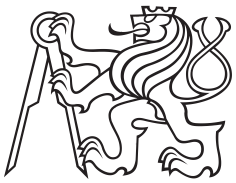
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Deep Learning for Autonomous Control of Robot's Flippers in Simulation

Teymur Azayev

Supervisor: doc. Ing. Karel Zimmermann, Ph.D
Field of study: Artificial Intelligence
January 2017

Acknowledgements

I thank CTU for being such a good alma mater and my supervisor for his advice and proofreading. I thank UC Berkeley, CMU and Standford universities and especially Sergey Levine and David Silver for making high quality video lectures available for free on the internet.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of the university theses.

Prague, date .....

.....
signature

Abstract

Neural networks have seen increasing use in various robotic tasks such as locomotion largely due to advanced in Deep Learning techniques and Reinforcement Learning algorithms. We examine several Deep Learning approaches to learning a semi-autonomous locomotion policy for a ground based search and rescue robot using only front facing RGBD camera and proprioceptive data. A supervised learning approach is suggested and implemented for the case where we only have a real robot and no simulated environment. We also suggest a method to deal with potential issues of multimodal action distributions using an alternative loss proxy based on Generative Adversarial Networks. Reactive as well as recurrent policies implemented using RNNs are compared. A simulator is used to train policies for the robot using Deep Reinforcement Learning. All policies are trained end-to-end, using convolutional neural networks for high dimensional image inputs. We examine the performance of policies trained with variously shaped rewards such as low control effort and smooth locomotion. Experiments are performed on the real robot using a learned RNN policy in the simulator and observe that the policy is transferable with no finetuning to the real environment, albeit, with some performance degradation. We also suggest two potential methods of domain transfer based on image modification using Gram matrix matching and Generative Adversarial Networks.

Keywords: Deep learning, imitation, reinforcement learning, neural networks

Supervisor: doc. Ing. Karel Zimmermann, Ph.D

Abstrakt

Diky současnému pokroku v algoritmech hlubokého a posilovaného učení, jsou neuronové sítě stále častejí používane v různých robotických úlohách jako je napriklad rizeni robotu pri prejizdeni nerovneho terenu. Zkoumáme různe přístupy hlubokého učeni semi-autonomního algoritmu pohybu pro terénniho robota určeneho pro učely 'Search&Rescue' misí s použití jenom čelní kamery a interoceptivnich dat. Navrhujeme nový algoritmus učení s učitelem a implementujeme ho pro připad kde máme pouze reálného robota bez simulovaného prostředí. Dále navrhujeme metodu řešící problém multimodality akcí pomocí Generative Adversarial Networks (GAN). Porovnávame reaktivní a rekurentní chováni implementované pomocí RNN sítí. Simulátor je použit pro trenování pohybu robotu pomocí hlubokého posilovaného učení. Všechny algoritmy chováni jsou trenované jako celek, s použitím konvolučních neuronových sítí pro vysokodimenzionální vstupy. Zkoumáme a experimentálně vyhodnocujeme různé metody pro reward shaping jako je napřiklad low control effort a smooth locomotion. Experimenty na realném robotu s použitím naučené rekurentní sítě ze simulátoru ukazují, že algoritmus je použitelný i bez nutnosti přeučení na reálném systému. Také navrhujeme dva algoritmy pro domain transfer založene na modifikací obrázku s použitím shody s Gram maticí a GAN sítí.

Klíčová slova:

Překlad názvu: Hluboké učení pro autonomní řízení fliperů robotu v simulaci

Contents

Figures

Tables

Part I

Introduction

Chapter 1

1.1 Motivation

Mobile robotics is a multidisciplinary field that is primarily concerned with robots that are able to move and navigate in a certain environment to perform one or more tasks. These robots are usually called agents and are either completely controlled by a user or have a certain degree of autonomy to them. Such robots have been finding use in various fields such as industry, astronomy and transportation. Mobile robots are capable of performing tasks that are monotonous, expensive, dangerous or outright impossible to do by humans. Some examples of these include cave or ocean exploration, search and rescue missions and small or large scale geographical data collection, etc. There are arguably two main challenges to building a mobile robotic system consisting of one or more agents. One of them is the design and implementation of the actual physical robot itself, called the embodiment. The second, is the control, or decision mechanism of the robot which enables the robot to perform its task in a specific environment. This behavioral part is usually referred to as the 'Intelligence' of the robot and has been actively research by the Artificial Intelligence (AI) community for decades. Designing more effective semi or fully autonomous robots can vastly benefit humanity in various aspects both in the present day with problems such as search and rescue or in the future in tasks such as space exploration.

1.2 Aims

We attempt to provide a Deep Learning [23] approach for learning a semi-autonomous locomotion policy for a mobile robot whose task it is to configure itself in such a way so that it is able to traverse a set of ground obstacles such as construction debris. By policy we mean a function which tells the agent what action to perform at each state. This policy is intended to decrease the cognitive load on a human operating the robot allowing them to concentrate on other mission tasks. The operator provides simple commands to the robot, such as steering and velocity, and the robot must autonomously control its configuration to achieve the desired locomotion. The contribution of this

thesis is an approach using Deep Learning to attempt to solve the above task on a robot using only a minimal amount of sensory input, namely a front facing RGB or RGBD camera such as the Realsense R200 [33]. Instead of using classical methods such as planning in a map obtained with SLAM [8], we turn to using Deep Learning with neural networks to obtain a reactive and recurrent policy in a simulator and attempt to transfer it to the real world. Successfully learning a neural network policy from a high dimensional visual input means that it can work in real time on very simple embedded hardware and with cheap sensors, reducing development costs and potentially further spreading the use of mobile robots.

1.3 Thesis structure

The next chapter gives a brief overview of the field of mobile robotics and related work. We take a look at the recent innovations in neural networks and deep learning and how the community has been approaching the problem and where it stands amongst classical methods. This is followed by a description of the NIFTi project, which encompasses the main robot that we will be demonstrating our algorithms on. Part I is concluded with an overview of Tensorflow [5] and why the concept of computational graphs is crucial in deep learning research.

In part II we explore how human input can be used to learn a locomotion policy in a simulator or on the real robot using a limited amount of demonstrations. We also discuss and attempt to overcome the partial observability problem when navigating from a single front facing camera using recurrent neural networks.

Part III focuses on self-supervised learning, namely Reinforcement learning [58]. Here the problem is tackled by providing the agent with a simulated environment and learning a policy essentially by trial and error. We take a look at state of the art algorithms that have been developed in the past few years that can handle high dimensional state-spaces and action spaces.

In Part IV we attempt to transfer the learned policy from the simulated environment to the real robot. We discuss the physics and input data distribution mismatches and proposed methods how to deal with them. It is then followed by a description of the experiments on the real robot as well as results.

In Part V we analyze the overall results and discuss the applicability of the proposed methods in this thesis, followed by a future work and conclusion.

Chapter 2

Related work

Robot control in mobile systems can usually be divided into a few different paradigms, namely reactive, hierarchical and hybrid. Reactive systems are typically very simple and consist of a function called a 'policy' which maps current sensory inputs to a given action. Their disadvantage is the inability to solve complex tasks which require long-term planning. Some examples of reactive methods in robots include bug algorithms and swarm behavior. Hierarchical systems consist of multiple modules which work asynchronously at different time scales. It may consist for example of a sensing system which usually builds a map representation of the environment, a planning module which communicates with the sensing module and decides what to do next, and a low-level controller which takes inputs from the planner and sends them to typically fast-paced hardware controllers. State of the art locomotion tasks usually use SLAM-based [8] map synthesis to localize the robot and continuous-space planning techniques such as RRT* [40] to plan a trajectory in configuration space for the robot from a start point to a goal point. It is to be noted that such planning techniques suffer from the typical curse of dimensionality and are slow or completely unusable for high-dimensional configuration spaces. SLAM-based techniques are error prone, especially in outdoor environments and are computationally expensive, especially in the case of Visual SLAM (vSLAM) [59]. Another disadvantage of the above methods is that they usually require data fusion from multiple sensors, some of which are bulky, expensive and error prone. An implementation/deployment of such a system consists of multiple software modules, and usually the requirement of an x86 hardware based system, which can be difficult or even infeasible to use on certain platforms due to size, power or price constraints.

Lately we have seen the emergence of reinforcement learning being applied to all sorts of robotic tasks ranging from simple gait learning to complex robotic manipulation. A typical problem of Reinforcement learning methods is the high sample complexity requirement which translates to long training times. Using deep neural networks as policy functions it has been shown that it is possible to learn tasks that have high dimensional action spaces (> 30) and very high dimensional input spaces (control from pixels). In most of these examples the policy that is learned is reactive, which is at first glance a severe limitation, but it has been shown that even reactive policies can

exhibit persistent complex behaviors [27] in dynamic and adversarial [10] environments. A reactive policy is an issue when the environment is partially observable, which is often the case in robotic locomotion. By the notion of partial observability we mean that we do not have enough information to make an optimal decision. In the case of locomotion usually the partial observable factor is the environment. This issue is mitigated by using some sort of state representation which includes a portion of the history. Another way of solving this issue is to use recurrent neural networks (RNN) which can remember the relevant state information from the history and enable the use of a reactive policy on that state representation. This will be discussed in detail in part II. Advances in neural network architectures enable complex designs such as Neural Turing Machines [25] and memory networks [66] which keep explicit memory storages and access them in a differentiable manner which allows the application of many standard neural network training techniques. This has even been extended to techniques such as Neural-SLAM [68].

Learning a policy from visual inputs in a simulator is usually an issue as the learned policy does not directly transfer to real life due to the dynamics of the real world differing from the simulation and distribution mismatch of the simulation render and the real-world camera image. This issue is an active area of research in the deep learning community. Various approaches exist including fine-tuning (partially retraining) the policy on the real world, learning robust transferable visual features, or modifying the input image of the simulator to look as the real image or the other way around. These will be discussed in detail in part V.

Chapter 3

NIFTi project

The main robot that we will be testing our methods on is an unmanned ground based vehicle (UGV) designed to navigate complex terrain cluttered with obstacles, such as construction sites, primarily for purposes of Urban search and rescue (USAR). It is part of a larger project named NIFTi [1], which is a human-robot collaboration that aims to assist in emergency and USAR situations. The NIFTi project was discontinued in 2013 and subsequently replaced with the TRADR project [4]. Figure 3.1 shows the UGV. To avoid confusion, we will refer to this robot as the NIFTi robot, UGV or simply 'robot' interchangeably from now on.

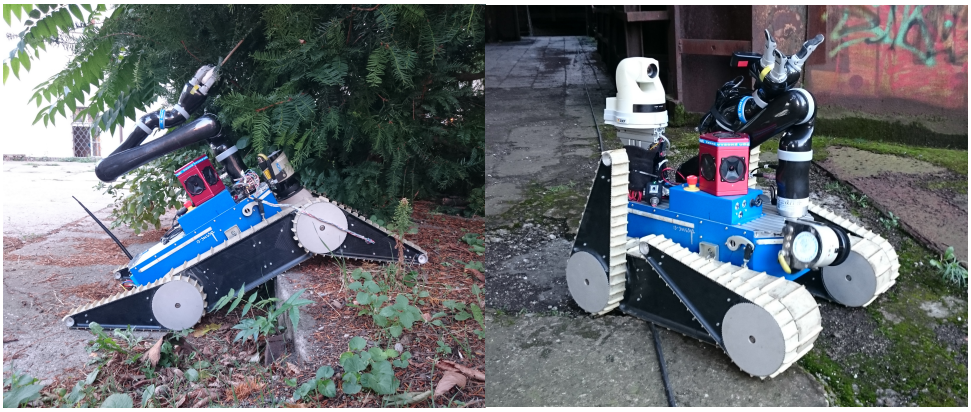


Figure 3.1: NIFTi UGV robot

Mechanical description. The UGV itself consists of two main tracks, called *bogies*, each of which has two independently controllable flippers attached. The track velocity of each bogie and the attached flippers is controlled by a single motor. The bogies are attached to the body via a revolving joint, although we use the inbuilt locking mechanism to disallow rotation, making the attachment fixed. The main body of the robot houses the battery and all the electronics. The motors for both the tracks and flippers are embedded inside the bogies themselves. The robot has a total height of about 41cm, width of 60cm and total length of about 115cm with the flippers extended. The platform weighs roughly 25kg fully loaded. The robot has an approximate maximum linear track velocity of $0.3ms^{-1}$ and maximum angular flipper velocity of $\frac{\pi}{4}rads^{-1}$.

The flipper torque is relatively low and the robot cannot lift itself up using the flippers.

Electronics. The electronics consist of an embedded x86 platform PC with a quad core processor, power distribution board for the whole platform and a sensor board. The communication between sensors and motors is done through a standard CAN bus.

Sensors. The sensors consist of a rotating laser scanner, omnicam, and IMU & GPS sensors. A Realsense R200 RGBD camera is connected to the main body using a 3-D printed mount, as shown in figure 3.2.

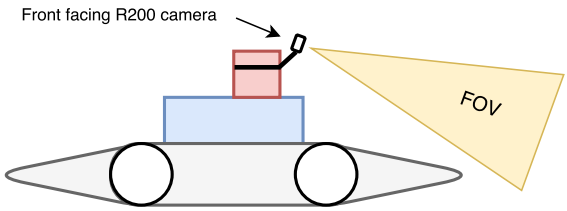


Figure 3.2: An illustration to show the front facing camera position on the robot. FOV is not to scale.

Software platform. The on-board embedded PC runs an Ubuntu Linux distribution with the Robotics Operating System (ROS) [48] serving as the middleware between the sensors, motors and processing unit. Robot control and sensor readings are done by reading or publishing from the appropriate ROS topics. This makes it easy to transition from a simulation to the real robot without much change in code. Along with the physical platform, a simulation model is provided for the Gazebo platform [36] for the UGV which we use in most of our experiments. In the simulation we set a simulation step value of 200 ms. This means that after publishing the actions we run the simulator for 200 ms before taking the next sensor readings and republishing the next actions. This rather large value is fine in our case where the robot is slow and dynamics don't play a significant role in locomotion, allowing for slower update rates.

Robot control. The robot control consists of the following main input signals: Two inputs for the track velocities of the left and right bogies, four inputs for the angle control of the flippers. The flippers are controlled by providing an angle reference, which is then realised using a built-in PID controller.

The locomotion task for this robot consists of having correct flipper angle configurations when trying to move through terrain or overcome an obstacle. This is manually controlled by the operator which is difficult and slow given the multiple degrees of freedom that the robot presents. In this thesis we essentially attempt to lessen the burden of controlling flipper position from the operator by delegating the flipper task to the robot.

Chapter 4

Tensorflow

One of the contributing factors to advancing neural network research is the ability to quickly and efficiently prototype and train neural network architectures. Tensorflow [5] is a framework, developed by a Google team which allows just that. It is based on a declarative programming style where the user defines a computational graph and then queries the graph for specific outputs given certain inputs. Every complex operation in the graph is decomposed into basic addition and multiplication operations. One of the advantages of this approach is that we can request gradients of any node with respect to any other node in the graph and get the gradients essentially "for free". Due to this feature, Tensorflow is sometimes called an *autodiff* software package, along with similar frameworks such as Theano [11] and Torch [17].

```
import tensorflow as tf

# Define input variable
x = tf.placeholder(dtype=tf.float32, name='X')

# Define constants a,b,c
a = tf.constant(0.3, name='a')
b = tf.constant(5., name='b')
c = tf.constant(1.7, name='c')

# Symbolic operation
y = a*tf.square(x) + b*x + c

# Open a session
sess = tf.Session()

# Run and print the symbolic variable 'y', feeding in the
# necessary x input having a value of '2'
print sess.run(y, feed_dict={x:2})

# Log the graph
writer = tf.train.SummaryWriter("/tmp/tbg", sess.graph_def)
```

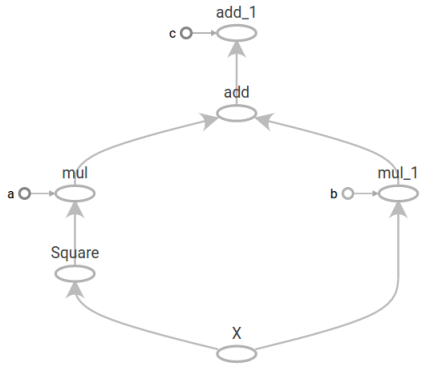


Figure 4.1: Simple code example in TensorFlow showing a computation of $y = ax^2 + bx + c$ along with the generated computational graph. Code example taken from [3]

Tensorflow also comes with multiple ready modules for neural networks, called layers which can be easily wired together and attached to an optimizer object. This leads to faster and less bug-prone prototyping. Tensorflow also comes with a sophisticated visualization tool called Tensorboard [2] which allows real-time monitoring and logging of almost all features inside the computation graph, such as neuron activation histograms, weight distributions, gradient magnitudes, network outputs, etc. This allows the user to tune

network parameters for more efficient operation or to debug a neural network architecture which is not performing as expected.

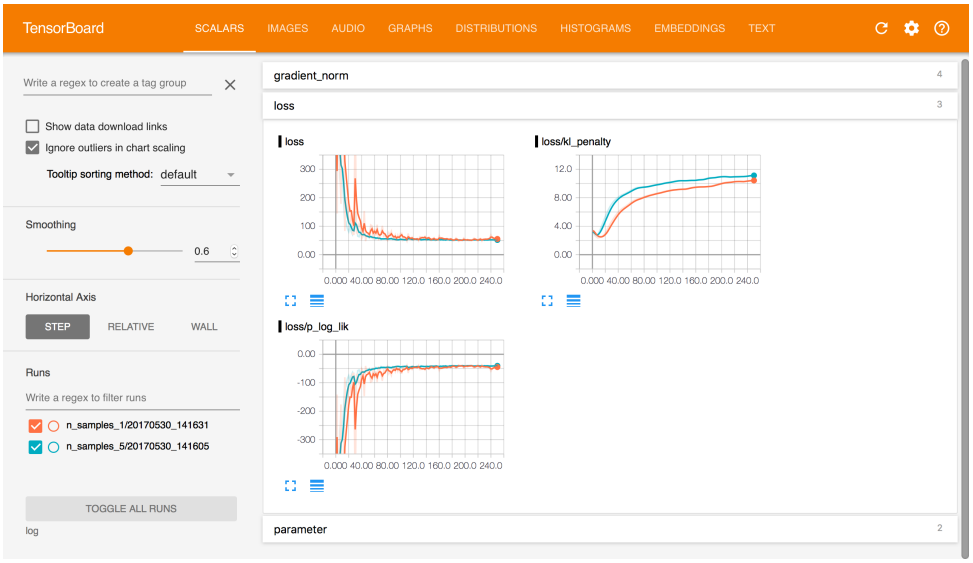


Figure 4.2: Tensorboard visualization example. Image taken from Edward: http://edwardlib.org/tutorials/tensorboard

Part II

Imitation Learning

Chapter 5

Definition of the task

What we are essentially trying to solve here is autonomous or rather semi-autonomous robot locomotion for the NIFTi robot. We define the task as follows: The robot is placed at position p_1 in the simulator and the task is to successfully navigate to p_2 through the placed obstacles towards the side of the goal. The exact y position of the goal is not important, rather, we only consider the x position and consider the task successfully solved when the robot passes the green finish line. Figure 5.1 shows a top-down 2 dimensional illustration of this setup, which we will refer to as *the scene*.

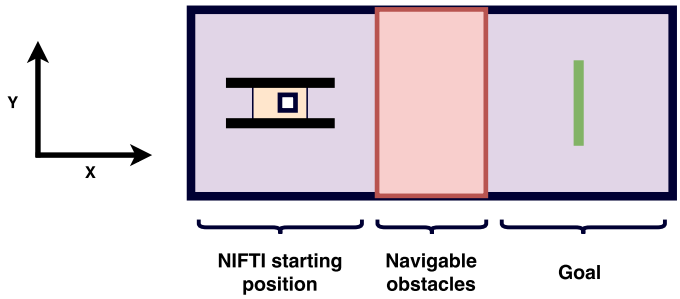


Figure 5.1: Task specification. The NIFTi robot starts on the left side and has to navigate to the right side through the red part with navigable obstacles past the green finish line.

For our task, we will use a subset of the NIFTi robots action space. Specifically, we will be controlling the continuous position of each of the 4 flippers in the range $[-\pi, \pi]$. The track velocities are controlled continuously by a human operator in the range $[-1, 1]$ where the boundaries of the interval denote the maximum track velocity. This makes the action space in total of 6 continuous actions. Concerning the inputs we have at our disposition the following sensor measurements: 6-DOF IMU, flipper angle measurements, and exteroceptive measurements including a front facing camera attached to robot and observer camera looking at the whole scene. We will be considering subsets of the action and input spaces for most of our tasks; e.g. The robot will be controlling the flipper angles only and the inputs can be either a single depth image or in combination with other proprioceptive data.

Policy expectation. For the robot to successfully navigate the obstacles to get to the other side it has to appropriately configure the 4 flippers at every step. If a certain flipper is too low it might get stuck under an obstacle. If it's too high, the angle of attack might be too high which will pose difficulty to the robot in ascending an obstacle. The robot has to support itself appropriately at every point, ideally keeping as much flipper contact as possible to maximize traction and to ascend/descend obstacles smoothly to avoid unstable positions where the robot can suddenly tilt and land hard on its flippers, potentially damaging the mechanism. These criteria are explained in more detail and with illustrations in part IV.

Obstacle generation. To train various policies throughout the thesis we use randomly generated obstacles at every episode. We use Palette models in Gazebo as our main obstacle objects which are of size $1.2m \times 1.2m \times 1.2m$. The palette is a good obstacle model because it features edges as flat top surfaces, holes in the side and other features which can make navigation over them tricky due to flippers getting stuck etc. We use 3 palettes which are generated randomly with x values varying across 1.5 meters, y values varying across 0.8 and z values varying across the range of the height of 2 palettes stacked on top of each other. Environments generated this way test the maximum physical limits of the robot. To evaluate various policies we define a dataset D_{25} which consists of 25 environments generated in the above fashion using a static seed. Environments also define a progress counter which measure the amount of x distance travelled by the robot in a single simulation step. If the robot takes to long or gets stuck, the episode counts as a fail.

Chapter 6

Learning from expert inputs

In this chapter we attempt to learn a locomotion policy from human *expert* demonstrations. This is commonly referred to as *learning from demonstrations*. In this case, the term *expert* is only used formally and refers to any human demonstrator who can perform the task to some extent. Learning from demonstrations is a useful approach in a scenario where we only have the real physical robot in the environment that we expect to deploy in, without any access to the robot or environment dynamics and no access to a simulator. Despite that, we do make use of the available simulator to test and verify the learning algorithms. In our setting, the robot is fully controllable by a human operator, meaning that it is not unreasonable to manually obtain a training set which can be used to train a locomotion policy. The many degrees of freedom make it difficult for an operator to perform the task at all, let alone in real time. A policy that successfully imitates some portion of the task alleviates the operator of some of the cognitive load required for it, freeing him/her up to perform other tasks that are vital to the mission.

6.1 Imitation as a supervised learning problem

We first consider the simplest form of learning from demonstrations, which is behavioral cloning [42]. This means that we simple want to imitate, or "ape" the demonstrations. To avoid the issue of partial observability we initially attempt to learn a policy from a representation which makes the task mostly fully observable. One simple way is to use a digital height map (DEM) to represent the obstacles around the robot and to provide localization and configuration information to the robot so that it knows where it is relative to the DEM. Another simpler example is to use pixel inputs of a camera looking at the whole scene, which we call the *observer image*. This arguably makes the task fully observable as we can see the location and configuration of the robot as well as all of the obstacles, albeit at an angle, meaning that the necessary 3D geometry has to be inferred by the policy. Due to the kinematic nature of our robot we can assume that a static image is all that is required for us to predict a correct action. Of course we cannot infer velocities from a single image which could lead to performance degradation but in our case this simplification suffices for some basic experiments. A simple solution to

this problem would be to provide the last 3 images as the state which is a commonly used technique in the atari game simulators. Figure 6.1 shows a montage view of the task from the observer image.

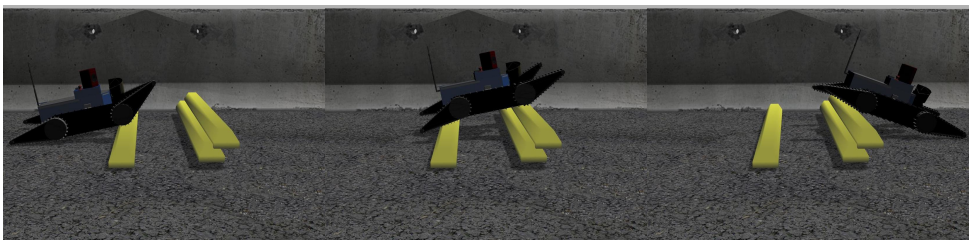


Figure 6.1: A sequence of observer images in the Gazebo Simulator

Markov Decision Process. Before formalizing the imitation learning problem we will first define the Markov decision process. We consider a setup where the agent interacts with environment E in discrete time steps t by performing actions. At each timestep the agent is in a state s_t and can transition to a state s_{t+1} by performing action a_t. The agent receives a scalar reward $r_t(s_t, a_t)$ from the environment which can be interpreted as a reward for taking action a_t at state s_t. It can also be interpreted as the reward $r_t(s_t, s_{t+1})$ of transitioning from state s_t to s_{t+1} due to have taken action a_t. Actions are taken according to a function $\pi : s \rightarrow a$, which we call the policy of the agent. The policy is non-deterministic in the general case and is denoted by $\pi(a|s)$ which is a distribution over actions given a state. The environment transitions from state s_t to s_{t+1} by probabilistic transition dynamics $p(s_{t+1}|s_t, a_t)$. A decision process is called *Markov* if it has the property where $p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_1, a_1) = p(s_{t+1}|s_t, a_t)$ for all t. The Markov Decision Process can be described by the following 5-tuple:

$$M = (S, A, P, R, \gamma) \tag{6.1}$$

Where S is a set of states of the environment, A are the actions of the agent, P is the state transition probability function, R is the reward function, and $\gamma \in [0, 1]$ is the discount factor which signifies the importance of later rewards as opposed to immediate ones. A gamma of $\gamma = 1$ means that we care about all rewards equally whereas a $\gamma = 0$ means that we only care for the immediate reward. For the imitation learning problem we will initially consider an MDP without rewards: $MDP \backslash R$

$$M_s = (S, A, P, \gamma) \tag{6.2}$$

We can then formalize the imitation learning as a supervised learning problem where for each state s_i the policy predicts an action a_i which is then compared to the expert action e_i for that state. This is a standard regression problem for which we will use the mean square error (MSE) loss as the optimization criterion. The MSE in this case is simply an easily manageable proxy for the optimal loss function which evaluates the action with respect

to the expert action, without taking into account their true distributions. Since this is inherently a sequential task where the state distribution depends on the policy, we have to take into account and take expectations over the distribution $\xi(\tau;\theta)$ of state trajectories τ which is induced by $\pi_\theta(s)$. The optimization problem can then be written out as shown in equation 6.4 where the policy $\pi_\theta(s)$ is parametrized by a parameter vector θ which can refer to the weights of a linear function for example.

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{\xi(\tau;\theta)} \sum_{t\in\tau} \|\pi_\theta(s_t) - e(s_t)\|_2 \tag{6.3}$$

Since the distribution of the robot policy is unknown before the actual optimization, we have insufficient information to solve the above objective directly. We can first relax the problem by ignoring the sequential nature of the state distribution and treat it on a per-example basis, making it into the following optimization problem which can be easily solved. The possible negative consequences of this relaxation are addressed later on in the chapter.

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \sum_{\tau\in T} \sum_{t\in\tau} \|\pi_\theta(s_t) - e(s_t)\|_2 \tag{6.4}$$

Here we denote $T = \tau_1, ...\tau_n$ to be the set of n available demonstration trajectories. In this experiment the track velocities are controlled by the user, leaving the policy to control only the flipper configurations. As the policy function $\pi_\theta(s)$ we use a convolutional neural network [21],[41](CNN) which takes which maps grayscale input images $s_i \in R^{d\times d}$ straight to actions $a_i \in R^4$, where the actions are the individual flipper positions of the robot. The structure of such a CNN policy is shown in fig 6.2. It is a fairly standard architecture with several alternating convolutional/pooling layers, followed by several fully connected layers. The convolutional layers extract lower dimensional features from high dimensional image inputs which are then fed into a fully connected multilayer perceptron. For this simple experiment and several following it we stick with standard CNN architectures and do not focus on state of the art techniques which offer improvements in sample efficiency and training stability as this is not the point of the experiment.

Training. The CNN is optimised with stochastic gradient descent, using minibatches of size 32 - 64. Minibatches improve training stability in deep neural networks by averaging noisy gradients and therefore preventing the parameters being knocked off too far from the solution manifold during updates. Algorithm 1 demonstrates this simple training procedure. We also add training from grayscale and depth images from the front facing camera attached to the robot for comparison. In this case, however, the robot also receives current flipper angle and IMU data which is integrated into the CNN at the first fully connected layer.

The trajectories are gathered in the following way: The user is given the same input observer image as the policy itself and controls the NIFTi robot using a plain game pad. The user uses the game pad joysticks to control both track velocities jointly (meaning the robot can't turn) and the 4

Algorithm 1 MSE behavioral cloning optimization

Require: Learning rate α, amount of iterations N
Initialize random parameter vector θ, expert trajectories $T = \{\tau_0, ...\tau_n\}$
for N iterations **do**
    Select random minibatch B of $\{s_i, a_i\}^m \in T$
    $\theta = \theta - \alpha \cdot \nabla_\theta \sum_{i=1}^{m} (\pi_\theta(s_i) - a_i)^2$
end for

flipper positions by varying the x and y positions of both joysticks, giving 4 continuous degrees of freedom, one for each flipper. Controlling four degrees of freedom this way results in quite noisy trajectories. During each episode all sensory and visual data is logged, along with the user actions. After each episode, the obstacles are repositioned randomly to prevent overfitting and to generate a more robust policy. We use the palette model obstacles as mentioned earlier. The objects are positioned at varying x, y and z coordinates and are made to be static in the simulator, meaning that they do not move. This is done primarily due to two important reasons. The first is that having many dynamic obstacles in the scene has a severe impact on performance due to the extra collision checking that the simulator has to perform. The second reason is that it would be more difficult to programmatically randomly reposition objects because static objects aren't checked for collisions between themselves so it is not a problem if their volumes overlap in the 3D space.

Evaluation. This simple approach gets decent results, even on relatively noisy and inaccurate user trajectories. The policy was trained on about 50 training trajectories with up to 100 time-steps each and fits them with no issue. We can also compare the test accuracy on 10 withheld trajectories. Typically in a given state the user will have performed various different actions. A visual inspection shows that the policy learns an "average" behavior over the noisy user trajectories which is to be expected given the MSE that we use to train it. Figure 6.3 shows the MSE training curve and results of supervised learning from the observer image and grayscale and depth images from the front facing camera attached to the robot. We can see that on the observer inputs the training error decreases rapidly and after about 500 iterations it starts to overfit because the test error starts rising slightly. Although the MSE gives an initial indicator of how well the policy fits the actions, it is not a good indicator of performance due to the sequential nature of the task. More detailed evaluation is discussed in the next section.

Discussion. We can see that using the observer image leads to overfitting with our small demonstration set and a relatively large test error. The front-facing grayscale image performs better than the observer image but both are outperformed by the depth image. It is, however, necessary to realise that we can easily constrain a regularize the network so that it does not overfit. This can simply be done by early stopping. If we stop training at 1000 iterations for the observer image we can see that the minimum test error

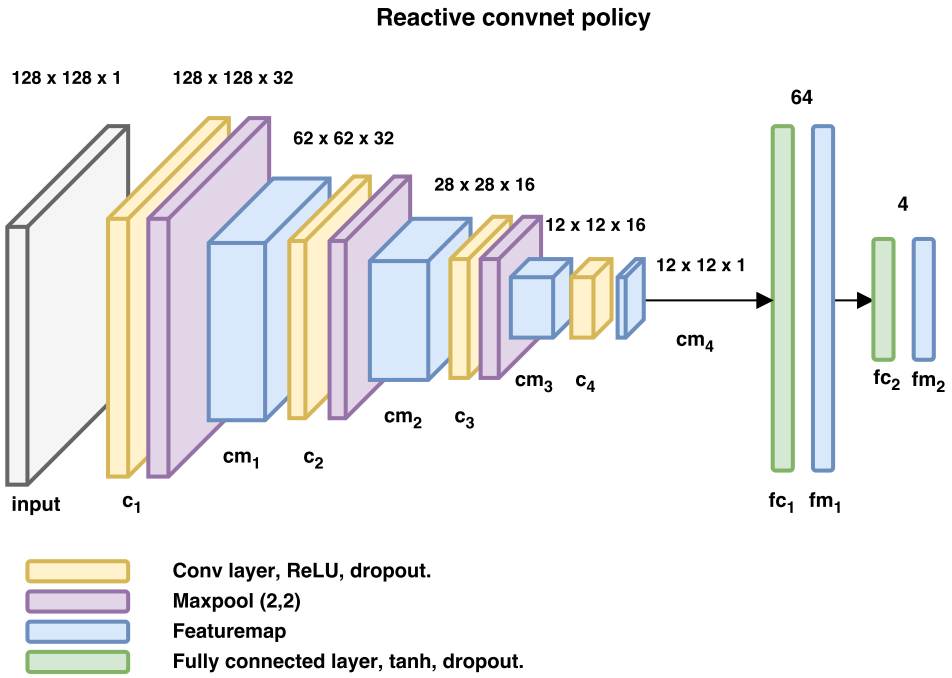


Figure 6.2: The architecture of the convolutional neural network policy. Convolutional layers are composed of a stack of 3x3 learnable filters, ReLU [46] activations, and dropout layers which set random neurons in a given featuremap to zero with a given probability. Maxpool layers are downsampling operations with a 2x2 window size and stride 2. The sized of the feature maps in the figure are approximate and might vary slightly depending on convolution filter sizes. The network has 8 convolutional filters per layer.

of 0.15 attained is similar to that of the rest of the runs, which means that all 3 runs have similar performance.

It is not unreasonable to expect that the agent can learn from pixel inputs of the whole seen as it arguably makes the task fully observable. It is, however, less clear as to why navigation from static front facing camera images works. Using a grayscale static image means that the agent has to infer the depth of the content to some extent so that it can judge what action to take. This is a relatively difficult task and requires the neural network to partially understand the underlying content in the image to be able to infer 3D structure. This has been done successfully using CNNs in several cases [51], [38] and has been shown to be useful for other tasks as well [15]. Using a depth camera such as the Realsense R200 alleviates the problem of having to infer the depth as it is provided as input directly. The disadvantage of using such as camera is typical failure cases on reflective surfaces, including floors.

6.1.1 The role of the convolutional layers

In our task we need to be able to somehow process high dimensional images into a lower dimensional vector which can be further processed by another

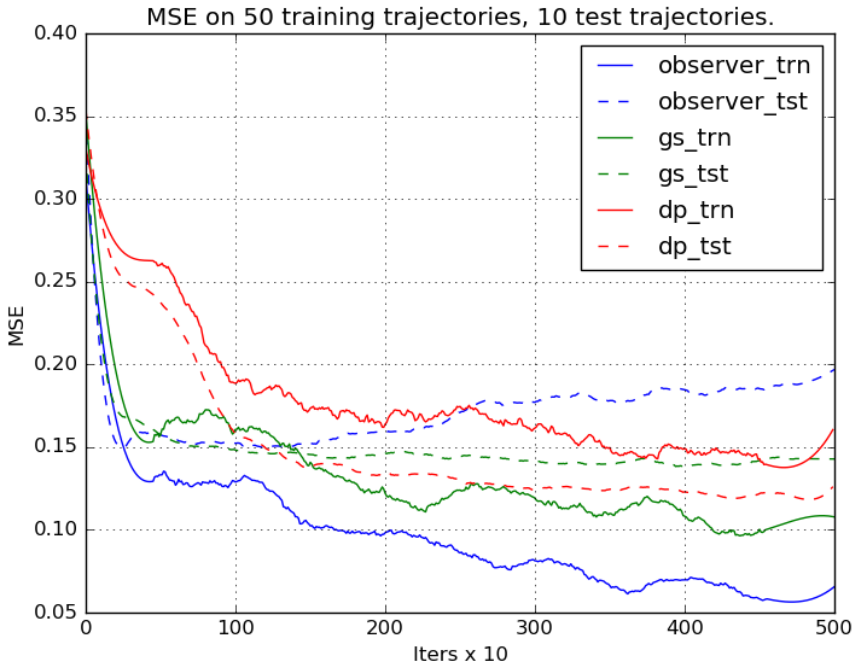


Figure 6.3: Training convnet policy to imitate expert trajectories from pixel inputs. Observer image is a 128×128 image looking at the whole scene. The *gs* run refers to the 64×64 grayscale image from the front-facing camera. The *dp* run refers to the front-facing camera image depth image.

classifier such as a fully connected neural network. The simplest and most naive way is to downsample the input image to a manageable size containing 50-200 features. This means having an image size of roughly 7-12 pixels which is way to coarse and will not work for most applications due to information loss. CNNs do a similar thing by subjecting the input image to a series of convolutions and downsampling layers, as shown in 6.2 which eventually gets the image down to a lower size. The difference here to the naive approach is that we downsample salient points in the image and therefore lose much less information.

After training a CNN it is useful to inspect the network to see what it has learned. One simple way to do this is to visualize the convolutional filters in the first layer. This layer operates directly on the image space so they should be interpretable as such. Figure 6.4 shows all 16 filters in the first layer before and after training.

Surprisingly, there are no noticeable patterns in the trained filters such as are obtained [37] from training CNNs on content classification on rich datasets such as ImageNet [18]. This is perhaps due to the fact that we are not learning to discriminate content and our environment is very static and the texture variance almost nonexistant. We further inspect the training process of the CNN and visualize the histogram of the weights in layers 1,2,3 and the activations of the embedding layer (first fully connected layer. Figure

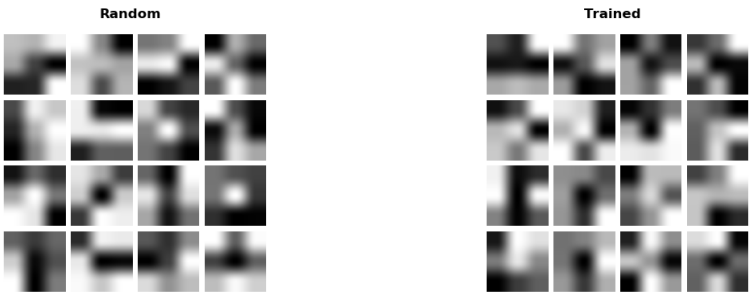


Figure 6.4: Random and trained filters from the first layer of the CNN. Filters are plotted with interpolation for easier perception.

6.5 is a screenshot from tensorboard showing the training process of the network on our supervised task. We can see that as the convolutional layers train the embedding layer activation features settle down into a smoother distribution.

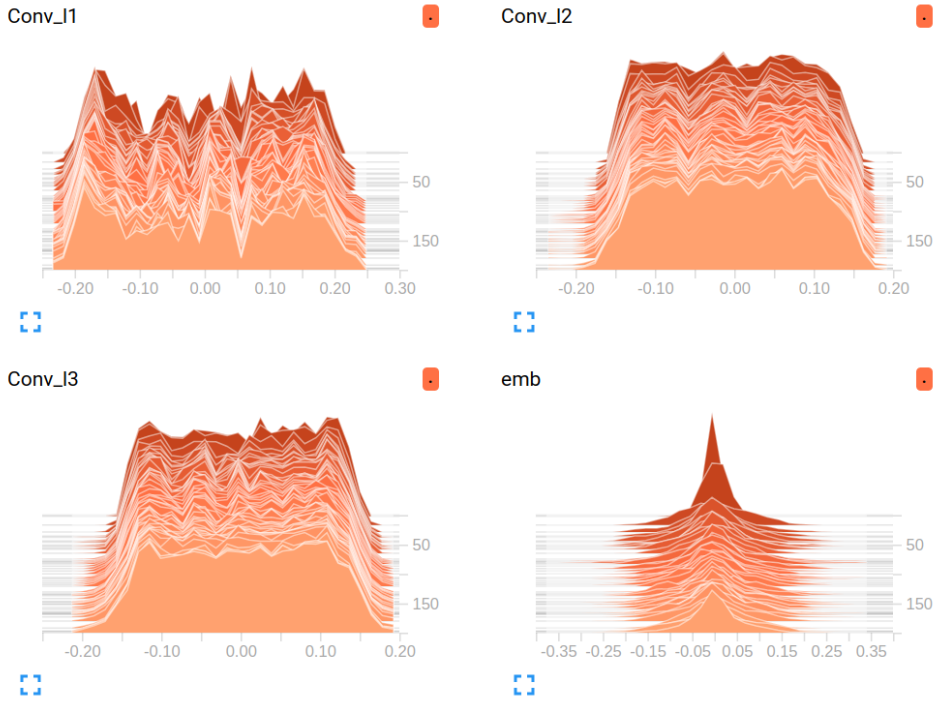


Figure 6.5: Tensorboard screenshot of the histogram distributions of the filters values in the first 3 layers of the CNN and the activations of the embedding layer. The horizontal axis denotes filter values and the individual slices are timesteps x10

Conclusion. In our task the CNN most likely only requires a rough understanding of 3D geometry in front of it and does not attempt to distinguish any objects by texture etc.

6.1.2 Addressing the covariate shift

We can see that the state-action mapping network generalizes to the test trajectories which is the first sign that tells us that the policy is fit to the demonstration examples and has learned somewhat of a meaningful policy that imitates the human expert. This can however be misleading because upon execution of the learned policy, the state visitation distribution will have changed and the agent will likely find itself in different states than it has seen in the training data. This can lead to a cascading failure of the policy where a small mistake leads the agent to a novel state, leading to further errors which compound and result in failure. Such an example is demonstrated in the car steering task in figure 6.6. Deviations from the usual trajectories compound and lead to agent crashing at the border. This issue has been known for quite a while and investigated in autonomous car driving but there are relatively simple fixes to this issue by using self-correcting feedback [12]. Covariate shift mostly affects dynamical systems where properties such as velocity play a significant role. Since the NIFTi robot is mostly kinematic, we expect that covariate shift will not be too much of an issue, but nevertheless merits investigation.

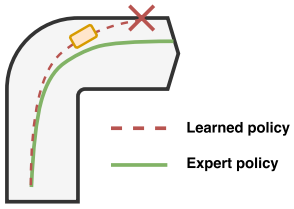


Figure 6.6: Steering problem: Small compounding errors leading to policy failure

The state visitation distribution mismatch of the learned policy to the supervisionary policy is often referred to as covariate shift in literature. Earlier we defined the trajectory distribution $\xi(\tau;\theta)$ on trajectories $\tau = \{x_0, a_0, x_1, a_1, ..., x_T\}$ where θ is the parameter vector of a policy π_θ.

$$\xi(\tau;\theta) = p(x_0) \prod_{t \in \tau} \pi_\theta(a_t|x_t) p(x_{t+1}|x_t, a_t) \tag{6.5}$$

There have been numerous works in the past that suggest techniques of dealing with covariate shift. An algorithm called Dagger [50] works by gathering rollouts from the current policy and querying the demonstrators for labels on those actions. Since the trajectories are gathered from the learned policy, the authors prove that the covariate shift eventually converges to zero. The algorithm steps are summarized in figure 6.7

The disadvantage of this algorithm is that it requires a human expert to provide actions on a state-action basis which is quite unnatural. It has been demonstrated that this approach can lead to noisy action labels which lead to poor results [39]. A different approach to imitation learning which deals with covariate shift is Inverse Reinforcement learning [6]. The idea is that instead of mimicking the demonstrations we learn the reward function

- Step 1 : Fit policy $\pi_\theta(s, a)$ on labelled demonstration dataset $D = \{x_0, a_0, ..., x_n, a_n\}$
- Step 2 : Run policy $\pi_\theta(s, a)$ to get unlabelled set $D_u = \{x_0, ..., x_n\}$
- Step 3 : Query expert for actions on dataset D_u
- Step 4 : Add new dataset to old $D \leftarrow D \cup D_u$, goto step 1

Figure 6.7: Outline of Dagger algorithm.

which the expert is acting upon and then use that to learn a policy using reinforcement learning. This approach however is computationally expensive and requires solving multiple reinforcement learning problems to compute the reward function which is unsuitable in our case since we assume that we only have the demonstrations at our disposal and no simulator. Other state of the art approaches such as GAIL [28] (Generative adversarial imitation learning) have been proposed but also rely on reinforcement learning, and thus a simulator which makes this method unsuitable for us as well.

Sequential evaluation. Having talked about various methods have been proposed to deal with this issue, we should first examine as to how much covariate shift affects our learned policy and if it is detrimental at all. We evaluate 3 reactive policies learned from pixel inputs on our D_{25} dataset, meaning that we run the policy on those environments and record the amount of successful trials. Table 6.1 shows the results. Surprisingly the observer image scores the lowest out of the 3.

	GS-Observer	GS-frontal	Depth-frontal
Success on D_{25}	15/25	17/25	20/25

Table 6.1: Evaluating reactive policies on D_{25} environments

We can see from the results in table 6.1 that covariate shift does not affect our system significantly because from a small amount of trajectories we are able to generalize to new randomly generated environments with good results. A visual inspection also shows that there are no situations where a compounding error leads to failure, although this is difficult to quantify. The observer image gets the worst results most likely due to the larger amount of information available in the image, requiring a more powerful network and therefore larger dataset to infer the required state information. Another reason is due to the small size of the robot compared to the entire image, making inference of the robot configuration and obstacles more difficult.

6.1.3 Potential issue of multimodality

There is a potential issue with the previous imitation learning setup that can come up the the case that the action distributions of the expert trajectories are

multimodal for some states. Consider the following scenario. If we are training an autonomous vehicle to avoid obstacles by swerving away from them, we are bound to have demonstrations in our dataset where we sometimes turned left and sometimes right to avoid the obstacle. Using MSE to train such a policy will result in averaging, and therefore an action which might not make sense. In this case the action will be to go straight forward into the obstacle. This is shown visually in figure 6.8.

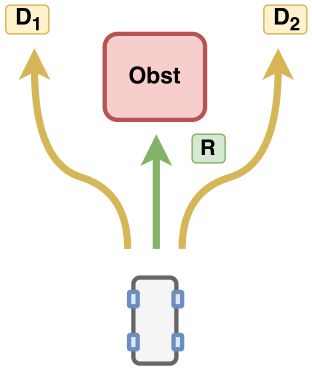


Figure 6.8: A scenario with two training examples, namely D_1 and D_2, both turning in opposite directions to avoid the obstacle. The resultant trained action R is the average turning action of both training examples and leads the vehicle directly into the obstacle.

The issue of multimodality does not affect us significantly in our case, at first glance, at least not detrimentally. This is because we are always navigating *over* obstacles and not under, so there is no significant disambiguation of the direction that a flipper should be turned to at a specific time-step unlike the steering problem described above. However, we propose a method to get around this problem in the case where it does affect us. To alleviate this issue we need an alternative loss function proxy that can handle multimodal distributions.

Discrete action spaces. If we are dealing with a discrete action space then we could use a cross-entropy loss L_{CE} at every timestep which allows for multimodal outputs.

$$L_{CE}(y, \bar{y}) = -\sum_i y_i \log(\bar{y}_i) \tag{6.6}$$

Where y is the true label for the given input and $\bar{y}$ is the predicted output probability vector. In neural networks outputs can be forced into probability distributions by feeding them through a Softmax layer where each unit $\bar{y}_i$ is a function of neuronal activations x_i:

$$\bar{y}_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{6.7}$$

As mentioned above, this requires a discrete action space which is not always possible to do. Using the most basic action discretization into 3

actions $\{up, no\text{-}op, down\}$, this leads to a total of $3^4 = 81$ actions in the case where we consider 4 flippers for our UGV robot, which is uncomfortably large, but tractable. This approach, however, clearly does not scale to larger action spaces due to the exponential growth. This approach would also not scale to sequences of actions for an action space larger than 1.

Continuous action spaces. It is clear that for larger continuous action spaces we need an alternative loss proxy. One such proxy can be in the form of Generative Adversarial Networks (GAN) [24] which will be addressed in detail in the next section.

6.2 Imitation using Generative Adversarial Networks

6.2.1 GAN theory

Generative Adversarial Networks (GAN) [24] have been a successful class of tools in deep learning for tasks such as learning complex data distributions, video representation learning [64], image super resolution [43] and missing data reconstruction. They have also been applied in the context of imitation learning for robotics [28] in numerous cases .

The idea of the GAN lies in a two network architecture consisting of a generator G and discriminator D. We assume that we have some real data $M = \{x_1, ...x_n\} \sim p(x)$ where p is the distribution that we would like G to model using the generator G using a latent variable z so that $G(z) \sim p$. The role of the discriminator $D(X)$ is to decide whether an input X was generated by distribution p (real) or by $G(z)$ (fake) by outputting a probability value of the input X being a Fake. The prediction gradient from the discriminator is then used to train both the discriminator and the generator using backpropagation. Essentially the discriminator is nothing more but a loss proxy which links the distribution of the generator and sampled data from an arbitrary other distribution. The networks form a minimax game with a value $V(G, D)$:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_x(x)}[\log D(x)] + \mathbb{E}_{z \sim p_{\mathbf{z}}(z)}[\log 1 - D(G(z))] \quad (6.8)$$

GANs can be extended to condition on inputs y to model conditional probability distributions. The inputs y can be any additional information such as class labels. The game value is then:

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_x(x|y)}[\log D(x)] + \mathbb{E}_{z \sim p_{\mathbf{z}|\mathbf{y}}(z)}[\log 1 - D(G(z))] \quad (6.9)$$

6.2.2 Applying GANs to our problem

In our case we have a setting where we are attempting to imitate an expert policy which is essentially a function $\pi(a|s)$ which probabilistically maps a given input state s to action a. For unimodal distributions we can model stochastic policies by mapping states s to a Gaussian distribution mean vector μ and diagonal covariance σ. This distribution is the easily sampled from and backpropagated to using the reparametrization trick [35]. However, when dealing with multimodal distributions there is no clear way on how to parametrically predict such a distribution and sample from it using a neural network. We can interpret the function of the generator $G(z|y)$ as implicitly forming a probability distribution $Q(y)$ and sampling from it using information from the latent variable z.

In our context of imitation learning our inputs y are the states s and variable z is a random vector drawn from an appropriate distribution, uniform or Gaussian. The data $X = x_1, ...x_n$ are actions from the expert trajectories. Informally, the generator generates an random action based on the state s and random vector z and the discriminator has to decide, conditioned on s, whether the action a was generated by the generator or whether it was a real action from the expert trajectories.

Training GANs. GANs are notoriously difficult to train due to their unstable training mechanics. There have been several documents and posts published on suggestions that improve the procedure based on empirical observation. We implement many of those suggestions in our generator and discriminator architectures. One common issue when training GANs is that the generator or discriminator eventually completely overpowers the opponent, leading to a degenerate solution for both sides. In our task, we notice that the generator often overpowers the discriminator. We attempt to remedy this by Training the discriminator 7 times as often as the generator. Algorithm 2, adapted from [24], gives an example of the training procedure using this setup. One disadvantage of GANs is that we do not have a clear performance criterium because we do not know the value V of the game. Because of this, the losses of the generator and discriminator are only used as debugging criteria. Since the generator is learning to generate actions conditioned on observations we can simply evaluate it using the MSE metric on our test trajectory dataset as a guide.

Evaluation. The GAN is trained for several thousand iterations until the test error doesn't go down any lower. The networks had a tendency to diverge after a while so we early stop then once a certain threshold is reached in the test error. We use the Adam optimizer [34] and learning rates of 10^{-4} for both generator and discriminator. Figure 6.10 shows a screenshot from tensorboard showing the generator and discriminator losses as well as the test loss on the withheld trajectories. We evaluate the GAN policy (generator) on our test trajectories to get a test error of 0.21 and a success rate of 15/25 on our D_{25} dataset. This is slightly worse than the evaluation of policies learned

Algorithm 2 GAN imitation learning

Initialize random discriminator and generator parameter vector θ and ϕ respectively, expert trajectories $T = \tau_0, ...\tau_n$, learning rate α, multiplier d
for N iterations **do**
  for d iterations **do**
    Select random minibatch of $\{(s^0, a^0), ...(s^n, a^n)\}$ $from$ T
    Select random minibatch of noise vectors $\{z^0, ...z^n\}$ $from$ $p(\mathbf{z})$
    Calculate discriminator gradient: $g_D = \nabla_\theta \frac{1}{m} \sum_{i \in 1..m} [\log D(x^i) + \log 1 - D(G(z^i))]$
    Update discriminator by $\theta = \theta + \alpha * g_D$
  end for
  Select new random minibatch of noise vectors $\{z^0, ...z^n\}$ $from$ $p(\mathbf{z})$
  Calculate generator gradient: $g_G = \nabla_\phi \frac{1}{m} \sum_{i \in 1..m} \log 1 - D(G(z^i))$
  Update generator by $\phi = \phi - \alpha * g_G$
  calculate generator and discriminator losses l_g, l_d
end for

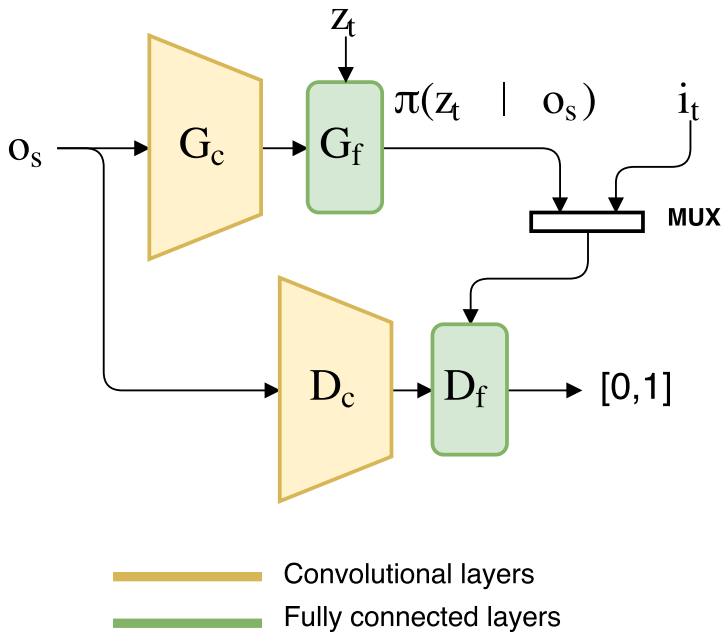


Figure 6.9: GAN structure for imitation learning.

by using the MSE metric.

Discussion. As a proof of concept, we saw that we could train imitation learning from an available dataset using GANs as the error proxy which we discussed would be very useful if the training dataset contained multimodal distributions of actions for a given state. We also argued that this is most likely not the case for our robot, or at least, it is not significant. We have to

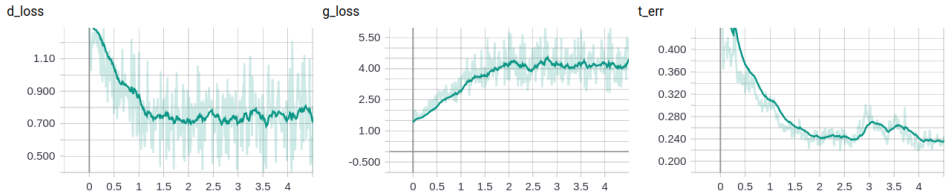


Figure 6.10: Screenshot from tensorboard showing GAN discriminator and generator and losses d_{loss} and g_{loss} respectively, as well as MSE on test dataset, t_{err}. The x-axis denotes hours of training

consider that what we attempted was only a state-action learning paradigm where we assume that the training data was generated by the user annotating given states with actions. This, however, is not the case. The actions were generated in a sequential manner, so it is not appropriate to model a distribution over actions for each state, but rather distributions over sequences of actions due to temporal correlation in the decision of the human annotator. This could be adapted to our GAN framework by considering conditioning not only on an observation o_t, but on a sequence of observations $o_t, ..., o_{t-n}$. This would also require a change in architecture for both generator and discriminator to something that could handle sequences efficiently, such as recurrent neural networks (RNN) which will be addressed in the next chapter. Since multimodality of action distributions is not an issue in our case, we not feel necessary to pursue this direction here and leave it as a suggestion for future work.

6.3 Dealing with partial observability by relaxing the markovian assumption

In the previous section we showed that it is possible to learn a locomotion policy from pixel inputs of the whole scene and from static images from a front facing camera attached to the robot. Due to the partial observability of the front facing cameras the robot could only fit the demonstration actions up to a point. This means that there can be crucial states between which the agent does not distinguish using this input. It becomes clear that to learn a better locomotion policy from on-board data of the robot the markovian assumption does not work and that we require some sort of memory. We require that our policy is a function of the history of states and actions. We can safely assume that for locomotion a only limited horizon of the history $h_n = \{s_{t-n}, s_{t-(n-1)}...s_{t-1}\}$ is necessary. The question then becomes how to efficiently use a limited horizon history h_n to predict the next action a given state s. Naively feeding the whole history h_n and state s to the policy to predict an action is very computationally expensive and will most likely lead to severe overfitting. We need a method that efficiently uses the structure in the history h_n. Ultimately, we want a function that efficiently maps every history h_n and current state s to a compact representation s_{h_n} which can

then be used by the policy to make a decision.

One way how we can efficiently compress a sequence of image data into a compact representation is to use 3D convolutional networks. These are an extension of the 2D convolutional neural networks that are used in 2D image computer vision. 3D Conv nets have been applied successfully in all sorts of time-series applications such as video analysis, motion classification [32], temporal feature extraction [62] and 3D tasks such as CT scan analysis [30]. At each time step we can use 3D conv nets to map a sequence of images $\{q_{t-n}, q_{t-(n-1)} ... q_t\}$ to a vector v_{img} of a fixed dimension m. A similar procedure using 1D convolutions could be used on sequences of low dimensional data inputs such as IMU readings. Although this method is not unreasonable, it does require recomputing all these convolutions on the entire history horizon at each time-step. An arguably better approach to represent s_{h_n} would be to use a Recurrent Neural network.

6.3.1 Using Recurrent Neural networks for history representation

Recurrent Neural Networks (RNN) [45] are a temporal extension to vanilla multi-layer perceptrons, first conceived around 1990 [19]. Their key feature is that they keep a recurrent state h_t at every timestep t which is a function of the previous state h_{t-1} and the current input x_t. The output y_t is simply a function of the current state h_t. This allows the RNN to learn dependencies over a sequence of steps since an output y_t at the current timestep is a function of all previous timesteps. In a way, we can interpret the hidden state of the RNN as a memory that the network decides to store given previous data, enabling it to make the correct decisions in future timesteps. The hidden state can also be seen as a compression of the complete previous history, but in a way, to retain only the features relevant to the task. This whole architecture is fully differentiable and can therefore be trained using a modification of backpropagation called backpropagation through time [65]. Training RNNs in their basic variant is difficult and faces similar issues to shooting methods in optimal control. This leads to problems such as the vanishing and exploding gradient problem. Failing to propagate the gradient back through the network through a sufficient amount of steps means that it will have trouble learning long-term dependencies over tens or hundreds of steps which is crucial for many temporal tasks, such as the one that we are trying to solve. Several improvements have been proposed that deal with the above issues, the most notable of which is the Long short term memory neural network (LSTM) [29]. This architectures deals with the issue by including auxiliary functions, called *gates* in each cell of the network. These gates regulate the flow of information to and from the hidden state, as well as the input at each time step. This allows the gradient to bypass certain parts of the chain in the backwards pass and travel longer distances. Figure 6.11 shows a vanilla RNN alongside and LSTM cell architecture.

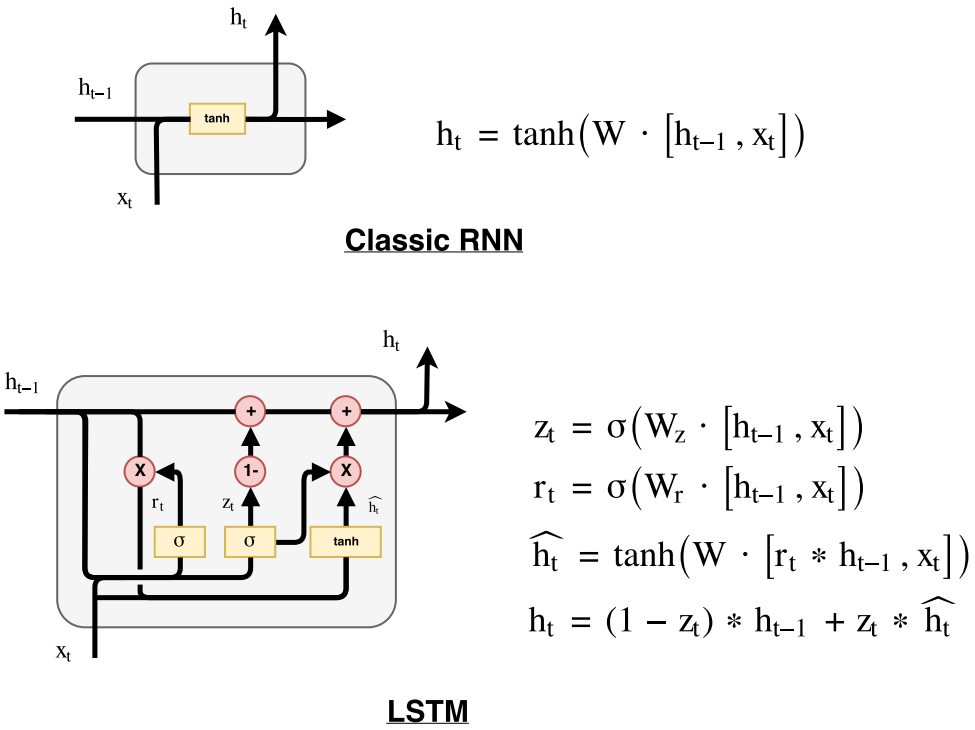


Figure 6.11: A vanilla RNN and LSTM architecture side by side for comparison.

History compression using LSTM. The question then becomes the following: Is an LSTM network capable of representing the entire history of input images and proprioceptive data in a compact low dimensional vector representation which contains the necessary information to make a decision? To attempt to answer that question we need to first think about what is inferable from a sequence of such data. To be able to navigate through an unknown environment the agent to be able to perceive the environment around it in a way that is relevant to the interaction of the agent and the environment. In the case of our robot, this means having a voxel map of the environment around it. The second relevant part is localization. The agent needs to know where in that environment it currently is. Classical planning techniques do exactly that through a technique called SLAM in which the agent iteratively constructs a map and localizes itself in it. One can argue that constructing the whole map and planning through it is excessive and wastes computation. To be able to navigate from point A to point B we do not need to build the entire map, but instead, infer the path through the relevant parts.

Architecture. We propose an architecture which uses CNNs jointly with RNN as a model which is sufficiently capable to learn a good policy from a front facing camera attached to the robot. The input space consists of a grayscale 64×64 image from the camera, as well as the current angles of the flippers and the robots internal IMU roll and pitch estimates. The image is fed into the CNN and the other two inputs are then joined with the convolutional embedding before being fed into the LSTM cell. The output

of the LSTM cell is then fed through two fully connected layers to give the 4-dimensional action. The architecture is shown in figure 6.12

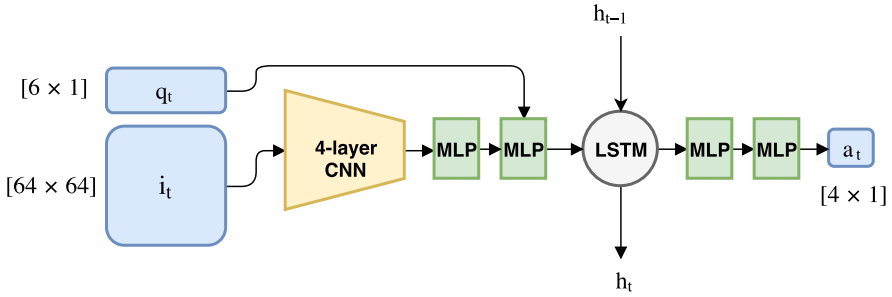


Figure 6.12: A vanilla RNN and LSTM architecture side by side for comparison.

Using LSTM to learn a supervised policy. We compare the results on our demonstration examples with the recurrent architecture using a raw grayscale image, depth image and the grayscale image modified with a Roberts edge detector. The motivation for using an edge detector is explained in part IV. The architecture is trained using the Adam optimizer and a learning rate of 10^{-3}.

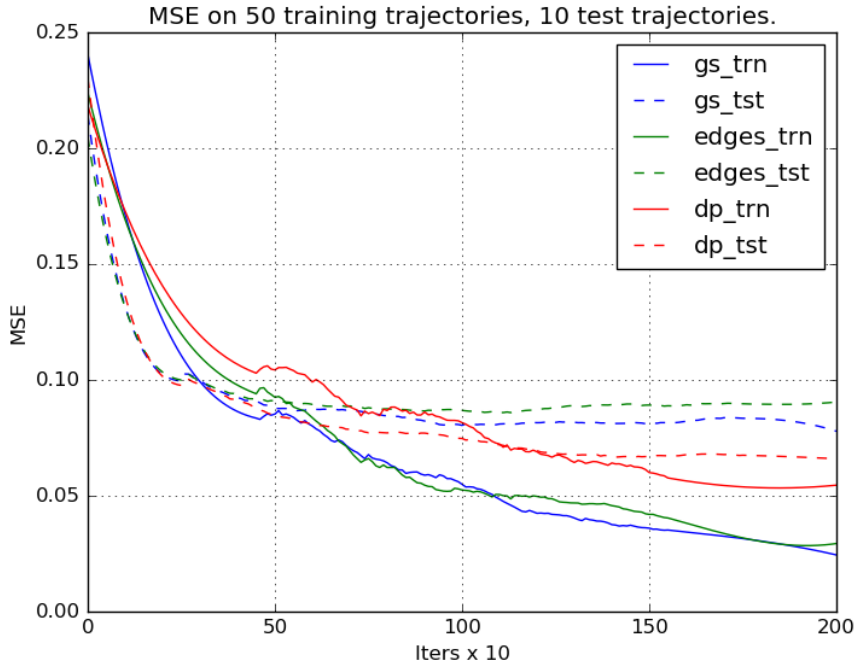


Figure 6.13: Training LSTM policy to imitate expert trajectories from pixel inputs. The *gs* run refers to the 64×64 grayscale image from the front-facing camera. The *edges* run refers to the grayscale image from the front-facing camera which has been preprocessed by a gaussian filter and subsequently a Roberts edge detector. The *dp* run refers to the front-facing camera image depth image.

	GS-frontal	GS-edge-frontal	Depth-frontal
Success on D_{25}	24/25	24/25	24/25

Table 6.2: Evaluating RNN policies on D_{25} environments

The results show that although the RNN policies are capable of getting significantly lower test errors on demonstration actions than their reactive counterparts, they perform similarly on the D_{25} test environments. This perhaps signifies that the D_{25} environments are not demanding enough to take advantage of the temporal capabilities of the LSTM networks.

Inspecting RNN memory usage. Having learned an RNN policy, we would like to have an idea of how much information it stores from previous inputs in its memory. We can attempt to do this by using gradient propagation of an output at a specific timestep with respect to all previous inputs. Formally we are interested in the gradient magnitudes

$$\|\frac{\partial a_t}{\partial o_i}\|_1 \quad \forall i < t \tag{6.10}$$

$$\|\frac{\partial a_t}{\partial o_i}\|_1 = \sum_{j \in \{1,2..Dim(o)\}} |\frac{\partial a_t}{\partial o_i^j}| \tag{6.11}$$

Intuitively we can interpret this as the amount of influence that input o_i had on the decision a_t. Figure 6.14 shows a plot of gradient magnitudes from the last timestep from an episode with respect previous inputs, and figure 6.15 is image showing the same thing but for outputs at each timestep.

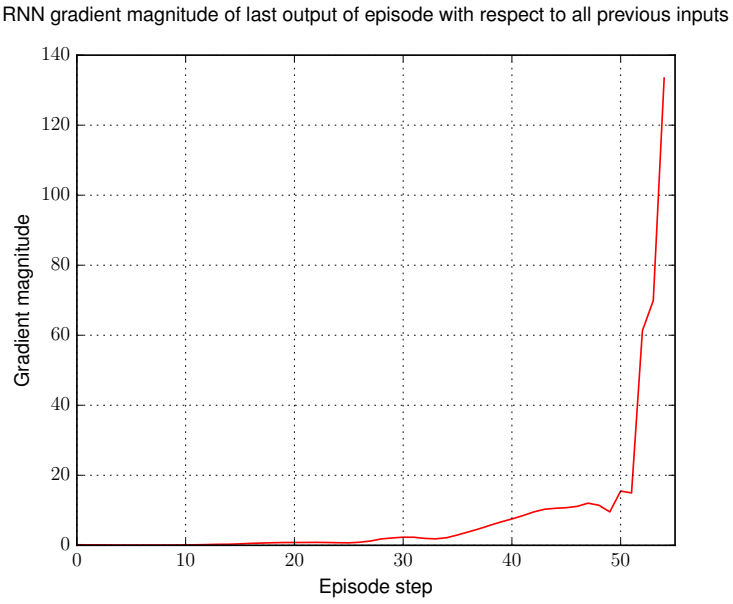


Figure 6.14: RNN gradients of outputs with respect to previous inputs.

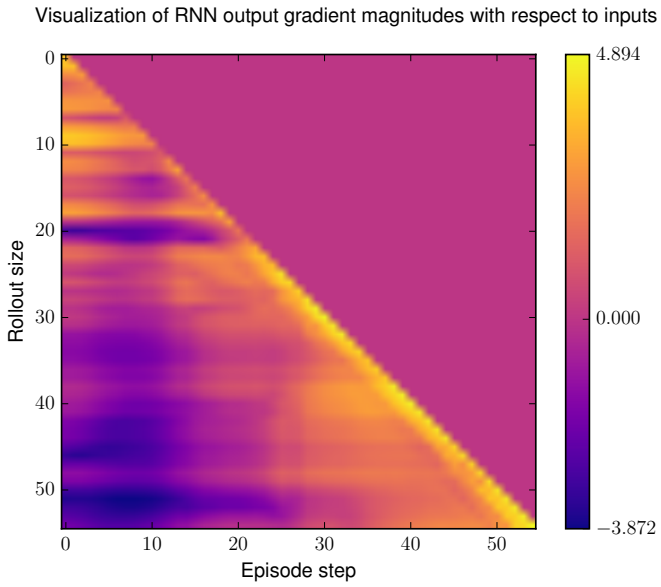


Figure 6.15: Log plot of gradient magnitudes for all timestep outputs.

We can see that the gradients decay very rapidly with each timestep. We also notice that the decay is not monotonic. Various timesteps can lead to a higher influence in alteration of what is being placed in the memory. This suggests that the RNN uses to an extent the past 20-30 frames to make a current decision, however, it is important to note that this is only a suggested interpretation.

Part III

Reinforcement Learning

Chapter 7

Policy learning through reinforcement learning

In part II we considered teaching an agent a policy by providing examples and having the agent attempt to clone those examples through supervised learning. This was done by the agent performing actions and learning from feedback. This method produced a decent policy which was able to generalize to new environments of the same type as it was trained on from only a 50 training demonstrations. One issue with this method is that the demonstration examples are imperfect due to human error. Another drawback is that we cannot tune the policy with respect to various criteria that we might desire without requiring demonstration examples that specifically adhere to those criteria which might be difficult or impossible to obtain. In this part we look at generating policies without requirement of any human demonstrations and that can be optimized to specific requirements provided by the user.

Learning without explicit supervision is a task which usually falls under the umbrella term of *self-training*. In this learning paradigm the agent performs sequences of actions from its current policy, called a rollout, then analizes the actions and improves upon them after receiving some feedback from the environment.

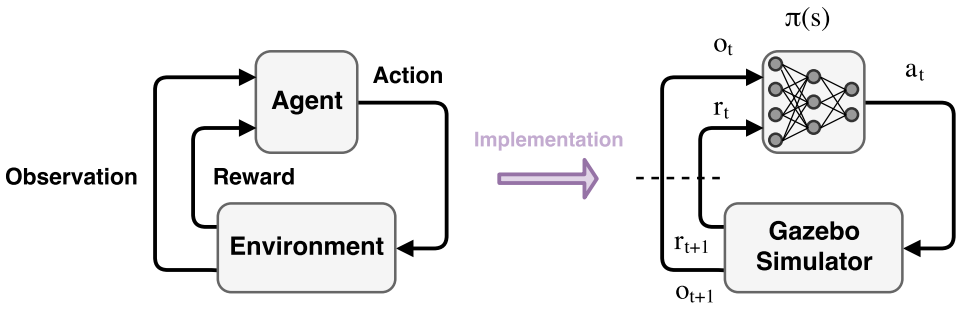


Figure 7.1: Agent-environment interaction. Since the interaction is a discrete process, the actual timestep has to happen in either the agent or environment. Here we assume that agent instantly returns action a_t for given o_t and the environment receives action a_t and returns r_{t+1}, s_{t+1}

This is essentially a trial and error method where the agent iteratively

performs actions and then improves upon them. The feedback is in the form of a sparse reward. The simplest (and sparsest) case is where the agent receives a reward of 1 if the sequence of actions leads to the agent solving the task, and 0 otherwise. Since the agent does not receive feedback for individual actions, but rather, for sequences of actions, it has to figure out which actions led to good rewards. This is called the credit assignment problem. The good actions of the agent are reinforced by providing high rewards and actions that lead to bad outcomes are suppressed by providing negative rewards. The task of the agent is to maximize the cumulative received reward. This specific learning paradigm is called *reinforcement learning* and loosely based on models of dopamine-based learning in mammalian brain.

7.0.2 Preliminaries

The goal of the reinforcement learning algorithm is to find a policy π which maximizes the expected cumulative sum of rewards $R_t = \sum_{t=0}^{T} \gamma^t r(s_t, \pi(s_t))$ for episodes with T timesteps, where $r(s, a)$ is the reward (feedback) that the agent receives from the environment. The γ coefficient discounts (reduces) rewards that occur in later timesteps which is a convenience for MDPs with a finite amount of steps and a necessity for infinite MDPs as a means to avoid infinite cumulative rewards. We will be dealing only with MDPs having a finite, but varying amount of timesteps.

The value function

$$V^{\pi}(s_t) = \mathbb{E}_{\pi}[R_t] = \mathbb{E}_{\pi}[\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}) | s_{t'} = s_t] \tag{7.1}$$

is a concept used in MDPs which can be intuitively interpreted as how good it is to be in a specific state s_t. We denote R_t as the return at timestep t and T is the length of the episode. It is to be noted that a value function $V^{\pi}(s_t)$ is tied to a specific policy π. One way to express the reinforcement learning objective is through the expected value of the initial state.

$$\mathbb{E}_{s_0 \sim p(s_0)} V^{\pi}(s_0) \tag{7.2}$$

A more useful concept for reinforcement learning is the action-value function $Q^{\pi}(s, a)$ which gives the expected return after taking action a in state s and following policy π afterwards.

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\pi}[\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'}, a_{t'}) | s_{t'} = s_t, a_{t'} = a_t] \tag{7.3}$$

A key property of both the value and action-value functions is the recursive Bellman Relationship.

$$V^{\pi}(s_t) = r(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) \tag{7.4}$$

$$Q^{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, \pi(s_{t+1})) \tag{7.5}$$

$$\tag{7.6}$$

This is useful because it tells us that the value at the current state is equal to the immediate reward added to the value in the next state. We can use this property to backup values through the MDP using a method such as TD-learning, which propagates temporal difference (TD) errors through the MDP.

$$TD_{target} = r(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) \tag{7.7}$$
$$TD_{err} = TD_{target} - V^{\pi}(s_t) \tag{7.8}$$
$$V(s_t) \leftarrow V(s_t) + \alpha \cdot TD_{err} \tag{7.9}$$

This is also called the $TD(0)$ update, because we are only looking one step ahead (the smallest amount). The value $V^{\pi}(s)$ can also be estimated using Monte carlo sampling by performing multiple rollouts from the state s and using the average return as the estimate. This is an unbiased but high variance estimate and is useful only is some cases. There exist methods such as $TD(\lambda)$ that lie between the $TD(0)$ and monte carlo updates. Multiple methods have been developed to solve MDPs such as Linear programming, dynamic programming and value iteration. These methods address MDPs with discrete states and actions and provide convergence guarantees in some cases using contraction mappings. Most of these algorithms work well on small problems but are very slow or completely intractable for more serious applications, where the state-space usually increases exponentially with problem size.

Q-learning. One of the most known algorithms for reinforcement learning is Q-learning. It is based on learning a state-action value function for the whole MDP, and after having learned that function we can obtain the policy simply by choosing the action with the highest value at each state.

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q(s, a) \tag{7.10}$$

Q-learning works by sampling actions from a sufficiently exploratory policy π and then updating the Q-function by using the temporal error from the recursive Bellman relationship.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{7.11}$$

The basic algorithm Q-learning algorithm is illustrated in figure 3. Typically we use an ϵ - greedy policy where with a probability $1 - \epsilon$ we choose action $a = \operatorname{argmax}_a Q(s, a)$ and a random action a_{rnd} otherwise. This is a basic exploration strategy for Q-learning. The issue of exploration-exploitation is explored more in the next section on deep reinforcement learning.

7.1 Deep reinforcement learning

So far we have been dealing with finite discrete state and action spaces where a tabular format (arrays) is used to represent value and action-value entries.

Algorithm 3 Tabular Q-learning

Require: Learning rate α, amount of iterations N
Initialize q table with zero values
for N iterations **do**
    initialize s_0
    for each step in episode **do**
        choose action a from ϵ-*greedy* policy using current Q values
        $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)]$
        $s \leftarrow s'$
    end for
end for

To get an idea of how much memory is required for a simple problem, we consider a grid world with $n \times m$ cells. If the world is static and only the agent moves then we have an $n \times m$ amount of states, meaning $n \times m$ entries for a value function and $n \times m \times q$ entries for all action-value entries, assuming that the agent has q distinct actions available at every timestep. Even for small grids where the environment is dynamic, with other moving entities, the amount of states increase exponentially with the degrees of freedom of the environment and this type of representation becomes intractable due to memory reasons and the fact that most entries will be unvisited after learning. In our case of the NIFTi robot, we have a continuous high dimensional state space which is an image and other proprioceptive sensors and a continuous action space with a dimension of 4-6. A tabular representation is clearly not suitable for our task. The rest of this section describes how we can use function approximators to represent Q-values for high dimensional and continuous state and action spaces to solve difficult problems.

Function approximators have been used in deep reinforcement learning to represent state and state-action value functions to solve many difficult problems. Linear functions are one choice where the we can represent a value by a linear combination of features $V(s) = \sum_{i \in \{1..n\}} w_i \cdot \phi(s^i)$ where w_i are weights of the function, ϕ is an optional feature map and s^i is the i_{th} feature of the state which can also be interpreted as dimensions, for a total of n features per state.

More powerful non-linear function approximators such as neural networks have been used in reinforcement learning with immense success and superhuman performance in tasks such as Atari game playing from pixel inputs [47] and playing the game of GO, ultimately defeating the world champion [55]. The word *Deep* simply means that we are using neural networks with more than one hidden layer.

It is common to use a convolutional neural network (CNN) to directly learn end-to-end from pixel inputs, either estimating a value for a given state (image), or directly as a policy function, mapping images to actions. We make use of this architecture extensively as we would like to learn a policy from observed camera inputs in an end-to-end fashion.

Many works have been published that provide methods of adapting reinforcement learning algorithms to work with neural networks. Unfortunately, when using non-linear function approximators most, if not all theoretical convergence guarantees are lost and the topic remains an active area of research. Training dynamics of large networks can also make training slow or unstable and require special techniques and adaptations to make them work such as batch normalization [31], dropout [57] and proper initialization [22]. A large number of parameters, such as in deep networks leads to other problems such as overfitting and large sample complexities leading to longer training times.

7.1.1 Deep Q learning

The first successful adaptation of Q-learning to difficult problems was playing atari games from pixel inputs [47]. Since then, DQN has had staggering success in the field and has been shown to be a good general algorithm that can solve a vast array of reinforcement learning problems. The basic principle is the same as in tabular Q-learning, adapted for use with function approximators. There are some key additions that were made to make the training stable. A deep neural network $Q_\phi(s, a)$ parametrized by parameter vector ϕ is used to approximate the state-action value function, mapping a given input and action to the expected value outcome. In the training phase episode rollouts are gathered using a typical ϵ-greedy fashion and the updates are in the form of gradients with respect to the mean squared temporal difference error at each timestep. Due to unstable training mechanics it was empirically found that it is advantageous to keep a target network Q_t which is used to sample new actions and is then updated as a moving average between the current and target networks using a small update constant τ

$$\phi_t \leftarrow \phi_t \cdot (1 - \tau) + \phi \cdot (\tau) \tag{7.12}$$

Due to the lack of i.i.d samples, highly correlated updates to large models such as deep neural networks can be detrimental. One method found by the authors to alleviate this issue is to keep a replay buffer from which state transitions are sampled and according to which the value network is updated. This makes the samples and therefore the updates less correlated, leading to greater stability.

Continuous action spaces. One thing to note is that the DQN works very well on high dimensional state spaces but expects discrete actions due to the requirement of being able to compute the $max_a Q(s, a)$ for the Q-learning updates. If we consider our NIFTi robot with 4 flippers we could discretize each flipper into 3 actions $\{up, no\text{-}op, down\}$ which leads to a total of $3^4 = 81$ actions. This is a relatively large amount of actions, albeit tractable, and might lead to optimization issues and hyperparameter tuning requirement. In any case, we want to be able to control the flippers in a continuous manner.

There is no straightforward way how to find the necessary $max_a Q(s, a)$ in a continuous action space. One thing that we attempted was to use gradient

ascent at every state s to find the $argmax_a Q(s,a)$ using the following iterative update rule.

$$a \leftarrow a + \nabla_a Q(s,a) \tag{7.13}$$

We found that unfortunately this method does not work, at least not in this naive way. There can be many reasons. The first is that deep non-linear functions have a multimodal landscape which means that we can get easily get stuck in a local optimum. There are, however, studies that suggest that local minima in neural networks are not so problematic and using gradient descent optimizers with momentum can overcome some of these issues. Another issue is the learned landscape of the Q function. Even though we expect neural networks to generalize from state to state, optimizing on unknown terrain can be expected to be more difficult. Even if this optimization process did work, it would still be impractical due to the computational overhead required at each step to optimize the $max_a Q(s,a)$.

DDPG. A method that directly extends Deep Q-learning to high dimensional action-spaces is the Deep Deterministic Policy Gradients method [44], which is also the algorithm that we use to learn a locomotion policy for the NIFTi robot. The DDPG is an actor-critic method where the actor is our policy and the critic is a function that evaluates the policies action. This is implemented as two separate neural networks; One for the actor (the policy), mapping states to actions, and an additional critic which is the Q-function. The role of the critic is to provide a proxy for the environment feedback which the policy can use to improve itself. The core of this method relies on the family of algorithms called policy gradients [58], which is a popular method to solve problems with continuous action spaces. The policy gradient is simply a way to relate the effect of the policy on a set of collected trajectories in a differentiable manner. The fundamental result of the policy gradient algorithms is the policy gradient theorem [58], [56], which reduces the performance gradient to a single expectation.

$$\nabla_\theta J(\pi_\theta) = \int_S \rho^\pi(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) da ds \tag{7.14}$$

$$= \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta log\ \pi_\theta(a|s) Q^\pi(s,a)] \tag{7.15}$$

Where s, a are the states and actions, $J(\pi_\theta) = \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[r(s,a)]$ is the performance objective, $\rho^\pi(s)$ is the state distribution, π_θ is a policy function parametrized by θ. Once the gradient is obtained, we can simply ascend the parameters of our policy in the appropriate direction. The advantageous property of this theorem is that even thought the state distribution $\rho^\pi(s)$ depends on the policy parameters θ, the gradient of the parameters does not depend on $\rho^\pi(s)$.

In the DDPG algorithm the critic is updated using the TD-error Bellman backup, similarly as in tabular Q-learning. The actor is updated from the

action gradient of the Q function through the chain rule with respect to the actor parameters.

$$\nabla_{\theta^\mu} J \approx \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}] \tag{7.16}$$

$$= \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta_m u} \mu(s|\theta^\mu)|_{s=s_t}] \tag{7.17}$$

One way to think about it, is that the gradient $\nabla_a Q(s, a)$ gives the direction that the action a should be moved in order to get the greatest increase of value Q at state s, and using the chain rule we can update the actor (policy) $\mu(s|\theta^\mu)$ with respect to actor parameters θ^μ. The whole DDPG algorithm, as taken from [44] is shown in figure 4.

Algorithm 4 DDPG algorithm

Require: Learning rate α, τ, amount of iterations N, minibatch size B
Initialize actor μ and critic Q network parameters θ^μ, θ^Q as well as their target Q' and μ' counterparts
Initialize replay buffer R
for N episodes **do**
    Initialize random process $\mathcal{N}$
    Initial observation s_1
    for each step in episode **do**
        choose action $a_t = \mu(s_t; \theta\mu) + \mathcal{N}_t$
        step the environment using action a_t and observe reward r_t new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample random minibatch of size B from R
        Set target $y_i = r_i + \gamma \cdot Q'(s_{i+1}, \mu'(s_{t+1}; \theta^{\mu'}); \theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i; \theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a; \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s; \theta^\mu)|_{s_i}$
        Update the target networks:
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu'}$
    end for
end for

7.1.2 Applying Deep RL to the NIFTi robot

We adapt the DDPG algorithm to the problem of controlling flippers of the NIFTi robot. The reason DDPG was chosen over other state of the art RL algorithms such as TRPO [52] and PPO [53] is that actor-critic methods tend to be the most sample efficient methods and that DDPG is an *off-policy* algorithm. *Off-policy* means that we evaluate a different policy to which the current rollouts are being sampled from. This allows us to deal with the exploration problem separately. The environment is again a closed, walled-off mini arena as shown in figure 5.1 where the robot starts on the left side and

the task is to navigate over the navigable obstacles to get to the finish line on the right side. Implementation-wise, this task is formed very similarly to the OpenAI gym environments [14] which are a set of environments created by OpenAI, defining a simple API compatible with reinforcement learning. We define the environment for the NIFTi robot according to this API so that it can be compatible with readily available RL algorithms with little or no modification.

- The environment starts in a deterministic initial state s, given by observation o which is returned by the $reset()$ function.
- A $step(a)$ function accepts an action, and returns a tuple (o, r, d) where o is the observation, r is the reward $r(s, a)$ and d is a boolean variable denoting whether the episode has terminated
- Upon termination the $reset()$ function is then used to reset the environment to its initial state and returns an observation o.

The state transition function is given by the simulator itself which is implemented by an inner $step_{sim}$ function. The observation, reward and termination conditions are user specified and have a profound effect on the performance of the RL algorithm.

Observation. As the observation at each timestep (state of the MDP) we will use the front facing depth image, as it had the best performance in the supervised task. The depth image has a resolution 64×64 and is preprocessed by clipping depth and NaN values to 1.5 meters. This input requires a CNN with over 10^3 parameters to process. Although the original authors of the DDPG algorithm demonstrate that there is no significant issue (besides more difficult training dynamics) to use DDPG straight on pixel inputs, we initially attempt to train the policy using a provided embedding of the image in attempt to save computational resources. There are several options on how to do this. We will explore the options which involve taking an already pre-trained policy from a similar task and using a subset of the neural network layers, namely the convolutional layers as feature extractors. One such candidate is to use the first fully connected layer of the policy used in training the policy from demonstrations in part II, giving a vector $\phi(o) \in R^{64}$. This policy used exactly the same input and the task was to map input images to actions in order to imitate the expert. We can therefore assume that it encodes relevant features that enable the policy to understand the configuration of the robot and infer the necessary information of the environment around it for locomotion in general.

Reward. The reward given to the agent is essentially the only source of information which it receives about how well it is doing in the environment which makes it a crucial part of designing an environment for RL. It is important to consider the behavior of the robot in mind when designing the reward function because it can often times happen that the agent abuses the

reward and performs unintended actions. Since we want the robot to traverse obstacles and get to the other side, it makes sense to provide a reward r_{dist} for every δx travelled at each timestep, or the velocity $\dot{x}$ at which it's travelling. This is scaled up by a coefficient c_{dist} so that the average travelled $c_{dist} \cdot \delta x$ value without obstacles is around 1 at each timestep. A common penalty imposed on agents in reinforcement learning is the timestep penalty where the agent gets a small valued penalty for every taken step because we usually want to achieve a task as fast as possible. We use a timestep penalty of $t_p = -0.6$. The last penalty that we will use is the euclidean norm of the actions $\|a\|_2$, also called the *control effort* in the context of optimal control. This penalty is also scaled by a coefficient c_{eff} so that a full control effort has a penalty of -1. The total reward is then

$$r(s, a) = c_{dist} \cdot (x_t - x_{t-1}) + c_{eff} \cdot \|a_t\|_2 + t_p \tag{7.18}$$

As shown above, there are a few hyperparameters that have to be tuned to get a reasonable behavior. A set of coefficients which highly rewards r_{dist} in relation to the others sometimes results in a behavior where the agent is locked and stays in an obstacle, causing the simulator to push the robot back and cause a periodic rocking motion which gets high reward but no result. Another issue excessive angles of the flippers if the control effort is not penalised enough, and a reluctance to do any action if it is set excessively high. It is also important not to have very high rewards because they can generate high value gradients and make the training process unstable.

Termination condition. The episode is terminated if:

- The robot crosses the finishline
- The maximal amount $s_{max} = 100$ of steps has passed
- The angle of robot IMU reading is too high (the robot is tipping over)
- The robot gets stuck in an obstacle

The last one requires the most tuning. To detect if the robot is stuck, a progress score is used which maxes out at a certain value and is decayed when the robot is not making any progress on the x axis or rotational axes. It is important that the detection of this state works well, otherwise the trajectories will consist of many steps of stuck configurations yielding no learning benefit.

DDPG hyperparameters. For the most part, the default parameters of the algorithm itself yield a successful training process. These include the learning rates of the actor and critic network, set at 10^{-4} and 10^{-3} respectively. The replay buffer size is set to 10^6 The target network update rate $\tau = 0.001$ gives stable updates, although a smaller update results in faster learning. The exploration noise used is the Ornstein Uhlenbeck process which is the recommendation by the authors in the original publication. One of the most

important parameters is the discount factor γ which weighs the importance of immediate rewards versus delayed rewards. We set a γ value of 0.95 so that the maximum obtainable cumulative reward did not exceed ~ 50. This was done to limit the update gradients of the critic network and also so that the critic does not have to predict a very high Q value in the order of hundreds. In the appendix we take a look at possible issue with regressing on very high numbers with neural networks.

Training variations. To create the navigable obstacles, we use a set of available objects in Gazebo and after every episodes, randomly move the obstacles in the x, y and z directions in a way which keeps the task challenging. This is done so that the agent learns a robust policy and doesn't simply fit to one specific obstacle manifold. This is unlike most reinforcement environments which are static or deterministic.

Replay buffer and dynamically changing environment. Since the DDPG uses a replay buffer of past state-action transitions which are randomly sampled and learned from, one could ask how it can be compatible with a constantly changing environment. This is a big obstacle in using DDPG for multi-agent environments where using past experiences does not necessarily make sense because the policies of the other agents differ at the current time step and it makes sense to do something else. It is not, however, a problem in our case because although the environment changes, the task itself does not. The abstract notion of navigating over obstacles does not change. If it had been the case where the physics of the simulation changes over time, then this might have posed an issue for the algorithm. In any case, this is a specific issue of using past experiences from the replay buffer, and in the worst case, the replay buffer could be removed or severely shortened, but at the great expense of sample efficiency and training stability.

Initial experiments. We perform some initial experiments using the reward function specified above and also compare this to the sparse reward function where the agent gets a 1 for crossing the finishline and 0 otherwise. The sparse reward runs usually take much longer to start doing something sensible and are less stable. Figures 7.2 and 7.3 shows screenshots from tensorboard of a sample of the training progress of the algorithm. The algorithm takes about 12 hours to train to saturation. This is mostly due to the very slow simulator which is running at a rate of at most 3x realtime. Figure 7.4 shows screenshots of various palette configurations that the robot is trained on.

7.2 Exploration

The the context of reinforcement learning, exploration refers to performing sequences of actions which lead to unvisited states of the underlying MDP. Typically we do not know the rewards attained in each state and would like to discover them. In the general case the rewards are stochastic and admit an

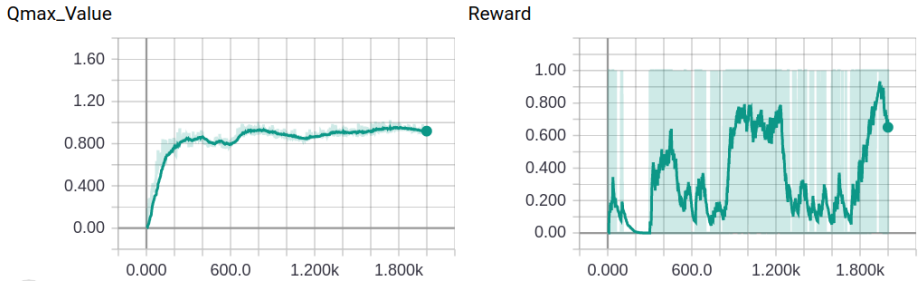


Figure 7.2: DDPG run on NIFTi robot using sparse reward function. X axis denotes episodes.

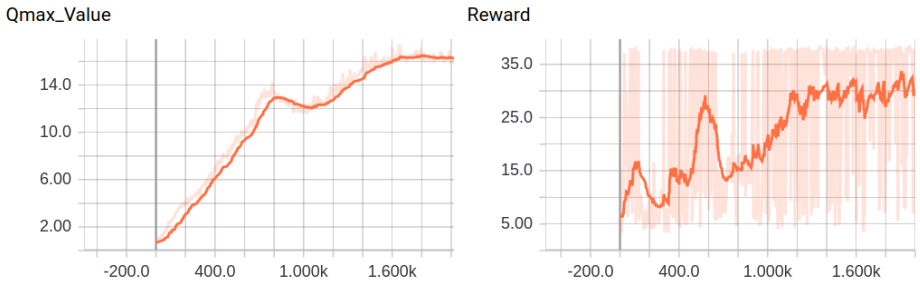


Figure 7.3: DDPG run on NIFTi robot using Shaped reward function. x-axis denotes episodes.

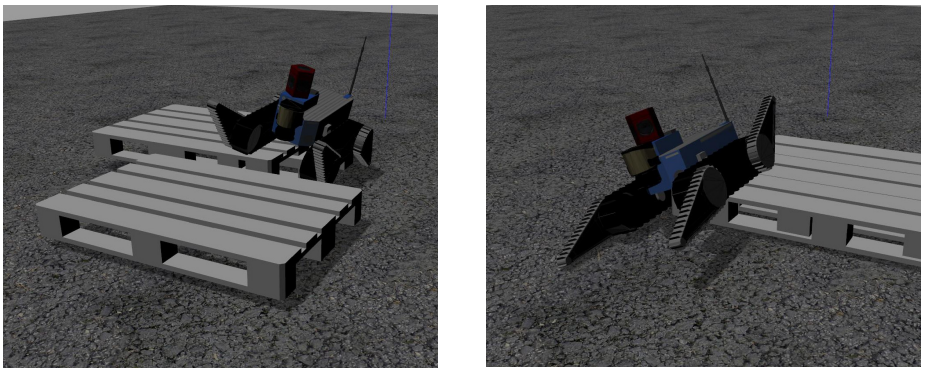


Figure 7.4: Screenshots of various trained palette configurations

unknown distribution for each state and action. Since the goal of reinforcement learning is to maximize cumulative future reward, the task becomes not only to visit the most novel states, but to exploit information about already visited states to get higher rewards. This is called the exploration-exploitation problem. Exploring efficiently is important especially in DRL where the sample complexity is high and therefore computation times very long. Improvements in exploration can significantly shorten training time and improve solution quality.

There are several exploration techniques that have been developed in classical reinforcement learning. The simplest approach is to simply select

random actions with probability ϵ and use the currently learned policy otherwise. This is called the $\epsilon-greedy$ policy. A more sophisticated approach is to track the visitation count of each state, along with its reward distribution (or simple the average), and prefer to visit states with low visitation counts and high reward. In the context of very small MDPs the exploration can even be computed optimally using Bayesian learning, but is intractable for most problems. The problem of exploration of exploration becomes even more tricky for problems with continuous and high dimensional state and action spaces. We will examine some of the techniques that we used for the NIFTi robot locomotion task.

7.2.1 Random processes

We first take a look at applying a simple $\epsilon-greedy$ approach. This essentially means that we will be introducing random noise at each time step in attempt to visit novel states. This can be problematic for systems with inertia or slow-moving parts, which is the case for the NIFTi robot. The flippers have an angular velocity lower than $45\,\mathrm{deg}\,s^{-1}$ which means that if we inject random noise at each time step, the result will be erratic movement about the zero point meaning that we will not explore the effect of interesting configurations of the flippers. What is instead required is temporally-correlated noise such as a random process.

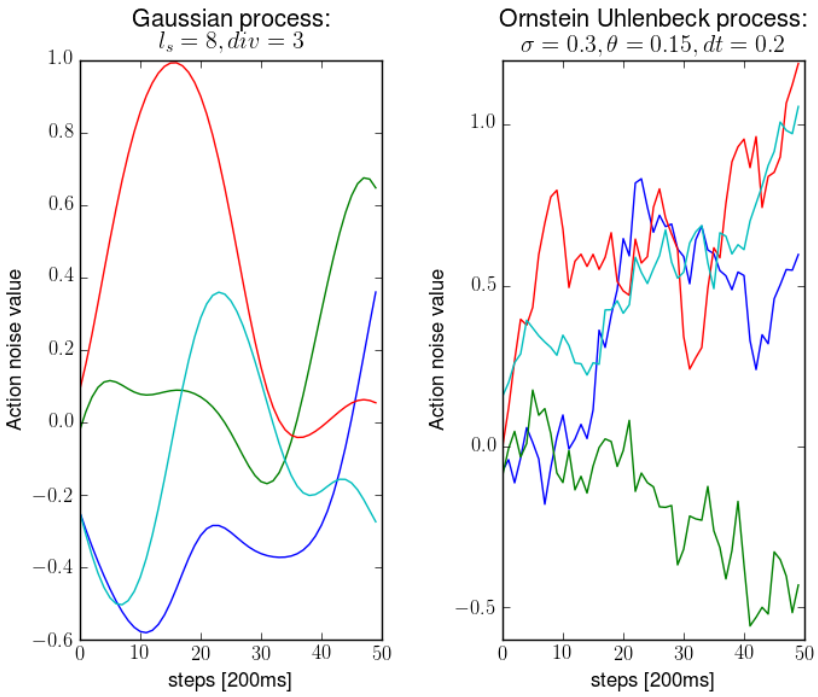


Figure 7.5: Samples from Gaussian process and Ornstein Uhlenbeck process

The process used in the original DDPG publication was the Ornstein-

Uhlenbeck process [63] which models the velocity of a Brownian particle with friction. The calculation of the noise value at timestep t is given by:

$$v_t = v_{t-1} + \theta \cdot (\mu - v_{t-1}) \cdot dt + \sigma \cdot \sqrt{dt} \cdot \rho \tag{7.19}$$

Where θ and σ are hyperparameters of the system, dt is the time delta between steps, and $\rho \sim \mathcal{N}(0,1)^m$ where m is the dimension of the action-space. Another way to get temporally correlated noise is to use points sampled from a Gaussian process with an exponential kernel. The parameters are fit such so that the noise values are smooth and range roughly from $[-1, 1]$. The time scale is such that the robot flippers have enough time to follow it. This method of exploration works fairly well on many environments, including our NIFTi robot environment and achieves a decent result.

7.2.2 State-based (Hashing)

A recent idea in deep reinforcement learning was to revisit count-based exploration [60]. In principle, what we want is for the policy to prefer actions which lead to novel states. One way how we can achieve this is to reward the policy for performing such fruitful actions.

Since we are working directly with high dimensional pixel-state images with dimensions of approximately 10^4, it is more practical hash lower-dimensional feature embeddings of the images. Since a convolutional neural network is used to map input images directly to actions, it's possible to use an intermediate layer of the network as a feature vector which contains relevant parts of the image for further classification of an action. The question is which layer to take features at. Layers closer to the pixel-space have more general lower-level features that pertain directly to image processing, such as edge detectors, and features higher up are more relevant to the task at hand. We will use features in the first and second fully connected layers. The architecture is shown in figure 6.2. We could also use an autoencoder [9] to embed the image into a latent representation which could then be used as our feature vector.

Since the state-space is continuous and high-dimensional, we require some discretization method. The authors [60] suggest a technique called locality-sensitive-hashing (LSH) to convert high dimensional continuous vectors to discrete hash codes which are more manageable. One such computationally-efficient method is the Simhash algorithm which buckets vectors according to their angular distance metric [16]. The Simhash mapping is shown in equation 7.20 where a k-dimensional hash code $g(s)$ is produced from state s. A is a $k \times D$ matrix containing entries drawn from a gaussian distribution $\mathcal{N}(0,1)$. Parameter k controls the granularity of the input separation. $\phi(s)$ is an optional feature embedding of the state s.

$$g(s) = sign(A\phi(s)) \in \{-1, 1\}^k \tag{7.20}$$

The SimHash function can bucket continuous states into discrete buckets, allowing counting of the states. The auxilliary exploration reward can then

be provided as addition to the regular reward. The total reward $r_t(s, a)$ is then

$$r_t(s, a) = r(s, a) + \frac{\beta}{\sqrt{N(s)}} \tag{7.21}$$

where β is a constant term which weighs the exploration bonus and $N(s)$ denotes the state count. The authors of the this method demonstrate significant improvements in the training time in some environments, especially in presence of sparse rewards.

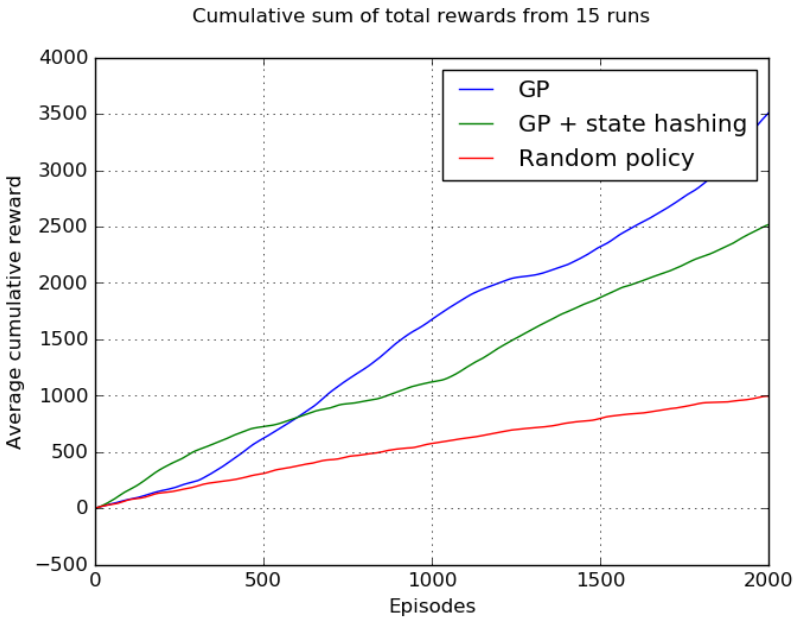


Figure 7.6: Average cumulative reward using random exploration and additional state-based hashing. Random policy is for reference

Evaluation. We compare the cumulative attained reward of a policy trained with sparse reward using random exploration and a policy using random exploration + state based hashing with a β exploration coefficient of 0.1 and granularity index $k = 10$ for a total of 1024 buckets. Figure 9.1 shows the average cumulative reward from 15 runs of each policy. The plot shows evidence that the state-based approach increases cumulative reward in the earlier stages but actually hurts performance further on. It is to be noted that each RL run has a high variance in terms of progress so 15 runs might not be statistically significant. Each run takes over 3 hours so it is difficult to get more data from a single PC. In addition, the runs were kept short, again due to computation time which means that we did not explore the full potential of the method here. Another potential source of the problem might by badly set hyperparameters, in this case the exploration and granularity coefficient and the layer which we use as the state embedding. Unfortunately, again due to computational reasons it is infeasible to perform a hyperparameter search in this case.

Chapter 8

Reward shaping

When defining the task for an agent to solve in an environment we usually have a goal state in mind, that is, a state or set of states that we want the agent to reach. The reward for the agent can then be defined as a value of 1 when the agent reaches a goal state, and 0 otherwise. Assuming a sufficient exploration and a discount factor of $\gamma < 1$, the agent will eventually learn a policy which in expectation minimizes the amount of steps taken to reach the goal. In this case, the only way that the agent can get any feedback at all, is to accidentally stumble upon the goal while exploring in an MDP which it knows nothing about. Once the agent reaches the goal then the actions that led to the goal will be reinforced and it will then reach the goal with a higher probability, with the probability of success increasing with every successful reaching of the goal. The problem is that in many environments, it can be practically impossible for an agent to reach the goal by a sequence of random actions. Figure 8.1 demonstrates an example of such an environment.

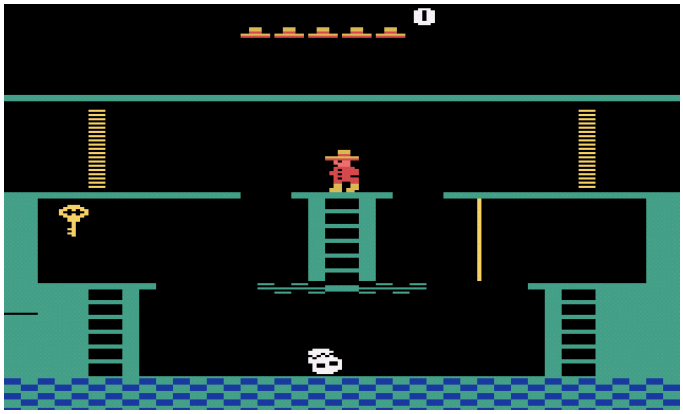


Figure 8.1: Montezumas revenge. A notoriously difficult environment for reinforcement learning due to the sparse delayed reward. To get any feedback the agent has to traverse the ladders to get the key and open the door while avoiding the skull and falling from heights.

In such cases we need to somehow guide the agent towards that goal by adding auxilliary rewards. This is called reward shaping. In essence, the designer of the environment uses their domain knowledge to assess what the agent should and should not be doing in order to achieve a certain goal and

implements this knowledge as a reward function. In the NIFTi robot case, we shaped the reward by adding a bonus when the robot successfully travelled in the direction of the goal which is a relatively common heuristic. We also do not want the robot using the actuators needlessly so a control effort penalty was added. Reward shaping directly affects policy, often leading to undesired results so it has to be done with care and sparingly. There are cases, however, where we would like to shape the behavior of the robot to suit our specific needs.

8.1 optimality with respect to specific criteria

We consider the locomotion task of a generic robot. There can be various criteria which we would like the locomotion task to respect. These criteria can include: Fastest time, lowest energy usage, most stable gait. We consider and optimize the NIFTi robot for the following criteria and compare results to the default locomotion policy that we used in our initial experiments with no specific criteria in mind.

Least steps. This is the most common criterium that we use to evaluate an agent in an environment. The reward that leads to this behavior can be binary where the robot gets a value of 1 for crossing the finish line and 0 otherwise. For this we require that the discount factor γ is strictly smaller than 1. The solution to the mdp will then simply be the shortest path to the goal. If $\gamma = 1$ then every state in the mdp that leads to the goal state will have a value of 1 eventually, meaning that every path would be the shortest path.

Smallest control effort. This criterium is useful for when energy savings are an important factor in the mission. Here the cost includes a very high penalty for control effort. The control effort is calculated by simply penalizing the norm of the action magnitudes, similarly to what is described in the reward section, but with a higher coefficient.

Smoothest ride. The robot might be carrying fragile cargo or might be itself fragile and it might be desirable to use a policy which leads to smooth locomotion behavior. The way we quantify smoothness is by analyzing high frequency changes in the IMU readings, in both translational and rotational axes. A simple way is to penalize a norm of the change between two consecutive readings weighed by a coefficient vector. More advanced methods could involve keeping a queue of values and applying a windowed FFT computation on the queue to obtain a clearer frequency criterium.

Level platform. We might require the body of the robot to maintain a level orientation if possible to obtain quality footage from an onboard camera. This can be easily implemented by simply penalizing any deviations of the robot body from the default position using IMU readings.

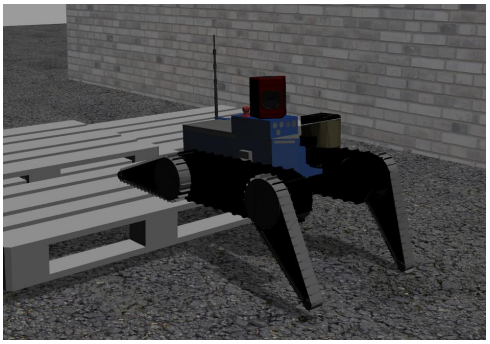


Figure 8.2: Screenshot of π_{level}, using flippers as support to stay level.

To evaluate policies trained on the above criteria we use our D_{25} environments which consist of 25 randomly generated palettes at various positions. Policies $\{\pi_{sparse}, \pi_{eff}, \pi_{smooth}, \pi_{level}\}$, correspond to policies trained using the four criteria described above in their respective order. Each policy is evaluated on D_{25} according to all four optimization criteria. The results are shown in table 8.1.

Comments on performance. We can see that the π_{sparse} has a slightly lower performance in terms of success rate and is worst off in all other regards which is expected given that we don't optimize for any of these. The π_{eff} policy has a mean actuation energy several times lower than the other policies, but at the cost of rough locomotion. One interesting observation made on π_{eff} is the difference in the way it performs the stage B maneuvre 10.5. Most policies use their back flippers to raise the robot forward, whereas π_{eff} uses the front flippers to shift the center of mass forward which is more energy efficient. Both π_{smooth} and π_{level} successfully minimize their own main criterium with respect to the other policies. One downside of the π_{smooth} policy is that since we optimized only for roll and pitch smoothness, the policy allows the robot drop vertically when running off a palette since this is not penalized. In hindsight, vertical acceleration should also have been part of the criterial function. for the π_{smooth} policy. Figure 8.2 shows a screenshot of the π_{level} using its flippers to support itself after rolling off a palette so that it stays level for as long as possible.

Policy	Finishlines passed	Mean actuation	Mean IMU activity	Mean angular deviation
π_{sparse}	18/25	5.39	0.038	0.063
π_{eff}	20/25	0.98	0.0414	0.066
π_{smooth}	20/25	2.17	0.011	0.0398
π_{level}	21/25	2.69	0.027	0.0344

Table 8.1: Comparison of policies trained according to various criteria, evaluated on the D_{25} environments. All policies are reactive using a front facing depth image, as well as roll, pitch and flipper angle data as inputs.

Chapter 9

Learning a recurrent policy

In part II we discussed how locomotion from static front-facing camera images is difficult due to the partial observability of the task. One suggested way around this issue was to use Recurrent Neural network cells as a compact representation of the history at every time step.

One simple way how to learn an RNN policy is to imitate a learned reactive policy in the simulator. This can be advantageous if the reactive policy is fully observable but has an impractical input requirement such as multiple observer images from various angles. In this case the RNN could simplify the input requirement by learning from such a policy and then enable deployment in the field from just a front facing camera.

Another method would be to adapt a reinforcement learning technique, for example, the DDPG algorithm to use RNNs. Using RNNs in reinforcement learning is significantly more difficult due to implementation reasons and unstable training mechanics which is one of the reasons why an alternative method was provided at the beginning of this chapter. The adapted policy update for recurrent networks was derived by Hess *et al* [26] to be:

$$\frac{\partial J(\theta)}{\partial \theta} = \mathbb{E}_{\tau}[\sum_{t} \gamma^{t-1} \frac{\partial Q^{\mu}(h_t, a)}{\partial a}\Big|_{a=\mu^{\theta}(h_t)} \frac{\partial \mu^{\theta}(h_t)}{\partial \theta}] \tag{9.1}$$

where h_t is the observation-action history. The DDPG algorithm has to be slightly modified to adjust for the above policy update; we now have to save and sample whole trajectories to and from the replay buffer. Both the actor and critic networks have recurrent layers in them. The architecture for the actor is the same as in chapter II, as shown in figure 6.12. The critic network is similar, with the main difference being that the action is added as an auxilliary input to one of the fully connected layers. The actual gradient update of both the actor and critic do not change as we are using autodiff software (Tensorflow) which propagates the gradient through time through the whole episode.

Training details. The input to the agent at each time step is again a front facing grayscale camera image, the current flipper angles and the roll and pitch from the internal IMU. Having already compared the effect of using various images at input, we only test the default grayscale image for this

algorithm. The action space is again $a \in R^4$. The learning rates are increased by half an order of magnitude compared to the reactive DDPG in the previous section. This is so that we get a decent gradient propagation through time. The training is done by sampling a small batch of whole episodes from the replay buffer at the end of each episode and training both actor and critic on the batch. We train the policy using the default shaped reward function with a distance motivation, timestep and control effort penalties as it provides the most stable and fastest training.

Results and discussion. A visual inspection shows that the RNN policy trained with RL performs better than its reactive counterparts. It has a 24/25 success rate on our D_{25} dataset. When trained using reward shaping it performs better in criteria such as smoothness probably due to the ability of the policy to keep important features in memory that it cannot see from the front facing image.

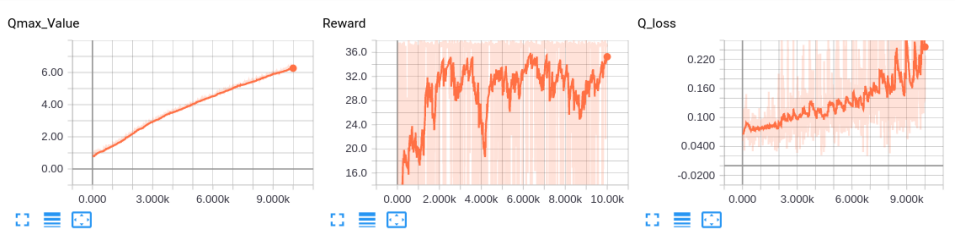


Figure 9.1: Screenshot from tensorboard showing training progress of DDPG with RNN policy. Max Q value shows the maximum Q value in the episode, Q_{loss} denotes the TD-error in that episode.

Part IV

Experiments in real environment

Chapter 10

Simulation to real robot transfer

So far we have learned locomotion policies in the simulator on a model of the NIFTi robot. The advantages of a simulator is that we can programmatically alter aspects of the environment, run the robot at faster than realtime speeds and perform an essentially limitless amount of runs without having to worry about mechanical wear, as we would get on the real robot. The disadvantage is that the model differs from the real robot, sometimes significantly. This means that transferring a policy learned on a simulator to the real robot might work well, in some cases with reduced performance, or completely fail in other cases. In the case of the NIFTi robot, we have an essentially kinematic system, but the physics of modelling the traction of the flipper tracks on various objects could be a weak point. Another issue might be a mismatch in the sensor input distribution, especially when using visual sensors such as cameras. Even though the simulator environment attempts to model the real environment, the visual appearance is significantly different than the real world.

Transferring learned policies from simulator to the real environment has been a very active area of research in the past few years. It has become an important topic since policies are being trained in an end to end fashion directly from pixel inputs. Several approaches have been proposed with varying success. Most of these approaches fall into the following three categories:

Finetuning. The finetuning approach is relatively simple and consists of retraining the already trained policy in the target environment. This is usually done carefully using smaller learning rates and typically retraining or reusing only a portion of the neural network. This works very well in many supervised learning cases in computer vision [67], [49]. A specific example is to use convolutional layers from a CNN trained on one domain such as ImageNet [18] and continue training them on a new domain. This assumes that the lower level feature detectors are general and reusable [67] and that we are finetuning to perform a different higher level task. In the case of simulation to real world transfer it is the other way around. We assume that our policy should remain the same at the higher level and only low level feature detectors should be slightly finetuned to recognise the new domain.

Using robust features. Another approach how to successfully transfer policies from a source to a target domain is to use robust features or to learn on a sufficiently diverse source domain in hope that the policy will be robust to changes and therefore generalize to the target domain. Robots have been trained in simulators with randomly varying parameters and have shown that this can vastly improve policy robustness. An effective approach in dealing with discrepancies in visual inputs is to use textural randomization during training in simulation to learn robust feature detectors in the convolutional neural networks [61]. Other methods such as adversarial feature learning have been proposed to learn more robust features [54].

Image modification. In many cases, the discrepancies of the dynamics of the system are not an issue, but the domain shift in the visual inputs is. We can denote the images that come from the source domain as $o_s \sim D_{source}$ and images from the target domain as $o_t \sim D_{target}$. It is possible to construct a function G which maps a source image to the corresponding image in the target domain such that $G(o_s) \sim D_{target}$. Such a function can be a convolutional neural network. One way that such a network can be trained is using GANs [13].

10.1 Setup description of experiments on real robot

To examine how our learned policy on the simulator generalizes to the physical NIFTi robot several tests are performed on various learned policies. Mostly we are interested in the following two topics.

- Does the physical model generalize?
- How do various different inputs generalize?

Used transfer techniques. In the previous chapter we mentioned 3 broad categories into which most transfer methods fall into. The simplest one was finetuning. We will not consider this as the robot is relatively slow and it would require a few hours of effort to gather several tens of trajectories on various obstacles. The second method, learning or using robust features is probably the simplest and most practical to implement. The third method would require the most work and algorithm tuning. We propose two different techniques that could be used for our task. We consider the following inputs and their transfer potential.

- Grayscale input image learned with random gamma shifts to account for various lighting conditions and texture intensities
- Preprocessing the grayscale image by a Roberts edge detector. The intuition of using an edge detector is that the image will look similar in the simulator and the real robot, minimizing the domain shift for the convolutional neural network.

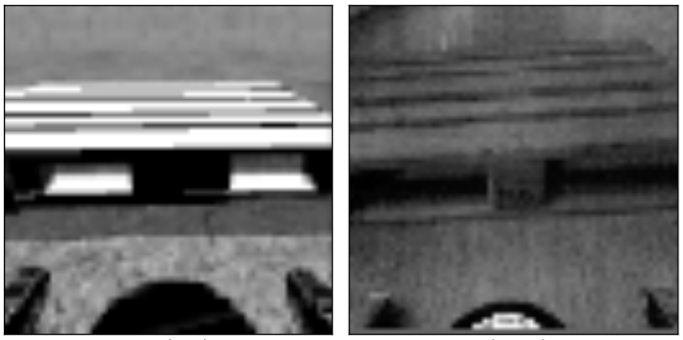


Figure 10.1: Comparison of front facing grayscale image from gazebo and R200 sensor at 64 × 64 pixels

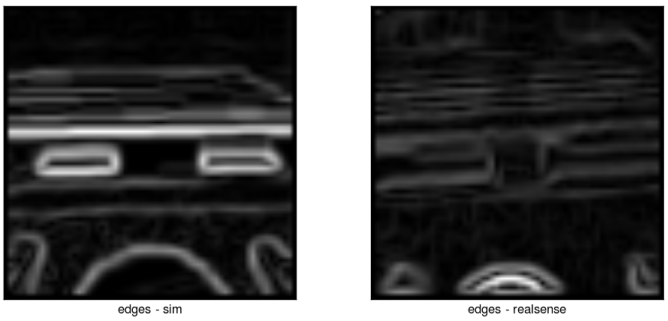


Figure 10.2: Comparison of front facing grayscale image preprocessed by edge detector from gazebo and R200 sensor at 64 × 64 pixels

- Using a depth image. The depth image is expected to generalize due to the image being content and texture agnostic, meaning that the depth image should look the same for a specific scene and a model of that scene in the simulator. In reality this is not the case as the real depth image will have significantly different characteristics, including heavy noise and reflectance on various materials. The depth image was preprocessed by clipping all the pixel values to 1.5 meters. All NaN values are set to the furthest 1.5 meter values.

Tested policies. We will be testing the following policies on the real robot. The actual method of learning for the policies in this case is not important as we are examining not the policy performance but the transfer performance. For this reason we simply use policies that were carefully learned using supervised methods from 50 trajectories that enable to navigate the real robot in a safe manner to avoid damage.

- RNN policy using default grayscale front facing image which was trained with randomized gamma shifts and pixel noise for robustness to lighting

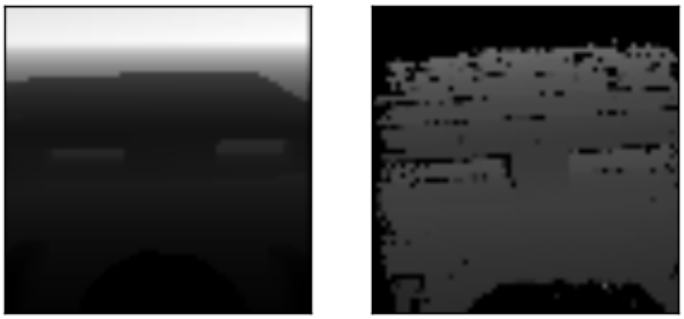


Figure 10.3: Comparison of front facing depth image from gazebo kinect model and R200 sensor. In this image the pixel values are taken from a screenshot and do not correspond to the preprocessed depth values used by the algorithm.

changes.

- RNN policy using default grayscale front facing image preprocessed by an edge detector.
- RNN policy using the depth image.

Real robot details. There are several important details that have to be addressed when performing the transfer from the simulator to the real robot to retain consistency of the model and to minimize perceptual difference for the policy.

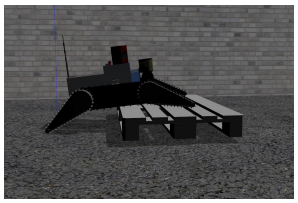
- Step size In the simulator we use a 200 ms step size. This is a rather large step size but since the NIFTi robot is essentially a kinematic system, this is not an issue. On the real robot this step size is emulated by a sleep delay of 200 ms in the control loop. The simulator model for the robot was modelled so that the velocity of the tracks and the flippers matches the real robot. When training the RNN policies in the simulator we random step sizes from 100 ms to 300 ms so that the policy stays temporally robust and does not fit to what it expects the simulator to produce. This should partially account for the change in physics between the simulator and the real robot.
- Physical robot parameters The real robot flipper torque is limited. A value for the torque is experimentally found on the real robot and adjusted in the simulator model so that the policy does not learn maneuvers in the simulator that it cannot perform on the real robot. The track torques are plentiful on the real robot so it is an insignificant parameter.
- Camera details In the simulator we use a model of the Kinect [20] RGBD camera to obtain grayscale and depth renders from the front facing camera. On the real robot a realsense R200 camera is used

which unfortunately has different parameters and characteristics. The parameters in the simulator Kinect model are adjusted as much as possible to match the R200. The horizontal field of view which is adjusted to 70 deg respectively. The simulator does not support changing vertical field of view. It is also important to match the position of the camera in the simulator and real robot. The R200 is mounted on the real robot using a 3D printed structure attached to the body of the robot. The transformation from the base link of the robot to the camera, including the camera mount is calculated and adjusted appropriately in the simulator.

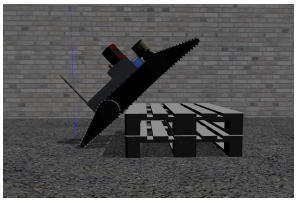
Hand tuning parameters. The camera angle was set to a smaller value that in the simulator because otherwise the robot did not recognize obstacles in front of it. This is most likely due to the discrepancy in position and camera characteristics between the simulator kinect model and the Realsense R200 on the robot. Both cameras differ significantly in their field of view angles which could not be adjusted in the simulator.

Results. The robot was tested on how well it can navigate over a real wooden palette which is approximately the same size as the models used in the simulator to train the policies. The robot is placed about half a meter from the palette and the user controls the velocity of the robot using a connected gamepad. The robot must configure its flippers appropriately so that it navigates the palette using only data from the front facing camera, roll & pitch data and the current flipper state angles. The robot successfully navigates the palette without the requirement of human intervention roughly half of the times. We evaluate the traversal of the palette, denoted in 4 stages described below.

- Stage A as the part before the robot has come close to the obstacle and has to configure the flippers as to start climbing the object. Ideally, the flippers should be more or less in a neutral position before it reaches the obstacle which shows that the robot is not falsely perceiving obstacles in front of it. Upon approaching the obstacle it should recognize the obstacle and raise one or both flippers to an appropriate angle. In our experiment the flippers were more or less neutral, with some movement before reaching the obstacle. The robot always recognizes the obstacle in front of it when approaching it. We also performed an experiment where the palette was placed in the way of only one of the flippers, and the result is that the robot successfully only lifts the required flipper to traverse the obstacle, meaning that it has a non-degenerate horizontal object detection resolution.



(a) : Correct flipper configuration



(b) : Typical failure case where the robot will abruptly tip forward and potentially cause damage upon impact

Figure 10.5: Stage B, ascent of the palette

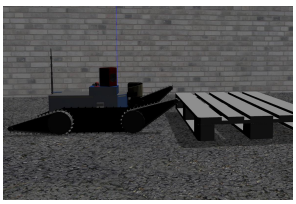


Figure 10.4: Stage A of the traversal, recognizing obstacle ahead

- Stage B as the part when the robot is climbing the palette and it needs to lower the front flippers to shift the center of gravity forward and lower the back flippers so that the back of the robot rises. This part is critical because if not done appropriately, the robot will ascend on an angle, and the tip over to the front abruptly after the half way point, leading to a relatively strong physical impact of the front flippers onto the top of the obstacle potentially damaging the robot. In our experiments this stage is sometimes problematic. The robot sometimes does not raise the rear flippers enough, which leads to a slight tip of the robot, requiring human intervention. This is most likely due to the discrepancy between the camera angle in the simulator and real robot. The policy learned in the simulator most likely prefers camera data to infer the pitch of the robot and therefore gets confused when the camera angle differs.

- Stage C as the part when the robot is on top of the palette and has to traverse it and lower the front flippers as it approaches the edge of the palette. In our experiments this is mostly successful with some unnecessary movement when on top of the palette but successfully lowers the flippers when reaching the edge.

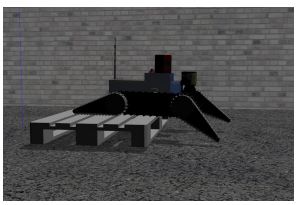


Figure 10.6: Stage C of the traversal, supporting with front flippers.

- Stage D as the part where the robot descends the palette and needs to raise both front and read flippers to ensure a smooth descent. On the real robot this works mostly, but if the robot is descended too slowly or is paused in the middle of the descent, the RNN policy "forgets" what is behind it, and straightens its flippers, leading to a non-smooth traversal.

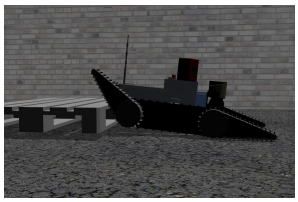


Figure 10.7: Stage D of the traversal, raising back flippers so that the robot doesn't fall abruptly as it rolls of the palette.

Comparison of various policies. All three policies seem to work and perform more or less similarly in terms of behavior. This is expected since they are the same policies, differing only in the type of input that they were trained on. It is difficult to tell which type of input is better due to the availability of only 1 palette in the vicinity of the robot and the lack of a diverse test environment.

Chapter 11

Suggested methods of improving transfer using image modification

11.1 Using a modifier network by matching GRAM matrices between old and new data distributions

Most methods that are used to transfer a policy from a simulated environment to a real one can be though of as *shallow* modifications because we are modifying information at the pixel level, without any change to the underlying semantic content. The rationale behind this assumption is that the simulator which we have at our disposition models the real world, which is our target and that the only difference between the two, visually, is at a textural level. This would mean that a relatively shallow modification of an image from the target domain could make it look like an image from the simulator. The concept of 'how deep' the modification has to be is very important when considering the architecture of the neural network modifier function G that we need to use. In terms of convolutional layers, we could conjecture that if our above rationale holds then at most two convolutional layers (with spatial pooling in between) can suffice. In any case, we will continue with the assumption that it holds.

Problem formulation. A policy $\pi(o_s)$ is usually trained on observations $o_s \sim D_s$ where D_s is the simulator domain. We instead train the policy on observations $G(o_s) \sim D_t$ which have been modified by the function G to resemble observations from the target domain D_t. In other words, the function G will modify the simulator images from the so that they look like real images in attempt to minimize distribution mismatch when deploying to the real environment. The difficulty is deciding on and training the function G.

Suggested algorithm. We assume that we have a dataset $U \sim D_t$ which are real images collected from the target environment which stylistically resemble images close to the environment that we will be deploying the robot on. At training time, we train two policy networks, each with their own convolutional

feature extractors C_{θ^A} and C_{θ^B}. Each network is parametrized and trained independently using a reinforcement learning algorithm. Network C_{θ^A} is meant to be trained on the unmodified simulator input image o_s and network C_{θ^B} is meant to be trained on the simulator image which is modified by network G_{θ^G}.

At each time step, the robot observes a simulator image o_s, and selects random image $i_t \in U$. The image o_s and modified image $G(o_s)$ is then used as inputs to networks C_{θ^A} and $C^1_{\theta^B}$ respectively. Image i_t is fed into a third network $C^2_{\theta^B}$ as shown in figure 11.1. We record the feature map activations in all 3 networks. We then want to update network G_{θ^G} such that the content of the feature maps in networks C_{θ^A} and $C^1_{\theta^B}$ match and such that the textural style between both networks $C^1_{\theta^B}$ and $C^2_{\theta^B}$ matches. This is done by defining and minimizing content loss function L_c and style loss function L_s. The gradient ascent update is shown in equation 11.3:

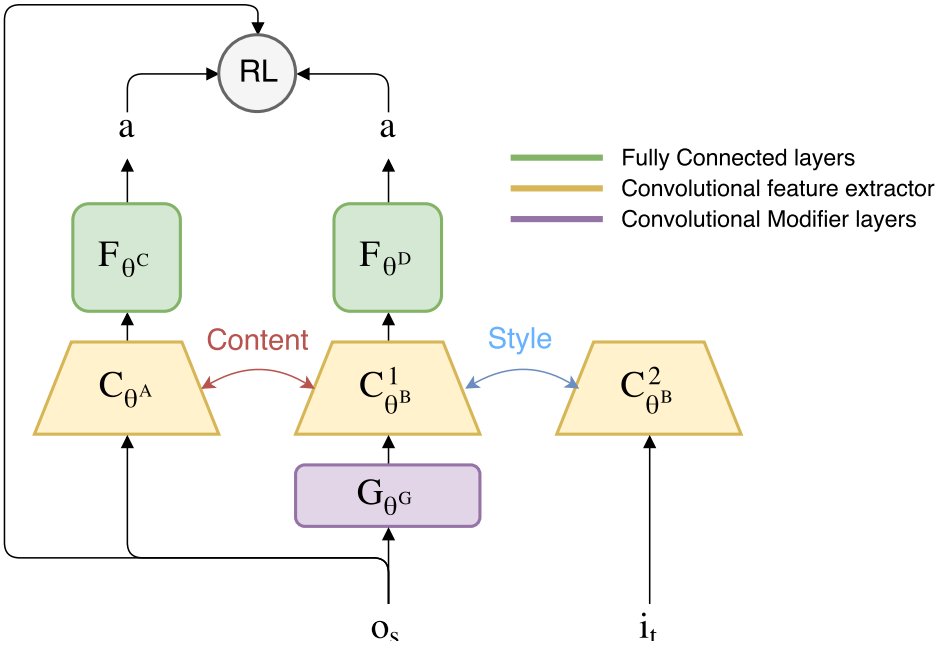


Figure 11.1: Diagram illustrating the network structure of the algorithm.

$$\theta_G \leftarrow \theta_G - \alpha \cdot \nabla_{\theta_G} L_s(o_s, i_t, \theta_G, \theta_B) - \beta \cdot \nabla_{\theta_G} L_c(o_s, \theta_A, \theta_B) \quad (11.1)$$

$$L_s(o_s, i_t, \theta_G, \theta_B) = \sum_{i \in \{1,2\}} \| gram(M^i_{C^1_{\theta^B}}(G(o_s))), gram(M^i_{C^2_{\theta^B}}(i_t)) \|_2 \quad (11.2)$$

$$L_c(o_s, \theta_A, \theta_B) = \sum_{i \in \{2,3\}} \| M^i_{C^1_{\theta^B}}(G(o_s)), M^i_{C_{\theta^A}}(o_s) \|_2 \quad (11.3)$$

Where $gram(M^i_Z(o_s))$ is the gram matrix of the i_{th} layer features maps M of network Z which are stretched out into a one dimension. This loss function is inspired by popular technique called *neural style* which attempts to transfer

the textural style from one image to another. We use only the first one or two layers of the network. The reason is that the higher we go, the more the features encode semantic content of the image, which is not what we want to transfer. The advantage of this technique is that we can see the results during training on the simulator and visually assess if the style transfer is working or not by looking at the modified images $G(o_s)$. This doesn't however guarantee that the policy will generalize to the target domain.

11.2 Using GAN to train modifier network

We look at one more approach on how to modify an image from one domain to look like it was sampled from another domain. In the previous section we approached the problem using a style transfer method which attempts to match gram style matrices from both images. A different and more popular approach is to use Generative Adversarial Networks (GAN) for this kind of task. Similarly as to the previous approach, we are looking to learn a modifier network G which modifies images from one domain A to look as if they came from domain B. In the context of GANs, the problem is formulated as follows: The generator $G(o_s, z)$ is conditioned on a random vector z and input image o_s. The discriminator randomly receives as input either an image from the generator $G(o_s, z)$ or real images from the target domain i_t. The task would then be for the generator to modify the input image in such a way so that the discriminator $D(X)$ cannot tell if it's being fed a real image or a modified image. This structure is illustrated in figure 11.2

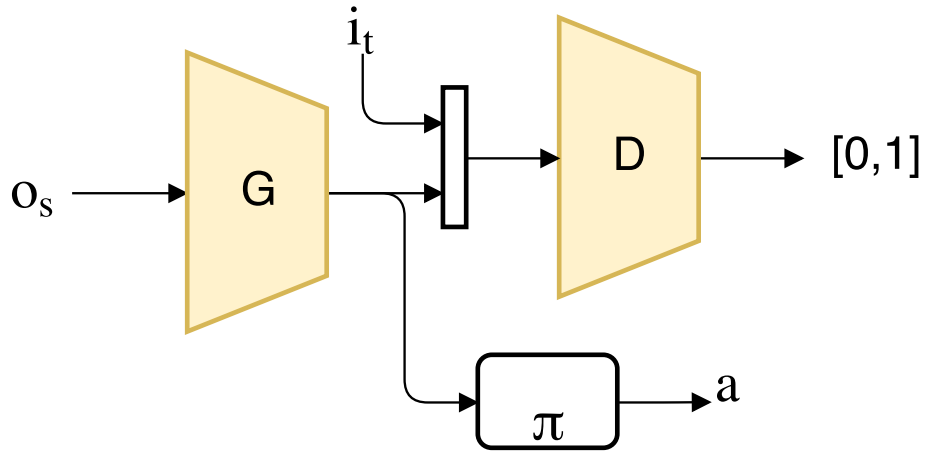


Figure 11.2: Using GANs to train a modifier network which changes the appearance of the simulator images to look like images from the target domain.

Such an approach was has been demonstrated to be effective [13] for some tasks. The impracticality with this method is that we do not have a good set of images that represent the target domain. This is due to the fact that the source environment that the policy is being trained on is very narrow in terms of content, and for this technique to work, we would have to gather a large

enough set of images which represents this environment in the real world. This is a difficult task and prone to failure since the discriminator will latch on any content discrepancies and judge the images in that manner which is not what we want. As in the style transfer approach, we can mitigate this issue by making the discriminator "weak", in the sense that it cannot understand image content, but only style. This means that the discriminator would not have access to the entire image that is provided to it, but rather to some statistic on that image which represents the textural style of the image, such as the gram matrix. It is however unclear on how to further process the gram matrix for the discriminator. Feeding it into a fully connected neural network means that we would have excessively many connections, which would lead to overfitting. In conclusion, unless there is a dataset from both the target and source domains which are large enough and similar content-wise, the previous suggested approach using gram matrix and content matching would probably work better.

Part V

Conclusion

11.3 Results comparison

In parts II and III we learn several different policies in the simulator using pixel inputs from an observer camera looking at the whole scene and from a front facing camera using various types of inputs. For reactive policies the depth image performed best, with up to 22/25 successful runs in our D_{25} test environments. RNN policies outperform reactive policies, especially when it comes to more difficult criteria which require memory. When comparing policies learned using reinforcement learning, RNNs are also superior to reactive policies. Our best trained RNN policy complete D_{25} with a 24/25 success rate while the best reactive achieves 22/25 and are visually superior. Typical failure cases are when the robot gets stuck in a position where it does not have traction between the obstacle and the flippers.

11.4 Computational performance of neural network policies

As mentioned in the motivation of this thesis, one of the reasons why being able to navigate from a single front facing RGB camera is advantageous is their very small size, availability and low price. This means that we could equip very small embedded robots with cameras as the only exteroceptive sensors and train a neural network as the policy of the robot. The computational time of a CNN or recurrent CNN policy such as the one shown figures 6.2 and 6.12 respectively, on a modern mid-tier desktop PC takes roughly 100-600 μs depending on the amount of convolutional filters and fully connected units used. The additional recurrent layers do not add significant computational cost.

We consider two different types of embedded systems. The first one featuring an ARM - M cortex microcontroller at 168Mhz including a hardware FPU running efficient C code with or without the overhead of an RTOS. If we don't consider SIMD instructions we can get a very rough clock-for-clock computation time requirement of 25-30x compared to a modern desktop $x86$ processor which would mean roughly 10-20ms, a rate that can be used realtime even for a relatively fast system such as a quadrotor. Implementing a specific neural network architecture in raw C code is relatively easy. The difficult part is implementing modularity and training which is not required for deployment. The second system that we will consider is a multi-core ARM A cortex system at several Ghz running a full OS. Such a system is outperformed by a modern desktop PC by only 3-10x so the computational time of a trained neural network policy would most likely be insignificant in this case.

11.5 Discussion and future work

The trained neural network policies works well in general as a means to configure the flippers in order to successfully traverse an obstacle both in the simulator and real robot. However, the robot was trained and tested on one type of obstacle in a relatively controlled setting. To be able to apply it to a real situation in indoor and outdoor navigation we would require to train the policy on a much more variable environment, something that resembles the sites that the robot is expected to travel in. In real world tasks we would also require the ability to navigate under various conditions such as outdoor daytime/night time, indoor light or dark conditions. This could theoretically be overcome by training several policies from various image types, such as RGB camera, Depth camera and IR camera which will cover all the conditions that we might face. In the real experiments we only considered a very small portion of potential options for policy transfer from the simulator to the real robot. One future direction in this regard would be to experiment with the suggested transfer algorithms for image modification to minimize the distribution shift between the simulated render and real sensor image inputs.

Another possible direction to improve this work would be to train the robot to control the track velocities and navigate from point A to point B using a goto(X,Y) form of input to the robot so that the operator only specifies coordinates where the robot should go and the robot should decide on how to get there. Something like this could be trained using a reinforcement learning algorithm called Hindsight experience replay [7] in which the agent is conditioned not only on current sensory inputs, but also on a given goal. The agent gets rewarded for getting to a goal successfully, but when it fails it does not get zero reward, but instead it saves that episode with the goal state which it achieved and receives a reward of 1. The agent then generalizes between attained goals and enables us to directly input a destination goal that we would like.

11.6 Conclusion

In conclusion, we showed several deep learning approaches for training a neural network locomotion policy for the NIFTi robot in a simulated environment. We showed that supervised methods can be a simple and effective way on how to learn a policy in some cases and that a small amount of training examples can suffice if done correctly. We demonstrated the GANs can be used as an alternative error proxy for learned from demonstrations and can be advantageous in the case where actions distributions are multimodal. We also showed that using reinforcement learning we could learn well performing policies in the simulator, with easily adjustable performance criteria such as control efficiency and locomotion smoothness. We compared the performance of reactive and recurrent policies and compared their performance, showing that the recurrent networks are more powerful for learning sequential tasks due

to their recurrent memory state. We also showed that using a convolutional neural network as a feature extractor is an effective method to reduce a high dimensional image to a lower dimensional embedding which can then be used as features for a policy. We showed that a policy learned even on a grayscale front facing image from the simulator transfers to a certain extent without any modification to the real robot. This further demonstrates that it is possible to learn policies in simulated environments and transfer them to a real robot. We also suggested and implemented a way on how to mitigate the discrepancy in physics between the simulator and real robot for RNN policies as well as two approaches on how to minimize the distribution shift between the simulation camera render and the real world camera sensor by using gram matrix style matching and Generative Adversarial Neural networks.

Appendices

Appendix A

A.1 Predicting large values with neural networks

In reinforcement learning cumulative reward values for a single episode can be up to several thousand in value. This cumulative reward is estimated by a value or Q function which is usually implemented by a neural network approximator in the case where we have high dimensional inputs for example. Neural networks are usually implemented with very small weight values and activation functions such as sigmoid and tanh which limit the output potentials to $[-1, 1]$. It is then not unreasonable to examine if the implemented neural network architecture can actually output values in the thousands because if it can't then the reinforcement algorithm will surely fail. An experiment was made where we force a neural network to fit to random samples from a gaussian process scaled by various coefficients. We found that it is possible to predict values ranging in tens of thousands if the neural network is correctly formed. Resultant weights of connections in all layers are roughly in the range $[-20, 20]$ after training on large values in the tens of thousands.

The following observation have been made which contribute to success or failure in training neural networks to predict very large values.

- If the neural network is shallow then many more units are required. Deeper network with less units is more stable. In general, the larger the values the more neurons required.
- Correct initialization matters, especially if the network is deep. Xavier [22] initialization works well.
- Inputs have to be scaled correctly, ideally $[-1, 1]$. Very high inputs (> 10) perform poorly or fail to train at all.
- Tanh activations should not be used when predicting large values due to saturation. Popular Relu units suffer neuron death due to high gradients. Leaky Relu [46] has been found to perform best due to their ability to recover.
- L2 weight decay value is not important as long as it is larger than 0.01

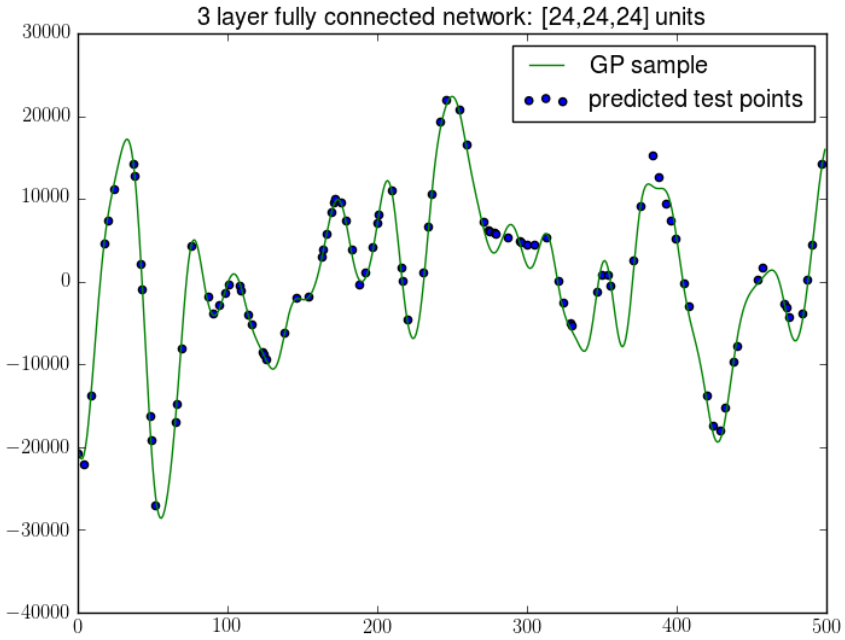


Figure A.1: Predicting large values with neural networks. The input is the index of the value divided by 100, the output is the predicted value.

Appendix B

Bibliography

[1] Nifti: Natural human-robot cooperation in dynamic environments. `http://www.nifti.eu`. Accessed: 2018-01-3.

[2] Tensorboard. `https://www.tensorflow.org/versions/r0.8/how_tos/summaries_and_tensorboard/index.html`. Accessed: 2016-05-12.

[3] Tensorflow api. `https://www.tensorflow.org/versions/r0.8/api_docs/index.html`. Accessed: 2016-05-09.

[4] Tradr: Long-term human-robot teaming for disaster response. `http://www.tradr-project.eu`. Accessed: 2018-01-3.

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

[6] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

[7] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight Experience Replay. *ArXiv e-prints*, July 2017.

[8] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó. The slam problem: A survey. In *Proceedings of the 2008 Conference on Artificial Intelligence Research and Development: Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*, pages 363–371, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[9] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. In I. Guyon, G. Dror, V. Lemaire, G. Taylor, and D. Silver, editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, volume 27 of *Proceedings of Machine Learning Research*, pages 37–49, Bellevue, Washington, USA, 02 Jul 2012. PMLR.

[10] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch. Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748, 2017.

[11] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. J. Goodfellow, A. Bergeron, N. Bouchard, and Y. Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[12] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars. *ArXiv e-prints*, Apr. 2016.

[13] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised Pixel-Level Domain Adaptation with Generative Adversarial Networks. *ArXiv e-prints*, Dec. 2016.

[14] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[15] Y. Cao, C. Shen, and H. T. Shen. Exploiting Depth From Single Monocular Images for Object Detection and Semantic Segmentation. *IEEE Transactions on Image Processing*, 26:836–846, Feb. 2017.

[16] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[17] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[19] J. L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[20] P. Fankhauser, M. Blosch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart. Kinect v2 for mobile robot navigation: Evaluation and modeling. *2015 International Conference on Advanced Robotics (ICAR)*, pages 388–394, 2015.

[21] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[22] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *JMLR W&CP: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 249–256, May 2010.

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

[24] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[25] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[26] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver. Memory-based control with recurrent neural networks. *CoRR*, abs/1512.04455, 2015.

[27] N. Heess, D. TB, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, S. M. A. Eslami, M. A. Riedmiller, and D. Silver. Emergence of locomotion behaviours in rich environments. *CoRR*, abs/1707.02286, 2017.

[28] J. Ho and S. Ermon. Generative Adversarial Imitation Learning. *ArXiv e-prints*, June 2016.

[29] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.

[30] X. Huang, J. Shan, and V. Vaidya. Lung nodule detection in ct using 3d convolutional neural networks. In *ISBI*, 2017.

[31] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[32] S. Ji, W. Xu, M. Yang, and K. Yu. 3d convolutional neural networks for human action recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(1):221–231, Jan. 2013.

[33] L. Keselman, J. I. Woodfill, A. Grunnet-Jepsen, and A. Bhowmik. Intel realsense stereoscopic depth cameras. *CoRR*, abs/1705.05548, 2017.

[34] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[35] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *ArXiv e-prints*, Dec. 2013.

[36] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, 2004.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[38] Y. Kuznietsov, J. Stückler, and B. Leibe. Semi-Supervised Deep Learning for Monocular Depth Map Prediction. *ArXiv e-prints*, Feb. 2017.

[39] M. Laskey, C. Chuck, J. Lee, J. Mahler, S. Krishnan, K. Jamieson, A. D. Dragan, and K. Y. Goldberg. Comparing human-centric and robot-centric sampling for robot deep learning from demonstrations. *CoRR*, abs/1610.00850, 2016.

[40] S. M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[41] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[42] Y. Lecun, E. Cosatto, J. Ben, U. Muller, and B. Flepp. Dave: Autonomous off-road vehicle control using end-to-end learning. Technical Report DARPA-IPTO Final Report, Courant Institute/CBLL, http://www.cs.nyu.edu/˜yann/research/dave/index.html, 2004.

[43] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. *CoRR*, abs/1609.04802, 2016.

[44] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[45] Z. C. Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[46] A. L. Maas, A. Y. Hannun, and A. Y. Ng. Rectifier nonlinearities improve neural network acoustic models. 2013.

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

[48] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[49] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382, 2014.

[50] S. Ross, G. J. Gordon, and J. A. Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *ArXiv e-prints*, Nov. 2010.

[51] A. Saxena, S. H. Chung, and A. Y. Ng. 3dd depth reconstruction from a single still image. *Int. J. Comput. Vision*, 76(1):53–69, Jan. 2008.

[52] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[53] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[54] J. Shen, Y. Qu, W. Zhang, and Y. Yu. Wasserstein Distance Guided Representation Learning for Domain Adaptation. *ArXiv e-prints*, July 2017.

[55] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[56] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[57] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[58] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[59] T. Taketomi, H. Uchiyama, and S. Ikeda. Visual slam algorithms: a survey from 2010 to 2016. *IPSJ Transactions on Computer Vision and Applications*, 9(1):16, Jun 2017.

[60] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. *CoRR*, abs/1611.04717, 2016.

[61] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *ArXiv e-prints*, Mar. 2017.

[62] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. Learning Spatiotemporal Features with 3D Convolutional Networks. *ArXiv e-prints*, Dec. 2014.

[63] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930.

[64] C. Vondrick, H. Pirsiavash, and A. Torralba. Generating videos with scene dynamics. *CoRR*, abs/1609.02612, 2016.

[65] P. J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. Reprinted in [**?**].

[66] J. Weston, S. Chopra, and A. Bordes. Memory networks. *CoRR*, abs/1410.3916, 2014.

[67] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.

[68] J. Zhang, L. Tai, J. Boedecker, W. Burgard, and M. Liu. Neural SLAM: Learning to Explore with External Memory. *ArXiv e-prints*, June 2017.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

DIPLOMA THESIS ASSIGNMENT

Student: Teymur Azayev

Study programme: Open Informatics
Specialisation: Artificial Intelligence

Title of Diploma Thesis: Deep learning for autonomous control of robot´s flippers in simulation

Guidelines:

1. Familiarize yourself with the available Gazebo model of the real Search&Rescue robot, see Figure 1.
2. Study state-of-the-art methods, codes and environments for deep reinforcement learning
3. Learn a policy network for autonomous control of robot?s flippers - four auxiliary independently articulated sub-tracks. Experiment and compare
4. reactive vs recurrent policies. The policy should use only minimal set of raw sensory measurements, i.e. no prone-to-failure data pre-processing such creating a 3D map of the environment should be used.
5. Evaluate proposed method in Gazebo simulator and discuss typical failure cases.
6. Try the policy learned in simulator on the real platform and discuss issues connected with model transfer. Discuss possibilities for efficient model transfer based on Generative Adversarial Networks

Bibliography/Sources:

[1] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end training of deep visuomotor policies JMLR, 2017, https://arxiv.org/pdf/1504.00702.pdf
[2] Ziyu Wang, Josh Merel, Scott Reed, Greg Wayne, Nando de Freitas, Nicolas Heess, Robust Imitation of Diverse Behaviors https://arxiv.org/pdf/1707.02747.pdf
[3] Lei Tai, and Giuseppe Paolo and Ming Liu2, Virtual-to-real Deep Reinforcement Learning: Continuous Control of Mobile Robots for Mapless Navigation https://arxiv.org/pdf/1703.00420.pdf
[4] Kostantinos Bousmalis, Nathan Silberman, David Dohan, Dumitru Erhan, Dilip Krishnan, Unsupervised Pixel?Level Domain Adaptation with Generative Adversarial Networks https://arxiv.org/pdf/1612.05424.pdf

Diploma Thesis Supervisor: doc. Ing. Karel Zimmermann, Ph.D.

Valid until the end of the winter semester of academic year 2018/2019

Z.S.