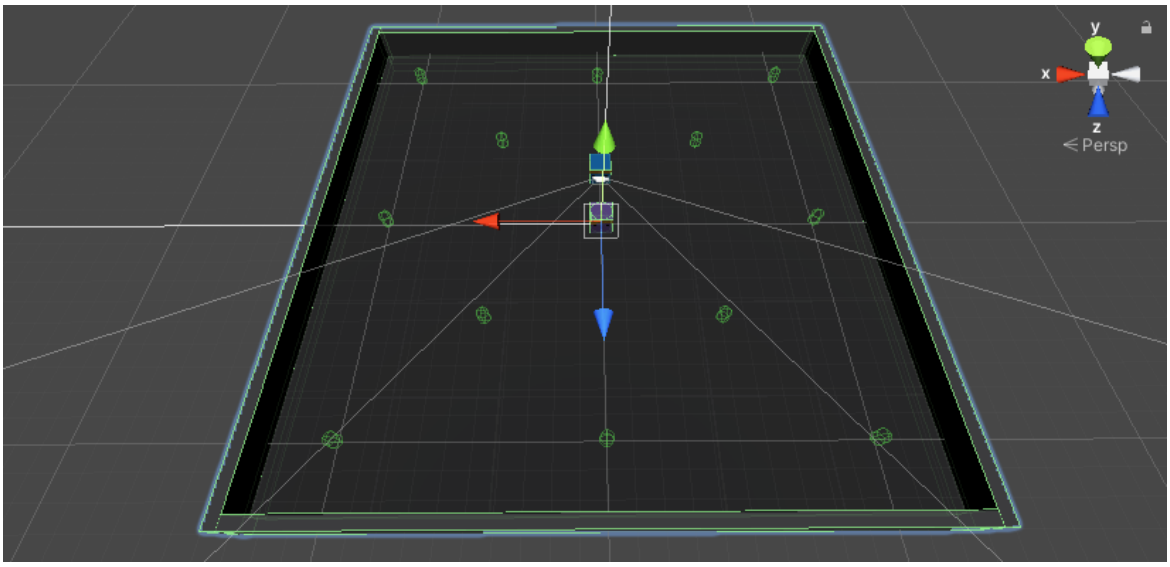


Fig. 3.5 Screenshot of CubeTrack in scene view. The green dots are the waypoints.

cube, with field of view (FOV) set to 60.

Importantly, the agent cube has several components made available after installing the ML-Agents C# SDK (the Unity package `com.unity.ml-agents`) via the Unity Editor's package manager registry. These components include the scripts 'BehaviourParameters', 'DecisionRequester' and 'CameraSensor'. The first has fields for declaring the type and size of the observation and action spaces, similar to how these would be declared in the initialiser of a Gym environment. The second defines the frame rate for action implementation (set to 1) and whether to implement an action repeatedly until the next decision is received (set to true). The third ensures the first person view (FPV) camera stays with the cube as it moves.

The agent cube also has a script 'AgentMovement', providing an implementation for the ML-Agents class `Agent` by extending functions `OnEpisodeBegin` and `OnActionReceived`. These functions define the Unity-side behaviour given Python-side calls to `reset` and `step` respectively. The `OnEpisodeBegin` function increments the episode counter, resets the agent's velocity to zero, and places both the target cube and agent cube in a random starting position and facing direction. The `OnActionReceived` implements the action decision (details provided in Section 3.3.3). With ML-Agents, the reward function code is also Unity-side, embedded in the `OnActionReceived` function. Details of the reward function are reserved for Chapter 5 (Section 5.3.2) alongside details of policy learning. Finally, the 'AgentMovement' script includes a check against the episode termination criteria (again, reserved for Chapter 5) within Unity's `FixedUpdate` function (called each tick of the physics

system), and a section of code allowing for keyboard control of the agent when the 'Behaviour Type' component is set to 'Heuristic'. The 'AgentMovement' script can be found in the CubeTrack repository [1] at [3].

3.3.2 Communication

Message passing between the Unity application and Python program is achieved over an open socket using the ML-Agents Python API. Details are extracted away from the user, however the GitHub repository documentation provides a high-level overview (see Figure 3.6). The box 'Python Trainer' refers to the algorithm implementations provided in the dedicated package `mlagents`, with single command-line utility `mlagents-learn`. This package interfaces with the package `mlagents_envs` (Python API), specifically the `UnityEnvironment` class residing in `environment.py`. Parts of this class resemble the Gym class `Env`, including functions like `step` and `reset`. However, the class also packs and unpacks messages to send to and receive from the external communicator residing in the Unity application. The Python side of this message passing (located in `rpc_communicator.py`) is achieved with gRPC (Wang et al., 1993), with Python as the server and Unity as the client.

Any agents with the 'behaviour type' field set to 'Default' within the 'BehaviourParameters' component will receive actions from the external communicator. It is possible for the same action to go to multiple agents (i.e. shared policy). The other two behaviour types are 'Inference Only' and 'Heuristic Only'. The former signals the agent to use a pretrained model. This requires the user to drag and drop a suitable file type into the 'BehaviourParameters' 'Model' field. Integrated models are made possible with the Unity Inference Engine, which uses compute shaders to run the neural network within Unity. The latter behaviour type signals the agent to interface with the `Heuristic` class, where the user can provide a rule-based policy or something like a keyboard listener to enable manual control.

3.3.3 Environment

Observations

As is good practice, the correctness of the algorithm and environment were first tested using what was deemed to be an easier problem – tracking with vector observations. The chosen vector observation had 12 elements: the x and z coordinate of the target's position, the x and z coordinate of the agent's position, the x, y and z coordinate of the target's forward vector

[3] Assets/Scripts/VisualAgentMovement.cs

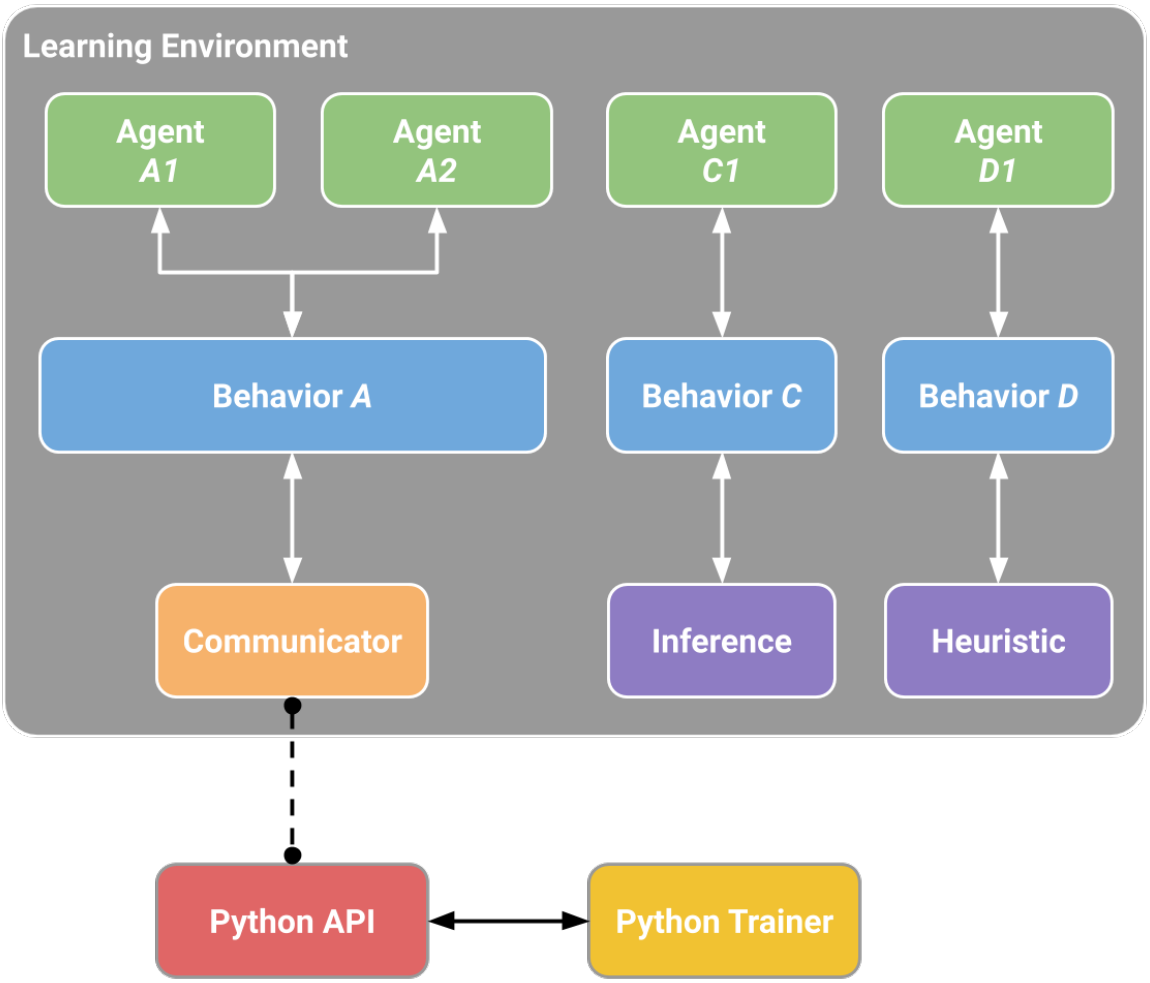


Fig. 3.6 Diagram of the ML-Agents framework, from Unity-Technologies (2017a)

(facing direction), the x, y and z coordinate of the agent's forward vector, the target's velocity, and the agent's velocity. Once the agent had successfully learnt to track with this explicit information, observations were switched to visual. Image observations in CubeTrack are 84×84 RGB images from the agent's onboard FPV camera, saved to file as a PNG.

Actions

CubeTrack is set up with a discrete action space. There are five levels: do nothing, forward, backward, turn left, and turn right. Since the agent cube has a Rigidbody component, it is possible to move the cube with an application of force. This is much more in line with the end goal of vehicle control than simply translating the agent's position. Algorithm 1 provides pseudocode for action implementation. For forward and backward, a positive and negative force were applied respectively to the agent's forward vector, using Unity's `AddForce()`

Algorithm 1 CubeTrack action implementation pseudocode

```
if action == 'forward' then
    transform.forward × forceMultiplier
else if action == 'backward' then
    transform.forward × - forceMultiplier
else if action == 'turn left' then
    transform.up × - torqueMultiplier
else if action == 'turn right' then
    transform.up × torqueMultiplier
else
    do nothing
end if
```

function. After experimentation, a multiplier of 500 was applied in both cases. For turn right and turn left, a positive and negative force were applied respectively to the agent's vertical y axis, using Unity's `AddTorque()` function. After experimentation, the multiplier for these turning forces was set to 250. These multipliers were selected on the premise that they had a visible effect on the cube, but were very slight. By mapping the action decision to a very subtle force application, the agent is afforded greater control; if the agent wants to make a very small movement, it can, and if it wants to accelerate it can select the same action decision for multiple consecutive time steps.

3.4 DonkeyTrack

The environment presented in this section is a novel environment, developed for the purpose of this doctorate research by modifying an existing open source project. Specifically, DonkeyTrack extends the Unity simulator 'SDSandbox' and the Python package `gym-donkeycar`, both developed by Tawn Kramer (2017; 2018). The code for DonkeyTrack is available at [4].

3.4.1 Simulator

The original simulation, 'SDSandbox', was developed to facilitate training a self-driving car in a racing context, with a particular focus on imitation learning. The version cloned was built with Unity 21.04.15. Shipping with a selection of indoor and outdoor racetracks, the simulator supports manual control, model training and model inference. The Unity project is advanced and difficult to unpick as an inexperienced game developer, however, with minimal changes, it has been possible to repurpose the project for training a self-driving car in the

[4] https://github.com/kncrane/gym-donkeytrack

context of VAT. Utilising the multiplayer functionality, a second car can be added to the simulation without any changes on the Unity side. The multiplayer functionality is set up to spawn one car per client-server connection, and so, by requesting two connections from the Python environment initialisation code, the scene dynamically loads with two instances of the car prefab. The data sent from and received by the Python server is with respect to the game object assigned to the client, and so both cars can send observations and receive control signals independently. To keep modifications to a minimum, one car can be sent action decisions randomly sampled from the action space. This car is the moving target. The other is sent action decisions output by a policy network. This car is the RL agent. Since the cars can also be configured from the Python end, the appearance of the two cars was made differentiable to aid evaluation. The agent car was given the white roll cage synonymous with 'Donkey Car', whilst the target was given a normal shell typical of RC cars, in blue. The target car does not require any of the data sent up from Unity to make an action decision, however this information is utilised in the reward function, and in `gym-donkeycar` the reward function is implemented server-side, not client-side. For this reason, the information exchange is left in place.

In this way, it is possible to train an agent for VAT without needing to rebuild the compiled Unity game binary. However, following experimentation, a handful of changes were introduced on the Unity side to facilitate learning. These changes were minor enough to not require any restructuring (or indeed deep understanding) of the complex Unity project. They were as follows:

- In order to reward the agent based on heading as well as distance (as was successful in CubeTrack), the Python environment requires the forward vector of both cars, as well as the position coordinates that are already sent up. Therefore, `transform.forward` was added to the `SendTelemetry()` message constructed in the C# script 'TcpCarHandler.cs'.

- When a second client connection is made, the default second instance of the car prefab is spawned 5 metres to the left of the first car, lining up the would-be racers on the start line. Additional racers, if included, are spawned behind. Spawning the second car to the left means the agent starts each episode with a relatively high reward signal (given the target is close) paired with an empty visual observation (given the target is outside of the agent's FOV). This data is not particularly helpful, especially to a naive agent. To optimise this, a modification was made so that the second car is spawned directly in front of the first, starting the episode with the agent in the optimal tracking position (see Figure 3.7). To do so, edits were made to the `CarSpawner` class in the C# script

Fig. 3.7 Screenshot of the DonkeyTrack simulation at the start of a new episode.

'CarSpawner.cs'. The `distCarRows` variable in the initialiser was changed from 5 to 15, and in the function `GetStartPosRot()`, the 'offset' on the x axis was commented out and the offset on the z axis was changed from negative to positive.

- The majority of the scenes offered by 'SDSandbox' include obstacles. This is problematic when the target object is driving randomly, as it can collide and become stuck. For this reason, the rebuild of the game binary was limited to the 'generated road' scene – a tarmac road in a sandy desert, free from any obstacles. The tarmac race track is not needed for VAT but was initially left alone. However, watching the agent, failure cases (i.e. losing the target) appeared correlated with crossing the race track. The layout of the scene meant that crossing the race track was an infrequent event when following a random target. As such, it was felt that the agent had insufficient data to learn the optimal actions in that scenario. To assist the agent and make the problem easier, the road generation boolean in this scene was set to False, producing a very homogenous environment of empty desert.

Producing a moving target from a 'player' addition was a quick and simple solution that works. However, at the point the user wants to incorporate obstacles, a better solution would

be to use a NavMesh Agent with built in collision avoidance, as in CubeTrack. Incorporating this new Game Object into the 'SDSandbox' simulator was going to be non trivial and so the simpler solution was taken in the first instance. An even better solution would be to use the multiplayer functionality, but feed the target car actions from a (separate) policy. This policy could be programmed, pretrained, or trained in parallel with an adversarial design similar to the one described by Zhong et al. (2018). Again, implementing this would be non trivial, especially when working with a third-party algorithm implementation. Both approaches offer feasible solutions, but would require more development time.

3.4.2 Communication

DonkeyTrack utilises the existing communication framework provided by `gym-donkeycar` – a client-server paradigm whereby the Python application is the server and the Unity application is the client. The network protocol is TCP – a protocol that guarantees the integrity of the data being communicated over a network (i.e. ensures all packets reach their destination by waiting for an acknowledgement before sending the next). This protocol requires an initial handshake to establish the connection. In `gym-donkeycar`, the design of the communication framework is to allow for continuous channels. That is, either process is able to send/receive at any point. Correct ordering of messages is maintained with a queueing system.

Given the requirement for message interpretation between client and server, the package utilises JSON, a universal and flexible language with support for object serialisation. All messages are JSON serialised into a UTF-8 string and then encoded to byte array, before being sent over the network. The server then needs to decode to JSON string, and load this into a JSON dictionary (the Python equivalent of a JSON object). Message fields with native data types (for example, the coordinates of the AUV) are directly accessible, whereas the image data needs additional unpacking. To collect image data, Unity renders the 2D image to a texture, and then encodes this render texture as a jpeg or png (i.e. a byte array). The server therefore has to reverse this encoding (with base64 decode), but may not cast to the right type, and so follows up with explicit byte array casting, reading the image data with Python Imaging Library, and casting to NumPy array.

3.4.3 Environment

Observations

DonkeyTrack supports both visual observations and feature vector observations; images encoded to a low-dimensional numeric vector using a pretrained autoencoder (the focus of the next chapter). If the environment initialiser is passed a filepath to a pretrained model, then feature vector observations are selected, otherwise image observations are assumed. If using feature vector observations, the observation space is declared as a `Box` space, with float values in the range $[-\infty, \infty]$ and shape $1 \times n\_z$. The value of n_z (the size of the learnt latent space) is determined at runtime by accessing the stored data of the pretrained model. If using visual observations, the observation space is declared as a `Box` space, with integer values in the range 0-255 and shape $w \times h \times 3$. The width (w) and height (h) are calculated dynamically at runtime using the resolution of the simulated camera, specified in the configuration file. The observation width mirrors the raw image width, whereas the height is reduced by one third. This is because raw images received by the client are cropped, removing the top third of the image which is entirely sykbox and therefore redundant information.

Actions

The DonkeyTrack action space is a `Box` space made up of two values, the first determining the steering angle and the second determining the throttle. Both are assigned the symmetric range [-1, 1]. The range [-1, 1] is advisory when using SAC because of the Tanh squashing function. Also, unlike racing, the ability to reverse in order to decelerate is important in VAT to maintain an optimal tracking distance and avoid colliding with the target.

3.5 SWiMM DEEPeR

The environment presented in this section is a novel environment developed from scratch for the purpose of this doctorate research. Whereas CubeTrack was developed independently, SWiMM DEEPeR was developed collaboratively with PhD student Samuel Appleby from the Game Engineering group (co-author of the IEEE Conference on Games 2023 paper in Section 1.4). Like the previous environments, there is both a Unity application and a Python application. As an experienced game developer, Samuel was responsible for coding the Unity application, guided by my requirements. This code is available at [5] in a public repository named SiM DEEPeR. The DokeyTrack Python application (including

[5] https://github.com/SamuelAppleby/SiM_DEEPeR

the implementation of server communication) provided a starting point for the Python-side development. Modifications to the Gym environment were handled by myself. Modifications to the communication network were handled by Samuel. This code is available at [6] in a public repository named SWiMM DEEPeR.

3.5.1 Simulator

The SWiMM DEEPeR project is built on Unity v.2021.3.20f1. Taking inspiration from footage captured during North Sea pelagic fieldwork, the simulation is of an open ocean environment. True to the real world footage, there is no sea bed, rock formations, vegetation, or marine structures, just open water. A depth-dependent colour gradient is implemented to simulate loss of light with increasing distance from the surface. Shaders help emulate a pseudo-realistic water surface. In-keeping with the other two environments, the tracking problem is reduced to single-object, distractor-free tracking, and so the only two game objects in the scene are the target (the dolphin) and the agent (the AUV). Both objects utilise third-party assets purchased from the Unity asset store. With the intention of sim-to-real transfer in future work, the AUV is a model of the BlueROV2 specifically, as opposed to a generic vehicle model.

An underwater environment is much more difficult to simulate with high physical and visual fidelity, given that water dynamics are not natively supported by Unity's physics engine PhysX. Whilst many hydrodynamic forces (e.g. water drag and currents) are reserved for future work, special attention was afforded to providing pseudo-realistic buoyancy.

Buoyancy modelling

The buoyant force enacting upon any submerged volume is directly proportional to the volume of fluid displaced:

$$F_\beta = \rho_f g V_f$$

where ρ_f = density of the fluid, g = gravitational constant, and V_f = volume of fluid displaced. According to Archimede's Principle, this can be simplified to

$$F_\beta = W_f$$

[6] https://github.com/SamuelAppleby/SWiMM_DEEPeR

where W_f is the weight of the displaced water. Upon full submersion, further increases to depth will have no effect on the resulting force, since the volume of water displaced is equal to the volume of the object. In a simulation, buoyancy can only ever be approximated, given that buoyant forces are acting on all volumes to the level of the atom. One accurate approach, proposed by Katara et al. (2019), is to apply a local force per every triangle of an object's 3D mesh which is below the surface. Any triangles partially submerged must be dissected into smaller triangles. The total buoyant force is calculated as:

$$F_{\beta}^{(t)} = \rho_f g \sum_{i=1}^{m} z^{(i)} S^{(i)} \vec{n^{(i)}}$$

where $z^{(i)}$ = distance from triangle centre to the surface, $S^{(i)}$ = surface area, $n^{(i)}$ = normal to the surface, and m is the submerged triangle count. Accurate buoyancy calculations such as this become increasingly computationally expensive with mesh complexity. In SWiMM DEEPeR, the BlueROV2 mesh has over 1.67×10^5 polygons, therefore performing such run-time calculations would be detrimental to system performance, and is not strictly necessary in this case given that physical fidelity is not the primary concern. The simplest implementation would be to apply a singular force at the centre of the mesh, however this would not provide the local forces required to stabilise an object on the surface of the water, resulting in a very unrealistic simulation. The chosen implementation is a compromise between realism and computational expense. Rather than apply a force per triangle, small objects we call 'floats' are provided at configurable positions around the mesh, and the net buoyant force is distributed at their local positions on the Rigidbody. The net buoyant force is assumed to be constant (as it would be once the vehicle was fully submerged).

AUV modelling

The selected AUV model is a 3D mesh of the Blue Robotics' BlueROV2 vehicle, designed by 3D Molier International and consisting of 1.67407×10^5 triangles. Like the agent cube and Donkey Car objects in CubeTrack and DonkeyTrack, the AUV in SWiMM DEEPeR displays realistic Netwonian physics through the use of a Rigidbody component, meaning the object responds to the effects of gravity, friction and other external forces. Setting the Rigidbody coefficients dictate the degree to which game objects are affected by these forces. The mass component was set to 1.4kg to match the weight of the real vehicle, provided in the vehicle's specification (BlueRobotics, 2022). As mentioned, realistic modelling of water drag and other hydrodynamic forces is reserved for future work. Reference was also made to the vehicle's specification when setting the attributes of the simulated camera, allowing for realistic camera rendering (sensor size, focal length etc). To avoid unnecessary complex

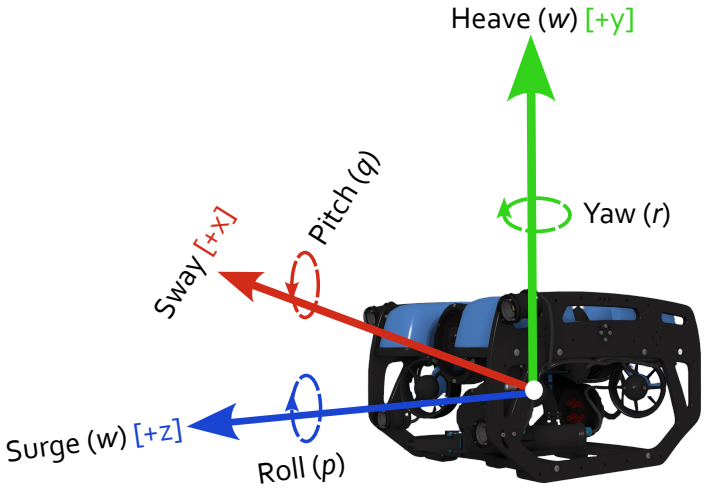
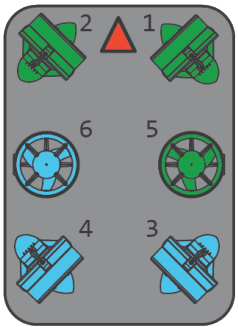


Fig. 3.8 BlueROV2 with labelled motion axes.

physics calculations, the AUV is given a 'Box Collider' component using the exact measurements from the vehicle specification. The simple box shape provides a rough approximation of the collision bounds of the complex mesh. It also guides the positioning of the 'float' objects included for buoyancy distribution.

Movement of the real BlueROV2 is achieved using fixed-position thrusters with both clockwise and counterclockwise propellers. Figure 3.9a provides an illustration of the BlueROV2's default 6-thruster vectored configuration. This standard configuration is unable to modify pitch and roll (von Benzon et al., 2022). The simulation stays true to this, offering four degrees of freedom: surge, sway, heave and yaw (see Figure 3.8). With the real vehicle, control signals are received from a joystick controller (see Figure 3.9b). These signals are sent from the joystick to a laptop, and from the laptop to the vehicle's onboard Raspberry Pi3, via the vehicle's tether. On the Raspberry Pi, commands are forwarded to the vehicle's Pixhawk Flight Controller across a UDP client/server connection, using the MAVLink protocol and Python library Pymavlink (Robotics, 2023). Here, the software ArduPilot (specifically, ArduSub) (ArduPilot, 2010) translates the signals into appropriate values for the individual thrusters. The ArduPilot project has nearly 400 contributors, and incorporates extremely sophisticated physics models to support features such as stabilisation. It would therefore make sense, when transferring the learnt behavioural policy to the real vehicle, to continue to utilise this software i.e. to take the outputs of the trained policy network (stored on the Raspberry Pi) and send them to the Pixhawk. This requires the RL agent to learn how to select the appropriate joystick signal, *not* how to select the appropriate actuator signal per



(a) BlueROV2 default thruster configuration.

(b) Xbox joystick controller for BlueROV2.

Fig. 3.9 BlueROV2 specifications.

thruster. For this reason, SWiMM DEEPeR does not need to simulate the individual thrusters, it need only simulate the resulting force and direction.

Dolphin modelling

The selected dolphin model is designed by Junnichi Suko[7]. This model consists of 3.5×10^3 triangles, and includes a rigged skeleton and pseudo-realistic animation patterns (see Figure 3.10). More expensive assets with higher visual fidelity were available, but these exploited the HDRP Unity rendering pipeline which does not have shader backward compatibility with the Unity built-in pipeline, which benefits from low graphical requirements. A simple Finite State Machine (FSM) is used to simulate pseudo-realistic dolphin movement. Similar to the target cube in CubeTrack, the dolphin moves toward an invisible waypoint, randomly generated within a threshold distance from the dolphin. Once the waypoint is reached, a new waypoint is generated. The dolphin model also uses a simulated FOV via cosine calculations with a threshold distance. If an object is detected within the FOV, the dolphin will soar above it, simulating realistic collision avoidance behaviour (when working with greater than two dimensions of control). A collision volume is included for when object contact still occurs. By giving the dolphin a collision volume but not a Rigidbody component, suitable movement and collision detection can be achieved without the computational expense that physical simulation incurs.

[7]https://junichistamesi.wixsite.com/my-personal-site

Fig. 3.10 Common bottlenose dolphin (*Tursiops truncatus*) model.

Configurability

Central to the design philosophy of SWiMM DEEPeR is server-side configuration. The `gym-donkeycar` package offers this to an extent, with the ability to configure the car object (body shape, colour, and floating player name) and onboard camera (resolution, encoding, FOV, offset, rotation etc.) by passing a dictionary variable of configuration options to the environment constructor. For anything outside of this (for example, changing the second car to spawn out in front rather than on the left), it was necessary to edit the Unity project directly and rebuild the application. This was difficult for someone outside the project development team, and particularly for a machine learning engineer with limited game development experience. With this in mind, SWiMM DEEPeR places a real focus on flexibility and data-driven design.

In SWiMM DEEPeR, the repository includes a single JSON file, populated with a default configuration for the user to edit. To aid reproducibility, the full configuration file is provided in Appendix A, showing the configuration used for the training run referenced in results Section 5.4. In terms of the AUV, it is possible to configure the motor, the camera, and the vehicle rig. Motor configuration includes stability force and stability threshold, linear thrust power and angular thrust power (force multipliers). Camera configuration includes resolution, sensor width, sensor height, and focal length. Configuration of the vehicle rig equates to configuration of the game object's mass, simulating the ability to add ballast weights to the rig of the real vehicle.

In terms of the dolphin target object, it is possible to set the minimum and maximum speed, the size of the dolphin, and the rotation offset. Booleans are provided for randomised movement (as opposed to straight), randomised starting positions, and spawning out in front of the vehicle. Individual waypoint axes (x, y, z) can be turned on or off, making it possible to limit the dolphin's movement to a single plane. In preparation for multi-object tracking support, target object spawning is highly configurable, with the user being able to set the

spawn timer, the spawn container ratio, the spawn radius, the maximum number of dolphins that can spawn at any one time, and the maximum number of dolphins that can exist in the world.

3.5.2 Communication

The communication framework in SWiMM DEEPeR takes inspiration from the project `gym-donkeycar`. It provides the same asynchronous (separate to the main thread) bidirectional communication, using a client-server paradigm and TCP as the chosen protocol. However, rather than allow for continuous communication, message passing is more strictly controlled (see Figure 3.11). Inspired by the uni-directional RL cycle, the client/server awaits a message from the other before sending a response. The cost of this is a reduction in step rate. The client cannot proceed with camera rendering, for example, until it has received an action, as opposed to the data sitting in a queue ready to go. The benefit, however, is that it guarantees observation/action handshakes i.e. the correct association between action and next observation. The strict design reduces network throughput and eliminates any risk of the server receiving a 'stale' observation, because the only time the client can send data is when it has received an action. The Python (server) side code for this communication cycle can be found in the SWiMM DEEPeR repository [6] at [8]. The Unity (client) side code for this communication cycle can be found in the SiM DEEPeR repository [6] at [9].

Connectionless protocols (such as UDP) provide a faster communication channel but do not offer the same data integrity guarantees. UDP was considered as an additional configurable option for the user to select when training on a local machine, given that in this scenario datagram loss or disordering would be much less likely. However, UDP is bottlenecked by the maximum datagram size (1500 bytes) supported on the physical layer (ethernet) and is therefore not able to send the amount of data required by this application; even very small images quickly exceed this threshold. Messages could be de(re)constructed across the network, however this process is costly and, at least in this context, there would need to be a re-request of datagrams if any message parts were lost (which is essentially what TCP does, very efficiently).

[8] gym_underwater/sim_comms.py
[9] Assets/Scripts/Network/Server.cs

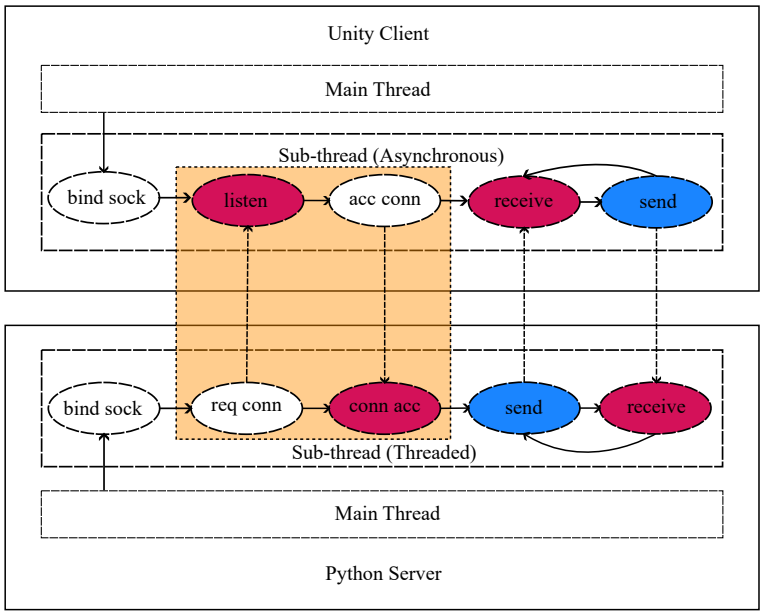


Fig. 3.11 Illustration of communication network. Red areas indicate busy waits, where the client(server) await data from each other, while blue indicates internal waits, where data is not sent until the internal conditions are met. Orange areas are required by TCP protocol.

3.5.3 Environment

Observations

In SWiMM DEEPeR there are three observation spaces: state vectors, feature vectors, and image observations. As in CubeTrack, there is the option to use vector observations of ground truth state information from Unity. Specifically, this is a 12-element vector containing the $\{x, y, z\}$ values of the AUV's position and forward vector, along with the same six values for the target object. Providing a state vector observation space facilitates environment and/or algorithm testing. As in DonkeyTrack, there is the option to use feature vector observations output from a trained autoencoder, the length of which is dictated by the number of learnt feature dimensions. Finally, there is the option to use image observations (8-bit pixel values), either at the resolution sent from the client, or with additional server-side scaling.

Both the Unity camera resolution and the resolution output by the server-side scaling are configurable. When using feature vector observations, images need to be passed to the encoder with the same dimensions used during training (for example, 64×64 for the models reported in Chapter 4). However, the real vehicle's low-light HD USB camera is 1920×1080 (BlueRobotics, 2023). One approach would be to set the Unity 'Camera' component's 'physical camera' attribute to have the same focal length and sensor size as the real camera,

and then scale the raw simulated images to the necessary encoder size within the Python application. This would exactly mirror the process on the real system. The problem is that simulation-based training requires data transfer across a TCP communication channel, and the larger the image the slower this transfer, significantly lengthening the already large wall clock time of training runs. Two experiments were conducted to compare a) client-side scaling versus server-side scaling, and b) rendering size. Both experiments utilise the *cosine similarity* metric for assessing the similarity ($\mathscr{S}$) between two images:

$$\mathscr{S} = \cos(\theta) = \frac{A \cdot B}{\|A\| \, \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Each pixel tuple is scaled between 0 and 1, and the metric ensures that each resulting score is normalised between 0 and 1, thus providing a percentage-based score that is easily comparable.

The first experiment revealed that, regardless of whether camera frames are resized client side or server side, the resulting images are near identical. At the time of running the experiment, a training and test dataset of example camera images had been rendered from the simulated AUV's onboard camera with a fully automated data collection script, ready for autoencoder training and testing in Chapter 4. This data collection script ensures uniformly distributed target poses (see Section 4.3.1). For this experiment, 1,000 images were randomly sampled from the test dataset. Images were resized from the simulated camera resolution (640×360) to the required encoder resolution (64×64) with two different methods – quad rendering on the client side, and interpolation techniques on the server side. Then, for each of the 1,000 images, the cosine similarity was calculated between the two resized outputs, using equation 3.1. Mean similarity was 99.998%. To ensure that this high degree of similarity was independent of starting resolution, the data collection script was re-ran, rendering images at 1920×1080 (the resolution of the inbuilt camera on the real vehicle). Again, 1,000 images were randomly sampled from the test dataset and resized to 64×64 with the two resizing methods. Mean cosine similarity was 99.997%. The conclusion was that, whilst the real system will need to perform resizing with Python on the vehicle's onboard computing device, for simulation-based training resizing can be done client side with Unity, minimising throughput across the network.

The second experiment revealed that the efficiency of simulation-based training can be further improved by rendering the low resolution directly, as opposed to resizing. The data collection script was modified to render the camera at three different resolutions with each

change to the target pose. This resulted in three training datasets and three test datasets, at the resolutions 1920×1080, 640×360 and 64×64. Image files were named numerically according to the order of collection, and so it was possible to randomly sample 1,000 images from one of the test datasets and then select the counterpart images from the remaining two test datasets. This ensured that the only difference across the three sets of 1,000 images was the resolution. Images rendered at 1920×1080 and 640×360 were resized to 64×64 with quad rendering on the client side. For both resized image sets, the cosine similarity was calculated between the resized image and the same image rendered at 64×64. Mean similarity was 99.286% with images resized from 1920×1080, and 99.289% with images resized from 640×360. The conclusion was that, whilst resizing will be necessary on the real system, for simulation-based training resizing can be avoided altogether. Rendering at a larger resolution requires greater computational power and therefore, since the results suggest that the comparative loss is negligible, it is preferable to directly render the low resolution.

Actions

The full action space is implemented as a five-part continuous action space. The first four actions correspond with the vehicle's four degrees of freedom given a default thruster configuration {x, y, z, r}, ($x, y, z, r \in [-1, 1]$), as illustrated in Figure 3.8. x, y, z represent linear force along the AUV's axis (i.e. lateral thrust or sway, vertical thrust or heave, and forward thrust or surge), and r represents proportional angular force around the AUV's y axis (i.e. yaw). The fifth action, d, is used to determine the AUV's 'depth hold' mode, M^d, according to

$$M^d_{t=0} = true,\ M^d_{t+1} = \begin{cases} true, \text{ if } d \geq 0.5 \text{ and } M^d_t \neq true \\ false, \text{ if } d < 0.5 \text{ and } M^d_t \neq false \end{cases}$$

The first term means that the simulation always loads with depth hold switched on. The second term means that depth hold mode is switched off when it is on and the value of d received from the server is less than 0.5, and it is switched on when it is not already on, and d is 0.5 or greater.

A 5D continuous action space presents a much greater challenge than the previous two environments, therefore SWiMM DEEPeR facilitates curriculum learning by making the number of control dimensions configurable. For example, if 2D is selected, then π will output values for z and r with all other values forced to zero. In this scenario, the user can

also limit the movement of the target to the *y* plane.

There is also greater consideration of *how* an action is implemented client-side. Real-time game simulations have core game logic running upwards of 60Hz, with physics operating at an even higher rate. Therefore, the communication network will always operate at a lower frequency than the simulation, since observations can only be sent at most once a frame. As the Unity client operates at a much higher frequency, we provide different behaviours for how to infer action messages:

- **maintain (default)**. Persists between frames. Upon receiving the next action, the previous is overridden.
- **onReceive**. Enacted for one physics tick only. No actions are applied until another message is received.
- **freeze**. Enabled for one physics tick. Upon sending an observation, the physics engine is frozen until the next action is received.
- **freezeMaintain (hybrid)**. A combination of the above. Each action persists for the frames between receiving an action and sending an observation, upon which the physics engine is frozen.

3.6 Summary

This chapter has introduced three novel data generation environments for using deep reinforcement learning to solve the problem of visual active tracking. CubeTrack offers a typical toy environment in the style of ML-Agents, DonkeyTrack offers a slightly more complex (but still low realism) racing car environment, and SWiMM DEEPeR offers a pseudo-realistic open ocean environment concentrated on the very specific problem of BlueROV2 control and cetacean filming. All three are a combination of a Unity application generating the data, and a Python application running the DRL algorithm. In addition, all three adopt a client-server architecture, with Unity as the client and Python as the server. In CubeTrack, aspects of the Gym environment such as the reward function sit client-side and server communication is handled by ML-Agents using gRPC. In DonkeyTrack and SWiMM DEEPeR, the entire Gym wrapper implementation sits server side, receiving raw game data from the client using TCP. In CubeTrack the action space is discrete and observations can be either image observations (i.e. camera frames) or ground truth, task-relevant state information such as positions and headings. In DonkeyTrack and SWiMM DEEPeR the action space is continuous, and there is

the additional option to use feature vector observations, integrating the autoencoder described in the next chapter as a preprocessing module for received image data. As a ground vehicle, DonkeyTrack addresses two-dimensional control, whereas SWiMM DEEPeR supports an action space of anywhere up to five dimensions. In Chapter 5, the suitability of these environments for agent training is demonstrated.

Chapter 4

Learning Task-Relevant Features from Image Data

4.1 Introduction

When machine learning algorithms take image data as input, the image is received as a matrix of pixel values per colour channel. The dimensionality of the data is therefore very high and directly related to the camera resolution. For example, the BlueROV2 camera has a resolution of 1920×1080, producing 6,220,800 pixel values for the algorithm to work with per data sample. Whilst DRL algorithms like DQN have demonstrated the ability to learn directly from pixel data (e.g. Mnih et al. (2015)), image-based DRL is notoriously difficult, requiring millions of training examples to solve a given task. In Section 2.4, this problem of sample inefficiency cropped up numerous times. For example, recall how off-policy learning improves sample inefficiency with the introduction of a replay buffer and the data re-use that this affords. Rather than tackle sample inefficiency from within the algorithm, this chapter takes one step back and looks at how to improve what is passed *in* to the algorithm.

Referring back to the RL terminology introduced in Section 2.3.1, the subtle distinction was made between the state of the environment (a complete description of the environment at a given time step), and an observation of the environment (what information the agent receives). Despite a camera frame constituting what is known as a 'partial observation' (the agent is not privy to all of the possible information available on the state), it is likely to have much higher dimensionality than the state it represents, for example, a position vector or velocity. This implies that the observation carries a lot of redundant information, requiring the DRL algorithm to filter out the task-relevant sensory information in order to make good

decisions. The policy network therefore has a dual task: mapping the observation to the underlying state, and then mapping the state prediction to an action prediction. Both of these tasks are highly complex tasks in and of themselves, for example, state estimation requires basic perceptual skills (the detection of edges, textures, shapes etc.) ahead of anything else. Therefore the dual-task is a big ask of what is typically a very shallow model when it comes to DRL policy networks. As has been discussed, DRL is notoriously data hungry, requiring a magnitude greater amount of data than supervised learning to converge. This presents a problem when training large networks with many parameters (Deisenroth et al., 2013), therefore it is common to opt for a policy network only two to three layers deep (see the default policies offered by Stable Baselines (Hill et al., 2018b)) so as not to have too large an optimisation space for the algorithm to have to explore with only indirect supervision. One way to remove some of the strain on the shallow policy network is to decouple representation learning (those more basic perceptual skills) from policy learning.

4.2 Related Work

A seminal paper on this approach from David Ha and Jürgen Schmidhuber (2018) likens this decoupling to the way in which the human brain uses internal models, built on prior experience, to help handle vast amounts of incoming sensory information. By separating out the two tasks, the first can be assigned to a neural network sufficiently large for learning rich spatial (and potentially temporal) representations, whilst the second can be assigned to a neural network sufficiently small for the credit assignment problem (learning the value of states and actions through experience). A smaller policy network means fewer parameters, reducing the search space for the policy optimisation algorithm, and a lower dimensional observation creates a smaller observation space, reducing the amount of required exploration and interpolation across unseen states. Furthermore, abstract representations can help toward a degree of invariance to perceptual variability, producing a policy that is more robust and able to generalise beyond the conditions of training. This is particularly important for simulation-trained policies intended for sim-to-real transfer. The hope would be that whilst the two domains present a large gap in raw data space, the gap is reduced or entirely closed in feature space.

The Ha and Schmidhuber 'World Models' paper presents an 8-layer ConvVAE as a 'vision module' for preprocessing data prior to reaching the policy network. Variational autoencoders were introduced in Sections 2.2.1 and 2.2.2. Another good example of autoencoder preprocessing is provided by Mattner et al. (2012), who demonstrate competitive performance

against an RL controller fed motor sensor data, in a real-world setup of the popular inverted pendulum task. Of particular relevance, a VAE has also been applied to the `gym-donkeycar` environment, on which the DonkeyTrack environment from Chapter 3 is based. This unpublished work is presented in the Medium article 'Learning to Drive Smoothly in Minutes' (2019). The author, Antonin Raffin, describes training a 64-dimensional version of the 'World Models' VAE implementation on 10,000 images collected from the simulated car's onboard camera, whilst running the `SD Sandbox` simulation in manual control mode. The trained VAE is then paired with SAC to achieve a high performing track racing policy in under 20 minutes.

All of these examples perform representation learning with a VAE, as is proposed here. These related works also provide examples of learning representations for the purpose of a downstream control task, as is proposed here. One limitation of this approach is that autoencoders are traditionally trained in an unsupervised manner, and as such, there is no control over what features are learnt, nor is it easy to interpret what has been learnt. It is therefore not guaranteed that the model will encode all of the information actually relevant to the downstream control task, or at least not without using a wide enough net (i.e. a large enough latent space), compromising on the benefits of dimensionality reduction. To address this limitation, the proposed solution utilises a specific type of representation learning referred to as state representation learning.

4.2.1 State representation learning

When representation learning and policy learning are not decoupled and the RL model is fed raw sensory data, the implicit process of feature learning performed by the RL model is supported by context-relevant feedback, i.e. the next state and reward issued by the environment. Explicit feature learning (as is the case when representation and policy learning are decoupled) can also be framed in this way, giving birth to an area of research commonly referred to as 'state representation learning'.

According to a review of the field, "state representation learning (SRL) focuses on a particular kind of representation learning where learned features are in low dimension, evolve through time, and are influenced by actions of an agent" (Lesort et al., 2018, p. 1). That is, rather than solely compress the observation, the idea is to learn a mapping from the observation to the underlying state. Böhmer et al. (2015) lay out some criteria for a good state representation, including being able to represent the true state well enough for policy improvement, allowing the value function approximation to generalise to unseen states, being low dimensional for

efficient estimation, and carrying the Markov property (all of the necessary information can be obtained from the representation of the current state, without looking at previous states). Some of these are a reframing of desirable characteristics for representations in general. For example, good support for value interpolation will come from a latent space which is smooth and continuous.

The review by Lesort et al. and a paper from Raffin et al. (2018) introducing the 'S-RL Toolbox' both provide a good overview of common approaches to SRL, including forward models, inverse models, and learning using generic priors. Many of these approaches can be implemented to involve an autoencoder, but in some way or other utilise system dynamics to constrain the learnt latent space. For example, given observation o_t, the encoding z_t can be used in combination with action a_t to predict $\hat{z}_{t+1}$, which is then remapped onto the pixel space to give $\hat{o}_{t+1}$. This is referred to as a forward dynamics model given the prediction of a future state. The prediction $\hat{o}_{t+1}$ is then compared to o_{t+1} and the pixel-wise error backpropagated through the entire model. The prediction, otherwise referred to as a transition, from z_t to $\hat{z}_{t+1}$ can either be fixed or parameterised (i.e. learnt). It can also be constrained, with many implementations opting for a simple linear constraint of the form $W * z_t + U * a_t + V$ (Goroshin et al., 2015; Van Hoof et al., 2016; Watter et al., 2015). An inverse dynamics model is then the reverse, predicting a_t from encoded observations z_t and z_{t+1}. Shelhamer et al. (2016) use the error between $\hat{a}_t$ and a_t as an auxiliary loss function within the policy gradients algorithm Asynchronous Advantage Actor Critic (A3C), for example. Many examples also exist of using a forward model and inverse model in combination (Agrawal et al., 2016; Duan, 2017; Zhang et al., 2018), including the well-known Inverse Curiosity Module (ICM) (Pathak et al., 2017).

A commonality between the methods discussed above is the reliance on action a_t. This poses several limitations. The first is the requirement to learn representations online, meaning the distribution of features is changing over time, adding yet another non-stationarity which could introduce instability into policy training. Secondly, if the autoencoder (or other feature learning model) is large and therefore slow to train, it would be preferable to train this model just once, offline, rather than train this model again and again across the potentially large number of RL training runs demanded by reward engineering and hyperparameter tuning. Thirdly, constraining feature learning with a_t only ensures that controllable elements of the environment are represented. The position of a target object, for example, may not be encoded, since the agent cannot directly act on it (Raffin et al., 2019). Clearly this would be an issue for VAT.

A huge benefit of training in simulation, as has already been mentioned, is the easy access to state information. With access to true state s_t, it is possible to take more of a supervised learning approach to SRL. A great example of this is a 2020 paper from Bonatti et al. on learning representations for aerial navigation. Interestingly, although the authors reference many of the seminal papers in the field, they choose not to use the term SRL and instead describe the approach as 'cross-modal', in line with a series of works using the same terminology (Aytar et al., 2017; Erin Liong et al., 2017; Ngiam et al., 2011; Spurr et al., 2018). This simple and elegant approach supports learning a regularised and task-relevant feature space fully offline. Bonatti et al. then take the encoder network from this trained autoencoder and use it in combination with imitation learning in order to achieve aerial navigation.

In this thesis, the reproducability of this approach is tested, applying the method to two new environments, DonkeyTrack and SWiMM DEEPeR. More importantly, the method is applied to the challenging problem space of VAT for the first time, in a new framework which utilises the constrained encoder network as a pre-processing module in front of DRL algorithm SAC. To the best of the authors knowledge, this is a novel solution which we refer to as T2FO (tracking with task-relevant feature observations). The next section provides a more detailed review of the original approach before presenting the methodology used here.

4.2.2 The 'cross-modal' approach

In Bonatti et al.'s cross-modal approach, training data is made up of two separate modalities: 1) raw camera data, sometimes referred to as the unsupervised data, and 2) a four-part vector of system states, collectively referred to as the 'pose' and sometimes as the supervised data. Specifically, the pose is the spherical coordinates and yaw of a large, red square on stilts, referred to as a gate. A series of these gates provides goals through which a drone must navigate. Different to the work presented in this thesis, Bonatti et al. opt for imitation learning from expert trajectories (human control) as their control method, but similarly adopt a sim-to-real approach, mastering the perception-control task entirely in simulation before attempting to transfer the learnt policy to a real-world set up of the same task. Prior to any policy learning however, the authors propose a method of offline representation learning, using the two data modalities, a VAE, and a number of MLPs, in a framework initially proposed by Spurr et al. (2018) in the context of hand pose estimation. The authors refer to the solution as a cross-modal VAE (CM-VAE), but essentially the proposed model is a constrained VAE.

As with a vanilla VAE, images are encoded into a $1 \times 2N$ vector of means and standard deviations, producing N normal univariate Gaussians, one per feature channel. A value can be sampled from each distribution to give latent vector z. Before passing the full vector to the image decoder, the first, second, third and fourth element of z are also passed, individually, to four small MLPs, each tasked with predicting one of the four system states. Predictions are compared to the ground truth value for each variable, producing four MSE loss values. These values are summed with the regular MSE and KL divergence term from comparing the image reconstruction with the input image. The sum provides a total loss value for the sample on which to compute and apply gradients.

The results show very clearly how the first four dimensions of the feature space represent the system variables they were allocated to predict (see Figure 4.1). The authors demonstrate this by singling out a feature dimension (i.e. setting all other features to zero) and decoding the feature at regular intervals. The decoded images of the red gates vary in size along the first dimension, horizontal offset along the second, vertical offset along the third, and rotation along the fourth, depicting a successful encoding of distance, azimuth, zenith, and yaw respectively. The remaining dimensions are much more abstract and it is difficult to understand what is being represented, as is often the case for *all* dimensions when using an unconstrained model. The authors describe the framework as providing implicit regularisation of the latent space, with the multi-task objective forcing the disentanglement of the first four elements of z.

Importantly, the control policy fed feature vectors from this regularised latent space was the best performing control policy. It markedly outperformed both the control policy fed feature vectors from the unregularised VAE, and the control policy fed raw images. It even slightly outperformed the control policy fed system state estimates produced by a direct regression model. In the real world, the control policy fed feature vectors from the regularised latent space achieved over three times the performance of the control policy fed state estimates from the regression model. The former was also stress-tested for transferability, with the authors evaluating the policy on unseen track configurations, in strong winds, in snow, and in scenarios of extreme visual variation to the simulated world in which it was trained (for example, a striped floor). The drone was still able to achieve over 1km of autonomous flight, demonstrating how regularisation can prevent the encoder from overfitting to the simulated data. The vanilla VAE and end-to-end policies are not reported on in the real-world setting, possibly because the lower level of performance in the simulation made it unsafe or unfeasible to evaluate in the real setting.

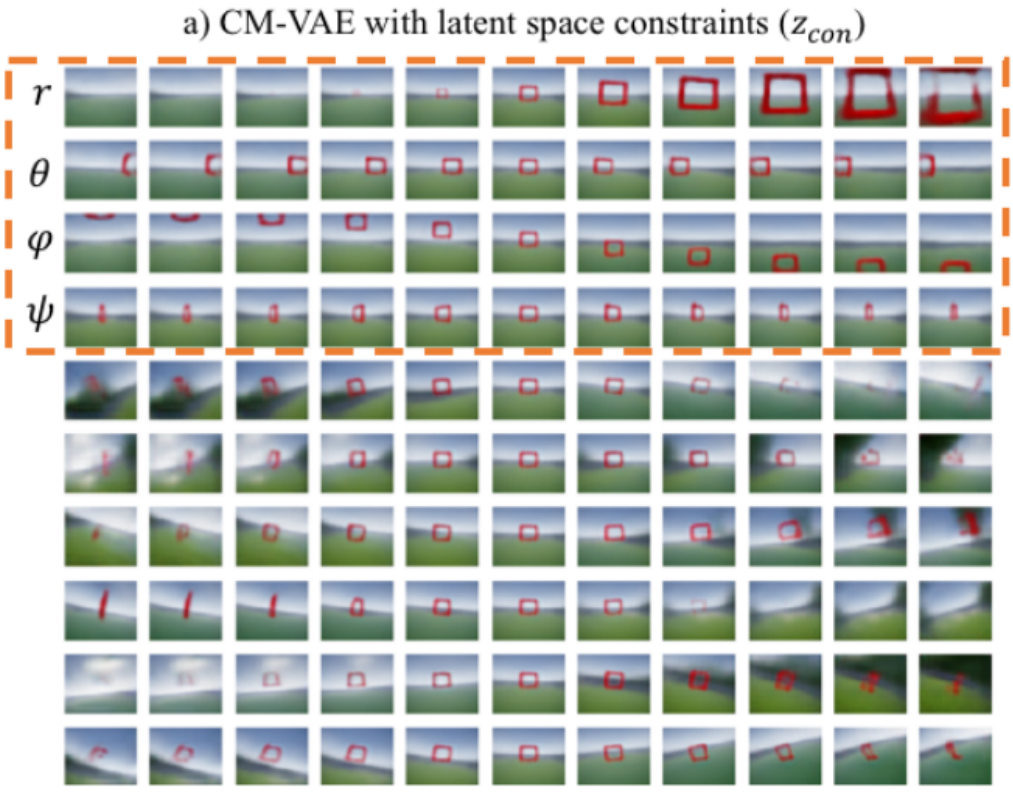


Fig. 4.1 Constrained latent space visualisation, from Bonatti et al. (2020).

4.3 Methodology

By far the greatest adaptation to the method presented by Bonatti et al. is the way in which this thesis utilises the encoder network post-training, using it to encode raw image observations of the DRL learning environment to task-relevant feature observations (the novel framework T2FO). However, the focus of this chapter is on the autonecoder training itself, reserving the wider framework for Chapter 5. In terms of the autoencoder training, the Bonatti et al. method required some modifications. A minor modification involved reducing the four-part system state vector to length three, since the ground vehicle in DonkeyTrack need not consider zenith (elevation), and AUV control in SWiMM DEEPeR was restricted to horizontal movement (reducing the problem complexity in the first instance). A much greater degree of modification came from a requirement to translate the Bonatti codebase from TensorFlow 2 to TensorFlow 1, so that the saved encoder network was compatible with the DRL policy and value networks and could be used in one unified framework; these networks utilise the widely used, well maintained DRL library Stable Baselines,available only in TensorFlow 1 or PyTorch at the time of writing. This migration was non-trivial, since

the library underwent a dramatic reinvention to meet the growing demand for higher-level programming. Changes included tight integration of Keras, default Eager execution, and Pythonic function execution. Migrating version 1 to 2 is supported by TensorFlow with the provision of automated scripts, however the reverse had to be coded manually. At the time of implementing, the SWiMM DEEPeR environment had not yet been developed, therefore Chapter 4 is largely in the context of the DonkeyTrack environment. Once the SWiMM DEEPeR environment was ready, the methodology reported here was replicated. Any differences are highlighted in Section 4.3.3 and the results are provided in Section 4.4.2.

4.3.1 Data collection for VAE training

The models described in the following section were trained on a novel synthetic dataset of 300K labelled images generated for this research. Images, here, are frames rendered from the simulated agent car's onboard FPV camera. Labels constitute a vector of three system state values: radial distance r, azimuth θ, and yaw ψ. These three values were used to describe the location and rotation of the target car **relative** to the agent car. The variables r and θ together make up the polar coordinates of the target car when treating the agent car as the origin. The variable ψ is then the rotation, in degrees, around the vertical axis, treating the yaw of the agent car as zero. The synthetic nature of the data afforded a high level of control over data collection. Initially, data was collected by driving the car manually with a 'teleoperation' script pulled from the Raffin and Sokolkov (2019) repository (originally implemented within the Rodriguez and Raffin (2018) repository). However, it soon became apparent that not only did this demand significant time and effort, it was also not the best way to generate a well balanced dataset.

Inspired by the Bonatti et al. codebase, data generation was then fully automated. The process was to first move the agent car to a random position and rotation in the simulation, and then move the target car, keeping the target within the agent's FOV. Once both cars were in place, the next step was to render the agent car's camera, save out the resulting image, and write the value for r, θ and ψ to a new row in the CSV file. Image files were given a numeric filename from 1 to 300K in sequential order, such that the filename matched the row number of the CSV file. Since the image was rendered from the agent car's FPV camera, it would have been possible to hold the agent car stationary and only move the target car, however changing the position and rotation of the agent car relative to the simulated light source introduced variation in shadows. This was important to simulate given that shadow variation would be present in real-world data. Algorithm 2 provides more detail on how this automated data collection process was realised. The code can be found in the DonkeyTrack

repository [4] in the file 'image_gen.py' at the root of the repository.

Selecting the range for r was experimental. A distance any smaller than 5 caused the cars to collide and be thrown into the air, producing a poor quality image. A distance of 30 was considered the maximum distance at which the target was still reasonably discernible in the image. As explained in Section 3.4.3, images sent from Unity were cropped, reducing the raw image height by one third. This is because the top third of the image is entirely sykbox and therefore redundant information. Generating a folder of 300K images, along with a 300k row CSV file, took approximately 42 hours wall-clock time on a Thinkpad T480s laptop with an Intel Core i7-855OU processor (4 cores, 8 threads). Whilst this is still a long runtime, no human effort was required beyond launching the script. In addition, the use of random sampling produced a balanced dataset, i.e. each state variable is uniformly distributed.

4.3.2 Feature learning

Feature learning within the cross-modal framework involved five DNNs: one image encoder network, one image decoder network, and three MLPs, one for each of r, θ and ψ. This was one fewer than the Bonatti et al. paper, since the three dimensional nature of aerial navigation involved the prediction of an additional state, zenith ϕ (elevation). The three networks allocated to state prediction were three instances of the same small network of two dense layers. The image decoder was also simplistic, made up of six transpose convolutional layers. The biggest network was afforded to image encoding (see Figure 4.2), using the 8-layer ResNet (He et al., 2016) architecture presented as 'Dronet' by Loquercio et al. (2018). Further details on all five models can be found in Tables 1 - 3 of Appendix B.1.

For training, the same hyperparameters were transferred across from the Bonatti et al. repository: batch size 32, $\beta = 8.0$, and an initial learning rate of 10^{-4}, optimised using the 'Adam' optimisation method (Kingma Diederik and Adam, 2014). The β value here represents the multiplier used in the loss function to control the weighting of the KL divergence term. The number of features (i.e. length of latent vector z) was set to 10, the same as the Bonatti et al. repository, but the number of epochs was reduced from 50 to 30, having seen the loss curve begin to plateau from as early as 10 epochs. Given the results were comparable with those reported in the Bonatti et al. paper, no hyperparameter tuning was deemed necessary.

The 300K dataset was randomly split into 90% training data and 10% validation data using the `Scipy` package and a seed of 42. The TensorFlow API `tf.data.Dataset` was used to support the input pipeline. The dataset object was populated both with the state data from

Algorithm 2 Automated data collection pseudocode.

1: **for** i in N, where N is the number of samples to collect **do**
2: Randomly sample Cartesian coordinates x_A and y_A, for agent position, where $x_A \in [-30, 30]$ and $y_A \in [-30, 30]$.
3: Randomly sample the yaw (in degrees), ψ_A, for agent rotation around vertical z-axis, where $\psi_A \in [-90, 90]$.
4: Express rotation $[\psi_A, 0, 0]$ as Euler angles, and then as Quaternion $(Q_A^x, Q_A^y, Q_A^z, Q_A^w)$.
5: Send the agent's Cartesian coordinates and Quaternion rotation to Unity.
6: Randomly sample radial distance, r, and relative azimuth, θ_R, where $r \in [5, 30]$ and $\theta_R \in [-\alpha, \alpha]$, and where

$$\begin{aligned} \alpha &= \frac{90 \times 0.7}{2} \\ &= 31.5. \end{aligned}$$

▷ Note: the numerator is the simulated camera's FOV multiplied by a value less than one to account for the fact that the FOV is a cone, not a square.

7: Calculate the target's Cartesian coordinates, x_R and y_R, relative to the agent, using r and θ_R as the distance and angle (in radians) in a polar coordinate system with the agent's position as the origin.

$$\begin{aligned} \theta &= \frac{\pi}{2} + \theta_R \\ x_R &= r \times \cos(\theta) \\ y_R &= r \times \sin(\theta). \end{aligned}$$

▷ Note: azimuth is measured from the x-axis but the onboard camera points down the y-axis, $\therefore \pi/2 + \theta_R$ indicates the target azimuth is w.r.t the agent's facing direction. Relative height $z_R = 0$.

8: Translate the target's relative coordinates to world coordinates (x_T, y_T, z_T) with

$$[x_T, y_T, z_T] = [x_A, y_A, z_A] + R(x_R, y_R, z_R)$$

where $R(\cdot)$ applies a rotation to the given Cartesian coordinates, and the rotation is the inverse of $(Q_A^x, Q_A^y, Q_A^z, Q_A^w)$.

9: Randomly sample the yaw (in degrees), ψ_R, for the target relative to ψ_A, where $\psi_R \in [-90, 90]$.
10: Calculate the yaw, ψ_T, for the target according to $\psi_T = \psi_A + \psi_R$ and express the rotation $[\psi_T, 0, 0]$ as Euler angles, and then as Quaternion $(Q_T^x, Q_T^y, Q_T^z, Q_T^w)$.
11: Send the target car's Cartesian coordinates and Quaternion rotation to Unity.
12: Request Unity to render agent's onboard camera and send image over the network.
13: Write r, θ_R, ψ_R to file to provide ground truth labels for target's radial distance, azimuth and yaw. Since θ_R was used in the context of a polar coordinate system, it represents an angle in radians and is converted to degrees before writing to file.
14: **end for**

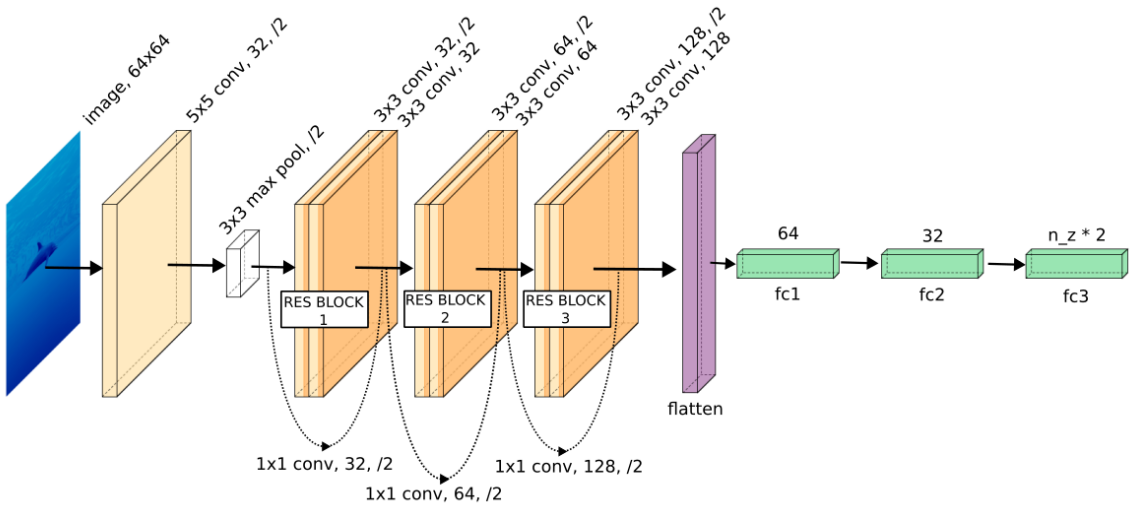


Fig. 4.2 Architecture diagram for the encoder network of the VAE. The architecture of the decoder is much simpler and does not require a diagram, but is detailed in Appendix B.1

the CSV and with the filepath to each image in the image folder. The large amount of data meant that a list of image arrays was too large to fit in memory, and so it was necessary to load images dynamically at runtime. It was therefore important to ensure that filepaths were ordered numerically by filename and *not* with character by character lexicographic sorting, as is default with string variables. This ensured that images were paired with their correct 'label' i.e. vector of ground truth state variables. All state variable values were normalised to be in the range [-1, 1]. At runtime, images were scaled down to 64×64 and normalised to be in the range [-1,1]. Each epoch of training was followed by an epoch of validation, with all loss values logged to Tensorboard. Model weights were saved every five epochs, as well as at the end of the training run.

Figure 4.3 provides an overview of the data flow per training iteration. As described in Section 4.2.2, images were first passed through the encoder network, returning a vector of means and standard deviations describing 10 univariate normal distributions. A random sample was taken from each distribution to form the 10-part latent vector z, passed as input to the image decoder. In addition, the first, second and third element of z were passed, individually, to each of the three MLPs. The MSE between model output and ground truth was then computed in each case – the image reconstruction with the input image, and the state prediction with the ground truth state data provided by the CSV. A weighted sum provided a total loss value to pass to the Adam optimiser.

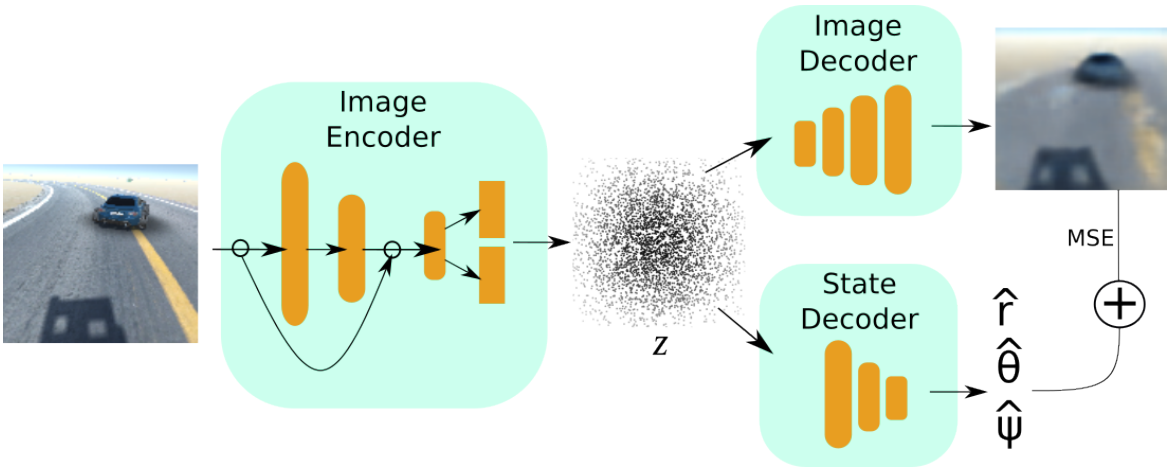


Fig. 4.3 Overview of the cross-modal method for learning a constrained VAE. The input is a 64×64 image. The output is a reconstruction of the same 64×64 image, and a prediction for target radial distance (r), azimuth (θ) and yaw (ψ).

4.3.3 Reproducing in SWiMM DEEPeR

Feature learning within the SWiMM DEEPeR environment was approached using the same codebase, retaining the three state dimensions (distance, azimuth, yaw). Although ultimately the SWiMM DEEPeR environment is intended for training 3D control, within the scope of this thesis policy learning is reduced to 2D control, training the AUV to follow the dolphin on a flat invisible y-plane (results reported in Chapter 5). Therefore, it was not necessary to reintroduce zenith (elevation).

Automated data collection was revisited, implementing the exact same procedure expressed in Algorithm 2, but with a Unity script (C#) as opposed to a Python script. This second implementation of automated data collection improves upon the Bonatti et al. (2020) implementation. Firstly, having the script client side removes the requirement to send messages across the network, drastically reducing the wall clock time for data collection. Secondly, wall clock time was reduced further by directly rendering 64×64 images, as opposed to rendering and saving out 120×120 images and then scaling them to 64×64 during autoencoder training. As discussed in Section 3.5.3, the resulting image is near-identical to rendering a higher resolution image and then scaling.

Automated data collection in SWiMM DEEPeR also differs with respect to the value ranges for the state variables. The r range is decreased from [5,30] to [2,20]. This is because r depends on the size of the world axes and the SWiMM DEEPeR world is much larger. In both the AirSim and DonkeyTrack environments, relative yaw values are sampled from a range of

180° (specifically, [-180,0] in AirSim, and [-90,90] in DonkeyTrack). In SWiMM DEEPeR, relative yaw values are sampled from a range of 360° (specifically, [-180,180]). For Bonatti et al., the smaller yaw range is permissible since the object (square red gate) has z+y-axis symmetry, i.e. it looks identical from the front and back. This is *not* the case for the car or dolphin target objects; there is asymmetry of head versus tail, and of bonnet versus bumper. In DonkeyTrack, this asymmetry is not important, since the target is always driving forward with only slight changes in direction, and so is only ever viewed from behind. However, in the application-focused SWiMM DEEPeR environment, this target behaviour was deemed unrealistic. Given that the dolphin target was controlled Unity side (with dedicated movement scripting) and not Python side (with random action space sampling), it was possible to have the dolphin express curiosity in the AUV. Occasionally the dolphin will loop round and travel directly toward the AUV. This produces front-facing camera images, hence the need to sample yaw from the full 360° range.

The same hyperparameters were carried across from the DonkeyTrack environment, which in turn utilised the Bonatti et al. hyperparameters: 10 feature dimensions, batch size 32, $\beta = 8.0$, an initial learning rate of 10^{-4}, and Adam optimisation (seed=31). The number of training epochs was raised from 30 back to the 50 reported by Bonatti et al.. Training took 45 hours and 25 minutes on a Razer Blade Pro laptop (Intel Core i7-10875H CPU @ 2.30GHz - 5.10 GHz, 64GB DDR4 2933MHz RAM, 265GB disk space).

4.3.4 Domain transfer experiment

In the opening to Section 4.2, it was explained that one benefit of using a feature vector as an environment observation when training an RL agent is that the feature vector presents a much more abstract representation of the raw data, providing the agent with a degree of invariance to perceptual variability. This is of course crucial when taking the sim-to-real approach to policy training, given the requirement to transfer the learnt policy from one visual and physical domain to another. Whether or not this claim holds true can be explicitly investigated prior to any attempt to control the real vehicle with a simulation-trained policy.

When comparing the real world and simulated environments, no matter how good the simulation, there will always be both a visual and a physical domain gap (hence why the term domain transfer is used to describe the challenge of performing real-world inference on a simulation-trained model). With a small enough visual domain gap, a vision-based, simulation-trained policy will map real world observations to the same action decisions as (contextually) identical simulated observations. This would address at least half the challenge

of successful sim-to-real policy transfer; whether or not the optimal action decision is the same in both domains depends on the physical domain gap. Ignoring, for now, this latter problem, an important question becomes: do real world raw image observations map to the same place in latent space as their simulated raw image counterparts? This section details the methodology of a proof-of-concept experiment designed to assess whether (and how well) image encoding closes the visual domain gap.

In order to run this proof-of-concept experiment, a small subset of 42 simulated images were manually selected from the DonkeyTrack 30,000 image test dataset. Images were selected such that there was good variation in target distance, azimuth and yaw, as well as variation in shadow appearance. For each simulated image, a real world mock-up image was collected using a standard blue RC car as the target car, and the official Donkey Car model purchased from Robocar Store [1] (see Figure 4.4a). The Donkey Car is fitted with a Raspberry Pi 4 and a wide angle Raspberry Pi camera. On a host machine, it is possible to SSH into the Raspberry Pi, instruct the camera to capture an image, and save to file. With the simulated image displayed on a tablet for reference, the position and rotation of the blue RC car was experimented with until the image captured by the Donkey Car's onboard camera was of similar likeness (see Figure 4.4b). This process was made slightly more difficult by the wide angle lens. Images were manually cropped following collection, both to alter the image ratio from rectangular to square before scaling, and as an easier means of getting the target car the right distance from the edges of the frame. Once real world data had been collected, both the real and simulated image sets were passed through the trained VAE. All images were scaled to 64×64 and pixel values were normalised to the range [-1,1] prior to passing through. State values were not collected given that the model is trained to predict Unity coordinates, which do not map to real world measurement units in any straightforward way.

A total of four methods were used to assess the similarity of simulated and real encodings. Firstly, if real world images are being mapped to the same place in latent space, then real world reconstructions should present the same quality as simulated reconstructions. Secondly, if real world images are being mapped to the same place in latent space, then real world state predictions should remain accurate, presenting a similar Mean Absolute Error (MAE). Since real world state values were not collected, state prediction error for real world image encodings was calculated using the state values for the associated simulated image as the ground truth. This introduces somewhat of a limitation – if the simulated and real images are not identical in terms of properties such as the car-to-image pixel ratio, then any error

[1] https://www.robocarstore.com/

(a) The real ‘Donkey Car’, from the Robocar Store

(b) Photograph of the data collection set up.

Fig. 4.4 Collecting real world image data for domain transfer experiment.

in prediction could be attributed to this difference as well as to any difference in encoding. However, it was still interesting to inspect the model’s performance on this front; even if the model was not able to accurately predict the value to the nearest unit, for example, would it at least respect *relative* distance, azimuth and yaw, i.e. assign a smaller distance value to an image with the car closer to the camera than another image.

Thirdly, image sets were passed through the encoder network to produce sets of latent vectors. The Euclidean distance between each latent vector pair was then calculated with

$$EU(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ... + (a_n - b_n)^2} \tag{4.1}$$

where a is the latent vector of the simulated image, and a_n is the value of this latent vector's n^{th} dimension, and where b is the latent vector of the real world image counterpart, and b_n is the value of this latent vector's n^{th} dimension. Euclidean similarity is then defined as

$$SimilarityEU(a,b) = \frac{1}{(1+EU(a,b))} \tag{4.2}$$

Finally, in order to visualise these distances, the 1×10 latent vectors were reduced to two dimensions with T-distributed stochastic neighbour embedding (t-SNE) – a popular non-linear dimensionality reduction technique, sometimes referred to as manifold learning (Van der Maaten and Hinton, 2008). Whereas traditional, linear dimensionality reduction techniques such as Principal Components Analysis aim to keep embeddings of dissimilar samples far apart, t-SNE aims to keep embeddings of similar samples close together, which is more effective for high-dimensional data that lies on or near a low-dimensional, non-linear manifold. The similarity of two samples, x_j and x_i, is the conditional probability $p(j|i)$ that x_i would pick x_j as its neighbour, if neighbours were picked in proportion to their probability density under a Gaussian centred at x_i. This approach turns out to be better than other nonlinear dimensionality reduction techniques at creating a single map which captures both the local structure of the data, and the global structure at many different scales.

4.4 Results

Models were evaluated against a separate 30k dataset, generated with the same automated script described in Section 4.3.1. Model performance was assessed with the same metrics and visualisations presented by Bonatti et al., to allow for direct comparison. Firstly, the quality of the learnt latent space was evaluated via a visual assessment of image reconstructions. Figure 4.5 provides 50 randomly selected input images from the 30K image test dataset described in section 4.3.1, together with their corresponding reconstructions (decoded outputs). Scanning the image tiles left to right, starting in the top left corner of the tile grip, image tiles are presented in pairs, with the input image on the left and the decoded output on the right.
Whilst reconstructions are blurry in comparison to the original image, in every single case the meaningful content of the image is captured accurately. Both the target car and the shadow produced by the tracking car is reconstructed in the right position, at the right angle, and to the right scale, even when the target has an extremely small pixel footprint on the im-

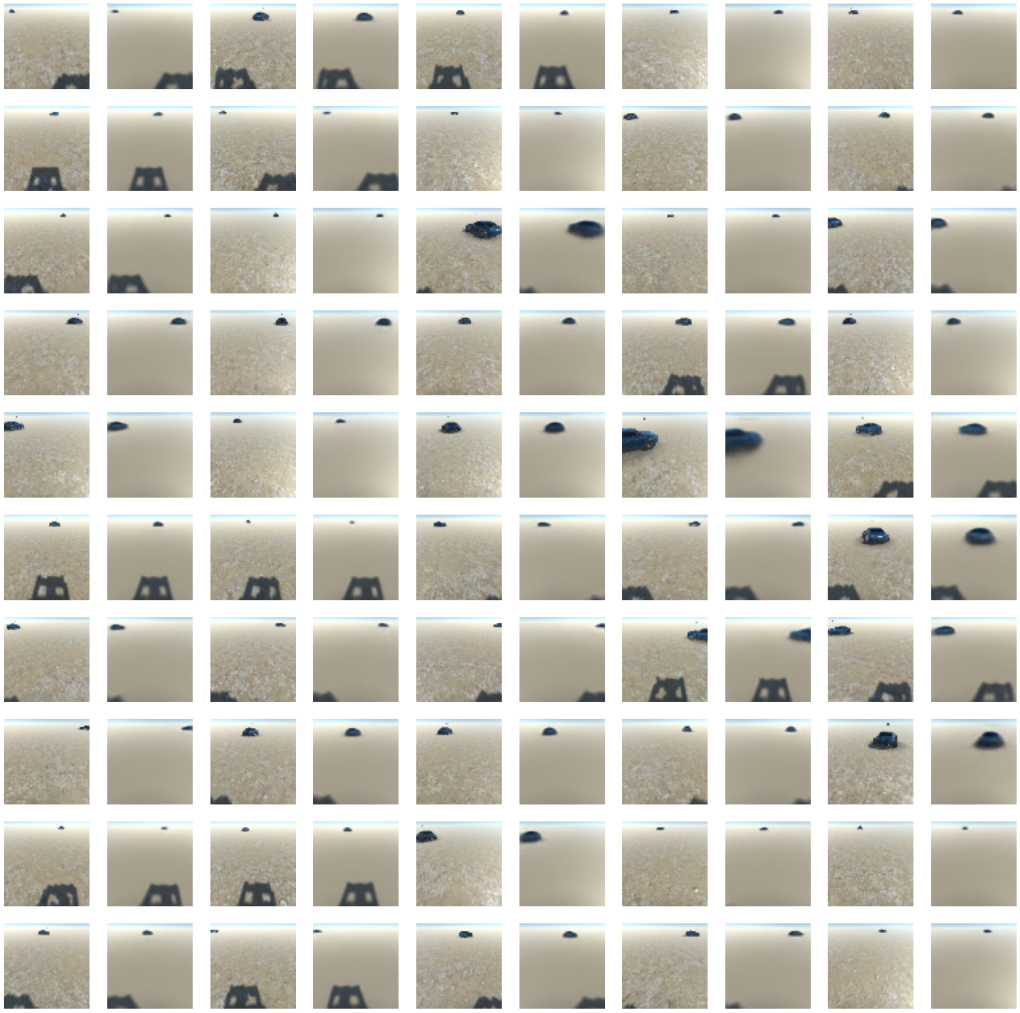


Fig. 4.5 Fifty randomly selected input images from the 30K image test dataset, together with their corresponding decoded output image. Starting with the top left image tile and scanning across the row, images are displayed in pairs, with the input image on the left and the decoded output on the right.

Env	r [m]	θ [°]	ψ [°]
AirSim (Bonatti et al.)	$\mathbf{0.39 \pm 0.023}$	2.6 ± 0.23	$\mathbf{10 \pm 0.75}$
DonkeyTrack	0.46 ± 0.003	$\mathbf{0.65 \pm 0.004}$	12.09 ± 0.097

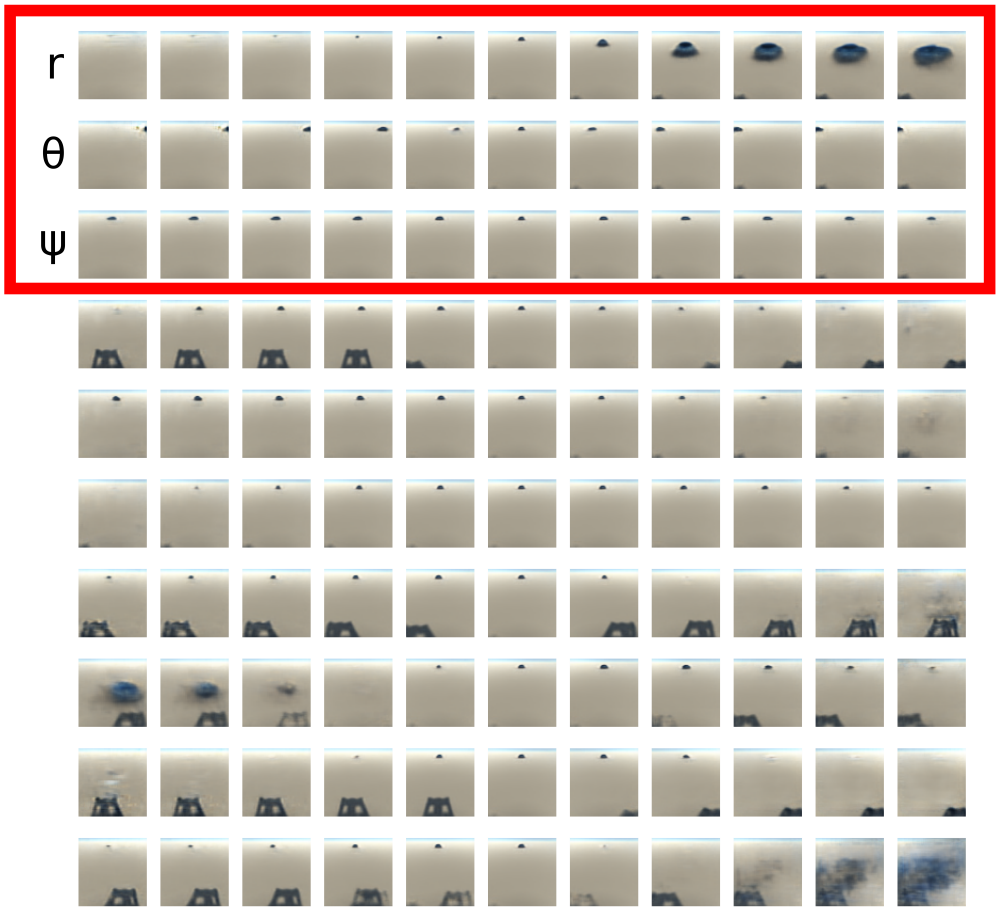
Table 4.1 Table of mean absolute error for predicted target car distance (r), azimuth (θ) and yaw (ψ) against ground truth values.

age. This qualitative analysis is supplemented with a quantitative analysis of state predictions.

The values in Table 4.1 represent the mean absolute error and standard error for the predicted distance, azimuth and yaw of the target car. The same results from the Bonatti et al. paper are presented above for reference (the lowest value across the two environments is in bold). As can be seen, results are comparable (within 3 units of each other), with a slightly higher error for distance and yaw, but a lower error for azimuth. As detailed in Section 4.3.2, the models were trained for 30 epochs and not 50 as in the original paper. It is possible that the additional 20 epochs would further lower the error for r and ψ to that reported by Bonatti et al. but a) the margins are already so small and b) the prediction task is only a secondary task used for regularisation purposes, and so it was deemed an unnecessary use of compute resources.

Of arguably greater importance is evidence for the disentanglement of the first three feature channels. Figure 4.1 (taken from the Bonatti et al. paper) provides a useful visualisation of the latent space. The same visualisation was generated for the DonkeyTrack data (Figure 4.6a). In this figure, all image tiles are image reconstructions output by the decoder, images are not presented in pairs as in Figure 4.5. Instead, each row is dedicated to one of the 10 feature channels. Per row, one of the 10 feature channels has a non-zero value and the remaining nine feature channels are set to zero. Each successive column left to right provides a decoding of a different value for the feature of interest, starting with -0.02 on the far left and increasing the value by equal increments up to a value of 0.02 for the image tile on the far right. It was necessary to experiment with decodings in order to find the appropriate feature value range, that is, the feature values representing the extremes of the property (e.g. distance).

The first row represents the feature channel passed as input to the prediction model for radial distance. As the value for this feature is increased, the decoded image goes from not containing the target car at all, to containing an increasingly large target car. This feature therefore successfully encapsulates the concept of target distance. The second row represents



(a) Mural of learnt latent space decodings, with each row representing a feature from z in isolation

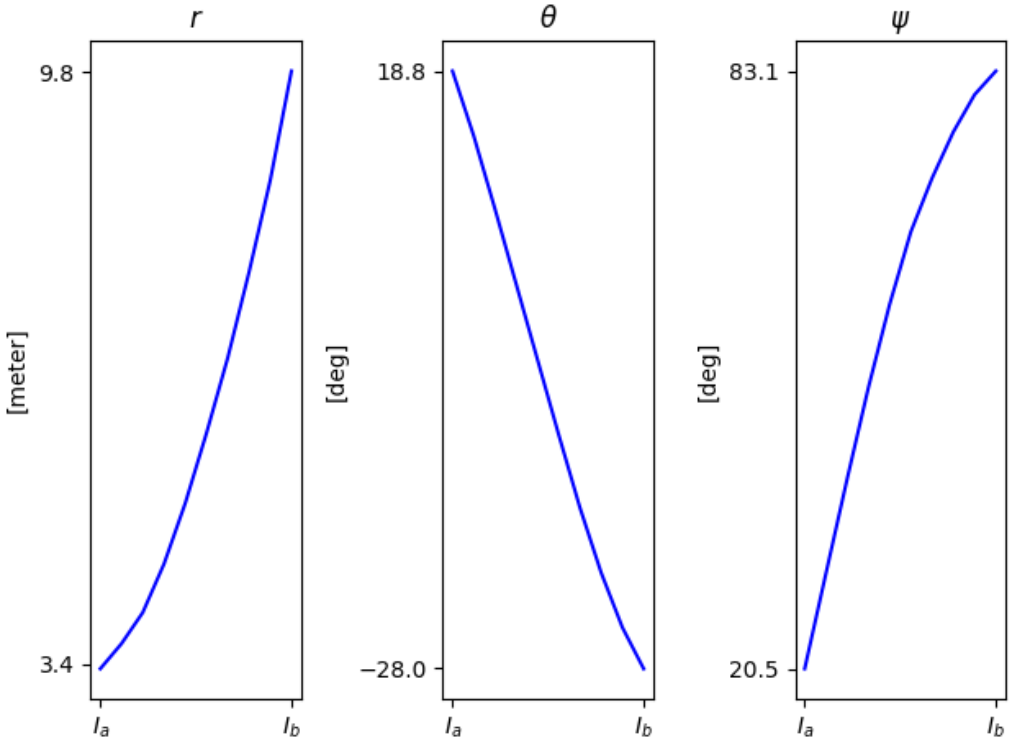
(b) Repeat of third row (ψ) but zoomed in, i.e. with $z[0]$ (r) increased from zero to 0.02

Fig. 4.6 Visualisations of the constrained VAE latent space decodings, providing insight into what the model has learnt and what the model is representing per channel.

the feature channel passed as input to the prediction model for azimuth. Similarly, the decoded images provide clear evidence for a learnt representation of azimuth. When the value of this feature is low the target appears on the far right of the image, moving across to the far left of the image as the value is increased. Both of these channels appear exactly as they do in the visualisation from the Bonatti et al. paper. The third row of Figure 4.1 is skipped, since no representation of zenith was learnt. Instead, the third row of Figure 4.6a

represents the feature channel passed as input to the prediction model for yaw. This state variable is harder to discern in Figure 4.6a compared to Figure 4.1 (row 4) because of the car's distance and solid shape – the space in the centre of the red gate makes it easier to appreciate rotation. It can just be made out that the car changes from side-on, to front-on, to side-on, however Figure 4.6b provides a clearer visualisation of the same feature channel, but with the 'distance feature' ($z[0]$) turned up. The remaining seven rows represent the unconstrained feature channels. These channels encode other information deemed relevant by the model, but it is not clear what features they represent, other than shadow shape and size.

(a) Synthetic images decoded from latent space interpolation



(b) Synthetic state values decoded from latent space interpolation

Fig. 4.7 Visualisations of the constrained VAE latent space interpolation. Image and state decodings from ten original latent vectors, generated by interpolating between the encoding for image a and the encoding for image b

Finally, in the brief introduction to variational autoencoders provided in Section 2.2.2, two desirable qualities were mentioned: completeness (all points in the latent space produce meaningful decodings) and continuity (two points close together should produce similar content once decoded). The interpretable first three rows of Figure 4.6a clearly demonstrate continuity. To further demonstrate the continuity and smoothness of the latent space manifolds, two points in the space are selected at random, by randomly selecting two images from the test dataset and encoding them. The resulting vectors z_a and z_b are then interpolated to produce 10 new vectors, by incrementing z_a by one tenth of the difference between z_a and z_b element wise. Figure 4.7 shows the results of decoding these evenly spaced new points in the latent space, both in terms of the image reconstruction and the state prediction. The images and values are described as 'synthetic' because there was no input image on which the latent vectors were based. Both the images and state values transition smoothly from a to b, as they do in the results presented by Bonatti et al..

4.4.1 Ablation experiment

An ablation study was conducted to properly decipher the contribution of the cross-modal approach to feature learning. This involved dropping the cross-modal element and training unconstrained variational autoencoders on the same dataset and with the same hyperparameters. Firstly, an unconstrained variant was implemented using the exact same architecture for the encoder (the 'Dronet' ResNet) and decoder (six transpose convolutional layers) but with the secondary objective omitted, i.e. latent vector z was passed through the image decoder but not the state prediction MLPs. The objective function therefore returned to the original two terms: the reconstruction error and the KL divergence. Image reconstructions produced by this model were just as accurate as image reconstructions produced by the constrained VAE, as can be seen by the example reconstructions in Figure 4.8. Again, reconstructions are blurred but the shape, size, position and rotation of objects is entirely faithful to the input image.

Figure 4.9 provides an interpolation between the same two images used to produce Figure 4.7a. The two figures are not dissimilar but the transition from left to right in Figure 4.9 is not as smooth, suggesting a lower degree of continuity in this learnt latent space. However, the difference between the two latent spaces is most apparent when viewed channel-wise. The 'z mural' for the unconstrained VAE, presented in Figure 4.10, is less interpretable; variations in target size, offset and rotation can be seen but are much more mixed across channels. It is perhaps most appropriate, therefore, to describe the contribution of the cross-modal approach as a disentangling of features, as they do in the Bonatti et al. paper. Still, it is not possible to

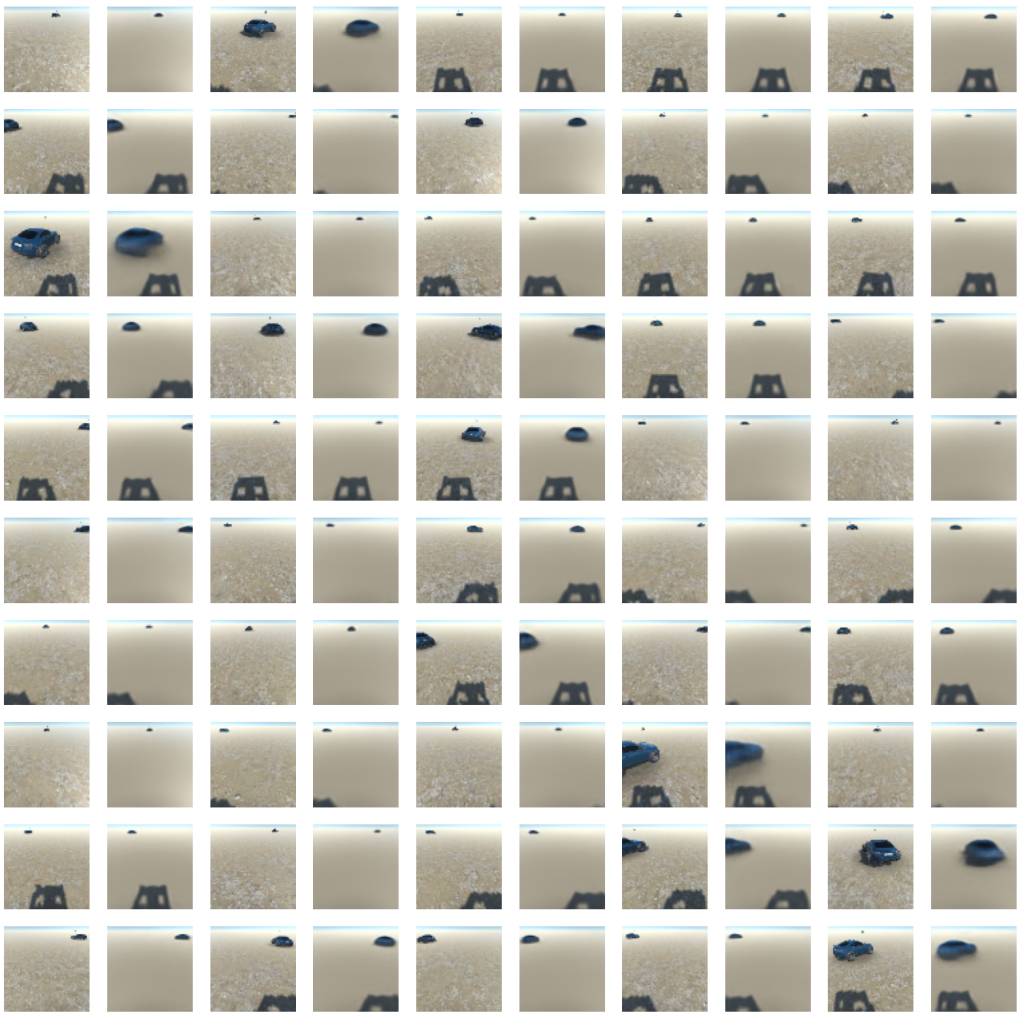


Fig. 4.8 Example image reconstructions from unconstrained VAE. Starting with the top left image tile and scanning across the row, images are displayed in pairs, with the original image on the left and the reconstruction on the right.

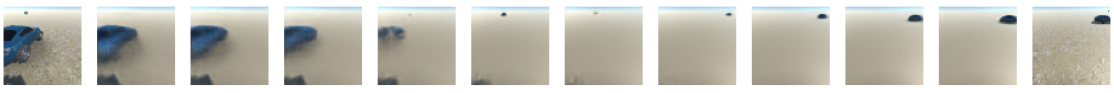


Fig. 4.9 Visualisation of unconstrained VAE latent space interpolation.

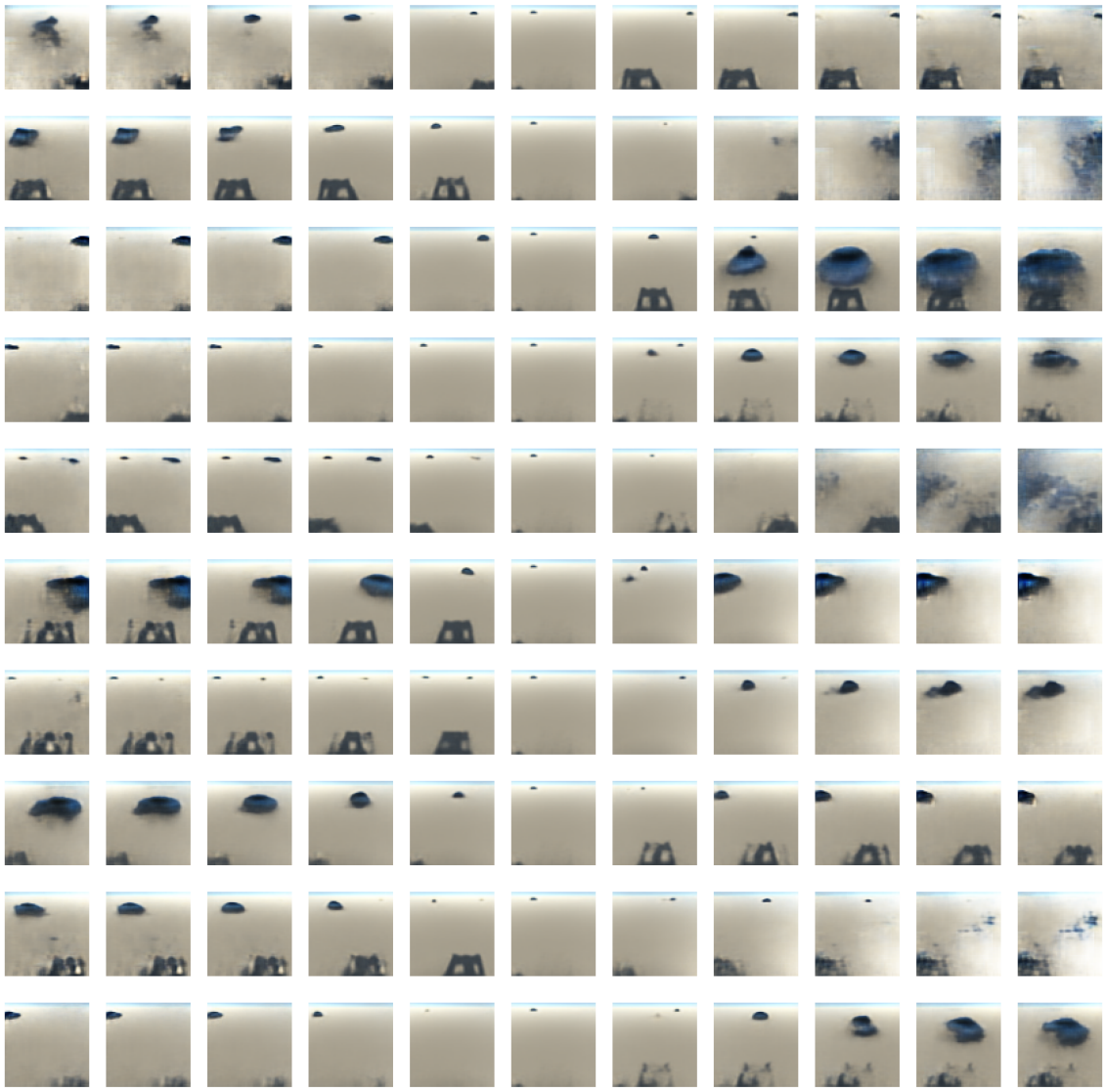
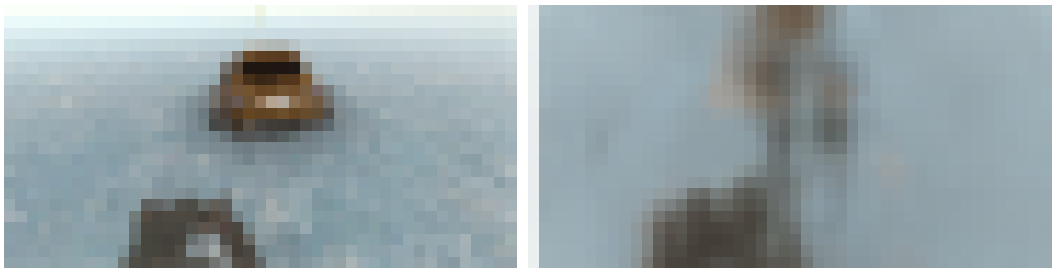


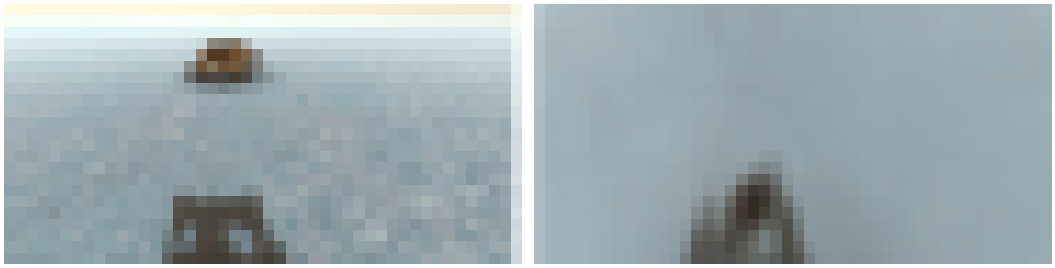
Fig. 4.10 Visualisation of unconstrained VAE latent space decodings, providing insight into what the model has learnt and what the model is representing per channel.

say whether one set of learnt representations is definitively *better* than the other. Whether constrained representations improve policy learning is explored in Chapter 5.

Prior to adopting the cross-modal approach, feature learning had been attempted using the VAE architecture from the Ha and Schmidhuber World Models paper, as was done in the 'Learning to Drive Smoothly in Minutes' (2019) repository. Importantly, this architecture does not include residual blocks. The performance of this model in the DonkeyTrack environment was poor. Not only did the reconstructions lack accuracy in terms of shape and size,



(a) An example image and reconstruction with a near (large) target



(b) An example image and reconstruction with a far (small) target

Fig. 4.11 Example image reconstructions from VAE trained on smaller dataset (10k images). *Note:* the environment is the same, the red and blue colour channels have been switched due to an OpenCV specificity

if the target car was distant and therefore had a small pixel footprint, the car object would not be encoded and the image reconstruction would be entirely background (see Figure 4.11).

The consensus was that, in these cases, the difference in reconstruction error between a reconstruction containing the target car and a reconstruction not containing the target car was too small to encourage target encoding. Attempts were made to correct this with a weighted reconstruction error, using OpenCV to work out the target to image ratio. The error was weighted with the inverse of this ratio, inflating the error in the case of small targets, but the issue persisted. When the issue of omitting small targets did not present in later work with the cross-modal trained VAE, there was a question of whether the latent space constraint was providing a resolve. Interestingly, Bonatti et al. suggest just that. However, this is not strictly true because small targets were not omitted by the uncontrained VAE (Figure 4.8). It was not clear whether the resolve was coming from the deeper ResNet-8 encoder and its use of residual blocks (compared to the 4-layer CNN World Models encoder), or the increase in data volume (training on 300k images for 30 epochs, compared to 10k images for 50 epochs). To answer this, the World Models implementation was revisited and ran on the 300k DonkeyTrack dataset. Figure 4.12 provides the reconstruction results.

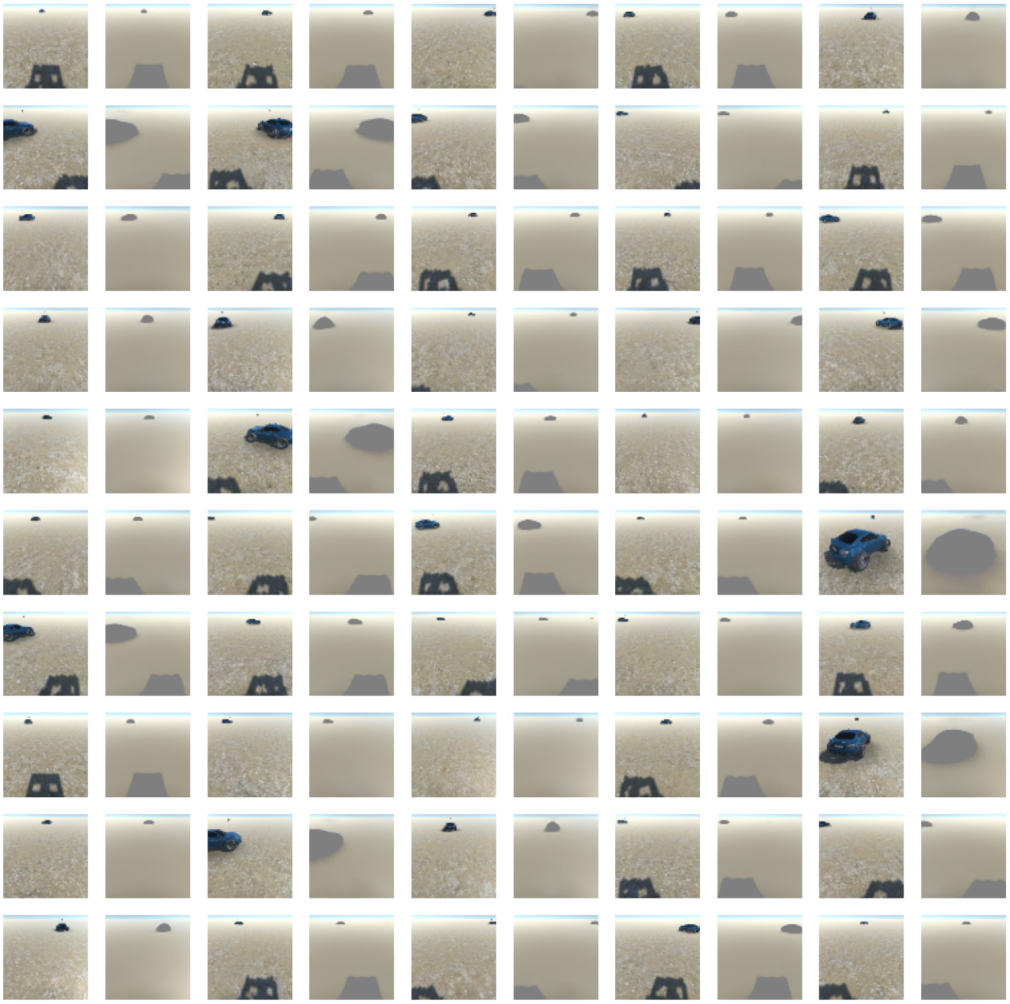


Fig. 4.12 Example image reconstructions from VAE sans residual blocks. Starting with the top left image tile and scanning across the row, images are displayed in pairs, with the original image on the left and the reconstruction on the right.

As can be seen, the World Models decoder **does** reconstruct the target in all cases, suggesting that data volume alone provided a resolve. However, note that, even when trained on the same amount of data for the same length of time, the reconstructions are far less detailed, failing to capture holes or distinct object parts. Instead objects are represented as grey blocks, capturing the correct shape and proportion but nothing further. This would suggest that something about the ResNet-8 architecture, whether it be the additional convolutional layers or the skip connections between them, is allowing the encoder to capture finer details.

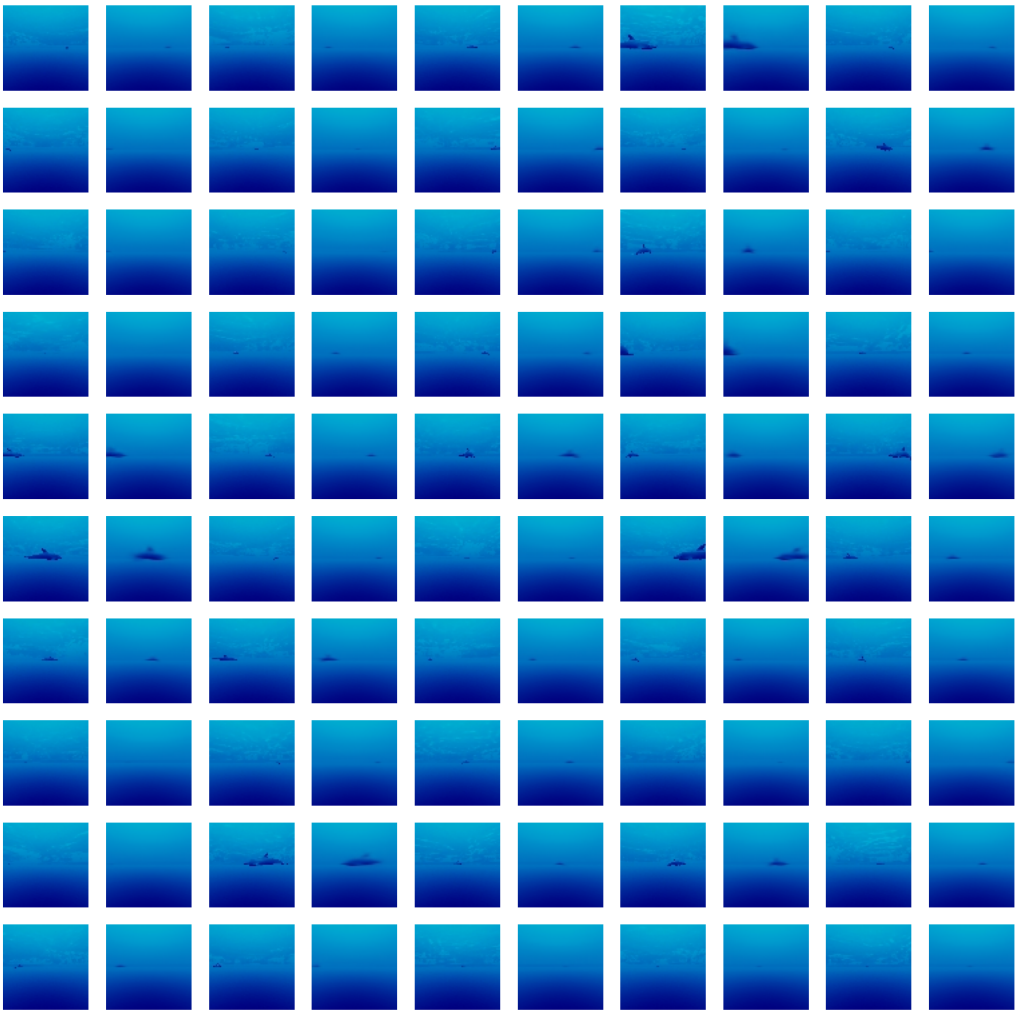


Fig. 4.13 Example image reconstructions from the constrained VAE in SWiMM DEEPeR. Starting with the top left image tile and scanning across the row, images are displayed in pairs, with the original image on the left and the reconstruction on the right.

4.4.2 SWiMM DEEPeR results

Figure 4.13 presents example image samples paired with their reconstructions. The lack of colour contrast between target and background makes the images difficult to discern, but it is still possible to make out that the shape, size, position and rotation of the target has been reconstructed accurately for images with the target in view.

Figure 4.14 presents the z-mural visualisation of the learnt latent space. This figure had to be generated slightly differently to previous z-murals. The first row represents feature 1 incremented from -0.02 to 0.02 left to right, whilst features 2 to 10 are set to zero, the same

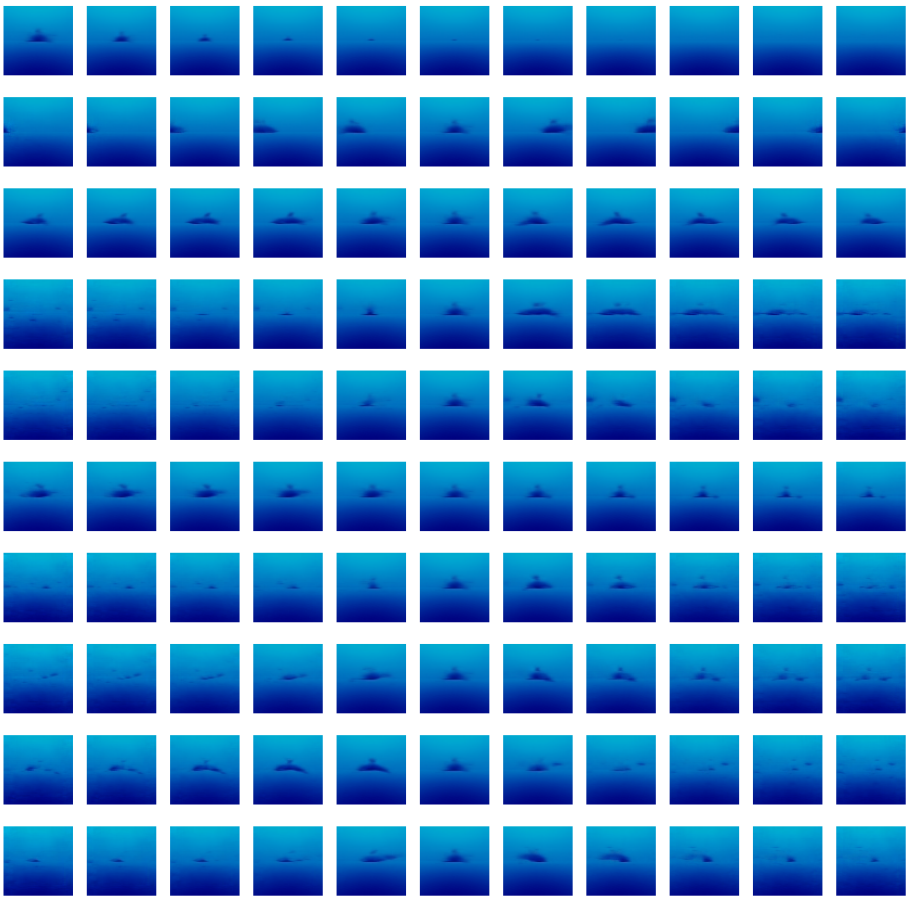


Fig. 4.14 Visualisation of the constrained VAE latent space decodings in SWiMM DEEPeR

Fig. 4.15 Visualisation of the constrained VAE latent space interpolation in SWiMM DEEPeR

as before. However, for the remaining rows, the process as it was before produced seemingly empty images. This is because the process as it was before sets feature 1 to zero on all rows except the first row. A value of zero for feature 1 ($z[0]$) represents the target at a middle distance (i.e. a value of r at the midpoint of the specified r range). In DonkeyTrack, the target was still discernible in the image tiles at this middle distance, whilst in SWiMM DEEPeR the target is not discernible at this middle distance (note how the target disappears by the 6th image tile on the first row). This is because 'distance' is different between environments since world scale depends on mesh scaling. To appreciate the representation of azimuth and yaw (and remaining feature channels), the value of $z[0]$ was set to -0.02 as opposed to zero on rows 2 to 10, bringing the target closer. With this change it is possible to see that the concepts of azimuth and yaw have been represented smoothly.

Figure 4.15 presents the latent space interpolation. As before, the image tiles on the far left and far right are taken from the test dataset. Images were selected which represent the extremes of azimuth and yaw. For example, in the left image the target is side-on and on the far left of the frame, pointing to the left (i.e. close to 180° yaw). In the right image the target is side-on and on the far right of the frame, pointing to the right (i.e. close to zero yaw). The intermediate image tiles are synthetic images generated by the model by interpolating between the latent vectors for the left and right images. The visualisation illustrates a learnt latent space with good continuity, as before.

Table 4.2 presents the state prediction results for SWiMM DEEPeR alongside the Donkey-Track and AirSim values. Although the MAE is marginally higher for both distance and azimuth, the model is still high performing for these state variables. However, for yaw, the MAE is considerably higher. As was explained in Section 4.3.3, relative yaw is sampled from a 360° range when collecting image data from the SWiMM DEEPeR environment. Therefore, the model is predicting yaw from a value range that is twice as large compared to the DonkeyTrack or Bonatti et al. AirSim environments. The shape of the target object and small pixel footprint in the image also add to the difficulty. At a resolution of 64×64, it is not possible to discern head from fluke when the target object is viewed from the front or back, making the large MAE for yaw quite understandable. These kinds of challenges are echoed in related work. For example, in Cai et al., the authors report that "Most fish in the dataset are extremely thin when viewed from the back or top, in these perspectives they lack any distinguishing visual characteristics. The homogeneous colours in these settings, due to colour absorption, and similar textures from the backgrounds, in the case of fish in reefs or squids in rocky terrain, tend to cause all the trackers to fail" (Cai et al., 2023, p. 16). Given

Env	r [m]	θ [°]	ψ [°]
AirSim	$\mathbf{0.39 \pm 0.023}$	2.6 ± 0.23	$\mathbf{10 \pm 0.75}$
DonkeyTrack	0.46 ± 0.003	$\mathbf{0.65 \pm 0.004}$	12.09 ± 0.097
SWiMM	1.25 ± 0.008	1.41 ± 0.012	49.95 ± 0.329

Table 4.2 Table of mean absolute error for predicted dolphin distance (r), azimuth (θ) and yaw (ψ) against ground truth values.

this context-specific challenge, future work ought to experiment with image resolution and the trade-off between processing speed and performance (specifically, yaw prediction).

4.4.3 Domain transfer results

Figure 4.16 provides the reconstruction results for the domain transfer experiment. Across the top row are six example images pulled from the simulated image test dataset, along with their corresponding reconstruction (decoded output from the constrained VAE) directly underneath on the second row. As has already been demonstrated in Figure 4.5, the reconstructions are accurate in terms of target position, rotation and size. The third row then shows the real world mock ups of the same images, along with their reconstructions on the fourth row. It is clearly visible from these reconstructions that the model has been negatively impacted by the

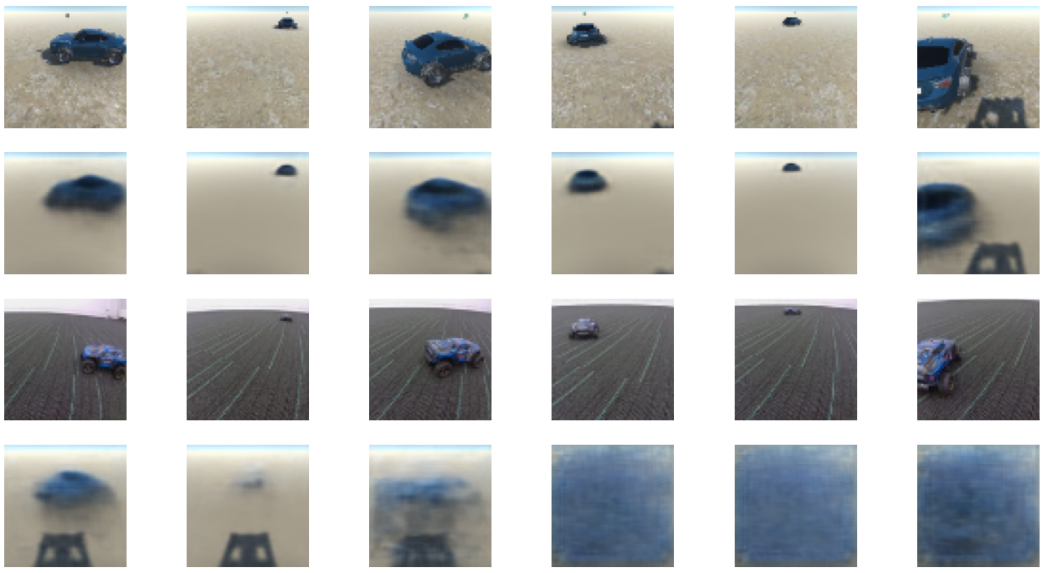
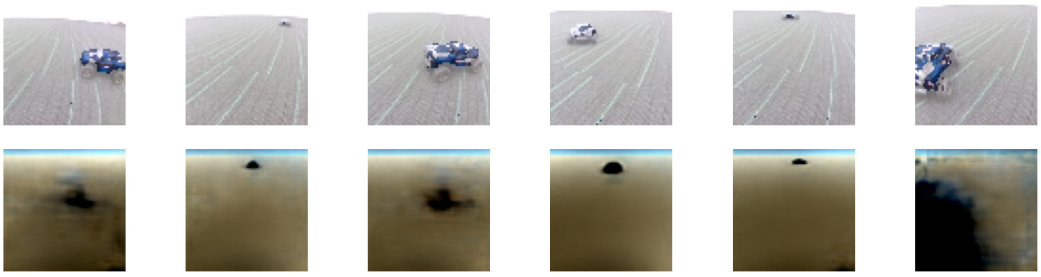
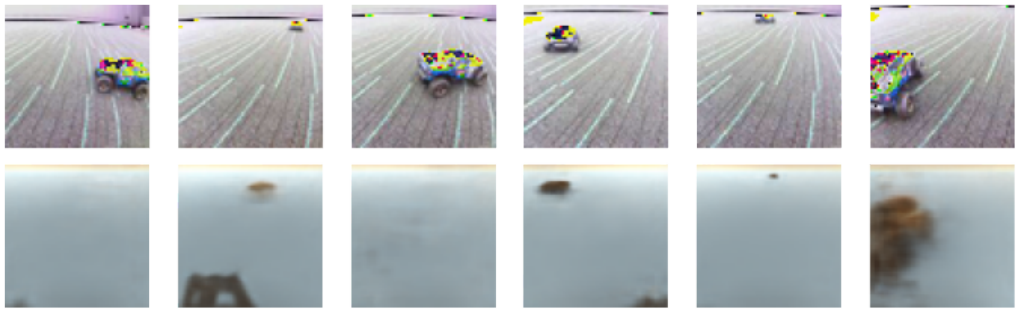


Fig. 4.16 Reconstruction results of domain transfer experiment. Real world images are shown across the third row, with their reconstructions underneath. For reference, the associated simulated images are shown across the first row, with their reconstructions underneath.



(a) Result of brightening background following colour masking of the target car.

(b) Result of changing the background colour following colour masking of the target car.



(c) Result of an incorrect application of pixel normalisation and de-normalisation.

Fig. 4.17 Reconstruction results of domain transfer experiment with pre-processing.

domain shift. When the model receives the real world input image, rather than treat it as an instance of the contextually identical simulated image, it is treating it as an unseen example and mapping it to a new point in the latent space. In columns three to six, the reconstruction is essentially just noise, and therefore this new point is likely *outside* of the learnt space. In the first and second column, the sample looks to have been mapped to a point within the learnt space, but not to the same point as the associated simulated image.

The most striking visual difference between the two domains is the colour of the ground. To test whether this was the cause of the poor performance, some basic approaches to image preprocessing were trialled using OpenCV. Figure 4.17 provides the results. In Figure 4.17a, an attempt has been made to isolate the pixels belonging to the target car using OpenCV's

`inRange` function to produce a mask for the provided HSV colour range. Pixels not belonging to the mask were then brightened, using the `convertScaleAbs` function with beta set to 100. Deciding on the colour range for the mask was a trial and error process, utilising the colour selection tool in Inkscape and other online tools. Whilst the real RC car is blue, it is not block blue as it is in the simulation; it is two-tone (blue and black) and covered in text logos, making it difficult to find a colour range that produced a complete mask without picking up any of the background.

Similarly, in Figure 4.17b, the same approach has been taken but with a straight colour swap, hard coding non-mask pixels to RGB colour (230,230,220) in an attempt to match the colour of the background in the synthetic images. This second approach produces a clear degradation in results. The results of the first approach are more ambiguous, but certainly do not offer any substantial improvement. Interestingly, the most marked improvement was observed in the results of an undergraduate dissertation project centred around this experiment. The student made a code error leading to the colour irregularities in Figure 4.17c. When images are loaded in from file, they are scaled and normalised in preparation for being fed to the VAE, using the equation $image = image/255.0 * 2.0 - 1.0$. Before displaying an image, or applying preprocessing, pixels are de-normalised, using the equation $image = ((image + 1.0)/2.0 * 255.0).astype(np.uint8)$ to take pixels from the range [-1.0,1.0] back to the range [0,255]. However, the student had applied the incorrect equation $image = (image * 255.0).astype(np.uint8)$. They then converted the image from RGB to BGR, increased the brightness and contrast, converted the image back to RGB, and applied the correct normalisation equation. Experimentation with this code revealed that the parameters passed to the `convertScaleAbs` function resulted in a negligible change in brightness/contrast, but the incorrect equation created a shift in pixel values. Why this particular preprocessing outcome was beneficial to the model is unclear. One explanation is that the shift in pixel values increases the colour contrast between target and background. Perhaps the magnitude of this contrast is more important than absolute colour values when it comes to reducing the visual domain gap. Irrespective of the explanation, what this result does indicate is that image pre-processing could provide a solution. However, additional time was not dedicated to experimenting with pre-processing, as it was not considered a robust solution; any change to the real environment (e.g weather conditions) would require modification to pre-processing. Chapter 6 discusses much more sophisticated and robust approaches.

Table 4.3 provides the state prediction results for the various image sets. Note that the MAE values for the simulated image set are different to those provided in Table 4.1 since the values in Table 4.1 are for the entire test set (30K images), whereas the values in Table 4.3 are only for the small sample set (42 images). The MAE values for the real world image set are considerably higher, particularly in the case of yaw. In most cases, this error reduces for processed real world images, but is still not competitive with predictions from simulated encodings. As with the reconstructions, this demonstrates that the domain shift is affecting how images are being embedded.

Image Set	r [m]	θ [°]	ψ [°]
Simulated	**0.23** $\pm$ 0.028	**0.64** $\pm$ 0.060	**4.90** $\pm$ 0.615
Real no proc.	7.78 $\pm$ 0.697	15.75 $\pm$ 1.674	63.16 $\pm$ 6.300
Real bright	4.92 $\pm$ 0.588	9.53 $\pm$ 0.999	43.02 $\pm$ 5.146
Real colour swap	6.15 $\pm$ 0.629	12.34 $\pm$ 1.12	66.55 $\pm$ 5.963
Real incorrect norm	7.69 $\pm$ 0.686	14.59 $\pm$ 1.281	45.51 $\pm$ 4.103

Table 4.3 Table of mean state prediction error for domain transfer experiment.

Table 4.4 provides the mean similarity scores. To reiterate, a similarity score is calculated by feeding the 1×10 latent vector of a simulated image, and the 1×10 latent vector of the real world counterpart image, into equations 4.1 and 4.2. The values in the table are then the means of these scores across all 42 images. With this metric, zero represents entirely dissimilar and one represents entirely similar. Therefore, these results suggest that encoding pairs are only moderately similar, despite representing the 'same image' contextually. Pre-processing the real images before encoding produces a very marginal increase in encoding similarity.

Image Set	Mean Similarity
Real no proc.	0.42
Real bright	0.46
Real colour swap	0.44
Real incorrect norm	0.42

Table 4.4 Table of average image similarity for domain transfer experiment. Each value is the mean similarity score between all image encodings from the simulated image set, and all image encodings from the image set described in the left column.

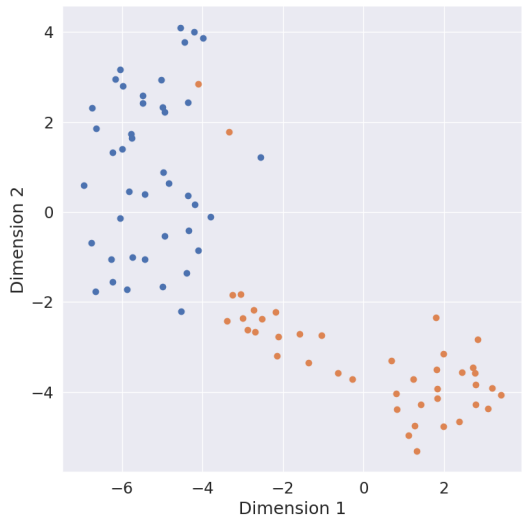


Fig. 4.18 Plot of simulated and real world encodings, sans pre-processing. Blue = simulated images. Orange = real images.

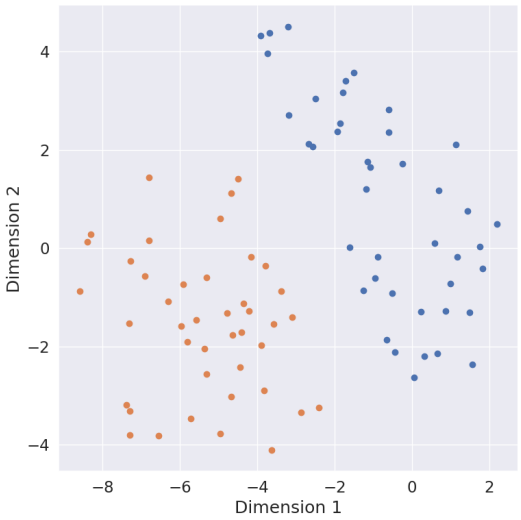


Fig. 4.19 Plot of simulated and real world encodings, with version one pre-processing. Blue = simulated images. Orange = real images.

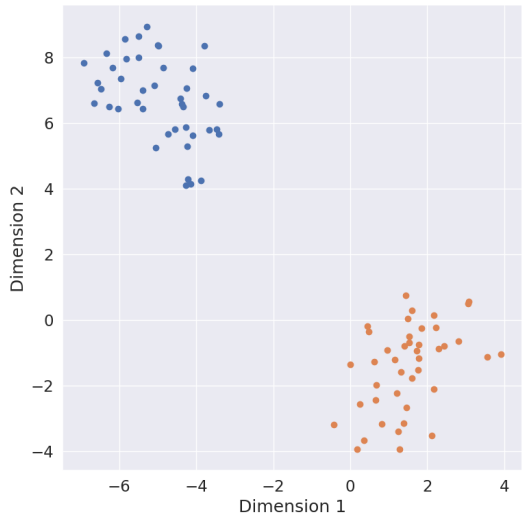


Fig. 4.20 Plot of simulated and real world encodings, with version two pre-processing. Blue = simulated images. Orange = real images.

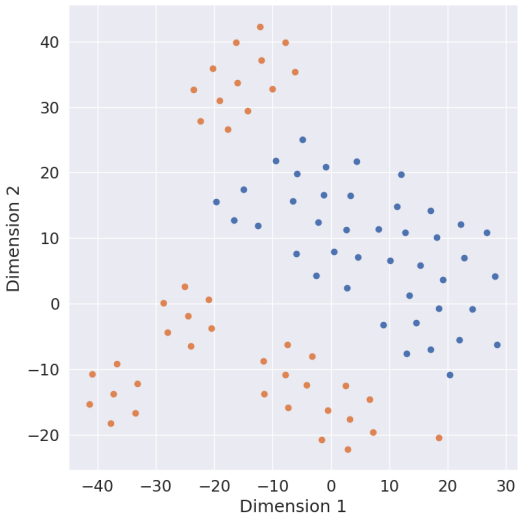


Fig. 4.21 Plot of simulated and real world encodings, with version three pre-processing. Blue = simulated images. Orange = real images.

Having a numeric representation of image set similarity can be useful, but interpretability is low. Figures 4.18 to 4.21 provide scatter plots of encodings. In order to visualise encodings on a two-dimensional plot, it was necessary to perform dimensionality reduction with t-SNE, reducing the ten feature values to two. Each point on the scatter plot represents a single image encoding according to these two reduced feature dimensions. Across all plots, blue points are simulated images and orange points are real images (with or without pre-processing). Figure 4.18 is a plot of real world images without any pre-processing. The plot shows a very clear dissimilarity between simulated and real embeddings; more so than the euclidean similarity score had suggested. Viewed in this way, it is clear that real images are being mapped to an entirely new area of the latent space. Figure 4.19 is a plot of real world images following the first version of pre-processing (masking the target car and then brightening the background). This version of pre-processing aligns the orientation of the cluster, but the two sets are still highly separable. Figure 4.20 is a plot of the real world images following the second version of pre-processing (masking the target and then colour swapping background pixels). This seems to push the two image sets further away in (reduced) latent space. Figure 4.21 is a plot of real world images following the third version of pre-processing (applying the incorrect de-normalisation function). Again, interestingly, this appears to provide some benefit. Although the two sets are not yet mixing, one is encircling the other.

4.5 Summary

Chapter 4 focuses on the concept of decoupling feature learning from policy learning. By learning basic perceptual skills ahead of control, the complexity of the DRL task is reduced, and therefore the complexity and depth of the DRL networks can be reduced, reducing in turn the DRL parameter search space. In addition, the observation search space is reduced, since there can be fewer variants of a 1×10 vector than a matrix of thousands of pixel values. The feature learning was approached with an autoencoder, trained on a dataset of camera frame examples. The model's task was to encode the input image into a 1×10 numeric vector of feature values, and then decode the vector back to the original image dimensions, using the reconstruction loss (difference between the input and output image) to optimise the encoding function.

There were two model optimisations to this autoencoder. Firstly, the autoencoder was 'variational', meaning the encoder network outputs 10 feature distributions as opposed to 10 feature values. A KL-divergence term is added to the objective function to penalise

'distance' from a standard normal distribution, encouraging feature space completeness and continuity. Secondly, three auxiliary decoder networks were introduced, with the task of predicting the distance, azimuth and yaw of the target, from the first, second and third feature values respectively. The error terms associated with these three predictions are then summed with the standard two error terms. Doing so encourages the encoder to represent these task-relevant metrics with the first three features, reserving the remaining features for unconstrained representation learning. This approach was introduced by Bonatti et al. (2020) as a cross-modal VAE.

The body of work described in this chapter successfully replicates this approach. Improvements in efficiency were made to automated data collection, bringing this functionality client side. In terms of feature learning, the same quality of results were observed despite migrating the codebase to TensorFlow 1 and despite applying the method to new environments. Evidence of completeness is provided by the visual analysis of reconstructions alongside their source image. Evidence of disentanglement is provided by a mural of the latent space channels, and evidence of continuity is provided by a latent space interpolation between two image samples. Quantitative analysis of state predictions presented marginally higher error for radial distance and yaw, but lower error for azimuth. An ablation experiment attributed disentanglement and continuity to the 'cross modal' framework, that is, the use of a secondary objective, state prediction, as a regularisation tool. A further ablation experiment attributed completeness and a resolve for the omitted target problem to dataset volume, whilst detail was suspected to be afforded by the ResNet-8 architecture.

Results were also successfully replicated in the SWiMM DEEPeR environment, although the encoding and therefore prediction of target yaw was more challenging due to the requirement to make predictions through the full 360° range. At a resolution of 64×64, the facing direction of the dolphin is not discernable at certain rotations, therefore future work will need to address this, either with an increase to image resolution or with alternative computer vision techniques. Yaw prediction aside, both the DonkeyTrack and SWiMM DEEPeR constrained VAEs were deemed as high performing and ready for use in control training.

In the introduction of this chapter, two claims were made about the benefits of decoupling feature learning from policy learning: improving algorithm efficiency and improving algorithm robustness. The former will be tested in Chapter 5, directly comparing policy training with raw image observations to policy training with feature observations. The latter was investigated with a domain transfer experiment, assessing the similarity of feature vectors

encoded from simulated and real world image counterparts. This experiment revealed that the constrained VAE was *not* robust to the visual domain gap. The model did not map real world image counterparts to the same or similar place in latent space. In fact, image reconstructions of noise revealed that the real images had been treated as unseen examples, mapping to an area outside of the learnt latent space.

Given the results of this experiment, we can conclude that a real vehicle driving autonomously according to a simulation-trained policy, would not succeed in actively tracking a real target car, since latent vectors serve as observations to the policy network, and real world latent vectors are not representing the correct contextual information. Therefore, the use of a VAE alone is not sufficient to combat the sim-to-real gap, even when using the cross-modal framework for VAE training. However, this does not rule VAEs out as an approach. The future work section of Chapter 6 discusses the two main avenues down which to proceed.

In their closing paragraph, Bonatti et al. suggest that the cross-modal, semi-supervised approach to learning task-relevant features be extended to other environments and to other robotic tasks, giving specific mention to autonomous driving and robotic manipulation. Through the work described in this chapter, this suggestion has been realised, extending the constrained VAE training to new environments and a new context (the downstream autonomous driving task of VAT). In the next chapter, the suggestion is realised further, extending the downstream control section part of the approach. As mentioned,Bonatti et al. proposed using the trained encoder network as a preprocessing module for an imitation learning model. Chapter 5 explores utilising the trained encoder in a different way, serving instead the policy network (or 'actor') of the reinforcement learning algorithm Soft Actor Critic, in a novel framework we call T2FO.

Chapter 5

Visual Active Tracking with Soft Actor Critic

5.1 Introduction

Visual active tracking (VAT) is a very specific type of object tracking that constitutes both a computer vision problem and a control problem. The aim is to follow a moving target object, where 'follow' can mean both orienting toward and travelling toward. For example, the task could involve rotating a camera on a camera gimbal, or the task could involve the autonomous driving of a camera-mounted vehicle. In either case, the input is the visual observation (i.e. camera frame) and the output is the control signal. In the visual domain, the goal is to keep the target(s) in frame, most likely central and with an optimal pixel footprint. More generally, active tracking, sometimes called active object tracking (AOT), can involve sensing devices beyond cameras and observations beyond visual observations. The defining characteristic is the element of control in addition to state estimation.

In the computer vision literature, more often than not, use of the term 'object tracking' refers not to VAT but to another closely related problem. This related problem is widely referred to as visual object tracking (VOT). Where one is *active*, the other is *passive*. VOT requires state estimation but not control. Specifically, the task is to estimate the state of a target object in each frame of a video sequence given an annotation in the first frame (see Fiaz et al. (2019a) or Li et al. (2013) for example surveys). Typically, this is an estimation of location and size, expressed with something like a bounding box, contour, silhouette or object centroid. The algorithm ought to be generalisable, capable of tracking any given target, hence most VOT benchmark competitions (e.g. Kristan et al. (2021)) do not disclose the intended target

until the first frame of model evaluation. The video data may be historical (filmed at an earlier date) or live streamed, but in either case the algorithm has no influence over the next incoming frame. The task is purely an annotation task, involving data processing but not data collection. As such, it is much more analogous to the classic computer vision task of detection (Zou et al., 2019) – the same task of drawing a bounding box around each instance of a certain semantic class, but constrained to a single image and therefore a single moment in time. Where one is static the other is dynamic. In object detection, there is no preservation of identity across images, whereas in VOT there is the need to associate detections across time into an object trajectory.

All three tasks, VAT, VOT and detection, require appearance modelling (understanding what to look for) and localisation (looking for it). An appearance model is a representation of the target composed of object descriptors called features. Whether those features are learnt, in the case of deep learning, or hand engineered, it is crucial for the algorithm to be robust to appearance change, requiring an object representation that is invariant to scale, illumination, viewpoint, and so on. The algorithm should aim for zero false negatives (missed detections) and zero false positives (incorrect detections), despite partial occlusions, background clutter, and objects with similar appearance (distractor objects). The dynamic problems, VOT and VAT, may need to deal with additional appearance-based challenges, such as motion blur, and additional problem strands, such as target recovery. There is also diminished data availability, with fewer labelled video datasets than labelled image datasets, and fewer still data generation environments, such as those discussed in Chapter 3. However, the biggest factor that sets apart VOT and VAT from object detection is the high processing speed demanded by the real-time nature of the problem.

The close relationship between the three tasks does of course mean they can be stacked. VOT can be approached by performing object detection on each frame of the video sequence, and then VAT can be approached by feeding the resulting state estimation to a separate control algorithm. However, the naive application of a deep object detector for VOT is highly inefficient. Since a detection algorithm has no contextual information, localisation involves evaluating the appearance model exhaustively at every possible location in the image, and the resulting tracking algorithm could not operate at real-time. To improve efficiency, most VOT algorithms constrain their search region, either with simple logic (for example, the last known target location) or with some form of motion modelling such as filtering or regression techniques.

VOT has been intensively studied, with numerous survey papers (e.g. Fiaz et al. (2019b); Li et al. (2018c); Soleimanitaleb et al. (2019); Verma (2017)) and annual benchmark competitions. As such, there is no shortage of high-performing, real-time solutions to pair with a controller for VAT; examples of which are provided in the opening of the next section under the heading of task-separated solutions. However, in 2018, Luo et al. proposed the first end-to-end solution to VAT, providing a direct mapping from pixels to control with deep reinforcement learning, as was famously introduced by Mnih et al. (2013) in the context of the Atari games. The authors argue that an end-to-end solution is optimal given that parameters in all components can co-adapt and cooperate to achieve a single objective, and report superior performance when compared to task-separated solutions (more detail provided in next section). Since this paper there have been a number of extensions, but in general the problem of VAT is relatively under studied, and therefore Section 5.2.2 draws on examples of end-to-end solutions from other perception-control tasks. Finally, the concept of decoupled feature learning from Chapter 4 is reintroduced, explaining where this sits relative to the task-separated and end-to-end approaches, and why this is expected to produce the best results for the given problem. By the end of the next section, it should be clear why the approach to policy learning reported in this chapter has been taken – an approach which provides a novel solution to visual active tracking.

5.2 Related Work

5.2.1 Task-separated solutions

To reiterate, a task-separated solution to VAT is a solution which treats perception and control as two separate problem components. A common approach to the control component is to use a linear controller, such as the Proportional-integral-derivative (PID) controller. A PID controller continually calculates the error between a desired state ('setpoint') and the current state (as measure by the 'process variable'), and applies a correction. The proportional term ensures the correction is proportional to the error, the integral term accounts for changes in the error over time, and the derivative term is a best estimate of the future trend, based on the error's rate of change (Borase et al., 2021). Due to their simplicity, PID controllers are the most extensively used controllers, both traditionally and in modern applications, making an appearance in 90-95% of control loops (Åström and Hägglund, 2001; Díaz-Rodríguez et al., 2019). However, they are fairly slow to respond, and would therefore require a slow-moving target in the case of VAT. For example, Yao et al. utilise three separate PID controllers (one for distance, one for height, and one for yaw) to control a real robotic blimp tasked with

following a human target, and report "the blimp is able to follow the human to a certain extent given that human is not moving too fast" (Yao et al., 2017, p.5). In this instance, the controllers are fed the motion primitives of the blimp along with an estimate of the human's 3D position using a Haar face detection model paired with a KLT (Kanade-Lucas-Tomasi) feature tracker (Suhr, 2009).

In the absence of additional sensors providing motion primitives, control can be implemented purely based on the error between current and desired features on the image plane. This is typically referred to as visual servoing, or image-based visual servoing (IBVS) specifically. In the context of VAT, this often involves keeping the target's bounding box near the centre of the image, and keeping the height of the bounding box near a predefined constant. For example, Zhu et al. (2019) use this approach to control a real mobile robot tasked with following a slow-moving human target. The bounding boxes are provided by a custom VOT solution, FlowTrack++, which combines Siamese Neural Networks with optical flow information, to simultaneously locate and regress a moving target. If the target is lost, the size of the search region is iteratively increased until recovered. The authors report that this solution outperforms the same control method paired with well-known VOT algorithms GOTURN (Held et al., 2016) and ECO (Danelljan et al., 2017). In Akhloufi et al. (2019), there is also a visual servoing approach to control, but paired with YOLO v2 (Redmon and Farhadi, 2017) and SAP (Search Area Proposal) based on particle filters. For this paper, the context is 'drones chasing drones', requiring 3D control. Whilst the video data is real data, the control output is evaluated offline, without actually flying the vehicle. In both of these examples, the control output is a high-level, discrete signal, which then needs to be realised by the vehicle's built-in controller.

A focused search for AUV vision-based tracking of marine animals identified a number of highly relevant papers which, in all cases, presented task-separated solutions. For example, Kumar et al. (2018) discuss the *premise* of having an AUV performing pre-planned route following, whilst simultaneously extracting features from each camera frame and comparing to a feature database pre-indexed with features of the target species. Once a match is made, the AUV tracks the animal using VOT algorithm ORB (Oriented FAST and Rotated BRIEF) (OpenCV, 2011), and uses visual servoing for control (specifically, the pixel distance between the centre of the bounding box and the centre of the image). Only the VOT component of this proposed solution is tested, with a static camera recording a goldfish in an otherwise empty tank. In 2021, Yoerger et al. successfully tracked stationary or very slow moving jellyfish and larvaceans in-situ, with a 'Mesobot' platform deep down in the Ocean Twilight Zone.

For VOT, they used a blob detector (OpenCV, 2010), taking input from a stereo camera and producing a range, bearing, and vertical offset value for the servo system. The authors profess the solution to be straightforward and suggest implementing a deep tracker in future work.

In the same year, Katija et al. did just this, using a multi-class RetinaNet (Lin et al., 2017) detection model on the left and right frames of a stereo camera onboard their MBARI MiniROV. A data association strategy was used to associate detections to a given target, before forwarding to a dedicated subprocess. Specifically, the bounding box pairs for a given target were fed to a UKF (unscented Kalman Filter) (Wan and Van Der Merwe, 2001) to generate a state estimate in 3D vehicle coordinates. In both Katija et al. and Yoerger et al., the controller is commanded to update the vehicle position in order to centre the target in the camera frame. Whilst the ResNet50 (He et al., 2016) underlying the RetinaNet detector can provide richer features than the much simpler ORB or blob detector, naive tracking-by-detection is inefficient, as discussed. It also requires a large amount of domain-specific labelled data, which is rarely available and difficult to acquire.

With this in mind, Cai et al. (2023) propose a semi-supervised approach to the perception component. In general, the VOT literature presents two families of approaches: discriminative trackers which consider tracking as a classification problem (as in the supervised tracking-by-detection paradigm), discriminating target object from background, and generative trackers which consider tracking as a matching problem, exploiting some similarity criterion to find the best-matched window. Within the latter family, there is an abundance of competition-winning trackers based on Siamese Neural Networks (e.g. Bertinetto et al. (2016), He et al. (2018b), He et al. (2018a), Li et al. (2018a), Wang et al. (2019)). Cai et al. selected 13 of these trackers and evaluated them on their domain-specific dataset VMAT (Visual Marine Animal Tracking). As a compilation of existing scientific sources and their own fieldwork footage, this dataset provides 33 fully labelled video sequences of varied length, across 17 marine animal species, a range of habitats, and a range of swimming behaviours. They then deployed the best performing VOT algorithm, KeepTrackFast (Mayer et al., 2021), on their custom-built AUV, CUREE (McGuire et al., 2023). The AUV performs closed-loop, visual servoing control using the KeepTrackFast detections; adjusting yaw to centre the target and adjusting range to maintain constant bounding box width (found to be more stable than height or area). With this solution they were able to track jacks, jellyfish and barracuda for up to 10 minutes, with occasional human operator interjection.

Finally, and of particular note, is a paper from Yu et al. (2020) presenting a task-separated solution, but with a DRL approach to control. The context is still marine animal tracking and AUV control, although not in-situ (in a pool environment), with a static or else slow moving model of a dolphin, with one-dimension of control, and with a greater focus on the bio-inspired movement of the AUV (a custom built robotic fish) and the camera stabilisation that this then requires. Rather than feed the algorithm image data and employ end-to-end control, camera frames are sent to an onshore computer to detect the dolphin with VOT algorithm KCF (Kernelized Correlation Filter) (Henriques et al., 2014). The position of the dolphin relative to the AUV (yaw, not distance) is then inferred from this image detection, using information from the potentiometer equipped on the motor shaft of the AUV's camera. The calculated yaw angle and its derivative make up the state vector fed to the DRL controller (specifically, DRL algorithm DDPG). Control of the AUV is governed using a central pattern generator (CPG) as the system actuation. According to this CPG-based locomotion control, the yaw of the AUV γ can be changed by adjusting the direction-related offset variable β. It is this variable, β, which the DDPG policy network provides. The DDPG controller was pretrained with a graph-like simulation in MATLAB, before training with the real system in a pool environment. Interestingly, the authors compared the DDPG controller with a conventional PID controller and report DDPG tracking to be more smooth and stable, with no steady-state error, whereas PID tracking presents overshoot above 0.5 radians and a steady-state error at 0.2 radians. The authors also report that their solution suffers from considerable lag, and suggest feeding camera frames directly to the DDPG controller for future work.

Across all of the provided examples, there are some common limitations. One is the capability of the more traditional control strategies such as visual servoing and PID controllers. These methods are widely used and can work well in certain scenarios, but are not appropriate for fast moving targets such as dolphins. PID controllers also require careful manual tuning, as reported by Yao et al. (2017). More sophisticated control strategies exist, such as 'Lyapunov-based' and 'fuzzy logic', however these approaches require accurate dynamic modelling to derive control laws. This is extremely difficult in an aquatic environment with parametric uncertainties and external disturbances. In contrast, reinforcement learning has the ability to accomplish adaptive controllers without access to accurate dynamic models or prior knowledge. Secondly, there is often a requirement for labelled image or video data in order to train the perception module, especially where this module is a deep VOT solution capable of providing robust features. In contrast, the sim-to-real DRL approach (proposed in this work) provides deep features, but the game engine offers unlimited labelled data without

the need for manual annotation. Finally, most of the provided examples rely on expensive sensors such as stereo cameras and not solely on a monocular camera, which is the aim here given the standard configuration of the BlueROV2.

5.2.2 End-to-end solutions

In 2018, Luo et al. proposed an end-to-end approach to visual active tracking, in order to alleviate the difficulties of data labelling, control implementation, and jointly tuning separated sub-tasks. As in Yu et al. (2020), they train the DRL controller in simulation. However, rather than simulate active tracking with two points on a 2-dimensional graph, the authors leverage a commercial game engine to provide a 3-dimensional environment, as proposed in Chapter 3. Specifically, they use two Unreal Engine environments – ViZDoom (Kempka et al., 2016), an RL research platform based on the monster video game Doom, and an unnamed custom urban environment with pedestrians as target objects. The raw camera frames from the agent's FPV camera provide the observation, the same as here. The action space is a 6-part discrete action space (turn-left, turn-left-and-move-forward, turn-right, turn-right-and-move-forward, move forward, and do nothing), similar to CubeTrack but different from the continuous action space of DonkeyTrack and SWiMM DEEPeR. The chosen DRL algorithm was A3C (Mnih et al., 2016). The core idea behind A3C is to decorrelate the agent's data into a more stationary process using parallelisation in an on-policy approach, as opposed to the use of replay buffers in an off-policy approach. During training, several threads are launched, each maintaining a separate agent and environment, but all utilising the same network parameters. These paramaters are updated asynchronously in a lock-free manner, and hence 'experience' is shared.

Luo et al. evaluated the learnt policy across 100 inference runs, and report that the agent is able to maintain target tracking (as indicated by an average episode length of 2959 ± 32 from a maximum of 3000), and position itself well (as indicated by an average cumulative episodic reward of 2547 ± 58). Moreover, the end-to-end RL policy massively outperforms a selection of VOT algorithms paired with a PID-like module operating over the same 6-part action set and in the same environments. For comparison, all of these VOT methods achieve approximately -450 cumulative episodic reward. The authors report that Meanshift (Comaniciu et al., 2000), Kernelized Correlation Filter (KCF) (Henriques et al., 2014), and Correlation (Danelljan et al., 2014) all struggle with camera shift between continuous frames, and so are not as suited to active tracking as they are to passive tracking. Multiple Instance Learning (MIL) (Babenko et al., 2009) works reasonably well in the active tracking scenario,

but drifts when the target turns suddenly.

To test the feasibility of sim-to-real transfer, the learnt policy is also evaluated on real-world video data. Although the camera cannot be influenced, the footage is fed through the model frame by frame, visualising each action prediction such that it is possible to qualitatively assess the appropriateness of the model's decision. The results of this are promising and, in a later paper (Luo et al., 2019), are backed up by real world experiments, utilising YOLOv3 (Redmon and Farhadi, 2018) not for tracking but for evaluation. In simulation, the policy is evaluated using the same reward function used for training, which in turn uses coordinates provisioned by the game engine. In the real world these coordinates would be expensive to collect, and so instead performance is measured according to target centering and size consistency, using the bounding box produced by YOLOv3. Calculating a reward value in this way would also make it possible to continue policy training in the real world setting, although Luo et al. suggest this was not necessary, the augmentation techniques used during training were sufficient to allow for zero-shot transfer.

Later, the same research group presented AD-VAT (Adversarial Dueling mechanism for learning Visual Active Tracking) (Zhong et al., 2019b) and AD-VAT+ (Zhong et al., 2019a), which uses the same end-to-end approach but within an adversarial framework. Rather than present the agent with an unintelligent target performing path following or randomised movement, the target itself is a DRL agent, trained in parallel. For the most part, the agents compete within a zero-sum game, although beyond a given distance threshold the game becomes non-zero-sum and the target is penalised for exiting the tracker's field of view. This design was found to be most conducive to learning, as was training a 'tracker-aware' target tasked with predicting the tracker's reward as an auxiliary task.

Outside of the tracking literature, a comparison between end-to-end and task-separated visuomotor control was addressed much earlier, in a seminal paper from the University of California Berkeley in 2016. Levine et al. use vision-based DRL on a real 7-DoF robotic arm, to perform a range of complex manipulation tasks such as inserting a block into a shape sorting cube and screwing a cap onto a bottle. The authors state that "by learning the entire mapping from perception to control, the perception layers can be adapted to optimize task performance, and the motor control layers can be adapted to imperfect perception" (Levine et al., 2016b, p.3). The results of their direct comparison corroborate, with the end-to-end trained policy markedly outperforming the task-separated solution, across all four tasks. For example, the success rate across 27 trials of the shape sorting cube was 96.3% for the

end-to-end solution and 0% for the task-separated solution.

The end-to-end solution of Levine et al. is slightly different to the end-to-end solution of Luo et al.. Whilst both map raw pixels directly to control with a CNN-based policy network, Luo et al. begin policy training from randomly initialised weights, whilst Levine et al. begin policy training with pretrained weights. Levine et al. argue that, whilst it is optimal to combine the training of vision and control, training both together *from scratch* leads to a large number of iterations spent learning low-level aspects of vision; a learning process that can be tackled much more efficiently with supervised learning in isolation. As such, Levine et al. append their chosen CNN architecture with a fully connected layer, and train the network to do pose regression with supervised learning – an approach much more comparable to the approach taken here. They then remove the fully connected layer and embed the trained convolutional layers into their policy network, further optimising the weights within the context of visuomotor control. In the VAT literature, the second of two solutions proposed by Akhloufi et al. (2019) has a similar flavour, using convolutional layers from a pretrained VGG-M network (Chatfield et al., 2014) within the policy network. Both of these examples can still be regarded as an end-to-end solution, albeit much more efficient. Luo et al. do not report on wall clock time, but report obtaining the best validation results at around 48×10^6 iterations (for comparison, the training runs reported in this chapter are typically 1×10^6 steps). Considering this is in a synthetic environment with a discrete action space reduced to yaw and thrust, it is fair to assume that this approach (training end-to-end from scratch) would take infeasibly long as the problem scaled (e.g. directly supplying torque to a robot with 7 degrees of freedom).

The concept of learning visual features ahead of policy learning revisits the same ideas introduced in Chapter 4. However, the two approaches are subtly different. Whereas Levine et al. continue to optimise their convolutional layers (perception) when learning control, in Bonatti et al. (2020) the convolutional layers are fixed when learning control, i.e. they become a preprocessing module. Therefore, arguably, this is no longer end-to-end. In fact, if the target pose (distance, azimuth and yaw) predicted by the auxiliary networks served as the agent's observation, then this would be exactly equivalent to the task-separated solutions presented in Section 5.2.1. However, because these predictions are dropped, and it is only the feature vectors that feed into the policy, the modularity of the approach is somewhat different to a task-separated approach. That is, the learnt appearance model is taken forward into policy learning but *not* the learnt localisation. Localisation is still (implicitly) learnt when learning control. Levine et al. also discuss this distinction, and include learning control from

pose estimation features in their results comparison. They report intermediate success, for example, a 70.4% success rate for the shape sorting cube, in comparison to 96.3% end-to-end and 0% task-separated. They conclude that although feature observations reduce computation time, end-to-end is superior, although this conclusion is with respect to object manipulation, with tasks that have tolerances of just a few millimetres. This accuracy requirement is not true of VAT.

In tasks more analogous to VAT, such as autonomous driving and flight, the literature lends support to the idea of learning control from features. In Bonatti et al. (2020), the success rate of the behavioural cloning control policy fed feature observations is five times higher than the behavioural cloning policy trained end-to-end with raw image observations, although it should be noted that this is end-to-end without any perceptual pretraining. In Toromanoff et al. (2020), they also decouple feature learning from DRL policy learning and utilise an auxiliary loss during feature learning. Where features have been described as 'task-relevant' in Chapter 4, here the exact same premise is coined as 'implicit affordances'. This proves extremely effective for urban autonomous driving tasks, with the resulting policy winning the "Camera Only" track in the 2020 CARLA Autonomous Driving Challenge. It would therefore appear a sensible direction to explore and, to the best of our knowledge, would be the first application of implicit affordances to visual active tracking.

5.3 Methodology

5.3.1 T2FO framework

To recap Chapter 4, the autoencoder approach to training was used to optimise a ResNet image encoder and a CNN image decoder, on a large dataset of example camera frames automatically generated with the Unity engine. Furthermore, three simple MLP networks were in place to perform the auxiliary task of predicting target distance, azimuth and yaw from the first, second and third values of the latent vector respectively. The total loss was calculated as the image reconstruction loss, plus the state prediction errors, plus the KL-divergence between the multivariate distrubition produced by the encoder and a multivariate standard normal distribution. Together, this produced a disentangled latent space, with continuity, completeness, and ensured task relevance across the first few dimensions. Once trained, the image and state decoding networks can be dropped, and the ResNet encoder taken forward into policy training.

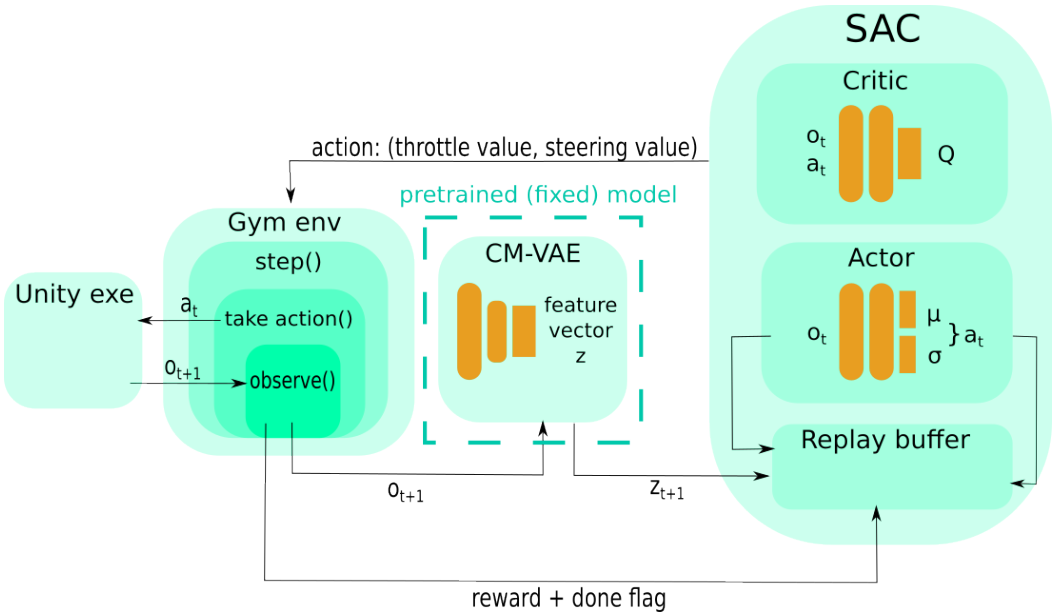


Fig. 5.1 High-level illustration of the simulation-based training framework.

Figure 5.1 illustrates the overall framework. Training begins with the server sending a 'reset' message to the client. The client responds to this message, resetting the game to its start state, rendering an image from the simulated camera, and collecting various other game data, such as the tracker's and target's position and orientation. The data is sent over the network where it is received and unpacked by the server. The image is resized (if necessary), normalised, and fed through the trained encoder network, producing a 1×10 feature vector z. It is this feature vector that is used as the RL observation, o_t. The agent is then ready to take the first 'step' in the Gym environment, feeding o_t through the policy network to produce action a_t. The action is sent off to the client, dictating the forces applied to the AUV and prompting the client to send the next data message. In the same 'step', the returned image is encoded to give next observation o_{t+1}, and the non-image game data is used to calculate r_t and the value of the 'done' flag according to the episode termination criteria (more detail in Section 5.3.2). Together, $\{o_t, a_t, r_t, o_{t+1}, done\}$ make up a single transaction, fed into the replay buffer. The next step begins with the policy network selecting a_{t+1} on the basis of o_{t+1}. The cycle continues and the step counter increments until the 'done' flag evaluates to `True`. If `True`, the server triggers an optimisation run (more detail in Section 5.3.3) before sending a reset message to the client.

5.3.2 Reward engineering

The reward function used for training agents across all three environments was taken from Luo et al. (2018). At each time step, r_t is calculated according to:

$$r_t = 1 - (\frac{d}{maxd} + \frac{a}{180}) \\ where\ d = \sqrt{x^2 + (z - optd)^2} \tag{5.1}$$

The coordinates x and z are taken from the heading vector from agent to target (i.e. the target's position vector minus the agent's position vector). The variable $optd$ is an environment hyperparameter expressing what distance the agent ought to leave between it and the target. The value of this variable was selected based on what looked appropriate in the Unity editor, and can be easily edited in the environment configuration file. By subtracting $optd$ from z prior to taking the magnitude, $optd$ expresses an optimal tracking position that is set back but in line with the target, i.e. the distance is with respect to the world z-axis only, it is not a horizontal distance or a radial distance. Once the magnitude is taken, the variable d becomes **a radial distance around the optimal tracking position** and not the distance to the actual target. In order for this optimal position to be expressed in terms of the target's local coordinate frame and not the world coordinate frame, additional lines of code were required to manipulate the sign of z. This code ensures that, regardless of the target's travelling direction, the agent is always rewarded for being *behind* the target, even when the agent is technically forward of the target according to the world z-axis.

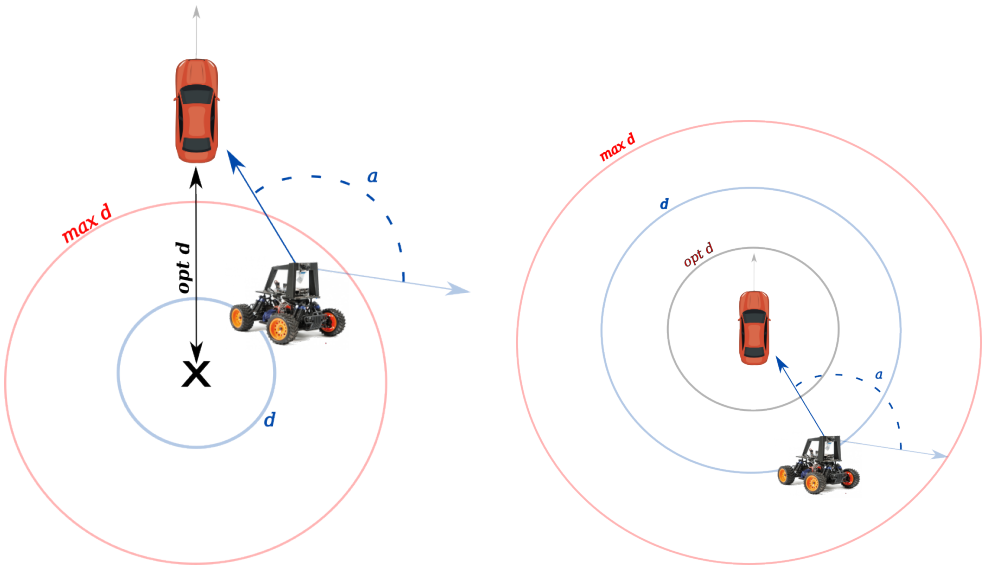
The variable a is the angle between the heading vector and the agent's forward vector. The angle is calculated in degrees and always returns the smallest angle and not the reflex angle, producing the range $[0, 180]$. This is subtly different to the definition of a in Luo et al., where a is the discrepancy in orientation i.e. the difference between the agent's forward vector and the target's forward vector. In the follow up paper from Zhong et al. (2018), the definition is changed to the definition used here. By measuring the angle between the agent's forward vector and the heading, the agent is encouraged to look directly at the target, bringing the object into the centre of the camera frame. Results in CubeTrack suggested that this implementation was optimal, which intuitively makes sense for a vision-based agent.

Artificial neural networks train best when the input to the network is normalized, as it keeps gradient magnitudes fairly homogeneous. Just as images are typically normalised to have pixel values in the range [-1,1], in DRL it is common to scale reward values also (Henderson et al., 2018). Here, both d and a are normalised to the range [0,1] with the formula $X - X_{min}/X_{max} - X_{min}$. For a, X_{min} is 0 and X_{max} is 180. For d, X_{min} is 0 and X_{max} is determined by $maxd$. In CubeTrack, because the environment is an enclosed platform, $maxd$ is simply the distance between the two diagonally opposite corners of the platform. That is, it is not physically possible for the agent and target to be any further apart than if the agent were in one corner and the target were in the other. In DonkeyTrack and SWiMM DEEPeR, because the environment is an open world, $maxd$ describes an invisible maximum threshold, the value of which is an environment hyperparameter. The sum of d and a is subtracted from 1, producing the range [-1, 1]. The resulting value r_t is issued per time step, providing a dense signal for the agent. It is also worth reiterating that this reward function is based on internal states and not raw visual information – a design made possible by the use of a game engine.

Episode termination

Deciding on episode termination criteria presents a trade-off between a need to 'fail fast' (i.e. not waste time collecting lots of data on incorrect behaviour) and the need to let the episode play out long enough to collect a sufficient amount of data to learn from. Moreover, the agent needs to collect varied data, which, without randomised starts, is not possible if the episode terminates shortly after starting. With CubeTrack, episode termination consisted of the same two criteria described by Luo et al.. The first is a maximum episode length, set to 3,000 steps. The second is a lower bound for the cumulative episodic reward. If the running cumulative reward for the episode falls below -450 at any point, the episode is terminated. This is a fairly relaxed criteria, producing relatively long episodes. The logic behind this was that, due to the small size of the enclosed platform, the target is never so distant that it is not visible in the camera image. Therefore, with the right action decisions, it is always possible to resume tracking. So long as the agent is able to recover, it can be conducive to learning to provide the opportunity to do so, within reason (up to the point of falling below -450 cumulative reward in this instance).

In DonkeyTrack and SWiMM DEEPeR, it is possible to stray so far from the target that it exceeds the horizon. Therefore, rather than use cumulative reward as the criteria for episode termination, instead the episode is terminated when d exceeds $maxd$. The value of the environment hyperparameters $maxd$ and $optd$ are reported in the results section along



(a) Tailgating design from Luo et al. (b) New radial design.

Fig. 5.2 Reward engineering illustrations. In both designs, d and $maxd$ are radial distances from $optd$. In a) $optd$ is a point, marked by 'x', whereas in b) $optd$ is itself a radius.

with the algorithm hyperparameters for a given run. In general, the value of $maxd$ was kept small, inspired by the strict termination criteria used in the 'Learning to Drive Smoothly in Minutes' (2019) project with great success. In addition, the relative size of $optd$ and $maxd$ was used to address an undesirable environment loophole presented by the modified 'SDSandbox' simulator in DonkeyTrack. In this environment, when the agent and target collide it is possible for the wheels to interlock and for the two vehicles to become attached. Such a scenario can lead to a high return, given that the agent remains consistently close to the optimal tracking position. Without intervention, the agent can learn to exploit this loophole, colliding with the target as early as possible. A number of designs were trialled around discouraging this behaviour, penalising the agent with negative reward or early termination, either at the point of collision or within a 'danger zone' surrounding the target. Often, it appeared these penalties were to the detriment of a good active tracking policy; the incentive to follow the target closely is in conflict with the deterrent of getting too close. One way to solve this issue was to set $maxd < optd$. The $maxd$ threshold is with respect to d (the distance between the tracker and the optimal tracking position), and the optimal tracking position is expressed by $optd$ (a measure of distance from the target). Therefore, if $maxd < optd$, the episode will terminate before the agent is able to reach and make contact with the target (see Figure 5.2a for an illustration).

Changing the definition of 'tracking'

The reward function described in 5.3.2 elicits what is best described as tailgating. In CubeTrack, this reward function proved effective, however in DonkeyTrack and SWiMM DEEPeR the agent was not performing so well. When reflecting on the difference between environments, one candidate for the cause of the performance difference was the target trajectory. In CubeTrack, although the target selects a waypoint at random, it travels to the waypoint in a straight line. On reaching the waypoint, it rotates on the spot to face the next waypoint, but then heads off in a straight line again. The behaviour of the target in the Luo et al. ViZDoom environment is similar; travelling in straight lines other than a sharp turn when it reaches the corner of the room, for example. Whilst the target in SWiMM DEEPeR also travels from waypoint to waypoint, it does not travel in straight lines, it is programmed to weave and swim in arcs, more akin to how a dolphin would move. In DonkeyTrack, the target randomly samples an action from the action space each time step and is therefore also prone to weave and change direction at any moment.

To make the environment more comparable, the target car in DonkeyTrack was temporarily given a straight trajectory by hard coding a_t to $[0, 0.2]$ (i.e. no steering angle and low throttle). As expected, performance immediately improved. The consensus was that, with the implementation in equation 5.1, the point of highest reward is anchored to the target's local negative z-axis and therefore, when the target turns, this point swings out in the opposite direction. The agent experiences a large change in the reward signal but with minimal change to visual input. To remedy this, the reward function was modified to:

$$r_t = 1 - \left(\frac{(d - optd)^2}{maxd^2} + \frac{|a|}{180} \right) \tag{5.2}$$

$$where\ d = \sqrt{x^2 + z^2}$$

Subtracting $optd$ from radial distance d as opposed to the target's z-coordinate creates a radius of highest reward, as opposed to a point of highest reward (see Figure 5.2b). This helps to make the reward signal a bit more homogenous; reward is no longer affected by the target's orientation so long as the *tracker's* orientation stays target-facing, and the same distance is maintained. Not only is this more consistent with visual servoing (keeping the target central and fixed size in the image plane), it also promotes more desirable behaviour. When filming marine mammals, there are several reasons why tailgating would not be appropriate. Firstly,

motoring directly behind the animal will likely be interpreted as predatory and startle the animal. Secondly, it is better to film the animal from varied perspectives (and in particular side-on) for the footage to be useful for photo identification. Thirdly, a tailgating policy does not lend itself to multiple object tracking (MOT) as well as the more relaxed radial design and, ultimately, the aim is to film the entire pod to provide the most complete data for behavioural study.

5.3.3 SAC implementation

Early work in CubeTrack centred around the on-policy DRL algorithm Proximal Policy Optimization (PPO), only because this is the default algorithm when using the ML-Agents toolkit and because this early toy environment served purely as a test-bed for the reward function. ML-Agents was not used to develop the DonkeyTrack and SWiMM DEEPeR environments; these environments have a custom API and integrate with DRL library Stable Baselines. When it came to training agents in these latter two environments, the algorithm of choice in both cases was SAC, the off-policy maximum entropy algorithm with a stochastic actor. This decision stemmed from the results of the 'Learning to Drive Smoothly in Minutes' (2019) project. The author, who is also a key maintainer of the Stable Baselines library, reports experimenting with PPO, SAC and DDPG when training an agent within the (original) `gym-donkeycar` environment, i.e. track racing. All three of these algorithms can be used with a continuous action space, as is required by DonkeyTrack and SWiMM DEEPeR, however the conclusion drawn from this project was that DDPG and SAC offered superior performance to PPO when trying to minimise training time, and that the more recent of the two, SAC, was easier to tune. Given that DonkeyTrack is essentially the same environment other than a slight change in problem specification (following a moving object as opposed to a race track), it was felt that these conclusions were likely generalisable. They also corroborate the proposed benefits of SAC (Haarnoja et al., 2018): improved sample efficiency and less brittle convergence properties.

A theoretical explanation of SAC was provided in Section 2.4. To recap, the unique selling point of SAC is the *learnt* exploration afforded by the introduction of an entropy term in the objective function. The entropy term can be thought of as a measure of randomness in the policy (refer back to Section 2.4 for a more detailed explanation). By maximising entropy as well as return, the network is encouraged to assign equal probabilities to actions with the same or similar Q-values. Smoothing the action probability distribution like this prevents the policy from collapsing into repeatedly selecting the same action (which could exploit some inconsistency in the approximated Q function) and allows the policy to capture

multiple modes of near-optimal behaviour. SAC also incorporates the double Q-learning trick from TD3. That is, even aside from the use of target networks, the algorithm optimises two entirely separate Q-networks, producing two different estimates of the state-action value for every passed state and action. Having more than one estimate and selecting the minimum has been found to help stability by solving the Q-value overestimation problem.

Given that there are multiple variants of the SAC algorithm, this section highlights any details specific to the Stable Baselines implementation used in this work. Firstly, a note on the number of networks. In line with the second version of SAC, published in 2018, Stable Baselines implements a value network V as well as the policy network and two action-value networks, Q_1 and Q_2. Both the policy network and V have an associated target network, with updates that lag behind main network updates. This can be achieved either with a delay (requiring the user to provide a target update interval) or with polyak averaging (requiring the user to provide a polyak coefficient between 0 and 1). In the third version of SAC, V and target V are dropped and are replaced with a target network for Q_1 and Q_2, but this is not the version used here.

Stable Baselines SAC generates four different state-action values. The first two, q_1 and q_2, are generated by sampling a state and action *from the replay buffer*, before passing through networks Q_1 and Q_2. They therefore provide the 'estimate' when updating $Q1$ and $Q2$ respectively. Note that the 'TD target' in both these updates uses the state-value produced by target V. To generate the third and fourth state-action values, q_1^π and q_2^π, it is necessary to feed the Q-networks the *current* state and the action returned by the *current* policy. In the SAC write up provided in Section 2.4, this was denoted with a bar symbol on the next action term. This is important because it makes the term dependent on the policy parameters. These latter two state-action values are used in the update of both the policy and V. For V, the update is a 'soft update', that is, the minimum of q_1^π and q_2^π is used. For the policy update, the minimum is not taken and the first of the two, q_1^π, is always used. This makes the update the same as the DDPG update barring the entropy term.

In terms of the policy network itself, there are two final *Dense* layers following the feature extraction layers: one producing the mean, μ, and one producing the log standard deviation, $log(\sigma)$. The second output is considered the log standard deviation as opposed to the standard deviation because the standard deviation needs to be positive, and it is easier to ensure this by taking the exponential of the log than it is restricting the neural network output. The second output $log(\sigma)$ is also clipped (specifically, to the range [-20,2] in this implementation) before

taking the exponential. Then, to get the action, the Stable Baselines algorithm implements the SAC reparameterisation trick introduced in Section 2.4. To recap, the reparameterisation trick provides a way of 'sampling the policy', by computing a deterministic function of state, policy parameters and independent noise. This was expressed as:

$$\tilde{a}(s,\xi) = tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \qquad \xi \sim \mathbb{N}(0,I)$$

meaning noise is sampled from a standard normal distribution and is then multiplied with the standard deviation and summed with the mean. The result is activated with a tanh function to give the action. Note that this is only during training. During inference, the action is $tanh(\mu_\theta(s))$.

Policy updates use the same objective function presented in Section 2.4

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{a_t \sim \pi}[Q_\phi(s_t,a_t) - \lambda log\pi_\theta(a_t|s_t)] \tag{5.3}$$

where $-log\pi_\theta(a_t|s_t)$ is the negative log probability of the action decision, used as the entropy term. As was explained in Section 2.4, entropy can be thought of as a measure of surprise and is therefore the opposite of how probable something is (hence the negative sign). The log is taken to make the term easier to differentiate when calculating gradients. The term $log\pi_\theta(a_t|s_t)$ can be calculated from the network outputs, μ and $log(\sigma)$, using the standard formula for the log-likelihood. Note that in the Stable Baselines implementation, $log\pi_\theta(a_t|s_t)$ is positive and $Q_\phi(s_t,a_t)$ is negative. The sign reversal provides a 'policy loss' for gradient descent, as opposed to a maximisation objective for gradient ascent.

Finally, the λ term in equation 5.3 represents the temperature parameter. This determines the weight of the entropy term and therefore the stochasticity of the policy. In the 2018 paper, the temperature parameter is fixed and is equivelant to the inverse of the reward scale. In the third version of the algorithm, SAC with 'autotuned temperature' (Haarnoja et al., 2019), the authors introduce automatic gradient-based temperature tuning, adjusting the expected entropy over the visited states to match a target value. The Stable Baselines implementation of SAC offers both fixed and autotuned entropy coefficients. If the entropy coefficient hyperparameter is passed as a float value, fixed is assumed. The alternative is to pass a string of the form 'auto_x', where x is taken as the initial value.

Network configurations

The double Q-learning trick prevents the `SAC` class from utilising the policies provided by the Stable Baseline's `common` module. Instead, the class has its own policies. Available policies include a CNN-based architecture (specifically, a 3-layer CNN followed by two 64 unit fully-connected layers) and an MLP-based architecture (specifically, two 64 unit fully-connected layers). Both are available with or without layer normalisation (denoted by 'Ln' in the policy name). For SAC policies, the activation is ReLU as opposed to tanh, in line with the original paper.

These architectures are described as 'policies' but can be thought of as the blueprint for feature extraction layers. In practice, the same architecture is used across all of the networks, appended with the appropriate output layer(s). For example, for the policy network, a final *Dense* layer with the same number of units as the action space size is used to produce a mean value, and another a standard deviation value. For V and Q, a final *Dense* layer with one unit is used. Of course, the feature extraction architectures are only default configurations. Stable Baselines supports custom policies. In the work reported in the following section, the default `MlpPolicy` is utilised, but is fed features output by the ResNet encoder network (refer back to Figure 4.2). It is therefore acting as a second layering of feature extraction. To reiterate, only the weights of the 2-layer MLP are optimised with SAC; the weights of the ResNet are fixed.

In the follow-up experiments reported in Section 5.4.6, a CNN-based architecture is required for working with image observations. Rather than utilise the default `CnnPolicy`, we implement a custom policy with the same ResNet architecture as the trained encoder network in order to make comparisons fair. A ResNet is also the underlying architecture of the policy network in CubeTrack training runs, undertaken prior to any of the work presented in Chapter 4. When training with state prediction observations in the Section 5.4.6 experiments, the `MlpPolicy` is used in order to match with feature observation training.

Hyperparameters

With any deep learning algorithm, there are a number of configurable parameters called hyperparameters. Each one has an influence on the model's performance to a greater or lesser degree. Some are common across algorithms and others are algorithm-specific. Stable Baselines provides default values for every hyperparameter per algorithm. These values are values that *typically* work well for the given algorithm, across different tasks and environments.

However, the optimal value for a hyperparameter will be environment and task-specific, so it is important to experiment – a process called hyperparameter tuning. Whilst it is not possible to test every value in a large and continuous search space, it is possible to tune hyperparameters in a systematic way. For example, in a method called grid search, the user discretizes the search space by defining a set of values for each hyperparameter. These value sets create a parameter grid, with each cell in the grid representing a different parameter combination on which to train and evaluate a model. In some cases the first grid search will produce an acceptable model, and in other cases the results are used to refine the grid for the next iteration.

In recent years, approaches to hyperparameter tuning have become increasingly sophisticated, with libraries such as Optuna (Akiba et al., 2019) providing a Bayesian framework for tuning. In this approach, the user defines an upper and lower bound for each hyperparameter and the space in between is explored probabilistically. This is a lot more efficient than exhaustive search, since the process ignores values which are unlikely to produce good results. Note that this efficiency is only relative to exhaustive approaches such as grid search; hyperparameter tuning can be extremely computationally expensive and time consuming, even in a supervised learning setting whereby the data is fixed. In DRL it becomes practically infeasible from a standing start, or at least when done in a way that produces meaningful results. Optuna refers to a hyperparameter optimisation run as a 'study' and each parameter configuration run as a 'trial'. Setting up a study presents some obvious trade-offs. For example, the bigger the search space for a hyperparameter, the longer the study will take, but too small and the space may not contain the optimal value. Similarly, the longer each trial the longer the study, but too short and the evaluation may not be representative. These problems are magnified in DRL for two main reasons. Firstly, DRL is notoriously sample inefficient, requiring lots of samples before the agent begins to learn anything. This requires trials to be sufficiently long to surpass the warm up period and provide a true reflection of model performance given the configuration. Secondly, DRL has poorer reproducibility than supervised learning due to the stochasticity in the environment and policy, on top of the usual level of stochasticity that can be controlled with seeding (more on this in Section 5.4.5). This means that the results of individual trials have lower confidence, since two different runs with the same hyperparameters can vary significantly.

These things considered, best practice guides to DRL often advise approaching hyperparameter tuning in a staged fashion (RL Wiki, 2020). These stages are 1) train the DRL model with the hyperparameters set to those reported for same or similar tasks, 2) based on the results of this, if necessary, manually tune a select few hyperparameters until performance is reasonably

good and reasonably stable (this process requires experience and a good understanding of the algorithm), 3) based on the results of this, if necessary, take the network weights of the best model forward and perform automated hyperparameter tuning (with the idea that the transfer learning ought to bypass or at least shorten the warm up period, making automated tuning over a large search space slightly more feasible).

Below is a list of the hyperparameters for the Stable Baselines implementation of SAC, along with a description and a default value.

- **Gamma (0.99)** – The discount factor, which tells us the importance of future rewards. When calculating return, the reward for time step t is multiplied by gamma to the power of t. Therefore, a gamma of zero creates a myopic agent that only cares about immediate reward, whilst a gamma of one creates a forward looking agent that assigns equal weighting to every reward in the time series.
- **Learning rate (0.0003)** – The step size for each gradient descent update step i.e. the degree to which the weights of the model should change in response to the estimated error calculated during backpropagation. In the Stable Baselines implementation of SAC, the same learning rate is used across all actor and critic networks. Note also that, across all Stable Baselines algorithms, the optimiser is Adam (Kingma Diederik and Adam, 2014) and the choice of optimiser is not provided as a hyperparameter.
- **Buffer size (5000)** – The maximum number of experiences (data traces) that can be held in the replay buffer before new data replaces old data.
- **Batch size (64)** – The number of data traces used for one iteration of a gradient descent update. This should always be smaller than the buffer size.
- **Tau (0.005)** – The coefficient, between 0 and 1, for the polyak averaging. The lower the value the slower the target network will update relative to the main network. This is typically left at the default value.
- **Entropy coefficient ('auto')** – The weight assigned to the entropy term of the objective function and therefore controlling the exploration/exploitation trade-off. Passing the string 'auto' as opposed to a float value indicates that the coefficient will be learnt not set. An underscore and float value can be incorporated into the string to provide an initial value for coefficient optimisation.
- **Train frequency (1)** – How many steps between model updates.

- **Learning starts (100)** - How many steps to collect data for before learning starts. Prior to this number of steps, the action is uniformly sampled from the action space. After this number of steps, the action is provided by the policy network.
- **Target update interval (1)** – How many steps between target network updates.
- **Gradient steps (1)** - Number of gradient decent steps to do per model update.
- **Target entropy ('auto')** – Used in the objective function when optimising the entropy coefficient. Passing the string 'auto' sets this value to -1 times the action space size – a heuristic used to afford a proportional budget of entropy for the policy per action dimension.
- **Action noise (None)** – Optional extra for explicitly injecting noise into the action decision, as in DDPG.
- **Random exploration (0.0)** – Probability of sampling a random action in an epsilon-greedy strategy. This explicit exploration strategy is not needed in SAC, but can help when pairing SAC with Hindsight Experience Replay.

5.4 Results

Results are presented in dedicated sections per environment. Across all three environments, Tensorboard was used for logging purposes, with some form of episodic return graph as the primary output, in line with what is most commonly reported in the DRL literature. Agent training is described in terms of 'runs', i.e. a continuous stretch of data collection and policy optimisation for a given number of environment steps (set by 'maximum steps'), using a particular set of hyperparameters. Unless stated otherwise, all runs were initialised with random weights, and were not building on previous runs. Behind each successful, final iteration run reported below is a very high number of experimental runs (as many as 70 in the case of DonkeyTrack) due to a mixture of environment debugging, reward engineering, and hyperparameter tuning. This enormous amount of time and compute is typically not reported on in the DRL literature, but is important to consider when evaluating DRL as an approach.

5.4.1 CubeTrack results

Runs in CubeTrack were performed locally on a Lenovo Thinkpad T480s laptop (Intel® Core™ i7–8550U Processor, 4 cores, 8 threads, 1.80 GHz processor base frequency). Hyperparameter tuning was not explored in CubeTrack; the purpose of this environment was

to implement the reward function, ensure its correctness (e.g. rewarding with respect to the target's local axes and not the world axes), and test whether the high VAT performance reported by Luo et al. was reproducible in a new environment. As mentioned, all training runs in CubeTrack used PPO, the default algorithm when using ML-Agents. Hyperparameters for PPO include batch size, buffer size and learning rate, the same as SAC. Hyperparameters specific to PPO include beta, epsilon, lambda and time horizon. In contrast to the entropy maximisation in SAC, PPO implements entropy regularisation, and beta dictates the strength of this regularisation. PPO also constrains updates, and epsilon dictates the maximum allowed divergence between the old and new policy. The hyperparameter lambda corresponds to the lambda parameter when calculating the Generalized Advantage Estimate (Schulman et al., 2015), that is, it dictates the emphasis placed on the current value estimate when calculating an updated value estimate. The hyperparameter 'time horizon' dictates how many steps of experience to collect before adding the transaction to the replay buffer, and therefore represents the 'temporal difference' of the underlying TD algorithm. Note that in the Stable Baselines implementation of SAC, this value is always one, i.e. the underlying algorithm is TD(0).

The algorithm hyperparameters for the run presented in Figure 5.3 were as follows: batch size 32, buffer size 10240, number of epochs 3 (equivelant of gradient steps), learning rate 0.0003, learning rate schedule 'linear', gamma 0.8, beta 0.005, epsilon 0.2, lambda 0.95, time horizon 64, summary frequency 10,000 and maximum steps 5 million. All of these values barring gamma are the default values used by ML-Agents. The default value for gamma is 0.99. In the ML-Agents documentation, it is recommended that gamma be large like this for sparse rewards, and smaller in cases of immediate reward. Given the density and immediacy of the Luo et al. reward function (equation 5.1), gamma was reduced to 0.8. The one environment hyperparameter in CubeTrack, $optd$, was set to 3 in Figure 5.3.

One of the problems encountered during agent training in CubeTrack was a consistent overshooting, resulting in a pendulum-like movement in front and behind the target. When using the ML-Agents plugin there are some additional hyperparameters besides the DRL algorithm hyperparameters (see section 3.3). The agent object has a 'Decision Requester' component with the integer field 'Period' (number of frames between each action decision implementation) and the Boolean field 'Take Actions Between Decisions'. By default, these are set to 10 and True. Changing the decision period to 1 got rid of the overshooting problem and did not deteriorate the performance of the simulation.

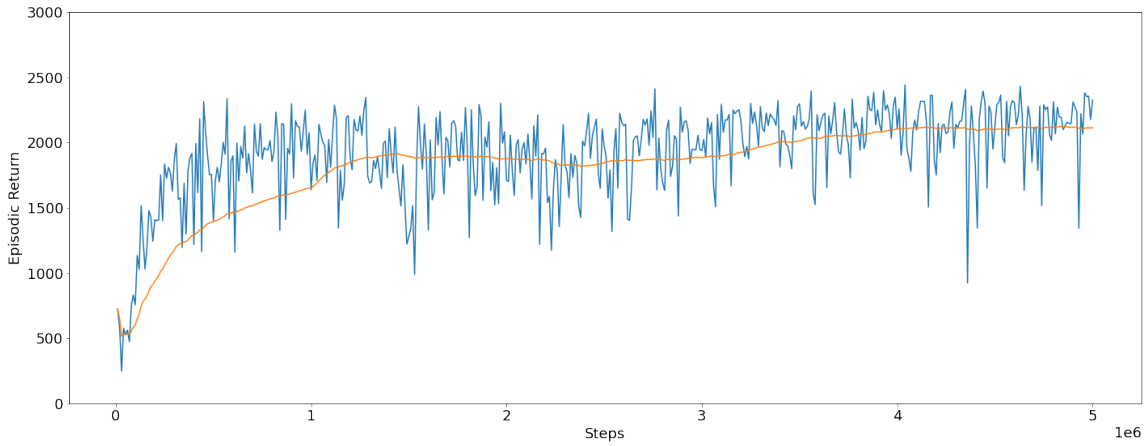


Fig. 5.3 Graph of episodic return for final version training run in CubeTrack.

In CubeTrack, there were two Tensorboard outputs, one for episode length (number of steps taken by the point of episode termination) and one for episodic return (the cumulative discounted reward achieved by the point of episode termination). The first is not shown since it presented a flat line at 3,000 steps. Recall that, in CubeTrack, there were only two episode termination criteria: a maximum episode length of 3,000, and a minimum return threshold of -450. The episode length graph therefore tells us that this second criteria was never hit. Figure 5.3 is the episodic return graph for the final iteration run. Note that this is episodic return *during* training. The blue line connects a time series of data points, plotted every 10,000 steps (i.e. the summary frequency). The y-axis value for each data point is the average episodic return across completed episodes that have occurred during the last 10,000 step period. Given that episodes were consistently 3,000 steps long, this is always an average over three episodes. The orange line provides a smoothed representation of the same data, using a moving average over the last 100 data points. The positive gradient is indicative of a successful training run.

Note that, theoretically, the maximum return achievable in any given episode is 3,000, given that the maximum episode length is 3,000 steps and the maximum reward per step is +1.0. However, this would require the agent to be in exactly the right position from the first step of the episode to the last. In practice, this is unrealistic, especially given that the episode starts with the two cubes spawning at random locations on the platform and therefore requires searching for the target and reaching the target before tracking. As such, achieving around 2,300 return is considered a good level of performance. Episodic return fluctuates, but within a reasonable range, rarely dipping below 1,500. The length of the training run was set to 5 million steps, with no early stopping criteria or callback to save the model mid-run. The

number of episodes is not explicitly set but determined by the number of times the episode termination criteria is hit. With episodes consistently terminating at 3,000 steps, the number of complete episodes in the 5 million step run was therefore 1,666. Wall clock time was 2 days, 7 hours, 54 minutes and 25 seconds.

5.4.2 DonkeyTrack results

DonkeyTrack runs were also performed locally on the same Lenovo Thinkpad T480s laptop. All training runs made use of the Stable Baselines implementation of SAC. Specifically, they used code from the open source 'Learning to Drive Smoothly in Minutes' (2019) repository, which provides its own class definition of SAC, inheriting from the Stable Baselines SAC class. The minor difference is that it forces model optimisation (and Tensorboard logging) to trigger upon episode termination, as opposed to after a set number of steps dictated by the 'train frequency' hyperparameter (i.e. episodic learning as opposed to step learning). In Stable Baselines 3, this is built in as a hyperparameter, however the 'Learning to Drive Smoothly in Minutes' project was developed with Stable Baselines 2 prior to the release of Stable Baselines 3, hence the re-implementation of SAC. Episodic learning creates a small change to the way episodic return graphs should be interpreted. On all of the episodic return graphs presented below, the x-axis value of each data point is the total number of environment steps taken at the point the episode terminates (as opposed to multiples of 10,000) and the number of data points on the graph directly reflects the number of episodes in the run (as opposed to the maximum step length divided by 10,000). It also means the y-axis value is the episodic return for a single episode, not an average over a number of episodes.

The algorithm hyperparameters for the runs presented in Figure 5.4 were as follows: batch size 64, buffer size 30,000, gradient steps 600, learning rate 0.0003, learning rate schedule 'threshold drop' (explained below), gamma 0.99, entropy coefficient 0.05 (with auto-tuning), tau 0.005, learning starts 300, target update interval 1, action noise 'None', random exploration 0, and maximum steps 1 million. As explained, the 'train frequency' hyperparameter was not called upon since policy optimisation was triggered on episode termination. Maximum steps was reduced from 5 million to 1 million since there was marginal improvement post 1 million steps in CubeTrack. A maximum step count of 1 million is also what is reported in the Haarnoja et al. (2018) paper which introduced SAC, having explained the improved sample efficiency of the algorithm.

The environment hyperparameters *optd* and *maxd* were set to 15 and 12 respectively. At the beginning of work on DonkeyTrack, *optd* and *maxd* were set to 10 and 30 respectively. As

explained in Section 5.3.2, the agent exploited an environment loophole, colliding with the target car as early as possible such that the wheels interlocked and the agent remained in close proximity to the optimal tracking position, not far from the target. The best resolve for this was to set $maxd < optd$, meaning that the episode terminated due to breaching the maximum distance from the optimal tracking position prior to colliding with the target.

The algorithm hyperparameters that are common across PPO and SAC were initially carried across from CubeTrack, with the remainder being set to their default value. Poor performance then prompted values to be carried over from the 'Learning to Drive Smoothly in Minutes' (2019) project. Although this project addresses a different task (road tracking as opposed to target tracking), it utilises the same simulation (SDSandbox) and the same continuous action space, and was therefore more comparable than CubeTrack. When the agent continued to fail, the randomly moving target car was made to drive forward in a straight line, to simplify the problem whilst working through a process of environment debugging and manual hyperparameter tuning. The hyperparameters experimented with during this process were learning rate, learning schedule, entropy coefficient, gamma, and maximum steps. Had this tuning been performed on the final version of the environment, it would be worthwhile presenting the results of each run in order to shed some light on the influence of each hyperparameter. However, the reality was that environment defects were found late in the process, and the design of the environment continued to iterate in parallel with hyperparameter tuning. It is therefore not possible to disentangle the effect of hyperparamter changes from environment bugs. This, and the time that was lost to inconsequential hyperparameter tuning, is a common challenge when working with a custom environment as opposed to a tried and tested benchmark environment.

Training runs in DonkeyTrack completed in under 24 hours. For example, for the run shown in Figure 5.4c, the number of episodes (and therefore number of model updates) was 1,000, and the wall clock time was 17 hours, 41 minutes and 24 seconds. Comparing Figures 5.3 and 5.4, the latter has larger fluctuations, with higher peaks but much deeper troughs as well. The deeper troughs can be attributed to the stricter episode termination criteria in DonkeyTrack, terminating as soon as the agent's radial distance from the optimal tracking position exceeds 12. Given that the reward function issues a reward per time step, early termination has a huge influence on episodic return; episodic return is being affected by episode length as well as the reward magnitude at any given step. The higher peaks could be attributed to two environment differences. First, the episodes start with the agent and target in the optimal position, as opposed to random starting positions. Second, although the target is driving randomly, it is

not able to spin or make a hard turn – it drives consistently down the world z-axis despite weaving as it goes. The situation whereby the agent and target car are facing each other is much less likely than in CubeTrack, and as such reward values rarely come from the negative tail end of the [-1,1] range.

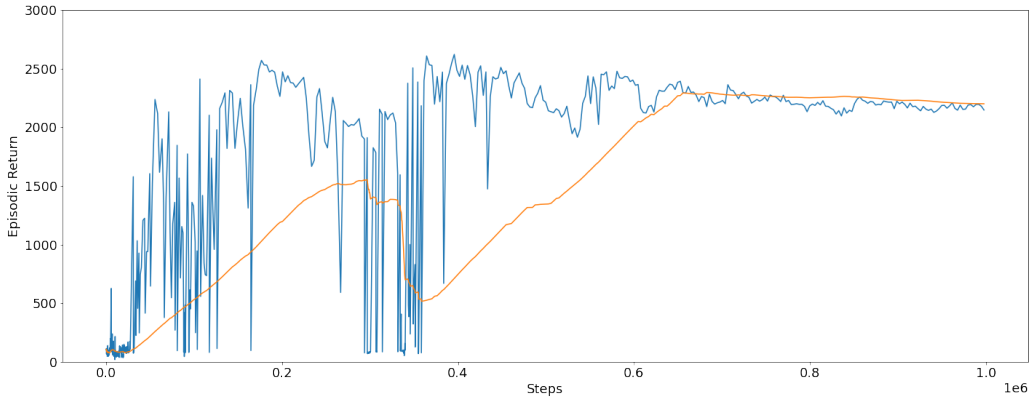
Custom learning rate schedule

Although the effects of hyperparameter tuning were obscured by environment defects, the effect of the learning rate was still relatively discernible. When the learning rate was too small (for example, 10^{-5}), the agent was not able to find a good policy, and the episodic return graph presented as flat. In this case, because network updates were small, variation in behaviour was low and therefore variation in experience was low. When the learning rate was too high (for example, 10^{-3}), the agent was able to find a good policy quickly, but then episodic return would fluctuate until the end of the run without any improvement in the depth of the troughs or signs of convergence. In this case, because network updates were large, the agent would step over the global minimum of the objective function.

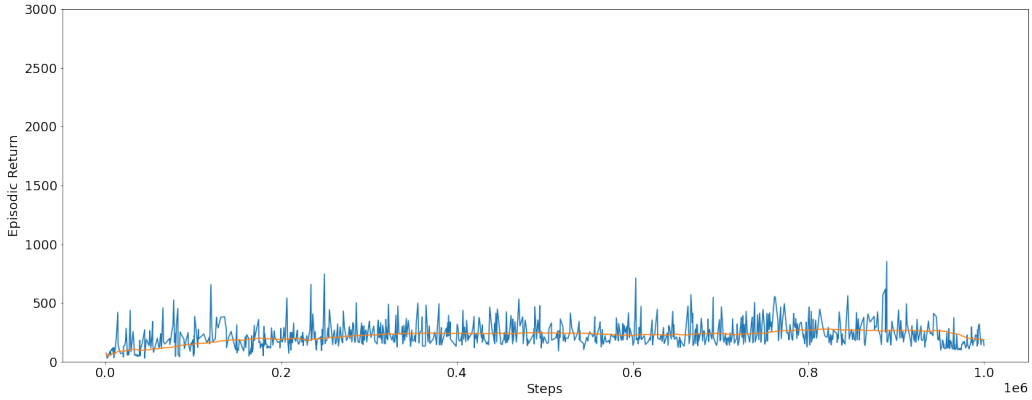
Stable Baselines provides a linear learning rate schedule in order to try and balance these two described problems, annealing the value from a provided starting value (hyperparameter) to zero at a constant rate. Finding the right starting value (and therefore gradient) for this schedule proved difficult. Another schedule provided by Stable Baselines is called 'middle drop'. This schedule is the same as the linear schedule for the first half of the training run and then drops the learning rate to a lower constant value (a tenth of the starting value). Since Stable Baselines supports the implementation of custom schedules, a schedule was designed whereby the learning rate starts off as linearly annealing and then drops to a tenth of the starting value, the same as the 'middle drop' schedule, but with two important differences. Firstly, the learning rate continues to anneal even after the drop. Secondly, rather than have the learning rate drop half way through the run, the drop is triggered by crossing a performance threshold. Specifically, the learning rate is dropped as soon as mean episodic reward (the moving average over the last 100 episodes) surpasses 1,000. By basing the learning rate around agent performance, the success of the run was less sensitive to the chosen starting value.

Comparing reward functions

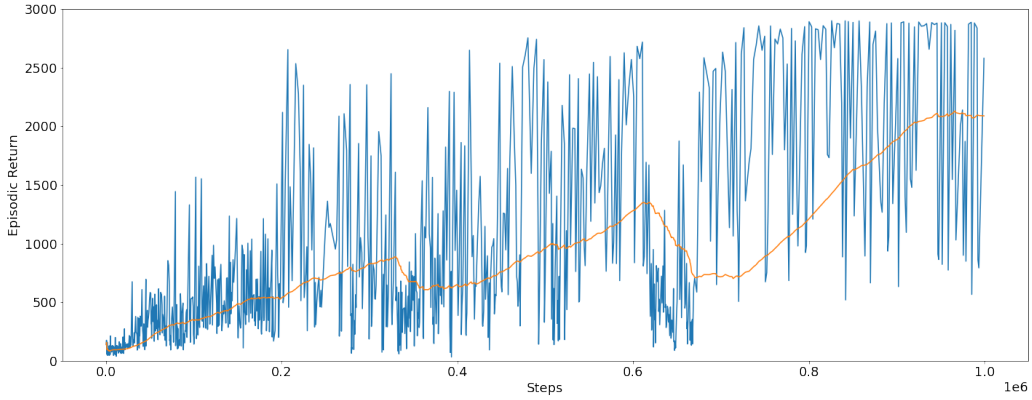
Section 5.3.2 describes two different versions of the reward function. The first is that adopted from Luo et al. (2018), whereby $optd$ is a fixed point somewhere along the target's negative



(a) Episodic return for training run with straight target and 'tailgating' reward function implementation.



(b) Episodic return for training run with random target and 'tailgating' reward function implementation



(c) Episodic return for training run with random target and 'radial' reward function implementation

Fig. 5.4 Comparing episodic return graphs across reward functions in DonkeyTrack.

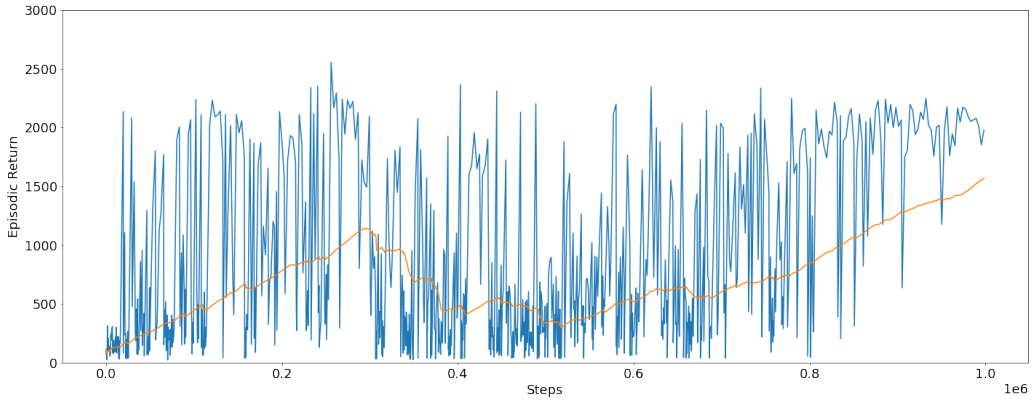
z-axis, promoting a tailgating behaviour. When the target travels in a straight line, the agent learns to successfully track the target, converging on a policy that achieves high episodic return of around 2,300 (see Figure 5.4a). However, when the target is able to turn, the same reward function, algorithm, and hyperparameters fail, with an average episodic return of around 200 (see Figure 5.4b). Figure 5.4c is then the same random target and hyperparameters, but with the second version of the reward function. This is the function detailed in equation 5.2, whereby the point of highest reward is now any point on a circle centred on the target, with a radius of $optd$.

Although the model does not converge as well as the run in Figure 5.4a, the agent regularly achieves an episodic return even higher than in the straight target environment, at around 2,800. In addition to saving out the network weights at the end of the training run, a callback was implemented to save the network weights whenever the return for the given episode was larger than the average episodic return over the last 100 episodes (depicted by the orange line). In the run shown in Figure 5.4c, the 'best' policy and the end policy are the same, however, for runs that deteriorate toward the end, this functionality has more value.

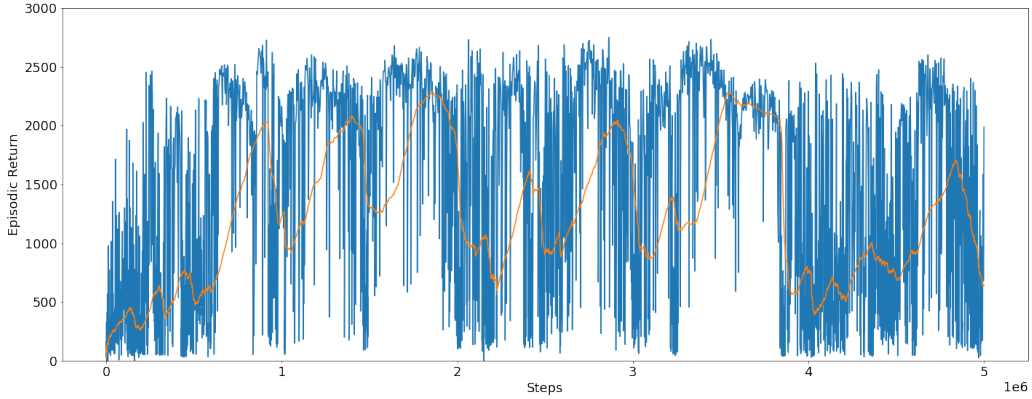
5.4.3 SWiMM DEEPeR results

Runs in SWiMM DEEPeR were performed locally on a Razer Blade Pro laptop (Intel Core i7-10875H CPU, 2.30GHz - 5.10 GHz processor base frequency). The SAC implementation was the exact same implementation used in DonkeyTrack. The environment configuration was set to two dimensions of control, with the dolphin target limited to the y-plane.

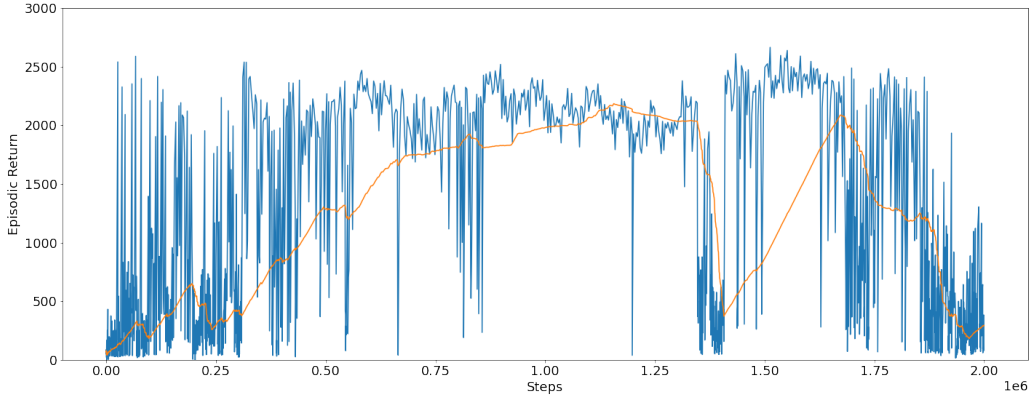
Hyperparameters were carried over from DonkeyTrack: batch size 64, buffer size 30000, gradient steps 600, learning rate 0.0003, learning rate schedule 'threshold drop', gamma 0.99, entropy coefficient 0.05 (with auto-tuning), tau 0.005, learning starts 300, target update interval 1, action noise 'None', random exploration 0, and maximum steps 1M. In terms of environment hyperparameters, because the simulated camera was matched to the specification of the real camera on the BlueROV2, it was necessary to reconsider the value of $optd$, making sure that the target was clearly visible in the camera image at the distance specified as optimal. Based on experimentation, $optd$ was set to 6 and $maxd$ to 4. SWiMM DEEPeR also has the additional environment hyperparameter of action inference. This was always left at it's default setting 'maintain', whereby the same forces are applied until the next action decision is received.



(a) One million step run with the same hyperparameters as DonkeyTrack.



(b) Five million step run with the same hyperparameters as DonkeyTrack.



(c) Two million step run with the batch size increased from 64 to 512.

Fig. 5.5 Episodic return graphs for the SWiMM DEEPeR environment.

Figure 5.5a provides the results for these reported hyperparameters. This run took 1 day, 2 hours, 57 minutes and 36 seconds. The slower wall clock time relative to the same length run in DonkeyTrack is due to the stricter implementation of the communication framework, with the client awaiting a decision from the server before sending the next environment observation. Comparing Figures 5.5a and 5.4c, model convergence is slightly better in SWiMM DEEPeR, as indicated by the much smaller fluctuations toward the end of the run, and the fewer number of episodes (910 compared to 1000). However, peak episodic return is lower, levelling off closer to 2,000 than 3,000. As with the DonkeyTrack run, the 'best model' was also the final model in this instance.

The maximum steps hyperparameter was increased from 1 million to 5 million to test whether simply lengthening the training run would offer any improvement, particularly to convergence. Figure 5.5b provides the results. Were the graph not so compressed (bearing in mind that the same width x-axis is representing a range five times as large), it would be easier to appreciate that there are some periods of consistently high return, however, these are always followed up by a period of large fluctuations – the model does not converge despite the longer training run.

The next experiment involved increasing the batch size. A batch size of 64 had been brought across from the 'Learning to Drive Smoothly in Minutes' (2019) project, however this is very low for DRL, especially in the context of a complex, three-dimensional environment (see McCandlish et al. (2018) for empirical examples of batch sizes ranging from thousands to millions of samples). With a small batch size and complex environment, the agent can end up with a batch that represents some idiosyncratic part of the problem. A larger batch size can help stabilise training and improve final performance (RL Wiki, 2020). Moreover, since advantage estimates only offer a faint signal, it is particularly important to reduce variance in gradient estimates by averaging over a greater number of samples. As such, DRL can tolerate (and benefit from) much larger batch sizes than other machine learning tasks.

Figure 5.5c presents the results of a 2 million step run with batch size increased to 512. In this run, episodic return deteriorates dramatically toward the end, however prior to this, there is a huge improvement in the shape of the graph, with the policy stabilising fairly well as early as 500k steps. The 'best policy' callback ensures that this stability and high performance is captured.

5.4.4 Inference results

For all three environments, model performance was evaluated over a 100 episode inference run, as is standard in the DRL literature (Henderson et al., 2018). Recall that, when performing inference, SAC changes the policy from stochastic to deterministic, with the mean of the action distribution being used directly as the action decision. In CubeTrack, mean episodic return was 2,145 and mean episode length was 2,967. In DonkeyTrack, mean episodic return was 2,057 and mean episode length was 2,197. In SWiMM DEEPeR, mean episodic reward was 2,120 and mean episode length was 3,000.

Whilst mean episodic return and episode length are useful metrics and are very widely used in the DRL field, it is important to remember that there is no guarantee that high values are associated with desirable behaviour. Although less objective, behavioural observation is therefore extremely important for policy evaluation. Videos of the trained agents, in all three environments, have been uploaded to a dedicated YouTube channel for easy reference[1]. Evidence of learnt active tracking is very clear in all cases. The agent will turn when the target turns, accelerate when the target strays too far, and decelerate to avoid collision. The latter is most pronounced in CubeTrack when the target pivots and heads toward a new waypoint, and the agent is able to reverse and quickly get out of the way. In the CubeTrack and first version of the DonkeyTrack environment, the agent tailgates the target in close proximity. Whilst this provides a good demonstration of the responsiveness of a dynamic method like DRL, it is not the appropriate behaviour for the application. Increasing $optd$ and changing to the radial implementation of the reward function produces a more application-appropriate behaviour; the agent keeps the target in frame without crowding or tailgating. In SWiMM DEEPeR, the solid blue background makes it difficult to appreciate that the AUV is moving. To help with this and to support debugging, a green ring was added to the simulation to mark out the optimal position, and red rings to mark out the maximum distance which triggers episode termination. These rings are not visible in the camera images fed to the algorithm.

5.4.5 Responsible reporting

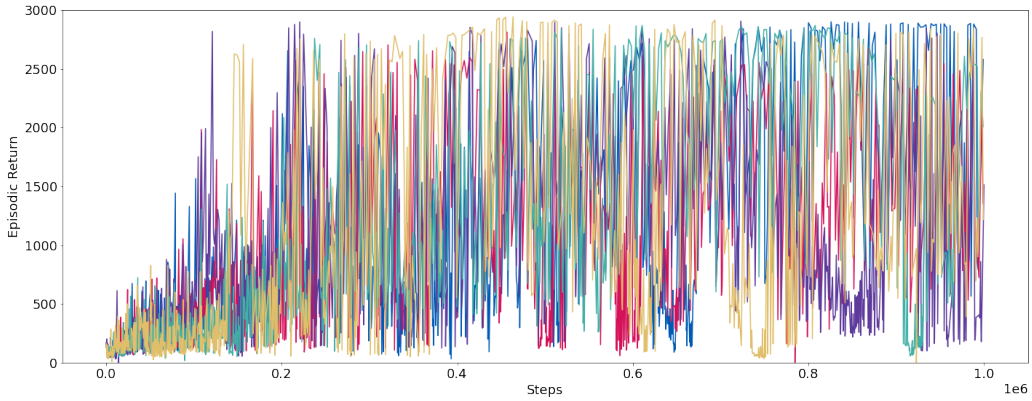
Reproducing results is particularly difficult in deep reinforcement learning because of the number of sources of non-determinism. As with any deep learning or machine learning algorithm, performance is affected by: hyperparameters (including architectures), random seeds, the number of averaged runs, network weight initialisation, the exact implementation provided by the codebase, and the data itself (or the environment, in the case of DRL).

[1] https://www.youtube.com/channel/UCA4fgSfe2IctRv5N-Gr0OrQ

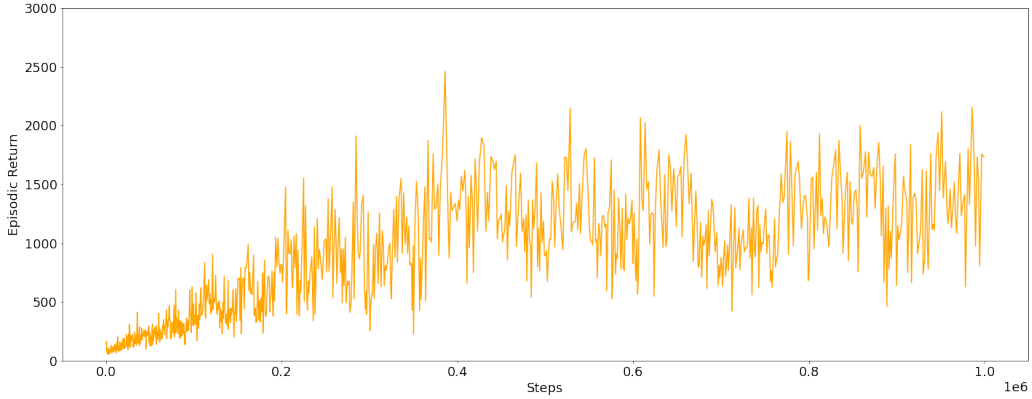
Additionally, a DRL environment will likely have its own hyperparameters, for example, the travelling speed of the target object. The reward function and episode termination criteria are also parameters of the environment, and will have huge influence on the success of a given algorithm. This includes not only the reward function itself, but details such as reward scaling (multiplying the outputs of the reward function by some scalar) which have been shown to have a large effect (Henderson et al., 2018).

In the supervised learning setting, it is possible to replicate across all of these components and get the same results (assuming that they have been properly reported). However, in online DRL, it is not possible to replicate the data component, even when all environment parameters are matched. Data comes from environment interactions and environment interactions depend on the weights of the policy network each episode, and so run-to-run variability is high. Within runs, varied environment experiences are integral to learning and so most DRL algorithms introduce further non-determinism, either by injecting noise into the policy or by making the policy itself stochastic. For example, in SAC, the values returned by the policy network are treat as the parameters of a distribution. At training time, actions are sampled from the returned distribution whereas at inference time, the mean is taken as the action decision.

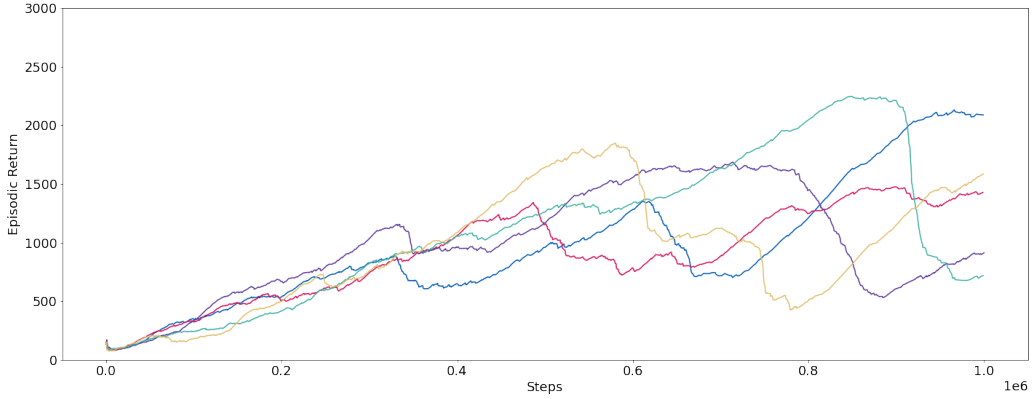
It is therefore particularly important that algorithms be evaluated with an average over multiple runs; reporting the best performing run is not a robust measure of algorithm performance. In 'Deep Reinforcement Learning That Matters' (Henderson et al., 2018), training runs are replicated five times before reporting the average, and this is considered a bare minimum that ought to become standard practice. In other papers around reproducability, the suggested minimum is as high as twenty (Colas et al., 2018), although arguably there is a trade off here between responsible reporting and responsible computing, with an environmental cost to huge numbers of compute hours. Figure 5.6 presents five replications of the DonkeyTrack training run, with the hyperparameters listed at the beginning of Section 5.4.2, the random moving target and the radial reward function. The five episodic return graphs presented together (5.6a) are impossible to decipher and so the average *across* those five runs is presented on separate axes (5.6b). The five runs are also shown together but as a 100-episode moving average within runs, to allow for a clearer comparison. As can be seen, there is a huge amount of variance across runs, with one run collapsing at around 900,000 steps, highlighting the importance of replication.



(a) All five runs shown as raw episodic return.



(b) Average episodic return across the five runs.



(c) All five runs presented as a moving average.

Fig. 5.6 Graphs of episodic return for five equivalent training runs.

Observation Type	Episodic Return	Episode Length
T2FO	**2,057**	**2,197**
Standard VAE	1,198	746
Raw Images	1,049	653
State Predictions	1,987	845

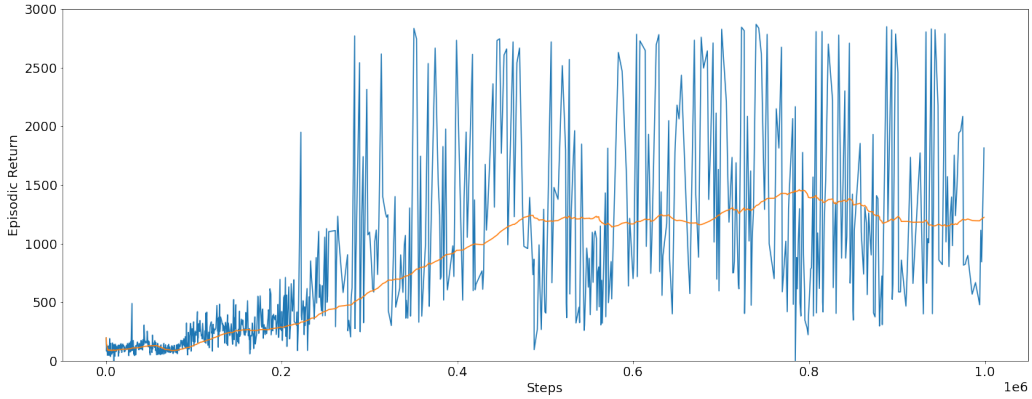
Table 5.1 Comparing agent performance metrics between the proposed framework T2FO and other observation types. Standard VAE: same framework of feeding the DRL policy network feature observations, but using an unconstrained VAE trained with the standard VAE objective function. Raw images: pixel data fed straight to RL policy network. State predictions: taking the auxiliary networks used in training the constrained VAE forward into agent training, and using the predictions of distance, azimuth and yaw as the observation

.

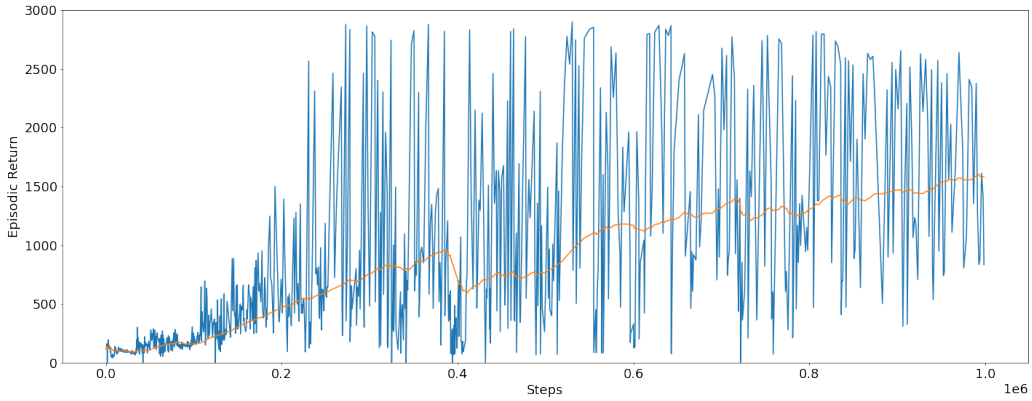
5.4.6 Ablation experiments

As with feature learning, an ablation study was conducted to properly decipher the contribution of the proposed solution elements to policy learning. Across all three ablation experiments, policy training was performed in the DonkeyTrack environment, with the hyperparameters listed at the beginning of Section 5.4.2, the random moving target and the radial reward function (the same as the five run replication). Firstly, the contribution of the feature disentanglement (i.e. the VAE constraint) was tested. In Chapter 4, this was tested purely with respect to feature learning performance. Here, the contribution is tested with respect to policy learning, by comparing policy performance (mean episodic return) to a policy fed features from an unconstrained VAE (the one trained and reported on in Chapter 4). Table 5.1 provides a comparison of inference results following a 100-episode inference run. Average episodic return drops from 2,057 for the policy trained with T2FO (task-relevant feature observations from the constrained VAE), to 1,198 for the policy trained with unconstrained VAE features. Similarly, average episode length drops from 2,197 to 746. Figure 5.7a provides the episodic return graph for this training run. This shows that high-return episodes were achieved, but much less consistently, with the 100-episode moving average (depicted by the orange line) never surpassing 1,500. These results suggest that the dissentangled, task-relevant features of T2FO improve learning of VAT.

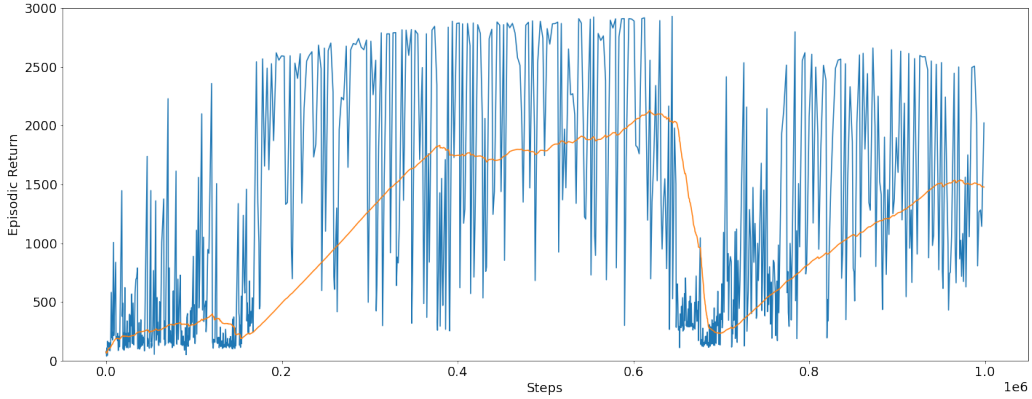
Next, the contribution of decoupling feature learning from policy learning was tested. Two additional ablation experiments were ran, one with raw images as observations, and one with $[\hat{r}, \hat{\theta}, \hat{\psi}]$ as the observation, i.e. a vector of state predictions from the auxiliary networks used to trained the constrained VAE. These observations represent end-to-end and task-separated solutions to VAT respectively. In Section 5.2.2, it was explained that the proposed solution



(a) Episodic return for run with feature observations from unconstrained VAE.



(b) Episodic return for run with raw image observations.



(c) Episodic return for run with T2FO framework.

Fig. 5.7 Comparing episodic return graphs between the proposed framework T2FO and other observation types

of policy learning from feature observations can also be considered end-to-end learning of VAT. Although not strictly end-to-end in the same sense as using images as observations, the DRL policy is still tasked with estimating target state from feature observations, and so state estimation and control are still jointly optimised. By feeding the policy network the decoder output of state predictions and not the encoder output of features, the transition is made into task-separated.

Again, Table 5.1 provides a comparison of inference results and Figure 5.7 provides episodic return graphs for training. Average episodic return drops from 2,057 for the policy trained with T2FO, to 1,049 for the policy trained with raw images. Similarly, average episode length drops from 2,197 to 653. Like with unconstrained VAE features, Figure 5.7b shows that high-return episodes were achieved, but less consistently. The 100-episode moving average is marginally better than the first ablation experiment, but still under-performs in comparison to T2FO. Figure 5.8 provides a separate graph of the moving averages to show this more clearly. The red line represents policy learning with T2FO (specifically, the run presented in Figure 5.4c) whilst the purple line represents policy learning from unconstrained VAE features and the yellow line represents policy learning from raw images. Note that training runs with raw image observations were more computationally demanding because the required CNN architecture for the policy and value networks increases the wall clock time of each update ten fold. When attempting to perform image observation training runs locally on the Lenovo Thinkpad T480s laptop, the laptop failed (overheated) before completing 1 million steps, having taken 1 day, 6 hours and 26 minutes to complete 500,000 steps. It was therefore necessary to perform this training run in the cloud, on a virtual machine with a GPU and 6 CPUs (specifically, the Standard NC6s v3 from the Microsoft Azure NCv3-series). This reduced training time to under 24 hours, the same as feature observation runs.

For state prediction observations, average episodic return drops less severely from 2,057 to 1,987, and average episode length drops from 2,197 to 845. Figure 5.7 shows that the policy was fairly consistently achieving very high episodic return from around 200,000 steps, but that performance crashed around 700,000 steps. The green line in Figure 5.8 shows that although end-of-run performance was below that of policy learning with T2FO, for a long period mid-training episodic return is equally high.

Insights from these ablation experiments are particularly valuable given that, to the best of our knowledge, the work presented in this chapter is the first report of VAT policy learning from task-relevant features. The finding that learning from raw images or unconstrained

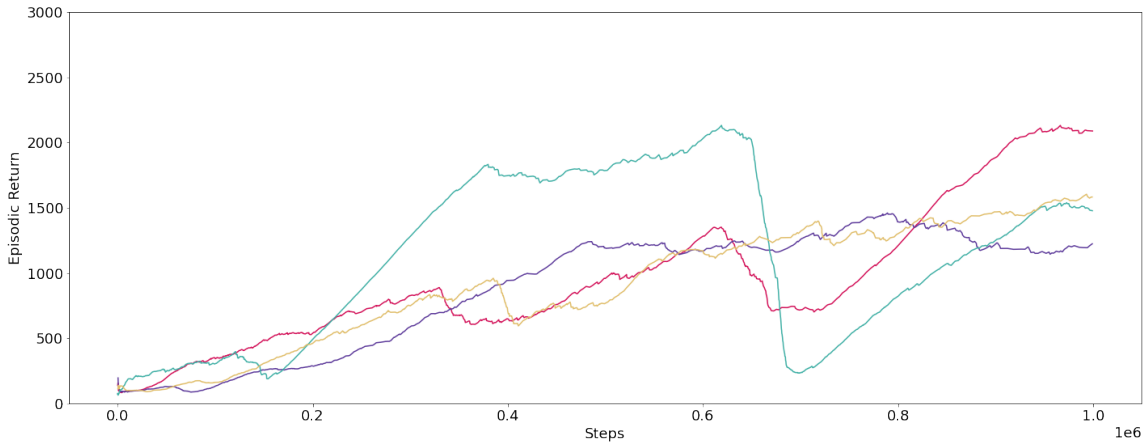


Fig. 5.8 Comparing moving averages of episodic return between the proposed framework T2FO and other observation types. Red = T2FO, purple = unconstrained VAE features, yellow = raw images, green = state predictions.

features is less effective than learning from task-relevant features corroborates with the findings of Bonatti et al. (2020) in the aerial navigation problem space, and Toromanoff et al. (2020) in the autonomous driving space. The finding that learning from state predictions was actually quite effective was more surprising. Interestingly, all of these results go against the results of Levine et al. (2016b), where the task is complex object manipulation with a robotic arm. Here, learning from raw image observations was best (albeit slower), state predictions was worst (specifically, pose estimation), and learning from features was intermediate. It is possible that this problem space is much less comparable due to the higher degree of accuracy required. Tolerance is at the millimetre level which is not the case for the VAT problem addressed by this thesis.

5.5 Summary

This chapter takes the novel environments detailed in Chapter 3, and the trained constrained VAE detailed in Chapter 4, and uses them in combination with a DRL algorithm to train a vision-based agent to perform active tracking. The Unity application renders the simulated camera and sends the image across the network to the Python application, along with the AUV and target coordinates. The image is passed through the constrained VAE encoder, which outputs a 1×10 vector of task-relevant features. The feature vector is passed through the policy network, which outputs a control decision, sent over the network to the Unity client where it is translated into force applications. The agent and target coordinates are passed

as arguments to the reward function, where distance (with respect to the optimal tracking position) and angle (between target heading and agent facing direction) are calculated. Distance and angle are normalised, summed, and subtracted from 1, producing a reward value in the range [-1,1]. When an episode is terminated or reaches the maximum number of steps (3,000), the environment is reset and the policy and value networks are updated according to the Stable Baselines implementation of SAC.

Successful VAT was achieved in all three environments. Algorithm performance was assessed according to graphs of episodic return throughout training, average episodic return and episode length over a 100 episode inference run, and visual inspection of agent behaviour. Several novel contributions were made throughout the work presented in this chapter. The DRL algorithm was the Stable Baselines implementation of SAC but with a custom learning rate schedule implementation. The reward function was taken from Luo et al. (2018) but modified to reward radial proximity and not tailgating, and to reward facing the target and not the direction the target is facing. The concept of pairing a constrained VAE with policy learning was taken from Bonatti et al. (2020), but was applied in a new problem space (VAT as opposed to aerial navigation) and with a new policy learning method (DRL as opposed to imitation learning). The efficiency of automated data collection was also improved.

Given that the solution involves two separate models, it is not 'end-to-end' in the usual sense of the word. However, in terms of approaches to active tracking, the solution is viewed as an 'end-to-end' approach and not a task-separated approach – a view also taken by Levine et al. (2016a). This is because, although the VAE handles some of the perceptual heavy lifting (reducing the dimensions of the input from raw image dimensions to one dimension), the perceptual task of state estimation is still handled by the same algorithm providing control. In other words, the perception and control elements of visual active tracking are jointly optimised, but the decoupling of feature learning from policy learning improves efficiency by learning the foundations of visual perception offline. Ablation experiments confirmed that the dissentangled, task-relevant features provided by the constrained VAE improve learning of VAT. Ablation experiments also confirmed that decoupling feature learning from policy learning both improves the learning of VAT and reduces computational requirements. What was less convincing in the results of the ablation experiments was the inferiority of a task-separated solution to VAT. This argument is the main premise of the Luo et al. (2018) paper, but here the finding was that learning from state predictions produced a similarly high performing policy. However, this is likely only the case because of the empty environment and distractor-free problem setting. As the complexity of the problem is scaled, the RL

agent will require more contextual information, and jointly optimising state estimation and control will perhaps become increasingly beneficial. Put differently, the state prediction observations (task-separated solution) provide information on target distance, azimuth and yaw only, whereas the feature vectors in T2FO provide this information *plus* a number of abstract features encoded by the model. As background information becomes more pertinent to the task, it is expected that the performance of these two solutions will diverge.

Training in custom, novel environments as opposed to tried and tested benchmark environments proved particularly challenging, since environment bugs failed silently and poor performance was attributed to reward function design or hyperparameters. The sheer number of hyperparameters (environment as well as algorithm) and the high inter-run variance added to the challenge. These factors, along with the particularly long warm up period, made automated hyperparameter tuning infeasible when training from randomly initialised network weights. Instead, hyperparameter values were copied over from the 'Learning to Drive Smoothly in Minutes' (2019) project and tuned manually. Much of this manual tuning process was inconsequential because of environment bugs, but clear improvements were observed from the custom learning rate schedule and increase to batch size.

After over 70 training runs and over a year of development work, agent behaviour is at a high standard, however there is still room for improvement, with mean episodic return falling short of the best results reported by Luo et al. (2018), and episodic return graphs often highlighting a lack of convergence. Given more time, it would be interesting to see if this performance gap could be closed with a further increase to the batch size, paired with a linear increase to the learning rate (suggested in RL Wiki (2020)). Given the large search space involved with tuning two interdependent hyperparameters, this would really need to be an automated process, leveraging a library like Optuna and transfer learning (initialising networks with the weights saved out from the best runs reported in this chapter).

Chapter 6

Conclusion

6.1 Thesis Summary

The work described in this thesis was motivated by a real-world problem put forward by the Marine MEGAfauna Lab of the School of Natural and Environmental Sciences (SNES). This research group is conducting essential species monitoring, and are forward-thinking in their approach to doing so. Alongside more traditional methods of data collection, they are keen to integrate unmanned underwater and surface vehicles in order to improve the variety and quality of data collected. The group has access to a commercial remote operated underwater vehicle, the BlueROV2, and are interested in the feasibility of developing autonomous capabilities for this high-performing but affordable underwater vehicle. As explained in Chapter 1, delivering this as a deployment-ready system is well beyond the scope of a single doctoral project from a standing start.

What this thesis provides is a first look at the computer science elements of the wider, multi-disciplinary project. Firstly, the development of custom pseudo-realistic DRL environments with a Unity-hosted front-end. Secondly, training a constrained VAE to encode environment observations (single frame images from the simulated vehicles onboard camera) to a one-dimensional feature vector, with task-relevant information represented in the first three feature channels. Thirdly, the training of a control policy in-simulation, using the pretrained constrained VAE as an observation preprocessing module, in a framework we are naming T2FO.

6.2 Evaluation Against Thesis Aims

In Chapter 1, three novel DRL environments were listed in the thesis contributions section: 1) CubeTrack, a toy VAT environment involving a target and agent cube on an enclosed platoform, perfect for starting out, reward engineering or testing new algorithms, 2) DonkeyTrack, a new, modified version of the open source car racing environment 'gym-donkeycar', adapted to support mid-difficulty (open world but still 2-dimensional) VAT 3) SWiMM-DEEPeR, a highly custom underwater environment developed according to the requirements of the very specific real-world problem addressed by this thesis. All three of these environment contributions have been developed to completion, tested, open sourced, and used to train VAT agents. The CubeTrack environment has been available as a public GitHub repository since November 2020, and has been forked three times and starred ten times. This development work was also written up as an online Medium blog, where it was picked up and published by online journal Toward Data Science (Crane, 2020). The DonkeyTrack and SWiMM DEEPeR repositories were made public at the point of thesis submission. The development of SWiMM DEEPeR was written up as a conference paper and submitted to IEEE Conference on Games 2023. The paper was accepted and presented in-person at the conference. Post conference, an invitation was received to contribute an extended version to the IEEE Transactions on Games journal, running a special issue to highlight the best works from the conference.

It is hoped that all three environments are picked up and used by the DRL community, and prove useful in the progression of this field. SWiMM DEEPeR in particular was developed with a focus on reducing dependencies, reducing runtime, maximising flexibility, and maximising accessibility. As discussed in Chapter 3, the number (and species) of target objects, and the number of control dimensions, can be configured by the user server-side without needing to rebuild the Unity application. This thesis only reports results for tracking a single target in two dimensions (a flat plane); the VAT problem was scaled back to this simpler problem with the intention of progressing to three control dimensions, however the reality was that the full project was spent tackling environment bugs and working to improve the stability and convergence of the algorithm. Given more time, it would have been good to utilise this environment in full, however it is hoped that the accessible design will allow others to use it for training control policies up to five dimensions of control, and for training multi-object VAT policies.

Beyond the development of the environments, the thesis contributes a new framework for training VAT agents, T2FO. This framework involves first training a constrained VAE, and then taking the encoder network from this trained model and using it to encode raw image

observations of the DRL environment to task-relevant features for the DRL agent. This framework was not utilised in CubeTrack, the only purpose of CubeTrack was to provide a simple testbed for reward engineering. It was however used in both DonkeyTrack and SWiMM DEEPeR. Both of these environments produced desirable agent behaviour, that is, the tracking vehicle followed the moving target object whilst avoiding collision and keeping the target in-frame. This can clearly be observed in the videos of the agents in action, but is also reflected in the mean episodic reward and episode length values of the 100-episode inference runs. The DonkeyTrack agent produced a mean episodic reward of 2,057 and a mean episode length of 2,197, whilst the SWiMM DEEPeR agent produced a mean episodic reward of 2,120 and a mean episode length of 3,000. Critically, when this performance was compared to same-environment, same-algorithm (SAC) runs without the proposed framework, performance deteriorated. With raw image observations (no decoupling of feature learning from policy learning), the mean episodic return of 2,057 in DonkeyTrack dropped to 1,049. With standard VAE feature observations (decoupled feature learning but no use of the cross-modal approach to constraining features) episodic return dropped to 1,198. Interestingly, when the cross-modal approach to VAE training was utilised, but the state prediction decoder networks were taken forward into agent training as opposed to the image encoder network, episodic return dropped but only very slightly to 1,987.

In terms of the constrained VAE training that preceded agent training, the performance of the VAE on simulated images was comparable to the in-simulation results of Bonatti et al. (2020). To aid direct comparison, the same visualisations as Bonatti et al. were generated. These visualisations provided the same indicators of high performance as the Bonatti et al. visualisations: the model produced reconstruction images that were faithful to the context of the input image, and latent space interpolations demonstrated the desirable latent space qualities of completeness and continuity. In DonkeyTrack, quantitative results (mean absolute error over state predictions) were also similar, at 0.46, 0.65 and 12.09 for distance, azimuth and yaw respectively (compared to 0.39, 2.6 and 10 in Bonatti et al.'s AirSim results). In SWiMM DEEPeR, there was the same quality of visual results, but greater prediction error, particularly in the case of yaw prediction. This was attributed to the requirement for 360° predictions when the target's front- and rear-facing profiles were indistinguishable at the chosen image resolution.

Despite these promising results, a number of limitations have been identified that would need to be taken into consideration going forward. With agent training, as discussed in 5.4, even same-environment, same-hyperparameter runs can be hugely variable due to the multiple

sources of stochasticity in SAC and DRL more generally. Repeatable convergence was only possible when the VAT problem was simplified to a straight-travelling target. Random moving targets were successfully tracked, but model performance was more prone to large fluctuations right through to the end of training, and even model collapse. It is also worth reiterating that, for the work in this thesis, the VAT problem has been significantly scaled back to distractor-free, single-object tracking with two dimensions of control. Given the difficulty of producing successful training runs even with this reduced problem specification, it is likely that the current method, as it is, will fail as the problem is scaled. There is however a lot of scope for optimisation, and a lot of scope for combining methods, as is explored in the future work section below.

With the constrained VAE, it was not possible to replicate the section of the Bonatti et al. results whereby they show that the simulation-trained VAE is capable of encoding real-world images. The domain transfer experiment in Chapter 4 revealed that image encoding was not sufficient for successful domain transfer, even with the cross-modal approach to VAE training. As soon as the model was fed real world images the encoder failed, treating the image as 'unseen data' and mapping the image to a point outside of the learnt latent space (i.e. the decoder output was noise). Again, this is not being viewed as a reason to abandon the approach. The literature presents a large number of potential resolutions, again, discussed in the future work section below.

6.3 Future Research Directions

6.3.1 Addressing the visual domain gap

The domain transfer experiment revealed that dimensionality reduction alone is not sufficient to close the sim-to-real gap. The problem with training a policy in simulation is that, even when the underlying environment state is the same, an observation of the simulated environment and an observation of the real-world environment will always present a domain gap to some degree. What this means is that the two observations will be treated differently by the policy network, and produce different behaviour in a situation that should elicit the same behaviour. The hope with integrating the VAE was that, whilst the two contextually identical images might present different values in raw image dimensions, the 10 feature values would be the same, given that features represent higher-level properties and concepts. If the policy is fed feature vectors as input, it will therefore be oblivious to the domain difference. Whilst

this logic is sound, the problem simply transfers to the performance of the VAE. A VAE trained on simulated images does not encode equivalent real images to the same point in feature space.

Largely, there are two avenues with which to proceed: leaving the VAE as is and attempting to narrow the domain gap between the two types of input data, or further optimising the robustness of the VAE itself. If selecting the former avenue, an obvious approach is to improve the visual fidelity of the simulation, with reference to footage collected with the real vehicle. Anything from water clarity to god rays could be incorporated and carefully tuned. Special attention was paid to the vehicle's camera specification when setting the attributes of the simulated camera. Even so, a comparison between real and simulated camera frames would still be important, tuning the simulation by-eye until those images were as similar as possible.

Another approach is to pre-process real images before encoding. Some very basic methods from the OpenCV library were trialled, but yielded poor results. However, there are much more sophisticated, *learnt* processing methods to pursue. One such method, or family of methods, is domain adaptation, used to update the data distribution of a simulation toward that of the real world, or vice versa. In the visual domain, this problem is known as style transfer, and has seen solutions such as CycleGAN (Zhu et al., 2017). In Truong et al. (2021), bi-directional domain adaptation is successfully used to facilitate sim-to-real deep reinforcement learning in the control context. The authors contend that using a small amount of real-world data to train a domain adaptation model (i.e. adapting the simulation) is easier and more effective than using a large amount of real world data to train on top of the simulation-trained policy in the real world (i.e. directly adapting the policy). This is especially true for a model like CycleGAN where data is unpaired and training is unsupervised.

In terms of the latter avenue (improving robustness), methods such as domain randomisation are widely used across the sim-to-real field. Domain randomisation is a cheap, unsupervised approach, focused on exposing the agent to as much variation in the simulated environment as possible. After all, the gap between the real and simualted domain can be modelled as variability in the simulated domain (Peng et al., 2018). By randomising simple, arbitrary parameters in the simulated environment, robustness to variability is incentivised. Examples of visual parameters include the position, shape and colour of objects, the position, orientation and FOV of the camera, material textures, and lighting. Examples of physical parameters include mass and dimensions of objects, mass and dimensions of robot bodies, damping,

friction, and action delay. Probably the most well-known use of domain randomisation to support sim-to-real deep reinforcement learning in the control context is the 'Learning dexterous in-hand manipulation' paper from OpenAI (Andrychowicz et al., 2020).

Another approach is to incorporate raw, real-world FPV camera frames into the VAE training data. A huge benefit to feature learning offline, prior to policy learning, is that real-world data can be incorporated without having to exercise real-world control. Furthermore, the factored architecture of the cross-modal approach (i.e. the use of separate network branches) means that there is no requirement for this real-world data to be labelled. A simple convention, such as a difference in filename, could indicate to the model whether or not to pass any latent features forward for state prediction. The expectation is that a simulation-trained policy fed by a vision module trained on both simulated and real images will perform better in a real environment than a policy served by a vision module trained purely on simulated data. Whilst this sounds like low hanging fruit for a future work consideration, Bonatti et al. report that, to their surprise, this approach was not advantageous. They suspect the reason is that, since the policy is still trained purely on simulated images, there is a distribution shift between the data used to train the vision module and the data used to train the controller. They suggest a more sophisticated approach to incorporating real-world images into the VAE training data, put forward by Zhang et al. (2019). In this adversarial approach, two perception modules are trained. The first is trained with a simulated-only dataset before a second is trained with both simulated and real images. A discriminator is tasked with distinguishing which domain, simulated or real, an encoded feature vector comes from. The second perception module is tasked with fooling the discriminator, with support from the first, pretrained module.

Improving model robustness is an arguably stronger approach, given that a change to the environment would require retraining a domain adaptation model, for example. That being said, the approaches are not mutually exclusive. In future work, it would be interesting to compare and contrast all of these methods, and explore whether a combined approach produces cumulative improvement.

6.3.2 Addressing the physics domain gap

The other domain gap to consider is the difference in physics. In this instance, the model will select the action with the highest value given the observation, but the value of the action will be based on experience from the simulation, and the same action in the real environment may not be optimal. This problem is very much magnified in an aquatic environment. An important next step in simulation development will be to incorporate hydrodynamics. Al-

though hydrodynamic forces such as water currents are not natively supported in Unity, they could be imported. Alternatively, a custom solution could be developed, as was done with buoyancy. It will also be important to calibrate the force multipliers per movement axis. The policy network produces values in the range [-1, 1] per control dimension being optimised, and these are then multiplied by a chosen integer value before being passed to Unity's `AddForce()` and `AddTorque()` functions. Currently, when training 2-dimensional control in SWiMM DEEPeR, forward thrust is multiplied by 70 and yaw (steering) is multiplied by 0.5, as per the server configuration in Appendix A. Like in CubeTrack, these values were selected based on what looked reasonable (a subtle but visible movement) when running the simulation in manual control mode. Experiments with the real vehicle will be needed to calibrate acceleration, deceleration and turning arcs more accurately.

Beyond improving the physical fidelity of the simulation, it is likely that real-world policy training will also be needed. Whilst zero-shot transfer has been proven possible (for example, Bonatti et al. (2020)), it would not be surprising if this did not apply here, where the deployment environment is the North Sea. Training could be performed in a pool environment available at the University within the School of Engineering, using a model dolphin. Sea trials would then be necessary, with the vehicle attached to a safety line.

6.3.3 Introducing a memory component

With the current implementation, the policy bases the control decision on an encoding of a static visual input; it can decipher where the target is relative to the camera, but not where it has been, where it is likely to go, or any properties such as velocity. For a dynamic problem such as active tracking, the policy would benefit from a memory component. A single environment observation could be comprised of a sequence of feature vectors (or raw images), starting with the most recent camera frame to be received by the Unity client, and progressing T steps back in time. The 'Learning to Drive Smoothly in Minutes' (2019) project incorporates this into the design of the observation space, treating T as a hyperparameter. In this implementation, no changes are made to the policy and value networks; the T $1 \times N$ feature vectors are appended into a single $1 \times N \times T$ vector and treated the same as the single image implementation. In other works, such as the Luo et al. (2018) paper, a long short-term memory (LSTM) is used as the underlying architecture of the policy and value networks. The latter approach would require a change in algorithm, since the Stable Baselines SAC algorithm does not support recurrent policies.

Alternatively, the memory component could be built into the VAE. In the Ha and Schmidhuber World Models paper, the authors couple the vanilla and fairly shallow VAE with a MDN-RNN – a LSTM type recurrent neural network (RNN) with a mixture density network (MDN) output layer. Whereas the VAE, referred to as the vision module V, compresses what the agent sees at each time step, the MDN-RNN, referred to as the memory module M, compresses what happens over time. It does this via the task of predicting the next latent vector z_{t+1}. Specifically, since many complex environments are stochastic, the output of M is a probability density function $p(z)$, approximated as a mixture of Gaussian distributions and conditioned on the latent vector, action and hidden state, so that $P(z_{t+1}|a_t, z_t, h_t)$. Just as the use of abstract representations was grounded in neuroscience, so is this ability to predict future states, affording humans fast reflexive behaviours (Mobbs et al., 2015). Replicating this with a VAE and RNN-based world model prior to a much simpler controller is a distillation of concepts from earlier papers such as 'Learning to Think' (Schmidhuber, 2015) and 'Dream to Control' (Hafner et al., 2019).

6.3.4 Scaling control dimensions

When it comes to problem scaling, the most pressing future work would be the introduction of depth control. Like aerial navigation, underwater tracking is at the very least a three-dimensional control problem. Adding a third component to the Gym environment action space, and accounting for this on the Unity side, is trivial. In fact, SWiMM DEEPeR is already set up for three-dimensional target movement and vehicle control, requiring only a change to the server configuration file. The increase to the search space for the algorithm is not so trivial, and would benefit from introducing one dimension at a time, building on the previous learnt model i.e. curriculum learning (dividing the training process into a sequence of subtasks with increased difficulty to improve convergence) (Bengio et al., 2009). For example, Xiao et al. (2021) accomplish narrow gap traversing with a quadrotor drone, using SAC, sim-to-real and a curriculum learning approach to the gap dimensions. Other approaches which might help tackle the increased difficulty are guided policy search (a supervised form of policy learning which iteratively constructs the training data with a trajectory optimisation procedure) (Levine et al., 2016a) and supervised assisted DRL (a combination of DRL and behavioural cloning)(Li et al., 2022).

Alternatively, the vehicle could be operated with combined control strategies. Vision-based DRL could be reserved for certain dimensions, whilst remaining dimensions are handled by a conventional controller such as a PID controller. Mosali et al. (2022) provide an example of this with aerial target tracking. One final note is on the transition from discrete actions

to continuous actions between CubeTrack and the other two environments. Discrete action spaces are easier for the model to optimise, however it was felt that a continuous action space would be necessary for the problem of controlling a real-world vehicle. In retrospect, knowing more about how the BlueROV2 operates (with its sophisticated autopilot software, ArduPilot) and more about how the AUV should behave (not moving too fast or tailgating the animal), perhaps discrete actions would be appropriate in this context, given they will be translated into hardware commands either way.

6.3.5 Multi-object tracking

The SWiMM DEEPeR simulation and communication framework has been designed to support multiple target objects. The number of dolphins and their spawning behaviour can be set in the server configuration, with the game data being sent as an array with the same number of elements. How this data is unpacked and utilised server-side is still to be addressed. The implementation of the reward function in particular will require reference to the literature on multi-object tracking (Kalake et al., 2021). With respect to simulation development, whilst it is possible to spawn multiple objects, the objects will behave entirely independently, each travelling to their own randomly generated waypoint. This is of course not representative of real dolphin pods, which tend to swim in small groups when travelling, or congregate in a given area when feeding. A first look into how this behaviour would be simulated has revealed plenty of open source algorithms which simulate flocking/swarming behaviour of organisms (referred to in game engineering as 'boids'). This has been trialled with a school of fish, but requires further development.

6.3.6 Distractor-aware tracking

Although a pelagic open ocean environment does not present the same challenges as, say, an urban driving environment in terms of distractor objects, future work should still look to develop a distractor-aware policy with more sophisticated collision avoidance. One paper which is particularly inspiring in this regard is 'Anti-distractor Active Object Tracking in 3D Environments' (Xi et al., 2021). In this work, the authors present a novel attention-based VAT solution. Their framework consists of two CNN branches with shared parameters, responsible for encoding the image observation and a target template image respectively. The two are combined with channel-wise operations, and a sequence of the last N encodings is passed through their multi-head attention module. The resulting features with temporal information serve as input to the DRL algorithm A3C.

Another approach would be to build in some form of classification task as an additional auxiliary task during offline feature learning. As well as state prediction, the model could be tasked with semantic segmentation or detection, for example. In 2023, Meta AI Research utilised our computer vision dataset, presented in the paper 'NDD20: A large-scale few-shot dolphin dataset for coarse and fine-grained categorisation' (Trotter et al., 2020), listed in Section 1.4. Their paper 'Segment Anything' (Kirillov et al., 2023) achieves fantastic results on these images, and would therefore be interesting to look at incorporating.

6.3.7 Vehicle integration

Prior to any of this problem scaling, it would make sense to test the sim-to-real transfer capabilities of the single-object, distractor-free, two-dimensional policy, to know for sure whether the high-level approach was worth pursuing. This presents an entire project in itself. The School of Engineering has agreed to provide a 'bread board' of the vehicle parts for experimentation, since it will be necessary to have an in-depth understanding of the control pipeline and camera. Values output by the policy network will not be sent directly to the hardware; they need only emulate the outputs ordinarily received by the game controller. Experiments are needed to understand this value range, such that the the [-1,1] range output by the policy can be suitably mapped. Once this is correct, the remainder of the pipeline ought to stay the same, passing through ArduSub and then ArduPilot on the vehicle's Pixhawk. In terms of feeding the model, the video feed will need to be split into individual frames, and the images resized to the dimensions expected by the VAE. The VAE is around 36 MB and the policy is around 130 KB, and so both will fit comfortably on the Raspberry Pi.

This practical side of the sim-to-real future work has a heavy robotics/engineering component. If constrained by resources or skill set, the sim-to-real transferability of the policy could be tested by feeding through frames of real underwater footage. It would then be possible to manually assess the suitability of the action decision per frame, as was reported in the supplementary materials of Luo et al. (2018).

6.4 Closing Remarks

The high-level research question put forward in Chapter 1 was

Can computer vision and deep learning facilitate a better performing, tagless method of AUV control for collecting data on free-roaming marine megafauna in the wild?

This research question cannot be answered in full for quite some time. From the beginning, the project was viewed as ambitious, and the work presented in this thesis has only demonstrated just how ambitious. At this point in time we have a working VAT algorithm **in-simulation**. It is a single-object, class-agnostic, distractor-free algorithm that provides two-dimensions of control. Answering the research question in full would require scaling the problem to multi-object, distractor-aware tracking, in greater than two dimensions, with the ability to classify objects. It would also require the introduction of hydrodynamic forces into the simulation, a progressively faster and more dynamic target, and the introduction of techniques such as domain randomisation and adaptation in preparation for the ultimate challenge of sim-to-real transfer. The results presented in this thesis have showcased the accuracy and impressive reaction speed of a DRL solution to VAT. Therefore, with the discussed limitations and their potential solutions in mind, the conclusion of this thesis is that visual active tracking with task-relevant features and deep reinforcement learning is a promising avenue to continue exploring.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Agrawal, P., Nair, A. V., Abbeel, P., Malik, J., and Levine, S. (2016). Learning to poke by poking: Experiential learning of intuitive physics. *Advances in neural information processing systems*, 29.

Akhloufi, M. A., Arola, S., and Bonnet, A. (2019). Drones chasing drones: Reinforcement learning and deep search area proposal. *Drones*, 3(3):58.

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631.

Andrews, R. D., Baird, R. W., Calambokidis, J., Goertz, C. E., Gulland, F., Heide-Jorgensen, M.-P., Hooker, S. K., Johnson, M., Mate, B., Mitani, Y., et al. (2019). Best practice guidelines for cetacean tagging. *Journal of Cetacean Research and Management*.

Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20.

ArduPilot (2010). Ardupilot project. https://github.com/ArduPilot/ardupilot.

Åström, K. J. and Hägglund, T. (2001). The future of pid control. *Control engineering practice*, 9(11):1163–1175.

Aytar, Y., Castrejon, L., Vondrick, C., Pirsiavash, H., and Torralba, A. (2017). Cross-modal scene networks. *IEEE transactions on pattern analysis and machine intelligence*, 40(10):2303–2314.

Babenko, B., Yang, M.-H., and Belongie, S. (2009). Visual tracking with online multiple instance learning. In *2009 IEEE Conference on computer vision and Pattern Recognition*, pages 983–990. IEEE.

Bale, K. (2008). osgocean. https://github.com/kbale/osgocean.

Bawa, P. and Ramos, F. (2021). Bagged critic for continuous control.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48.

Bertinetto, L., Valmadre, J., Henriques, J. F., Vedaldi, A., and Torr, P. H. (2016). Fully-convolutional siamese networks for object tracking. In *European conference on computer vision*, pages 850–865. Springer.

BlueRobotics (2022). Bluerov2 datasheet. https://bluerobotics.com/wp-content/uploads/2020/02/br_bluerov2_datasheet_rev6.pdf.

BlueRobotics (2023). Low-light hd usb camera. https://bluerobotics.com/store/sensors-sonars-cameras/cameras/cam-usb-low-light-r1/.

Böhmer, W., Springenberg, J. T., Boedecker, J., Riedmiller, M., and Obermayer, K. (2015). Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations. *KI-Künstliche Intelligenz*, 29(4):353–362.

Bonatti, R., Madaan, R., Vineet, V., Scherer, S., and Kapoor, A. (2020). Learning visuomotor policies for aerial navigation using cross-modal representations. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1637–1644. IEEE.

Borase, R. P., Maghade, D., Sondkar, S., and Pawar, S. (2021). A review of pid control, tuning methods and applications. *International Journal of Dynamics and Control*, 9:818–827.

Bossart, G. D. (2011). Marine mammals as sentinel species for oceans and human health. *Veterinary pathology*, 48(3):676–690.

Bottou, L. and Bousquet, O. (2007). The tradeoffs of large scale learning. *Advances in neural information processing systems*, 20.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

Cai, L., McGuire, N. E., Hanlon, R., Mooney, T. A., and Girdhar, Y. (2023). Semi-supervised visual tracking of marine animals using autonomous underwater vehicles. *International Journal of Computer Vision*, pages 1–22.

Cardona, M., Cortez, F., Palacios, A., and Cerros, K. (2020). Mobile robots application against covid-19 pandemic. In *2020 IEEE ANDESCON*, pages 1–5. IEEE.

Çetin, E., Barrado, C., and Pastor, E. (2020). Counter a drone in a complex neighborhood area by deep reinforcement learning. *Sensors*, 20(8):2320.

Chatfield, K., Simonyan, K., Vedaldi, A., and Zisserman, A. (2014). Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*.

Colas, C., Sigaud, O., and Oudeyer, P.-Y. (2018). How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*.

Comaniciu, D., Ramesh, V., and Meer, P. (2000). Real-time tracking of non-rigid objects using mean shift. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No. PR00662)*, volume 2, pages 142–149. IEEE.

'Convolutional Neural Networks for Visual Recognition' (2017). Convolutional neural networks for visual recognition.

Cook, D., Vardy, A., and Lewis, R. (2014). A survey of auv and robot simulators for multi-vehicle operations. In *2014 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pages 1–8. IEEE.

Crane, K. (2020). Cubetrack: Deep rl for active tracking with unity + ml-agents. *Towards Data Science*.

Crick, C., Jay, G., Osentoski, S., Pitzer, B., and Jenkins, O. C. (2017). Rosbridge: Ros for non-ros users. In *Robotics Research: The 15th International Symposium ISRR*, pages 493–504. Springer.

Crowston, K. (2012). Amazon mechanical turk: A research tool for organizations and information systems scholars. In *Shaping the future of ict research. methods and approaches*, pages 210–221. Springer.

Danelljan, M., Bhat, G., Shahbaz Khan, F., and Felsberg, M. (2017). Eco: Efficient convolution operators for tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6638–6646.

Danelljan, M., Häger, G., Khan, F., and Felsberg, M. (2014). Accurate scale estimation for robust visual tracking. In *British Machine Vision Conference, Nottingham, September 1-5, 2014*. Bmva Press.

Deisenroth, M. P., Neumann, G., Peters, J., et al. (2013). A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142.

Díaz-Rodríguez, I. D., Han, S., and Bhattacharyya, S. P. (2019). *Analytical design of PID controllers*. Springer.

Do, T. T. and Ahn, H. (2018). Visual-gps combined 'follow-me' tracking for selfie drones. *Advanced Robotics*, 32(19):1047–1060.

Dodge, K. L., Kukulya, A. L., Burke, E., and Baumgartner, M. F. (2018). Turtlecam: A "smart" autonomous underwater vehicle for investigating behaviors and habitats of sea turtles. *Frontiers in Marine Science*, 5:90.

Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. (2017). Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR.

Duan, W. (2017). Learning state representations for robotic control: Information disentangling and multi-modal learning.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Dutton, P. H., Komoroske, L., Bejder, L., and Meekan, M. (2019). Integrating emerging technologies into marine megafauna conservation management. *Frontiers in Marine Science*, page 693.

Erin Liong, V., Lu, J., Tan, Y.-P., and Zhou, J. (2017). Cross-modal deep variational hashing. In *Proceedings of the IEEE international conference on computer vision*, pages 4077–4085.

Espié, E. and Guionneau, C. (2016). Torcs - the open racing car simulator.

Fiaz, M., Mahmood, A., Javed, S., and Jung, S. K. (2019a). Handcrafted and deep trackers: Recent visual object tracking approaches and trends. *ACM Computing Surveys (CSUR)*, 52(2):1–44.

Fiaz, M., Mahmood, A., Javed, S., and Jung, S. K. (2019b). Handcrafted and deep trackers: Recent visual object tracking approaches and trends. *ACM Computing Surveys (CSUR)*, 52(2):1–44.

Fowler, M. C., Bolding, T. L., Hebert, K. M., Ducrest, F., and Kumar, A. (2016). Design of a cost-effective autonomous underwater vehicle. In *2016 Annual IEEE Systems Conference (SysCon)*, pages 1–6. IEEE.

Fragapane, G., Hvolby, H.-H., Sgarbossa, F., and Strandhagen, J. O. (2020). Autonomous mobile robots in hospital logistics. In *IFIP International Conference on Advances in Production Management Systems*, pages 672–679. Springer.

Fujimoto, S., Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR.

Fukushima, K. (1988). Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Goroshin, R., Mathieu, M. F., and LeCun, Y. (2015). Learning to linearize under uncertainty. *Advances in neural information processing systems*, 28.

Grando, R. B., de Jesus, J. C., Kich, V. A., Kolling, A. H., Guerra, R. S., and Drews-Jr, P. L. (2022). Mapless navigation of a hybrid aerial underwater vehicle with deep reinforcement learning through environmental generalization. *arXiv preprint arXiv:2209.06332*.

Greaves, J., Robinson, M., Walton, N., Mortensen, M., Pottorff, R., Christopherson, C., Hancock, D., Milne, J., and Wingate, D. (2018). Holodeck: A high fidelity simulator.

Guerra, M., Dawson, S., Brough, T., and Rayment, W. (2014). Effects of boats on the surface and acoustic behaviour of an endangered population of bottlenose dolphins. *Endangered Species Research*, 24(3):221–236.

Guo, X., Liu, X., Zhu, E., and Yin, J. (2017). Deep clustering with convolutional autoencoders. In *International conference on neural information processing*, pages 373–382. Springer.

Ha, D. and Schmidhuber, J. (2018). World models. *arXiv preprint arXiv:1803.10122*.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. (2019). Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*.

Haas, J. K. (2014). A history of the unity game engine.

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. (2019). Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*.

He, A., Luo, C., Tian, X., and Zeng, W. (2018a). Towards a better match in siamese network based visual object tracker. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, pages 0–0.

He, A., Luo, C., Tian, X., and Zeng, W. (2018b). A twofold siamese network for real-time object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4834–4843.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Heiler, J., Elwen, S. H., Kriesell, H., and Gridley, T. (2016). Changes in bottlenose dolphin whistle parameters related to vessel presence, surface behaviour and group composition. *Animal Behaviour*, 117:167–177.

Held, D., Thrun, S., and Savarese, S. (2016). Learning to track at 100 fps with deep regression networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 749–765. Springer.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32.

Henriques, J. F., Caseiro, R., Martins, P., and Batista, J. (2014). High-speed tracking with kernelized correlation filters. *IEEE transactions on pattern analysis and machine intelligence*, 37(3):583–596.

Hermann, K. M., Hill, F., Green, S., Wang, F., Faulkner, R., Soyer, H., Szepesvari, D., Czarnecki, W. M., Jaderberg, M., Teplyashin, D., et al. (2017). Grounded language learning in a simulated 3d world. *arXiv preprint arXiv:1706.06551*.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018a). Stable baselines. https://github.com/hill-a/stable-baselines.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018b). Stable baselines. https://github.com/hill-a/stable-baselines/blob/master/stable_baselines/common/policies.py.

Horning, M., Andrews, R. D., Bishop, A. M., Boveng, P. L., Costa, D. P., Crocker, D. E., Haulena, M., Hindell, M., Hindle, A. G., Holser, R. R., Hooker, S. K., Huckstadt, L. A., Johnson, S., Lea, M.-A., McDonald, B. I., McMahon, C. R., Robinson, P. W., Sattler, R. L., Shuert, C. R., Steingass, S. M., Thompson, D., Tuomi, P. A., Williams, C. L., and Womble, J. N. (2019). Best practice recommendations for the use of external telemetry devices on pinnipeds. *Animal Biotelemetry*, 7(1):1–17.

Horning, M., Haulena, M., Tuomi, P. A., Mellish, J.-A. E., Goertz, C. E., Woodie, K., Berngartt, R. K., Johnson, S., Shuert, C. R., Walker, K. A., Skinner, J. P., and Boveng, P. L. (2017). Best practice recommendations for the use of fully implanted telemetry devices in pinnipeds. *Animal Biotelemetry*, 5(1):13.

Inzartsev, A. (2009). *Underwater vehicles*. BoD–Books on Demand.

Iversen, N., Schofield, O. B., Cousin, L., Ayoub, N., Vom Bögel, G., and Ebeid, E. (2021). Design, integration and implementation of an intelligent and self-recharging drone system for autonomous power line inspection. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4168–4175. IEEE.

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

Kalake, L., Wan, W., and Hou, L. (2021). Analysis based on recent deep learning approaches applied in real-time multi-object tracking: a review. *IEEE Access*, 9:32650–32671.

Katara, P., Khanna, M., Nagar, H., and Panaiyappan, A. (2019). Open source simulator for unmanned underwater vehicles using ros and unity3d. In *2019 IEEE Underwater Technology (UT)*, pages 1–7.

Katara, P., Khanna, M., Nagar, H., and Panaiyappan, A. (2019). Open source simulator for unmanned underwater vehicles using ros and unity3d. In *2019 IEEE Underwater Technology (UT)*, pages 1–7. IEEE.

Katija, K., Roberts, P. L., Daniels, J., Lapides, A., Barnard, K., Risi, M., Ranaan, B. Y., Woodward, B. G., and Takahashi, J. (2021). Visual tracking of deepwater animals using machine learning-controlled robotic underwater vehicles. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 860–869.

Kelly, S. (2016). Basic introduction to pygame. In *Python, PyGame and Raspberry Pi Game Development*, pages 59–65. Springer.

Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. (2016). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *2016 IEEE conference on computational intelligence and games (CIG)*, pages 1–8. IEEE.

Kermorgant, O. (2014). A dynamic simulator for underwater vehicle-manipulators. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 25–36. Springer.

Khalid, S., Khalil, T., and Nasreen, S. (2014). A survey of feature selection and feature extraction techniques in machine learning. In *2014 science and information conference*, pages 372–378. IEEE.

Kingma, D. P. and Welling, M. (2019). An introduction to variational autoencoders. *arXiv preprint arXiv:1906.02691*.

Kingma Diederik, P. and Adam, J. B. (2014). A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., et al. (2023). Segment anything. *arXiv preprint arXiv:2304.02643*.

Koenig, N. and Howard, A. (2004). 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)*, volume 3, pages 2149–2154. IEEE.

Koryakovskiy, I., Vallery, H., Babuška, R., and Caarls, W. (2017). Evaluation of physical damage associated with action selection strategies in reinforcement learning. *IFAC-PapersOnLine*, 50(1):6928–6933.

Koubâa, A. et al. (2017). *Robot Operating System (ROS).*, volume 1. Springer.

Kramer, T. (2017). Sd sandbox. https://github.com/tawnkramer/sdsandbox/.

Kramer, T. (2018). Openai gym environments for donkey car. https://github.com/tawnkramer/gym-donkeycar/.

Kristan, M., Matas, J., Leonardis, A., Felsberg, M., Pflugfelder, R., Kämäräinen, J.-K., Chang, H. J., Danelljan, M., Cehovin, L., Lukežič, A., et al. (2021). The ninth visual object tracking vot2021 challenge results. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2711–2738.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

Kukulya, A. L., Stokey, R., Fiester, C., Padilla, E. M. H., and Skomal, G. (2016). Multi-vehicle autonomous tracking and filming of white sharks carcharodon carcharias. In *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*, pages 423–430. IEEE.

Kukulya, A. L., Stokey, R., Littlefield, R., Jaffre, F., Padilla, E. M. H., and Skomal, G. (2015). 3d real-time tracking, following and imaging of white sharks with an autonomous underwater vehicle. In *OCEANS 2015-Genova*, pages 1–6. IEEE.

Kumar, G. S., Painumgal, U. V., Kumar, M. C., and Rajesh, K. (2018). Autonomous underwater vehicle for vision based tracking. *Procedia computer science*, 133:169–180.

'Learning to Drive Smoothly in Minutes' (2019). Learning to drive smoothly in minutes.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Leike, J., Martic, M., Krakovna, V., Ortega, P. A., Everitt, T., Lefrancq, A., Orseau, L., and Legg, S. (2017). Ai safety gridworlds. *arXiv preprint arXiv:1711.09883*.

Lesort, T., Díaz-Rodríguez, N., Goudou, J.-F., and Filliat, D. (2018). State representation learning for control: An overview. *Neural Networks*, 108:379–392.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016a). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016b). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

Li, B., Yan, J., Wu, W., Zhu, Z., and Hu, X. (2018a). High performance visual tracking with siamese region proposal network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8971–8980.

Li, H., Xu, Z., Taylor, G., Studer, C., and Goldstein, T. (2018b). Visualizing the loss landscape of neural nets. *Advances in neural information processing systems*, 31.

Li, P., Wang, D., Wang, L., and Lu, H. (2018c). Deep visual tracking: Review and experimental comparison. *Pattern Recognition*, 76:323–338.

Li, X., Hu, W., Shen, C., Zhang, Z., Dick, A., and Hengel, A. V. D. (2013). A survey of appearance models in visual object tracking. *ACM transactions on Intelligent Systems and Technology (TIST)*, 4(4):1–48.

Li, X., Wang, X., Zheng, X., Jin, J., Huang, Y., Zhang, J. J., and Wang, F.-Y. (2022). Sadrl: Merging human experience with machine intelligence via supervised assisted deep reinforcement learning. *Neurocomputing*, 467:300–309.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2016). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988.

Linnainmaa, S. (1970). *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. PhD thesis, Master's Thesis (in Finnish), Univ. Helsinki.

Liu, Y., Anderlini, E., Wang, S., Ma, S., and Ding, Z. (2022). Ocean explorations using autonomy: Technologies, strategies and applications. In *Offshore Robotics*, pages 35–58. Springer.

Loquercio, A., Maqueda, A. I., Del-Blanco, C. R., and Scaramuzza, D. (2018). Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095.

Luo, W., Sun, P., Zhong, F., Liu, W., Zhang, T., and Wang, Y. (2018). End-to-end active object tracking via reinforcement learning. In *International conference on machine learning*, pages 3286–3295. PMLR.

Luo, W., Sun, P., Zhong, F., Liu, W., Zhang, T., and Wang, Y. (2019). End-to-end active object tracking and its real-world deployment via reinforcement learning. *IEEE transactions on pattern analysis and machine intelligence*, 42(6):1317–1332.

Manhães, M. M. M., Scherer, S. A., Voss, M., Douat, L. R., and Rauschenbach, T. (2016). UUV simulator: A gazebo-based package for underwater intervention and multi-robot simulation. In *OCEANS 2016 MTS/IEEE Monterey*. IEEE.

Martz, P. (2011). osgbullet. https://github.com/mccdo/osgbullet.

Matsebe, O., Kumile, C., and Tlale, N. (2008). A review of virtual simulators for autonomous underwater vehicles (auvs). *IFAC Proceedings Volumes*, 41(1):31–37.

Mattner, J., Lange, S., and Riedmiller, M. (2012). Learn to swing up and balance a real pole based on raw visual input data. In *International Conference on Neural Information Processing*, pages 126–133. Springer.

Mayer, C., Danelljan, M., Paudel, D. P., and Van Gool, L. (2021). Learning target candidate association to keep track of what not to track. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 13444–13454.

McCandlish, S., Kaplan, J., Amodei, D., and Team, O. D. (2018). An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*.

McGuire, N., Cai, L., Jamieson, S., McCammon, S., Claus, B., Soucie, J. E. S., Todd, J. E., Mooney, T. A., et al. (2023). Curee: A curious underwater robot for ecosystem exploration. *arXiv preprint arXiv:2303.03943*.

Mehta, M. N., Mylraj, S., and Bhate, V. N. (2021). Development of auv for sauvc during covid-19. In *OCEANS 2021: San Diego–Porto*, pages 1–8. IEEE.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533.

Mobbs, D., Hagan, C. C., Dalgleish, T., Silston, B., and Prévost, C. (2015). The ecology of human fear: survival optimization and the nervous system. *Frontiers in neuroscience*, 9:55.

Moore, S. E. (2008). Marine mammals as ecosystem sentinels. *Journal of Mammalogy*, 89(3):534–540.

Mosali, N. A., Shamsudin, S. S., Alfandi, O., Omar, R., and Al-Fadhali, N. (2022). Twin delayed deep deterministic policy gradient-based target tracking for unmanned aerial vehicle with achievement rewarding and multistage training. *IEEE Access*, 10:23545–23559.

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Icml*.

Nelms, S. E., Alfaro-Shigueto, J., Arnould, J. P., Avila, I. C., Nash, S. B., Campbell, E., Carter, M. I., Collins, T., Currey, R. J., Domit, C., et al. (2021). Marine mammal conservation: over the horizon. *Endangered Species Research*, 44:291–325.

Ngiam, J., Khosla, A., Kim, M., Nam, J., Lee, H., and Ng, A. Y. (2011). Multimodal deep learning. In *ICML*.

NVIDIAGameWorks (2015). Physx. https://github.com/NVIDIAGameWorks/PhysX.

OpenCV (2010). Simple blob detector. GitHub.

OpenCV (2011). Orb (oriented fast and rotated brief). GitHub.

OpenSceneGraph (1999). Openscenegraph. https://www.openscenegraph.com/.

Osa, E. and Orukpe, P. (2021). Simulation of an underwater environment via unity 3d software. In *BOOK OF PROCEEDINGS*, page 384.

Packard, G. E., Kukulya, A., Austin, T., Dennett, M., Littlefield, R., Packard, G., Purcell, M., Stokey, R., and Skomal, G. (2013). Continuous autonomous tracking and imaging of white sharks and basking sharks using a remus-100 auv. In *2013 OCEANS-San Diego*, pages 1–5. IEEE.

Parberry, I. (2017). *Introduction to Game Physics with Box2D*. CRC Press.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR.

Peng, X. B., Andrychowicz, M., Zaremba, W., and Abbeel, P. (2018). Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE.

'Policy Gradient Algorithms' (2018). Policy gradient algorithms. Lil'Log.

Portugal, D., Santos, L., Alvito, P., Dias, J., Samaras, G., and Christodoulou, E. (2015). Socialrobot: An interactive mobile robot for elderly home care. In *2015 IEEE/SICE International Symposium on System Integration (SII)*, pages 811–816. IEEE.

Potokar, E., Ashford, S., Kaess, M., and Mangelson, J. (2022a). HoloOcean: An underwater robotics simulator. In *Proc. IEEE Intl. Conf. on Robotics and Automation, ICRA*, Philadelphia, PA, USA.

Potokar, E., Ashford, S., Kaess, M., and Mangelson, J. G. (2022b). Holoocean: An underwater robotics simulator. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 3040–3046. IEEE.

Pérez Soler, J., Sales, J., Prats, M., Martí, J., Fornas, D., Marín Prades, R., and Sanz, P. (2013). The underwater simulator uwsim benchmarking capabilities on autonomous grasping. *ICINCO 2013 - Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics*, 2:369–376.

Raffin, A., Hill, A., Traoré, R., Lesort, T., Díaz-Rodríguez, N., and Filliat, D. (2018). S-rl toolbox: Environments, datasets and evaluation metrics for state representation learning. *arXiv preprint arXiv:1809.09369*.

Raffin, A., Hill, A., Traoré, R., Lesort, T., Díaz-Rodríguez, N., and Filliat, D. (2019). Decoupling feature extraction from policy learning: assessing benefits of state representation learning in goal based robotics. *arXiv preprint arXiv:1901.08651*.

Raffin, A. and Sokolkov, R. (2019). Learning to drive smoothly in minutes. https://github.com/araffin/learning-to-drive-in-5-minutes/.

Ray, A., Achiam, J., and Amodei, D. (2019). Benchmarking safe exploration in deep reinforcement learning. *arXiv preprint arXiv:1910.01708*, 7:1.

Redmon, J. and Farhadi, A. (2017). Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271.

Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.

'RL Course by David Silver' (2015). Rl course by david silver.

RL Wiki (2020). Reinforcement learning discord wiki. GitHub.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.

Robotics, B. (2023). Ardusub. https://www.ardusub.com/developers/pymavlink.html.

Rodriguez, S. and Raffin, A. (2018). https://github.com/sergionr2/RacingRobot.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252.

Samoili, S., Cobo, M. L., Gómez, E., De Prato, G., Martínez-Plumed, F., and Delipetrev, B. (2020). Ai watch. defining artificial intelligence. towards an operational definition and taxonomy of artificial intelligence.

Sanders, A. (2016). *An introduction to Unreal engine 4*. AK Peters/CRC Press.

Schmidhuber, J. (2015). On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *arXiv preprint arXiv:1511.09249*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer.

Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review*, 5(1):3–55.

Sharma, S., Sharma, S., and Athaiya, A. (2017). Activation functions in neural networks. *towards data science*, 6(12):310–316.

Shelhamer, E., Mahmoudieh, P., Argus, M., and Darrell, T. (2016). Loss is its own reward: Self-supervision for reinforcement learning. *arXiv preprint arXiv:1612.07307*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Skinner, B. F. (1988). Preface to the behavior of organisms. *Journal of the experimental analysis of behavior*, 50(2):355.

Soleimanitaleb, Z., Keyvanrad, M. A., and Jafari, A. (2019). Object tracking methods: A review. In *2019 9th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 282–288. IEEE.

'Spinning Up in Deep RL' (2018). Spinning up in deep rl. OpenAI.

Spurr, A., Song, J., Park, S., and Hilliges, O. (2018). Cross-modal deep variational hand pose estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 89–98.

'Stella - A multi-platform Atari 2600 VCS emulator' (1996). Stella - a multi-platform atari 2600 vcs emulator. Github.

Suhr, J. K. (2009). Kanade-lucas-tomasi (klt) feature tracker. *Computer Vision (EEE6503)*, pages 9–18.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, R. S., McAllester, D. A., Singh, S. P., Mansour, Y., et al. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer.

Swett, B. A., Hahn, E. N., and Llorens, A. J. (2021). Designing robots for the battlefield: state of the art. *Robotics, AI, and Humanity*, pages 131–146.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., et al. (2018). Deepmind control suite. *arXiv preprint arXiv:1801.00690*.

Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.

Toromanoff, M., Wirbel, E., and Moutarde, F. (2020). End-to-end model-free reinforcement learning for urban driving using implicit affordances. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7153–7162.

Trotter, C., Atkinson, G., Sharpe, M., Richardson, K., McGough, A. S., Wright, N., Burville, B., and Berggren, P. (2020). Ndd20: A large-scale few-shot dolphin dataset for coarse and fine-grained categorisation. *arXiv preprint arXiv:2005.13359*.

Truong, J., Chernova, S., and Batra, D. (2021). Bi-directional domain adaptation for sim2real transfer of embodied navigation agents. *IEEE Robotics and Automation Letters*, 6(2):2634–2641.

Uhlenbeck, G. E. and Ornstein, L. S. (1930). On the theory of the brownian motion. *Physical review*, 36(5):823.

Unity-Documentation (2021). Unity manual. https://docs.unity3d.com/Manual/.

Unity-Technologies (2017a). Ml-agents toolkit overview. https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/.

Unity-Technologies (2017b). Unity ml-agents toolkit. https://github.com/Unity-Technologies/ml-agents.

Vajapeyam, S. (2014). Understanding shannon's entropy metric for information. *arXiv preprint arXiv:1405.2061*.

Van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Van Hoof, H., Chen, N., Karl, M., van der Smagt, P., and Peters, J. (2016). Stable reinforcement learning with autoencoders for tactile and visual data. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 3928–3934. IEEE.

VanBressem, M.-F., Burville, B., Sharpe, M., Berggren, P., and VanWaerebeek, K. (2018). Visual health assessment of white-beaked dolphins off the coast of northumberland, north sea, using underwater photography. *Marine Mammal Science*.

Verma, R. (2017). A review of object detection and tracking methods. *International Journal of Advance Engineering and Research Development*, 4(10):569–578.

von Benzon, M., Sørensen, F. F., Uth, E., Jouffroy, J., Liniger, J., and Pedersen, S. (2022). An open-source benchmark simulator: Control of a bluerov2 underwater robot. *Journal of Marine Science and Engineering*, 10.

Waldrop, M. M. et al. (2015). No drivers required. *Nature*, 518(7537):20.

Wan, E. A. and Van Der Merwe, R. (2001). The unscented kalman filter. *Kalman filtering and neural networks*, pages 221–280.

Wang, J. X., King, M., Porcel, N., Kurth-Nelson, Z., Zhu, T., Deck, C., Choy, P., Cassin, M., Reynolds, M., Song, F., et al. (2021). Alchemy: A benchmark and analysis toolkit for meta-reinforcement learning agents. *arXiv preprint arXiv:2102.02926*.

Wang, Q., Ma, Y., Zhao, K., and Tian, Y. (2022). A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212.

Wang, Q., Zhang, L., Bertinetto, L., Hu, W., and Torr, P. H. (2019). Fast online object tracking and segmentation: A unifying approach. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1328–1338.

Wang, X., Zhao, H., and Zhu, J. (1993). Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86.

Watter, M., Springenberg, J., Boedecker, J., and Riedmiller, M. (2015). Embed to control: A locally linear latent dynamics model for control from raw images. *Advances in neural information processing systems*, 28.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Würsig, B. and Würsig, M. (1977). The photographic determination of group size, composition, and stability of coastal porpoises (tursiops truncatus). *Science*, 198(4318):755–756.

Wynn, R. B., Huvenne, V. A., Le Bas, T. P., Murton, B. J., Connelly, D. P., Bett, B. J., Ruhl, H. A., Morris, K. J., Peakall, J., Parsons, D. R., et al. (2014). Autonomous underwater vehicles (auvs): Their past, present and future contributions to the advancement of marine geoscience. *Marine geology*, 352:451–468.

Xi, M., Zhou, Y., Chen, Z., Zhou, W., and Li, H. (2021). Anti-distractor active object tracking in 3d environments. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(6):3697–3707.

Xiao, C., Lu, P., and He, Q. (2021). Flying through a narrow gap using end-to-end deep reinforcement learning augmented with curriculum learning and sim2real. *IEEE transactions on neural networks and learning systems*.

Yao, N., Anaya, E., Tao, Q., Cho, S., Zheng, H., and Zhang, F. (2017). Monocular vision-based human following on miniature robotic blimp. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3244–3249. IEEE.

Yoerger, D. R., Govindarajan, A. F., Howland, J. C., Llopiz, J. K., Wiebe, P. H., Curran, M., Fujii, J., Gomez-Ibanez, D., Katija, K., Robison, B. H., et al. (2021). A hybrid underwater robot for multidisciplinary investigation of the ocean twilight zone. *Science Robotics*, 6(55):eabe1901.

Yu, J., Wang, Z., Vasudevan, V., Yeung, L., Seyedhosseini, M., and Wu, Y. (2022). Coca: Contrastive captioners are image-text foundation models. *arXiv preprint arXiv:2205.01917*.

Yu, J., Wu, Z., Yang, X., Yang, Y., and Zhang, P. (2020). Underwater target tracking control of an untethered robotic fish with a camera stabilizer. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(10):6523–6534.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I 13*, pages 818–833. Springer.

Zhai, J., Zhang, S., Chen, J., and He, Q. (2018). Autoencoder and its various variants. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 415–419. IEEE.

Zhang, A., Satija, H., and Pineau, J. (2018). Decoupling dynamics and reward for transfer learning. *arXiv preprint arXiv:1804.10689*.

Zhang, F., Leitner, J., Ge, Z., Milford, M., and Corke, P. (2019). Adversarial discriminative sim-to-real transfer of visuo-motor policies. *The International Journal of Robotics Research*, 38(10-11):1229–1245.

Zhang, Q., Lin, J., Sha, Q., He, B., and Li, G. (2020). Deep interactive reinforcement learning for path following of autonomous underwater vehicle. *IEEE Access*, 8:24258–24268.

Zhong, F., Sun, P., Luo, W., Yan, T., and Wang, Y. (2018). Ad-vat: An asymmetric dueling mechanism for learning visual active tracking. In *International Conference on Learning Representations*.

Zhong, F., Sun, P., Luo, W., Yan, T., and Wang, Y. (2019a). Ad-vat+: An asymmetric dueling mechanism for learning and understanding visual active tracking. *IEEE transactions on pattern analysis and machine intelligence*, 43(5):1467–1482.

Zhong, F., Sun, P., Luo, W., Yan, T., and Wang, Y. (2019b). Ad-vat: An asymmetric dueling mechanism for learning visual active tracking. In *International Conference on Learning Representations*.

Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232.

Zhu, Z., Ma, H., and Zou, W. (2019). Human following for wheeled robot with monocular pan-tilt camera. *arXiv preprint arXiv:1909.06087*.

Zou, Z., Shi, Z., Guo, Y., and Ye, J. (2019). Object detection in 20 years: A survey. *arXiv preprint arXiv:1905.05055*.

Appendix A

SWiMM DEEPeR server configuration

```
{
    "msgType": "process_server_config",
    "payload": {
        "serverConfig": {
            "roverConfig": {
                "motorConfig": {
                    "stabilityForce": 0.2,
                    "stabilityThreshold": 1,
                    "linearThrustPower": [
                        70,
                        180,
                        70
                    ],
                    "angularThrustPower": [
                        0.5,
                        0.5,
                        0.5
                    ]
                },
                "camConfig": {
                    "resolution": [
                        64,
                        64
                    ],
                    "sensorWidth": 4.96,
                    "sensorHeight": 3.74,
                    "focalLength": 2.97
                },
                "structureConfig": {
                    "ballastMass": 1.2
```

```
                }
            },
        "envConfig": {
            "actionInference": "maintain",
            "faunaConfig": {
                "spawnTimer": 5,
                "spawnContainerRatio": 0.8,
                "spawnRadius": 10,
                "boidGroups": [
                    {
                        "prefabName": "boid_fish",
                        "prefabNum": 0,
                        "numGroups": 0,
                        "fragmentedGroup": 50,
                        "fragmentedIndividual": 30,
                        "soaring": 1
                    }
                ],
                "aiGroups": [
                    {
                        "prefabName": "dolphin",
                        "maxAmount": 1,
                        "maxSpawn": 1,
                        "enableSpawner": true,
                        "randomMovement": true,
                        "randomizeStats": false,
                        "waypointAxes": [
                            true,
                            false,
                            true
                        ],
                        "rotationOffset": [
                            0,
                            0,
                            0
                        ],
                        "scale": 1,
                        "minSpeed": 0.04,
                        "maxSpeed": 0.04,
                        "spawnInfront": true,
                        "scaleVariance": 0
                    },
                    {
                        "prefabName": "whale",
```

```
                        "maxAmount": 1,
                        "maxSpawn": 1,
                        "enableSpawner": false,
                        "randomMovement": true,
                        "randomizeStats": false,
                        "waypointAxes": [
                            true,
                            false,
                            true
                        ],
                        "spawnInfront": true,
                        "rotationOffset": [
                            0,
                            180,
                            0
                        ],
                        "scale": 5,
                        "minSpeed": 1,
                        "maxSpeed": 7
                    }
                ]
            }
        }
    }
}
```

Appendix B

Autoencoder network architectures

B.1 Architectures used in the cross-modal framework

Layer (Block)	Type	Filters	Kernel size (K)	Stride (S)	Activation	Output size
0	input	-	-	-	-	$64 \times 64 \times 3$
1	conv	32	5	2	linear	$32 \times 32 \times 32$
2	max pool	1	2	2	-	$16 \times 16 \times 32$
3 (1)	conv	32	3	2	linear	$8 \times 8 \times 32$
4 (1)	conv	32	3	1	linear	$8 \times 8 \times 32$
5 (1)	conv (skip)	32	1	2	linear	$8 \times 8 \times 32$
6 (2)	conv	64	3	2	linear	$4 \times 4 \times 64$
7 (2)	conv	64	3	1	linear	$4 \times 4 \times 64$
8 (2)	conv (skip)	64	1	2	linear	$4 \times 4 \times 64$
9 (3)	conv	128	3	2	linear	$2 \times 2 \times 128$
10 (3)	conv	128	3	1	linear	$2 \times 2 \times 128$
11 (3)	conv (skip)	128	1	2	linear	$2 \times 2 \times 128$
12	flatten	-	-	-	-	1×512
13	FC	-	-	-	relu	1×64
14	FC	-	-	-	relu	1×32
15	FC	-	-	-	linear	1×20

Table B.1 Architecture for cross-modal 'Dronet' encoder network (i.e. ResNet-8)

Note. Padding is set to 'same' i.e. P zero padding evenly around the input such that, if $S > 1$, $(Input - K + 2 * P)$ *is divisible by S. Convolutional layers within residual blocks are preceded by batch normalisation and ReLU non-linearity. The final FC layers are also preceded by ReLU non-linearity. The input to each layer is the output of the previous layer, except in the case of conv (skip) layers. These skip connections take the same input as the input to the residual block and then sum the output with the output of the previous layer, using the* 1×1 *convolution to match up dimensionality.*

Layer	Type	Filters	Kernel size	Stride	Activation	Output (Dilation)
0	input	-	-	-	-	1×10
1	FC	-	-	-	None	1×1024
2	reshape	-	-	-	-	$1 \times 1 \times 1024$
3	deconv	128	4	1	relu	$4 \times 4 \times 128$ (3)
4	deconv	64	5	1	relu	$14 \times 14 \times 64$ (2)
5	deconv	64	6	1	relu	$23 \times 23 \times 64$ (1)
6	deconv	32	5	2	relu	$53 \times 53 \times 32$ (1)
7	deconv	16	5	1	relu	$59 \times 59 \times 16$
8	deconv	3	6	1	tanh	$64 \times 64 \times 3$

Table B.2 Architecture for cross-modal decoder network

Note. Padding is set to 'valid' i.e. off. The input to each layer is the output of the previous layer. If a layer applies dilation D, the input (i.e. output of previous layer) is zero padded around each cell, making $I = I + 2D$

Layer	Type	Units	Activation	Output
0	input	-	-	1×1
1	FC	64	relu	1×64
2	FC	32	relu	1×32
3	FC	1	linear	1×1

Table B.3 Architecture for state prediction MLPs

B.2 Architectures used in the World Models framework

Layer	Type	Filters	Kernel size	Stride	Activation	Output size
0	input	-	-	-	-	$64 \times 64 \times 3$
1	conv	32	4	2	relu	$31 \times 31 \times 32$
2	conv	64	4	2	relu	$14 \times 14 \times 64$
3	conv	128	4	2	relu	$6 \times 6 \times 128$
4	conv	256	4	2	relu	$2 \times 2 \times 256$
5	reshape	-	-	-	-	1×1024
6	FC	-	-	-	None	1×10
7	FC	-	-	-	None	1×10

Table B.4 Architecture for World Models encoder network

Note. Padding is set to 'valid' i.e. off. The input to each layer is the output of the previous layer, except for layers 6 and 7, which both take the output from layer 5.

Layer	Type	Filters	Kernel size	Stride	Activation	Output size
0	input	-	-	-	-	1×10
1	FC	-	-	-	None	1×1024
2	reshape	-	-	-	-	$1 \times 1 \times 1024$
3	deconv	128	5	2	relu	$5 \times 5 \times 128$
4	deconv	64	5	2	relu	$13 \times 13 \times 64$
5	deconv	32	6	2	relu	$30 \times 30 \times 32$
6	deconv	3	6	2	sigmoid	$64 \times 64 \times 3$

Table B.5 Architecture for World Models decoder network

Note. Padding is set to 'valid' i.e. off. The input to each layer is the output of the previous layer.