Learning an End-to-End Steering Model for an Unmanned Surface Vehicle

Inverse Optimal Control for end-to-end mapping of input sensor data to action parameters

Alexandra S. Vedeler

Project thesis for department of Technical Cybernetics
Submission date: December 2018
Supervisor: Kristin Y. Pettersen, NTNU
Co-supervisor: Narada Warakagoda, FFI

Problem Formulation

This task deals with the ability of an Unmanned Surface Vehicle (USV) to move autonomously without colliding with other static or moving objects found in the environment. The vehicle is endowed with this ability through several modules such as scene understanding, navigation, route planning, and the control system. Traditionally, these modules are designed and optimized separately and combined afterward. For example, scene understanding module is designed to detect interesting objects such as other boats, ships, and buoys etc. and the control system itself needs to convert this information into appropriate action parameters, such as steering angle, engine power or degree of braking. In this proposal, we advocate a tighter coupling among individual modules, where the system directly generates the action parameters based on the sensor data. This end-to-end approach allows one to make the overall system more compact, efficient and accurate. Deep learning is an excellent candidate to implement such an end-to-end mapping of the input sensor data onto the action parameters.

One well-known area of deep learning suitable for similar problems is deep reinforcement learning (DRL). However, DRL depends on a known reward function. But in the case of a USV, it is difficult to manually design a good reward function because at least in certain maneuverings the relevant reward structure is not clear. A solution to this problem is learning the reward function together with the control policy based on training data using a technique known as Inverse Reinforcement Learning (IRL).

The tasks of this project include a study of IRL algorithms, with a special focus on the Guided Cost Learning (Finn et al.; 2016a), and data collection from manual steering of the USV. The collected data will be used in the master thesis following this project.

Abstract

While deep Reinforcement Learning (RL) has produced some impressive results over the last years, the requirement of a manually crafted cost function hinders its use in many fields. Inverse Reinforcement Learning (IRL) aims to solve this by optimizing a cost function through the use of a set of expert demonstrations. In this project, we study the combination of RL and IRL in preparation for later use in learning an end-to-end steering model for an Unmanned Surface Vehicle (USV). We perform a literature study on deep RL and IRL with a special focus on Guided Cost Learning (GCL), which combine the two. As part of our study, we have developed an implementation that combines the Guided Policy Search with the cost learning of GCL in order to learn an end-to-end steering model through the use of expert demonstrations. We call our implementation reworked GCL (rGCL) and it has been tested in a simple 2D simulated environment. Though training using the rGCL implementation did not yield optimal behavior, tendencies towards the target state were noticeable. We thus conclude that the implementation shows promise if developed further. The process of implementing the GCL algorithm allowed us hands-on experience and a deeper look into its workings, both of which we deem valuable for future work. In addition to this, expert demonstrations have been gathered using an USV. The project will be continued in a following master thesis.

Contents

Abstract **iii**

1 Introduction **1**

1.1 Previous works 2
1.2 Contribution and Background 4
1.3 Objective and Scope 5
1.4 Abbreviations and Notation 5
1.5 Outline 5

2 Background and Previous Work **9**

2.1 Supervised and Unsupervised Learning 9
2.2 Neural Networks and Deep Learning 10
2.3 Reinforcement Learning 12
2.3.1 Marcov Decision Process 12
2.3.2 Optimal policy 13
2.3.3 Policy Search 15
2.3.4 Cost 16
2.4 Guided Policy Search 17
2.4.1 The outer and inner loop 18
2.4.2 KL 19
2.4.3 Challenges 20
2.5 Inverse Reinforcement Learning 20
2.5.1 The Inverse Reinforcement Learning concept 20
2.5.2 Maximum Entropy IRL 21
2.5.3 Deep Maximum Entropy IRL 22
2.6 Guided Cost Learning 23
2.6.1 The GCL Cost Optimization 24

2.6.2 The GCL Policy Optimization Step 25

3 Method **27**

3.1 Python & Tensorflow 27

3.2 Reworked GCL Implementation 28

3.2.1 Implementation overview 28

3.2.2 `irl_cost` module 29

3.2.3 `reworked_gps` module 31

3.2.4 `reworked_gcl` module 32

4 Results **35**

4.1 rGCL 35

4.1.1 Experiment setup 35

4.1.2 The training results 38

4.1.3 Possible improvements 40

4.1.4 From state to observation in rGCL 41

4.2 Expert data collection 41

5 Conclusions and future work **43**

References **45**

List of Tables

1.1 Summary of the abbreviations frequently used in this paper . . . 6
1.2 A summary of the notation frequently used in this paper 7

List of Figures

2.1 A Neural Network structure example where each colored dot represents a neuron in the network and the arrows the weighted connections between them. The input is passed to the network through the input layer, forwarded through a number of hidden layers, and finally, mapped to the output through the output layer. Input neurons are activated through sensors that are perceiving the environment, while other neurons are activated through weighted connections from previously activated neurons (Schmidhuber; 2015). In this example, the network consists of an input layer of size three, two hidden layers, both of size three and an output layer of size two, but these sizes may vary. 10

2.2 An illustration of the agent-environment interaction in Reinforcement Learning. The same illustration is valid when replacing reward with cost (negative reward). 12

2.3 An illustration of Guided Policy Search (GPS) and its parting into an inner and an outer loop. i represents the time step, j represents the initial state and θ represents the policy parameter. 17

2.4 An illustration of the GCL algorithm. i represents the time step, and ϕ represents the cost parameters. 23

3.1 A simplified illustration of the three modules, `reworked_gcl`, `reworked_gps` and `irl_cost`, that make up our reworked GCL program. 29

3.2 An illustration of the `irl_cost` module. The illustration depicts the class `GCL_cost_fc`, its functions and the neural network as well as the function `train_cost_fc`. Simplified pseudo code for `train_cost_fc` is provided. dx and du is the dimension of the sensor state, x, and the action parameter, u, respectively. Otherwise, notation is as in Section 2.6.1. The dashed arrow represents the reference to the class. 30
3.3 A simplified flowchart of rGCL, v1 (left) and v2 (right). The illustration was created using Draw.io. 33
3.4 A simplified illustration of the interface of the three modules `reworked_gcl`, `irl_cost` and `reworked_gps` for rGCL v1. The illustration depicts the flow between the modules during one iteration of the `reworked_gcl` main loop. The flow is slightly different for rGCL v2. 34

4.1 The two-joint arm in its initial position. The yellow outline represents the target position. 36
4.2 The Graphical User Interface (GUI) of the GPS part of the program. For each set of samples taken for the policy optimizing, the GUI presents the average cost, step-multiplier (based on KL-divergency) and entropy (the randomness of a distribution) in the top left corner. The mean cost is also expressed graphically in the top right corner. In the graph to the lower left, the linear-gaussian controller means and distribution is represented by the red crosses and lines respectively, while the sampled trajectories are represented by the green lines. 37
4.3 Crop out from the training GUI during run time with v2. Only the first two decimals of a number is shown, thus the printout may be misleading. E.g. the GPS code reveals that step is never at a complete zero, but rather at a minimum value. 39
4.4 Approximate illustration of obstacle avoidance during expert demonstration. The dark circle represents the obstacle in question. . . . 41

Chapter 1

Introduction

Traditionally, USV steering has come through the use of a set of several modules, each designed and optimized separately before they are combined into the full system. E.g. a scene understanding module may be designed to detect interesting objects such as other boats, which may be used by the navigation module to pinpoint its location relative to the USV. The guidance system then uses this information to generate new trajectories which in turn is delivered to and executed by the control system through the use of appropriate action parameters such as a change in rotor angle or a decrease in speed. While this approach has given good results it requires a high level of engineering, tuning, and knowledge in several disciplines, e.g. object detection from sensory input, knowledge of the system dynamics, path planning, and control system algorithms. The resulting system is also not optimized with respect to an overall objective function, but rather a collection of locally optimized subsystems.

An alternative approach advocates a tighter coupling between the sensor input and the actions taken by the vehicle. This project studies such an approach, often dubbed an end-to-end approach, where the system directly generates action parameters based on the sensory data, thus making the system overall more compact, efficient and accurate. Deep Neural Networks (NN) offer a high level of expressive power. They are able to handle highly nonlinear relationships between their input and output (Schmidhuber; 2015; Goodfellow et al.; 2016), an ability which is necessary in order to perform end-to-end steering in USVs. Systems that couple deep NNs with Reinforcement Learning (RL), have also proved to be able to learn complex tasks (Silver et al.; 2016; Mnih et al.; 2015; Martinsen and Lekkas; 2018;

Levine et al.; 2016; Cheng and Zhang; 2018; Kober et al.; 2013; Sutton and Barto; 2017). However, RL requires a known cost (or reward) function to guide its training. We argue that such a function can be highly difficult to craft manually for USV steering as certain maneuverings do not express a clearly weighted cost structure (n.d.). Because of this, combining RL with Inverse Reinforcement Learning (IRL), where the goal is to learn the cost function from a set of demonstrations, may be a preferable solution. Though there are some examples of varying degrees of use of RL in USVs (Martinsen and Lekkas; 2018; Cheng and Zhang; 2018), we have not been able to find examples of the use of IRL in USV steering.

In this project, we evaluate IRL and its combination with RL in order to gain insight into its usability for end-to-end training of an USV. As part of our evaluation, we have developed an implementation that combines the guided policy search approach Finn et al. (2016b) with the cost learning of Finn et al. (2016a) in order to learn an end-to-end steering model through the use of expert demonstrations. We call this implementation the reworked GCL (rGCL). In addition to this, we have collected a set of expert demonstrations of manual steering of an USV during object avoidance. The project will continue in a following master thesis, where rGCL will be further developed and the collected expert demonstrations will be used for training of an USV steering model that allows it to move autonomously without colliding with other static or moving objects.

1.1 Previous works

With the use of Neural Networks (NN), recent advancements in Reinforcement Learning (RL) have achieved impressive performances, rivaling, or even beating, that of a human. Examples lie in their use in playing Atari games (Mnih et al.; 2015) or in the program AlphaGo (Silver et al.; 2016), which famously beat the human world champion in the game of Go.

An example of deep learning in steering, is Bojarski et al. (2017), who mapped raw visual input to steering parameters of a car, quite successfully using their deep NN PilotNet. However, this treats steering more like a deep NN classification problem, a type of supervised learning. Deep NN demand large datasets in order to be trained well (Goodfellow et al.; 2016) and as a supervised learning approach, the data must be labeled before training. PilotNet trained on 6 hours worth of video

and sensor data from a human driving a car. Such demands make this approach problematic in the case of USV steering, where large amounts of data is hard to come by or generate. In addition, because a small mistake on the part of the policy will place the system into states that lie outside the distribution in the training data, supervised learning will in general not generate a policy with good long-horizon performance (Levine et al.; 2016).

Martinsen and Lekkas (2018) used a policy search RL approach, (specifically the Deep Deterministic Policy Gradient method,) to find the desired policy for straight-path following for an underactuated marine vessel exposed to unknown ocean currents. As described in the article, the approach is model-free, requiring no prior knowledge of the system it is assigned to control. Another example of RL in an USV is Cheng and Zhang (2018) who proposes a deep RL approach for obstacle avoidance. However, as RL approaches, these require a pre-made cost function.

There are some examples of the use of Inverse Reinforcement Learning (IRL) in steering. One such example is Wulfmeier et al. (2016), who used a Maximum Entropy-based (Ziebart et al.; 2008), non-linear IRL framework with Fully Convolutional NNs to represent the cost model underlying expert driving behavior. However, we found no examples of IRL in use in USVs, much less the use of both IRL and RL in the steering of USVs.

Finn et al. (2016a) were the first to combine the benefits of sample-based IRL under unknown dynamics with nonlinear cost representations, using the raw state of the system directly, without the need for manual feature engineering. Their approach, Guided Cost Learning (GCL), learns both the cost and the policy simultaneously. However, for complex skills, their cost function seemed to only express the task locally. They also observed that the agent had a higher success rate at its task when the cost and policy were learned together, than when the resulting cost function was re-used for slightly different set-ups of the task. Nevertheless, their results are promising and have thus been identified as a focus for our study.

An approach to learning a policy from demonstrations which has not been explicitly explored in this paper is through the use of Generative Adversarial Networks (GANs). GANs uses a second NN (D) to guide its general model towards producing patterns similar to those of the expert (Baram et al.; 2017). An example of such an approach lies in GAIL (Ho and Ermon; 2016), a model-free approach which draws an analogy between imitation learning and GANs. GAIL is able to extract

a policy by alternating between policy optimization and cost fitting, much like GCL. Though both are more sample efficient than earlier IRL approaches, the fully model-free approach of GAIL makes it less sample efficient in terms of environment interaction during training than the model-based approach of GCL. Baram et al. (2017) attempts to ease this issue with their approach MGAIL, a model-based version of GAIL.

While GAN based approaches have not been focused on in this report, it might be of interest in future work.

1.2 Contribution and Background

Our contribution is a study of IRL and its possible use in learning of an end-to-end steering model for an USV. We focus on the Guided Cost Learning (GCL) algorithm (Finn et al.; 2016a) and, as part of our study, we have created an implementation that combines Guided Policy Search of Levine et al. (2016) with the cost learning of GCL. To the best of our knowledge, no implementation of the GCL algorithm described in Finn et al. (2016a) is available online. We call our implementation reworked GCL (rGCL). In addition, we have collected data in the form of expert demonstrations, from an USV during manual object avoidance.

Our implementation builds upon the open source GPS codebase of Finn et al. (2016b), using their implementation of GPS from Levine et al. (2016), and merging it with our implementation of the IRL cost optimization from Finn et al. (2016a). The code is implemented in python, using tensorflow (*Tensorflow home page*; 2017) for Neural Network (NN) support. The IRL code we developed is a substantially altered version of the open source implementation of deep Maximum Entropy IRL from Lu (2017). All illustrations in this paper were created using Draw.io.

The USV data was collected in cooperation with the Norwegian Defence Research Establishment and the use of their USV, Odin. We would like to thank them for their cooperation. We also express gratitude towards Narada Warakagoda for his excellent supervision and help throughout the project.

1.3 Objective and Scope

There were mainly three objectives for this project:

- Perform a literature study on IRL, focusing on Guided Cost Learning (GCL) (Finn et al.; 2016a)
- Implement and run an IRL algorithm, specifically GCL
- Collect expert demonstrations using an USV

All of the above objectives have been accomplished and described in this paper. For the literature study, we felt the need to include Reinforcement Learning (RL) as IRL and RL are closely related and because the goal ultimately is to combine the two. This combination has been successfully done in GCL (Finn et al.; 2016a), which is why it has been given special attention.

The second objective was originally limited to running an already implemented version of GCL. However, as far as we know, there is no implementation of GCL as described in Finn et al. (2016a), available online. Thus, we extended the objective to include implementing the algorithm. The scope was limited to creating a barebone implementation and test it on a simple two joint arm in a 2D environment.

The expert demonstrations were collected on December 4th. A total of 66 demonstrations of manual obstacle avoidance with the USV, was collected.

The project is intended as a preparation for a master thesis.

1.4 Abbreviations and Notation

For convenience, we use a number of abbreviations in this paper. A list of the most notable abbreviations is included below in Table 1.1. In addition, we have included a table of notations that are frequently used in Table 1.2. We have chosen to use notation similar to that which is used in control theory.

1.5 Outline

The report is organized as follows. In Chapter 2, a review of background information and previous works is presented, explaining the theory and the recent

Abbreviation	Full text
USV	Unmanned Surface Vehicle
NN	Neural Network
RL	Reinforcement Learning
IRL	Inverse Reinforcement Learning
GPS	Guided Policy Search
GCL	Guided Cost Learning
rGCL	reworked Guided Cost Learning

Table 1.1: Summary of the abbreviations frequently used in this paper

works that this project is built upon. Chapter 3 explains the details of the rGCL implementation. The results and discussion of the testing is presented in Chapter 4. This chapter also describes the collection of USV data. Lastly, conclusions and future work is presented in Chapter 5.

Symbol	Definition	Example/detail
x_t	System state in a Markovian process at time step $t \in [0, T]$	E.g. a combination of sensor measurements radar, LiDAR, heading angle, etc.
u_t	control or action at time step $t \in [0, T]$	E.g. a combination of USV action parameters
o_t	Observation at time step $t \in [0, T]$	E.g. a combination of sensor measurements LiDAR, camera image, etc.
τ	Trajectory: $\tau = (x_0, u_0), (x_1, u_1), ...(x_t, u_t)$	The sequence of state-action pairs in an episode
$r(x_t, u_t)$	Reward function that defines the goal of the task, $r = -c$	When using reward, the optimization comes in the form of a max problem
$c_\phi(x_t, u_t)$	Cost function that defines the goal of the task $c = -r$. Parameterized by ϕ	When using reward, the optimization comes in the form of a min problem. May be parameterized as a Neural Network (NN), in which case ϕ denotes the network weights
$p(x_{t+1}\|x_t, u_t)$	Unknown system dynamics, the transition function from one state to another	The physics of the environment of the agent
π_θ	Policy, parameterized by θ	May be parameterized as a Neural Network (NN), in which case θ denotes the network weights
$p_i(u_t\|x_t)$	Learned local time-varying linear-Gaussian controller for initial state x_1^i	Used in Guided Policy Search
π_τ	Trajectory distribution $p(\tau) = p(x_t)\Pi_{t=1}p(x_{t+1}\|x_t, u_t)p(u_t\|x_t)$	Used in Guided Policy Search and in Guided cost learning

Table 1.2: A summary of the notation frequently used in this paper

Chapter 2

Background and Previous Work

In this chapter, some background on the fields of Reinforcement Learning (RL) and Inverse Reinforcement Learning (IRL) will be presented and relevant concepts will be explained. A special focus is placed on the Guided Cost Learning algorithm and work related to it.

2.1 Supervised and Unsupervised Learning

We here dedicate a section to defining supervised and unsupervised learning as it will be relevant in later articles. We also highlight its relationship with RL, which is discussed in Section 2.3.

As explained in (Goodfellow et al.; 2016), in supervised learning, training is performed on data sets, where each input example has a corresponding label or target, corresponding to the ground truth. As this label is known, the function may be trained by evaluating its ability to predict correct labels, similarly to a teacher showing its student what to do. In contrast, unsupervised learning concerns the case where the true labels of the examples are not known. Algorithms of this kind handle a dataset containing many features and learn useful properties of the structure of this dataset. While supervised learning can e.g. be used to create a function that distinguishes cats from dogs, unsupervised learning may be used to e.g. discover clusters among a data set, like correlations between age and social media usage. RL is usually considered to be a third machine learning paradigm, alongside supervised learning and unsupervised learning (Sutton and Barto; 2017).

According to (Levine et al.; 2016), supervised learning will in general not yield a policy with good long-horizon performance, because a small mistake on the part of the policy will place the system into states that lie outside the distribution in the training data, thus causing compounding errors. We refer to Goodfellow et al. (2016) for further reading about supervised and unsupervised learning.

2.2 Neural Networks and Deep Learning

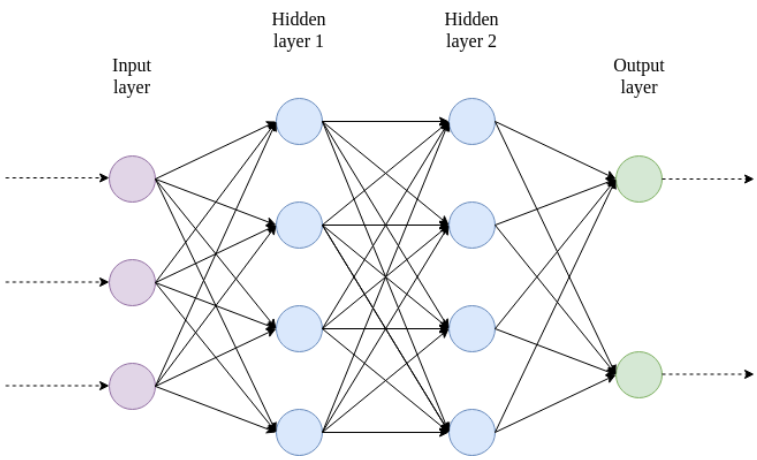

Figure 2.1: A Neural Network structure example where each colored dot represents a neuron in the network and the arrows the weighted connections between them. The input is passed to the network through the input layer, forwarded through a number of hidden layers, and finally, mapped to the output through the output layer. Input neurons are activated through sensors that are perceiving the environment, while other neurons are activated through weighted connections from previously activated neurons (Schmidhuber; 2015). In this example, the network consists of an input layer of size three, two hidden layers, both of size three and an output layer of size two, but these sizes may vary.

The following section summarizes some key points about Neural Networks (NN) and deep learning. This is relevant for us as deep NNs are responsible for some of the most impressive abilities of machine learning systems today, including Reinforcement Learning (RL) (Sutton and Barto; 2017). For deeper insight into NNs and their use in RL, we refer to Goodfellow et al. (2016) and Sutton and Barto (2017) respectively.

Neural networks (NN) are, in simple terms, a finite set of units (nodes or neurons) and a finite set of directed edges between these neurons (Schmidhuber; 2015). Each

neuron in the network takes in a certain number of inputs and calculates an output based on its input, and a specified activation function. The input each neuron receives comes from the weighted collection of outputs from previous neurons, allowing the NN to express complex and nonlinear functions. NNs thus map, or transform, their input to their output by passing it through the network of neurons. The number of neurons in a network may vary, and they are grouped together in layers as depicted in Figure 2.1. The term deep NNs is awarded NNs that have many such layers, as opposed to shallow NNs.

The weights of the neurons' inputs may be adjusted through cycles of training, in which a loss function determines the error of the NN output for each NN input and then adjusts the weights in the network. In the case of image classification, an example may be the task where a NN is to determine whether its input image depicts a cat or a dog. The image is passed through the NN and produces two probabilities at its output, the probability of the image depicting a cat and the probability of it depicting a dog. In training, a loss may be found by calculating the difference in the true values of the image, it being a cat or dog, and the suggested output. These losses are then back-propagated. This means the error of the prediction is passed backwards through the network, and the gradients of the error with respect to the weights of the network, are calculated. These gradients can then be used to update the weights of the NN. This process is repeated for a set of different inputs, images in this example.

NNs and deep learning, the process of adapting the weights of the deep NN to make it exhibit the desired behavior (Schmidhuber; 2015), have attracted widespread attention and recognition. As pointed out in Schmidhuber (2015), they have especially found success in tasks of visual aspects, like in the case of image classification, and have made great accomplishments in certain areas of reinforcement learning (Mnih et al.; 2015; Silver et al.; 2016; Sutton and Barto; 2017). In our case, the expressive power and compatibility with RL and IRL make it a good candidate for end-to-end learning of a steering model. A notable drawback of deep NNs, however, is their need for extensive training data in order to produce good results.

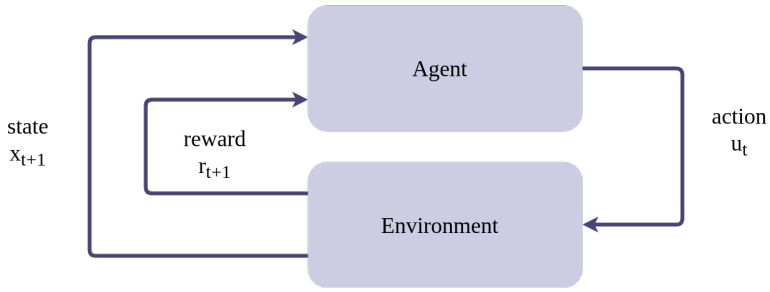

Figure 2.2: An illustration of the agent-environment interaction in Reinforcement Learning. The same illustration is valid when replacing reward with cost (negative reward).

2.3 Reinforcement Learning

As defined in Sutton and Barto (2017); "Reinforcement learning is learning what to do—how to map situations to actions — so as to maximize a numerical reward signal." At its core, the idea of Reinforcement Learning (RL) is to approach learning in the highly intuitive way of most organisms we know of, learning by trial and error. In RL the decision maker is called the agent. Everything outside of the agent is called the environment. The agent in learning is not told what to do in order to achieve a task. It is simply put into its environment, without prior knowledge of its workings. It must then discover the objective and the suitable approach by performing actions and observe the results that they cause. This feedback comes in the form of a numerical value from a given reward (or cost) function and, like a toddler, given encouragement and scolding, the agent will eventually learn the desired behavior. A simple illustration of the RL process is presented in Figure 2.2.

This section will present some important topics in Reinforcement Learning (RL). Further details can be found in Sutton and Barto (2017) and Gosavi (2009).

2.3.1 Marcov Decision Process

The process of the agent interacting with its environment and the resulting reward is, in reinforcement learning, formulated as a Marcov Decision Process (MDP), a tuple $< \mathcal{X}, \mathcal{U}, \mathcal{P}, \mathcal{R} >$. At each time step $t = 0, 1, 2, 3...$, the agent experiences some state of the environment, $x \in \mathcal{X}$, and must decide on some action, $u \in \mathcal{U}(x)$. The state is here defined as a unique characterization of the environment and the process of the task that the agent must solve. In practice, this is often expressed

by a collection of sensory input of the agent, such as e.g. the pose of a robotic arm combined with visual input (Finn et al.; 2016a). The actions can be used by the agent, to control the system state. In our case, the case of an USV, such an action could be to apply a change in the steering parameters, e.g. change the heading of the USV. The choice of action results in the agent transitioning into a new state, x', and receiving a scalar reward, $r \in \mathcal{R}$, as a consequence of this transition. The transition is modeled by a transition function $\mathcal{P}$ which defines the probability, $p(x'|x, u) \in [0, 1]$, of ending up in state x' after performing action u in state x. Meanwhile, $\mathcal{R}$, or $\mathcal{R}(x, u, x')$, represents the reward function. It denotes the reward of transitioning from one state, x, to another, x', with the use of action u. The resulting reward may be positive or negative and is used to guide the agent towards solving the task.

RL is not alone in relying on a set of states, actions, and an underlying model. Both in this regard and others, RL share similarities with classical optimal control (Kober et al.; 2013). However, as pointed out by Kober et al. (2013), a key difference is the fact that optimal control assumes knowledge about this model is available, while RL does not. RL learns a task without prior knowledge of the inner workings of the environment in which it resides. $\mathcal{P}$ and $\mathcal{R}$ are both unknown to the agent. Only the string of states and actions the repeated interactions produces, $x_0, u_0, r_0, x_1, u_1, r_1, x_2, \ldots$, is observable by the agent. RL's ability to learn without a given model makes it an appealing choice when it comes to our task of steering an USV, as the dynamics of its environment is challenging.

An assumption of MDPs is that the current state provides enough information in order to choose the optimal action. In other words, the past states can be ignored when deciding the action. While this is unproblematic in some cases, an argument could be made that this does not properly make use of the temporal information that may be needed to perform the task in an optimal way (Chi and Mu; 2017). A task can be episodic, meaning it has an ending state that stops the process, or non-episodic, in which case the task could continue indefinitely.

2.3.2 Optimal policy

A policy, π, is a function that maps states to actions, $\pi : \mathcal{X} \rightarrow \mathcal{U}$. When the agent acts in the environment, it does so based on the state to action mapping its current policy provides it with. The goal of RL is to learn a policy that solves a given task,

in an optimal way. In RL, this optimal policy, π^*, is learned by interacting with the environment and optimizing the policy of an agent such that it maximizes the reward it receives for its behavior. Many different RL techniques have been developed in order for an agent to learn the optimal policy.

The value function, $V^\pi(x)$, denotes the value of a state, x, under a policy, π. Specifically, $V^\pi(x)$ yields the expected reward when starting at x and then following the policy π thereafter. There are several ways of defining how future rewards should account for the current value of the state. E.g. one could count all rewards (infinite horizon), only the rewards of the closest time steps (finite horizon), or one could weigh the rewards so that those closest in time will weigh the most (discounted infinite horizon). Using the discounted infinite horizon reward, the value function is defined as in equation 2.1 and 2.2, where the latter highlights the recursive property of the function.

$$V^\pi(x) = E\Big\{\sum_{k=0}^{\inf} \gamma^k r_{t+k} | x_t = x\Big\} \tag{2.1}$$

$$= \sum_{x'} \mathcal{P}(x, \pi(x), x')(\mathcal{R}(x, u, x') + \gamma V^\pi(x')) \tag{2.2}$$

Here, $\gamma \in [0, 1]$ denotes the discount factor. The closer γ is to zero, the more the agent values immediate reward over future rewards. With $\gamma = 1$, the expression becomes an infinite horizon expression.

An optimal policy, π^*, is one that yields the highest returned value for all states, i.e. $V^{\pi^*}(x) \geq V^\pi(x), \forall x \in \mathcal{X}$. This leads to the famous Bellman optimality equation:

$$V^{\pi^*}(x) = \max_{u \in \mathcal{U}} \sum_{x' \in \mathcal{X}} \mathcal{P}(x, \pi(x), x')(\mathcal{R}(x, u, x') + \gamma V^{\pi^*}(x')) \tag{2.3}$$

Value iteration and policy iteration are two dynamic programming planning algorithms which aim to find the optimal policy using the Bellman equation. We call them planning algorithms because they use $\mathcal{P}$ and $\mathcal{R}$ in their calculations, functions that are not available in RL. In RL these planning algorithms are thus often part of model-based techniques.

Based on its approach to the lack of a given system model, an RL technique can

either be model-based or model-free. While both approaches use their sequential experiences with the environment, model-based methods use them to estimate the unknown model of the system. After having built this model, they then use planning techniques, such as value iteration or policy iteration (Gosavi; 2009), to find an optimal policy. In contrast, model-free techniques adhere to a true trial and error fashion, optimizing their policy directly from their experiences with the system without building such a model (Kober et al.; 2013). Though model-based methods require model estimation, they are often more sample efficient than their model-free counterparts. However, because of their reliance on a model, a good estimation of the model must be learned.

2.3.3 Policy Search

While the use of value functions to calculate the policy provide good results in some cases, it poses a set of challenges. One of these is the fact that a small change in the calculations of value functions may cause big changes in policies. From a safety point of view, this can be problematic in physical systems such as robotics or in our case, the use of an USV.

Policy search methods learn the policy function without estimating the value function. Instead, they aim to maximize the expected reward.

$$\pi^* = \max_{\pi} E[\mathcal{R}] \tag{2.4}$$

Denoting τ as the trajectory, the expected return for the policy parameter τ is thus defined.

$$J(\theta) = E_{p(\tau|\theta)}[\mathcal{R}(\tau)] = \int p(\tau|\theta)\mathcal{R}(\tau)d\tau \tag{2.5}$$

where $p(\tau|\theta) = p(x_1)\Pi_{t=1}^{T} p(x_{t+1}|x_t, u_t, \theta)\pi(u_t|x_t, \theta)$, is the probability density of observing trajectory τ under the policy parameter θ, and $E_{p(\tau|\theta)}$ represents the expectation over τ, drawn from $p(\tau|\theta)$. The optimal policy parameter, θ^* can thus be found by optimizing the returned $J(\theta)$.

$$\theta^* = \operatorname*{argmax}_{\theta} J(\theta) \tag{2.6}$$

Like other RL methods, Policy Search methods may be model-free, e.g. Williams (1992), or model-based, e.g. Levine et al. (2016).

2.3.4 Cost

We have until now been using the notation of reward as this is the most common notation in the field of RL. However, the reward can instead be annotated by its negative counterpart, cost, $c \in C$, where $C = -\mathcal{R}$. The objective of maximizing the reward is then transformed into that of minimizing the cost. Because the articles that influence our work the most (Finn et al.; 2016a; Levine et al.; 2016) use the notion of cost and cost function, we will in the rest of this paper do the same.

The cost function in RL, like the reward function, is constructed by the programmer. The function implicitly describes the task and thus guides the agent towards its solution, as an underlying motivation for the agent. Thus, because of the agent's reliance on the cost it perceives, a well-constructed cost function is important. However, the complexity of the cost function and its creation may vary. In some cases, as in the case of many games, the goal can be explicitly described by the use of such a cost function. In the game of Snake, the player moves a snake around on a board, eating apples and growing its body as it goes. The game is lost if the snake bites itself. Here, the goal is identifiable and may be described in a relatively straightforward fashion through a cost function. E.g. by giving a negative cost (a reward) if the snake eats an apple, but a positive cost if the snake bites itself. In other cases, for the programmer to create an appropriate cost function to motivate the task may be impractical, or close to impossible.

An example related to USV steering is the case of obstacle avoidance. It is clear that a high cost should be given if a collision occurs, but how much should be given for a near-collision? If we give a cost that is inversely proportional with the agent's distance to the object, is that to say that the agent should make large detours to keep as far away from all objects as possible? And should the quality of the agent's movements matter? These and more aspects, impact the learned policy, making the crafting of a good cost function important, but difficult.

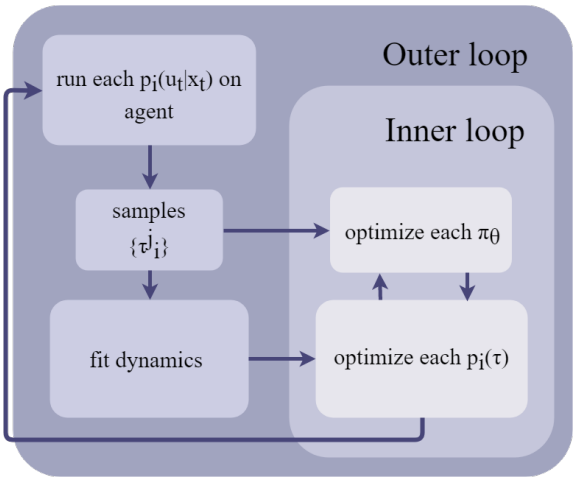

Figure 2.3: An illustration of Guided Policy Search (GPS) and its parting into an inner and an outer loop. i represents the time step, j represents the initial state and θ represents the policy parameter.

2.4 Guided Policy Search

Levine et al. (2016) presents a method used to learn policies that map raw image observations directly to torques at a robot's motors. A core component of their approach is a Guided Policy Search (GPS) algorithm, which separates the problem of learning a policy into two phases, a supervised learning phase, and a trajectory-centric reinforcement learning phase. Using this, and representing the policy through the use of a deep Convolutional Neural Network (CNN), Levine et al. (2016) were able to perform real-world manipulation tasks, such as screwing a cap onto a bottle. Using GPS, the agent learns a policy, $\pi_\theta(u_t|o_t)$, which determines its actions, u_t, based on the current observations, o_t, of the state, x_t. During training time, however, both the observations and the full state, x_t, is used to optimize its policy. At test time, only the observation based policy is employed, meaning the actions are predicted using only the observations. Other Guided Policy Search methods have been presented in Levine and Abbeel (2014) and Levine and Koltun (2013). However, the GPS method presented in Levine et al. (2016) is the only one the three that learns an observation based policy. When using the term Guided Policy Search (GPS), in this paper, we will be referring to the algorithm in Levine et al. (2016) unless otherwise stated. An implementation of the GPS algorithm is available as open-source (Finn et al.; 2016b).

In this section, $p_i(u_t|x_t)$ represents a learned local time-varying linear-Gaussian controller for the initial state x_1^i on the form $\mathcal{N}(\mathbf{K}_{ti}x_t + \mathbf{k}_{ti}, \mathbf{C}_{ti})$. $\pi_\theta(u_t|o_t)$ represents the learned nonlinear global policy, i.e. the policy which provides the action from the observation for all initial states. As in Section 2.3.3, $p(\tau|\theta) = p(x_1)\Pi_{t=1}^{T} p(x_{t+1}|x_t, \theta)\pi(u_t|x_t, u_t, \theta)$, represents the probability density of observing trajectory τ under the policy parameter θ. For simplicity, we will omit the notation of θ and refer to this density as $p(\tau)$.

2.4.1 The outer and inner loop

GPS uses a model-based RL approach, and can be separated into two loops, as depicted in Figure 2.3. In the outer loop, the agent uses its current controller in the environment, sampling the resulting trajectories, τ_i^j, for each initial condition. These trajectories are then used to estimate a dynamics model, a model for transitioning to a state given the previous state and action, $p_i(x_{t+1}|x_t, u_t)$. Both the trajectories and the fitted dynamics are then used in the inner loop of the method, in which a policy is optimized. The dynamics are modelled as linear-Gaussian and are defined as $p_i(x_{t+1}|x_t, u_t) = \mathcal{N}(f_{xt}x_t + f_{ut}u_t + f_{ct}, \mathbf{F}_t)$.

The inner loop of the GPS algorithm can be further divided into two components. The first of the two is a supervised learning algorithm which trains policies of the form $\pi_\theta(u_t|o_t) = \mathcal{N}(\mu^\pi(o_t), \Sigma^\pi(o_t))$. While Levine et al. (2016) point out that different nonlinear functions could be used to represent $\mu^\pi(o_t)$ and $\Sigma^\pi(o_t))$, they use a deep Convolutional NN for $\mu^\pi(o_t)$ and an observation-independent learned covariance for $\Sigma^\pi(o_t))$. The second component uses a trajectory-centric RL algorithm to generate local time-varying linear-Gaussian controllers, $p_i(u_t|x_t)$. $p_i(u_t|x_t)$ also doubles as guiding distributions, which serve as the supervisor for the first component. GPS thus generate its own supervision for the supervised learning of the policy, turning policy search into a supervised form of learning.

The policy, $\pi_\theta(u_t|o_t)$, trains by attempting to predict the actions along each trajectory using the observations, o_t, and not the full state, x_t. In addition, it learns a global policy, meaning it is trained to be used for all initial states, as opposed to the guiding distribution, $p_i(u_t|x_t)$, which learns one specific local controller for each initial state. Alternating between these two components' optimization can be framed as a Bregman ADMM (Wang and Banerjee; 2014) algorithm, in which the trajectory distributions and the policy converge to the same state distribution.

This means that the agent can learn a policy that captures the same behavior as the guiding distributions, the local controllers $p_i(u_t|x_t)$, using the observations instead of the state. Levine et al. (2016) performed experiments where a robot was to execute tasks such as screwing the cap on a bottle or hanging a hanger on a rack. In their experiments, they included the true position of the objects that were to be manipulated as part of the state, x_t, during training. These positions were not included in the observations, however. This meant that during testing, the agent had to discover these by the use of the visual data it received from the robot's camera. In the case of an USV, this would mean the USV can train with an assortment of sensors, e.g. a precise radar, LiDAR, video, and heading measures, while learning a policy that only relies on a subset of the sensors, e.g. only LiDAR and video. At test time, the excess sensors can be removed and the USV would be able to perform the task with only the remaining observations, i.e. relying only on LiDAR and video.

2.4.2 KL

Large changes in the controller risk large changes in the perceived dynamics. This is problematic as the new dynamics are estimated using trajectory samples from the distribution at the last iteration $\hat{p}(\tau)$. If the current distribution $p(\tau)$ is too different from $\hat{p}(\tau)$, this will not give a good estimate of the current dynamics $p(x_{t+1}|x_t, u_t)$, causing the optimization to diverge. In physical systems, too large changes in the controller have the added problem of the agent risking acting in a way that might cause harm to itself or its environment.

In order to avoid this, Levine et al. (2016) applies a KL-divergency constraint to the optimalization problem concerning p. They bound the change from $\hat{p}(\tau)$ to $p(\tau)$ by adding the constraint in Equation 2.7 to the optimization problem of p.

$$D_{KL}(p(\tau)||\hat{p}(\tau)) \leq \epsilon \tag{2.7}$$

Here, ϵ is the step size which binds the update, and $D_{KL}(p(\tau)||\hat{p}(\tau)) = E_{p(\tau)}\log p(\tau) - \log \hat{p}(\tau)$, denotes KL-divergency, the dissimilarity between the two distributions (Zhang; 2017). We can see from the available open source code, that the step size, ϵ, is chosen based on the difference in what the agent predicted it would receive in cost with its new improvements, and what it actually received in cost. In GPS, this

constrained optimization is performed in the inner loop.

2.4.3 Challenges

As pointed out by Levine et al. (2016), while the use of the whole state during training allows the GPS agent to learn the policy efficiently using a small number of samples, the same requirement may limit the number of tasks that can be learned. After all, the full state may be hard to acquire even during training. We also consider it another potential drawback of the GPS algorithm that it requires fixed length episodes. This may not be an issue in certain cases, like in tasks of episodic nature such as screwing a cap on a bottle, but in other cases, like that of playing a game of pong, it may be infeasible. Fortunately, the case of obstacle avoidance with a USV, it is likely the episodic nature of the training is unproblematic.

2.5 Inverse Reinforcement Learning

Reinforcement learning relies heavily upon the cost function it is provided with. However, not all behavior is easily described by such a function and in many cases, they are either unknown to the programmer, abstract, or overly complex. In addition, it is not always clear which features of the environment or sensor data, provide the best information for the agent to solve the task. An example is the case of driving a boat, in which a plethora of motivating factors lie behind the reasoning of a human driver, such as collision avoidance, the sea current, waves, etc. Evaluating these factors, generalize them, and hand-picking features to create a cost function is not a simple task for even the most experienced captain.

Inverse Reinforcement Learning (IRL) is a response to the issue of constructing a cost function. It is a relatively new field, in which the focus lies on the cost function, not the policy (Zhifei and Joo; 2012). In this section, we will take a look at IRL and some of the techniques that have evolved over the past decade.

2.5.1 The Inverse Reinforcement Learning concept

The agent is given a set of demonstrations, $\mathcal{D} = \tau_1, \tau_2, ..., \tau_n$, performed by an expert, where each demonstration consists of a set of state-action pair, $\tau_i =$

$(s_0, a_0), (s_1, a_1), ...(s_k, a_k)$, similarly to a driving instructor demonstrating correct behavior to their student. The goal is then to infer the motivation behind the demonstrations, the underlying goal of the task. In other words, the purpose of IRL is not to find the unknown policy of which the expert is performing, but the unknown cost function, C, that lies behind that policy. Unlike with RL, IRL has traditionally required the transition function of the environment to be known (Zhifei and Joo; 2012).

Because IRL attempts to understand what lies behind the behavior of an agent, it may be used to replicate behavior by e.g. using working RL methods with the learned cost function. This form of learning is known as imitation learning and works under the presupposition that the cost is the most concise and transferable definition of the task. (Ziebart et al.; 2008)

A challenge in IRL is that the task of finding a cost function that describes the observed behavior is underdefined. Several cost functions might describe the same behavior. For example, a red ball is moved to the left, and onto a green cloth. The motivation behind this action might not easily be discerned. Its movement could be motivated by some rule that red always should go on top of green, or it could be that the ball should not be in direct contact with the table, or it could be that the ball should always move to the left and the cloth was simply there. Although this is a simple example, it illustrates an important issue with learning from observations.

2.5.2 Maximum Entropy IRL

Maximum entropy IRL (Ziebart et al.; 2008) attempts to ease the issue of the underdefined problem in IRL. By making use of the principle of maximum entropy (Jaynes; 1957), they present a probabilistic approach. Intuitively, the algorithm assumes that the optimal policy is the most likely to appear in the dataset and the algorithm will favor the reward function that maximizes the likelihood of the observed expert distribution. This is done by formulating a probabilistic problem. With τ representing a trajectory, $\tau = \{s_1, a_1, ..., s_t, a_t, ..., s_T\}$ and $R_\phi(\tau)$ the hidden reward for the trajectory, $R_\phi(\tau) = \sum_t r(s_t, a_t)$ where ϕ denotes the reward weights.

$$p(\tau) = \frac{1}{Z} exp(R_\phi(\tau)) \tag{2.8}$$

Z is the partition function, $Z = \int exp(R_\phi(\tau))d\tau$ a normalization constant gained

by summing $exp(R(\phi))$ over all paths. A benefit of this is the implicit handling of the uncertainty and noise of the observed path, making the approach more robust.

The expression of the trajectory probability comes as a result of the assumption that trajectories with higher rewards will have exponentially higher probability of occurring in the set of trajectories than those yielding lower rewards. I.e. it expresses the wish to find a reward function such that if the expert were to use it, the likelihood of it receiving high rewards would be exponentially larger than it receiving low rewards for its demonstrations, maximizing the likelihood of the occurrence of the observed trajectories. The weights of the reward function can thus be found through maximizing over the set of demonstrations $\mathcal{D} : \{\tau_i\} \sim \pi^*$

$$\phi^* = \underset{\phi}{\operatorname{argmax}} \sum_{\tau \in \mathcal{D}} \log p_{r_\phi}(\tau) \tag{2.9}$$

The reward function is assumed to be linear. As such, the gradient of the log-likelihood function can be found by the use of dynamic programming.

Formulating or estimating a partition function can be challenging, especially in higher dimension problems or when the dynamics are unknown. Despite this, the Maximum Entropy approach has been heavily used and built upon in the IRL field. In its work on the distribution of trajectories, the algorithm not only provides a way to handle the issue of the underdefined problem but it also somewhat facilitates another issue of IRL. In contrast to the IRL approaches proceeding it, Ziebart et al. (2008) does not assume the expert observations are optimal.

2.5.3 Deep Maximum Entropy IRL

Deep Maximum Entropy IRL (DeepIRL) (Wulfmeier et al.; 2015), builds upon the Maximum Entropy IRL, using Neural Networks (NN) to approximate the reward function. In their paper, they advocate the use of NNs in IRL because their layered structure allows for a compact representation of highly nonlinear functions through composition and reuse of results from previous layers. In addition, they point out how NNs computational complexity allows them to scale well to problems with large state spaces and complex cost structures. NNs thus opens the door for end-to-end learning, where the cost may be learned from raw sensor data rather than from hand-crafted features.

Wulfmeier et al. (2015) suggests a Fully Convolutional Neural Network (FCNN) which calculates the cost of each state-action pair. This current cost function is then used to calculate a policy before the policy is propagated through the MDP. From this, the frequency of which each state is visited, can be calculated and compared to that of the demonstrations. Finally, the results can be backpropagated and the wights of the network can be updated before possibly re-iterating the process.

The paper indicated promising results that outperformed methods such as the Gaussian Process IRL of Levine et al. (2011) when it came to complex situations where higher order relationships between features were of large importance. However, like most IRL approaches, DeepIRL uses the transition function in its calculation of the cost and thus requires it to be known beforehand. Another limitation of Wulfmeier et al. (2015) is that they use dynamic programming that traverses over the whole state space to calculate the partition function. In large state spaces, this is highly problematic from a computational view.

2.6 Guided Cost Learning

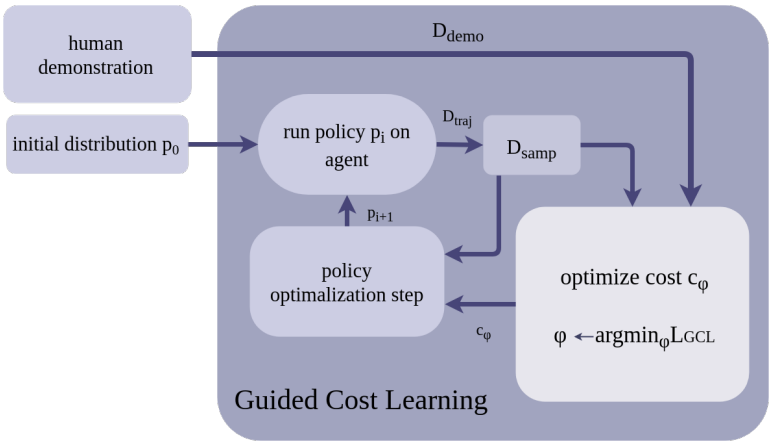

Figure 2.4: An illustration of the GCL algorithm. i represents the time step, and ϕ represents the cost parameters.

Motivated by the prospect of using IRL in fields where the dynamics are complex or unknown, such as in robotics, Finn et al. (2016a) presents the Guided Cost Learning algorithm (GCL). GCL uses an adaptive IRL approach that tests the current cost function and policy through training in the environment, samples this and uses

the samples to update its environmental model and policy. The cost function is then updated through a sample based approximation of Entropy IRL (Ziebart et al.; 2008) which, like Wulfmeier et al. (2015), expresses the cost function by the use of a Neural Network.

A regular challenge in IRL is how to evaluate the cost function. As pointed out in Zhifei and Joo (2012) and Finn et al. (2016a), a common approach is to find the optimal policy for the current cost function, a task which has been dubbed 'the forward problem' and compare the actions of this policy to that of the expert. The aforementioned authors note that standard IRL solutions solve this forward problem inside the cost optimizing loop. They argue that finding an optimal policy for a cost function can in itself be a complex and costly task that requires intricate knowledge of the dynamics of the system.

Thus, instead of finding a policy inside the loop searching for the cost, the approach of Finn et al. (2016a) is to find the cost in a loop searching for the policy. In other words, it constructs a cost function that can motivate the expert's behavior as a step in optimizing a policy that can imitate that same behavior. The policy is then tested in the environment and samples are collected. These samples are used to improve the cost function that gives rise to the policy. This way, knowledge of the system dynamics is not required as the agent will estimate them by itself when creating its current policy in the environment. This also grants it the benefit of building knowledge of the dynamics that is closer to the true local dynamics only for the areas where it is most important. GCL gets its name from the fact that the optimizing of the policy guides the sampling towards regions with lower cost.

A result of this approach is that it yields not only the cost but also a trajectory function, $q(\tau)$, that corresponds to a time-varying linear-Gaussian controller which can be used to execute the learned behavior.

2.6.1 The GCL Cost Optimization

In the GCL algorithm, the loop updating the cost function is located inside the loop that performs the policy search. GCL builds upon Ziebart et al. (2008) and assumes the demonstrated trajectories, τ_i, are drawn from the distribution

$$p(\tau) = \frac{1}{Z} exp(-c_\phi(\tau)) \tag{2.10}$$

where $c_\phi(\tau) = \sum_t c_\phi(x_t, u_t)$, the sum of the cost for each state, x_t, and action, u_t, pair that make up the trajectory. Because the partition function, $Z = \int exp(R_\psi(\tau))d\tau$, is unknown, it is estimated from samples of a background distribution, $q(\tau)$ which is estimated as part of the policy optimizing part of the GCL algorithm.

The GCL objective can then be expressed using the set of N demonstrations, $\mathcal{D}_{demo}$, and M samples, $\mathcal{D}_{samp}$ as denoted below (Finn et al.; 2016a)

$$\mathcal{L}_{GCL}(\phi) = \frac{1}{N} \sum_{\tau_i \in \mathcal{D}_{demo}} c_\phi(\tau_i) + \log Z \tag{2.11}$$

$$\approx \frac{1}{N} \sum_{\tau_i \in \mathcal{D}_{demo}} c_\phi(\tau_i) + \log \frac{1}{M} \sum_{\tau_j \in \mathcal{D}_{samp}} \frac{exp(-c_\phi(\tau_j)}{q(\tau_j)} \tag{2.12}$$

Finn et al. (2016a) then compute the corresponding gradients, with respect to the cost parameters, ϕ, and denoting $w_j = \frac{exp(-c_\phi(\tau_j)}{p(\tau_j)}$ and $Z = \sum_j w_j$

$$\frac{d\mathcal{L}_{GCL}}{d\phi} = \frac{1}{N} \sum_{\tau_i \in \mathcal{D}_{demo}} \frac{dc_\phi}{d\phi}(\tau_i) - \frac{1}{Z} \sum_{\tau_j \in \mathcal{D}_{samp}} w_j \frac{dc_\phi}{d\phi}(\tau_j) \tag{2.13}$$

The parameters of the cost function, ϕ, can thus be updated iteratively, using this gradient. As the cost function is represented by a NN, Finn et al. (2016a) back propagate $-\frac{w_j}{Z}$ for each trajectory $\tau_j \in \mathcal{D}_{samp}$ and $\frac{1}{N}$ for each trajectory $\tau_i \in \mathcal{D}_{demo}$ and update the weights of the NN based on the resulting gradients.

2.6.2 The GCL Policy Optimization Step

GCL optimize both a cost function and a policy. As part of the algorithm, a policy is optimized as a step in estimating the background distribution $p(\tau)$ used in the cost optimization. As described in Finn et al. (2016a), this step in the GCL algorithm uses the method of Levine and Abbeel (2014). Their method is highly similar to the GPS algorithm described in Section 2.4. Similarly to the GPS approach described in Levine et al. (2016), the policy optimization procedure consists of iteratively generating samples from $p(u_t|x_t)$, fitting the dynamics, $p(x_{t+1}|x_t, u_t)$, to these samples, and updating $p(u_t|x_t)$ under these fitted dynamics. One difference between the method of Levine and Abbeel (2014) and Levine et al. (2016) is that

the latter offers the simultaneous learning of a global, observation-based policy π_θ.

Chapter 3

Method

A main point of this project was to study the combination of cost function generation through Inverse Reinforcement Learning (IRL), and policy optimization through deep Reinforcement Learning (RL). Through our literature study in Section 2, we found the approach and results of Finn et al. (2016a) Guided Cost Learning (GCL) promising and, as a part of our research, decided to examine their approach more deeply through implementing a version of the GCL algorithm.

To avoid confusion between our version of GCL and the method described in Finn et al. (2016a), we refer to our as reworked GCL (rGCL). For the same reason, we also refer to our altered version of the GPS implementation (Finn et al.; 2016b) as reworked GPS (rGPS).

3.1 Python & Tensorflow

Our implementation builds upon Finn et al. (2016b), which is written in python. For this reason, the continuation was kept in the same language. Python is an easy and highly popular language that has an active community and a wide range of compatible open source libraries and implementations.

Tensorflow (*Tensorflow home page*; 2017) is an open source software library, originally developed by Google for high-performance numerical computing such as in the use of machine learning techniques like Neural Networks (NN). It is usable through python and allows the user to construct a graph of data, in the form of

tensors, and operations in a high-level python abstraction. A tensor is a symbolic handle to the output of an tensorflow operation, and the passing of tensors as input to other operations builds data flow connections. These graph structures define the steps of a multi-step computation such as e.g. an NN. Though tensorflow allows the user to build this structure using the Python language, the computations themselves are calculated using high-performance C++. This way the user is able to enjoy the high-level abstraction of python and tensorflow while still keeping the benefits of highly optimized code. Tensorflow is able to use GPU for computation.

3.2 Reworked GCL Implementation

To the best of our knowledge, no open-source version of the GCL implementation of Finn et al. (2016a) is currently available. Thus, in order to explore the GCL method and its potential for use in USV steering, we implemented one as part of the project. This was done by expanding on the open source GPS implementation from Finn et al. (2016b) and combining it with IRL cost learning as described in Finn et al. (2016a). Because GCL uses a method similar to Deep Maximum Entropy IRL (Wulfmeier et al.; 2015), an open source implementation (Lu; 2017) of Deep Maximum Entropy was chosen as a base for the GCL cost optimization. This code was then significantly altered into the IRL algorithm of GCL. The resulting IRL module was then combined with a slightly altered version of the GPS implementation to create reworked GCL (rGCL). An explanation of the rGCL implementation is given below.

3.2.1 Implementation overview

GCL uses a method described by Levine and Abbeel (2014) to calculate the guiding distribution and the samples used in the GCL cost optimization. Levine and Abbeel (2014) uses a GPS method similar to Levine et al. (2016), both optimizing linear gaussian distributions $p(\tau) = p(x_t)\Pi_{t=1}p(x_{t+1}|x_t, u_t)p(u_t|x_t)$. These Gaussian distributions, local for the initial conditions, are what generates the samples used in GCL cost optimization and in the calculation of the optimization gradients. Thus, we chose to use the GPS implementation available open-source through Finn et al. (2016b) in order to calculate these necessary distributions. We chose to adapt the already existing code rather than creating a full GCL implementation

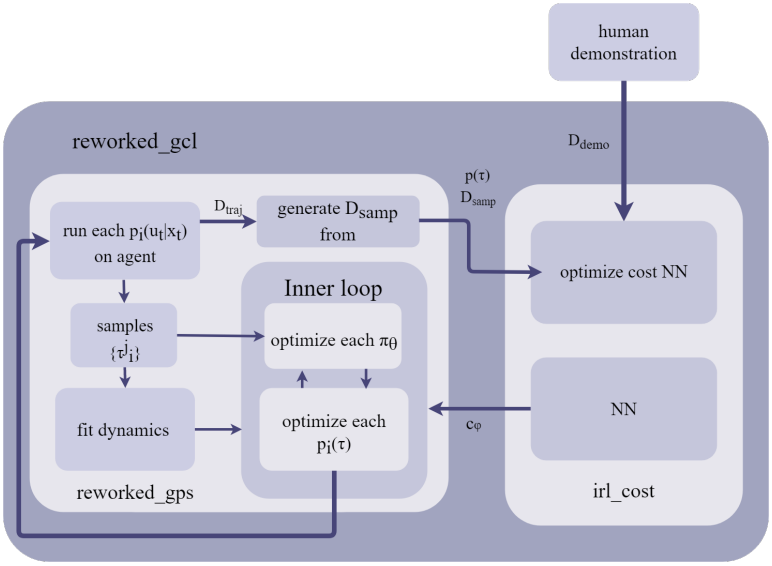

Figure 3.1: A simplified illustration of the three modules, `reworked_gcl`, `reworked_gps` and `irl_cost`, that make up our reworked GCL program.

from scratch due to the complexity of the task and the amount of time available to us.

rGCL combines the open-source GPS implementation (Finn et al.; 2016b) with the cost learning described in Finn et al. (2016a), using a Neural Network (NN) to express the cost function. In our description, we focus mostly on the tree major modules: `reworked_gcl`, `reworked_gps`, and `irl_cost`.

As indicated in Figure 3.1, rGCL alternates between running rGPS in module `reworked_gps` and the cost optimization in module `irl_cost`. `reworked_gps` executes GPS as described in section 2.4, and iteratively optimizes a policy to the current cost function by running the controller in the environment. `reworked_gps` is also responsible for generating sample trajectories to be used for cost optimization in `irl_cost`.

3.2.2 `irl_cost` module

The `irl_cost` module contains one class, and one function. The class, `GCL_cost_fc`, holds the cost function that is used for optimization of the policy in `reworked_gcl`. When initialized, `GCL_cost_fc` builds a NN of two layers using tensorflow. Each layer of the network has a number of neurons specified by the caller, `reworked_gcl`,

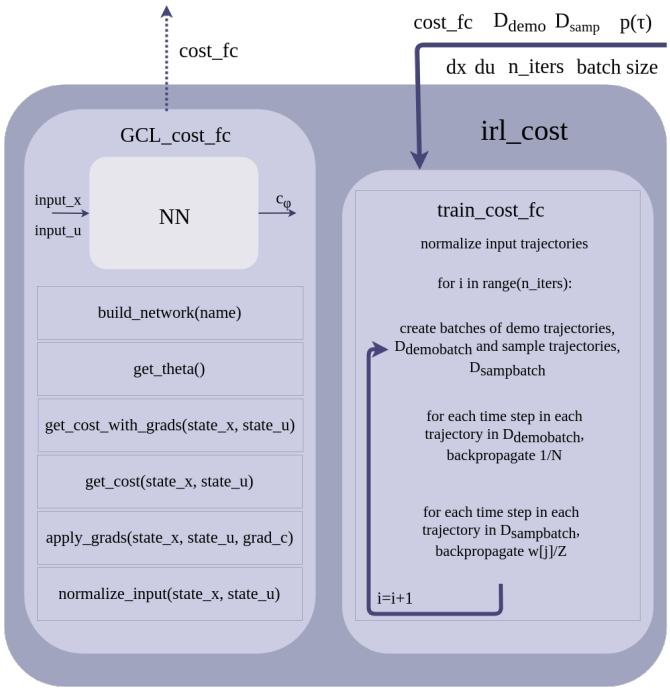

Figure 3.2: An illustration of the `irl_cost` module. The illustration depicts the class `GCL_cost_fc`, its functions and the neural network as well as the function `train_cost_fc`. Simplified pseudo code for `train_cost_fc` is provided. dx and du is the dimension of the sensor state, x, and the action parameter, u, respectively. Otherwise, notation is as in Section 2.6.1. The dashed arrow represents the reference to the class.

and the network maps two input arrays, `input_x` and `input_u` to a scalar output, `cost`. The class contains functions for accessing the network and for adjusting its weights. Outside of the class, is a function training the cost function. See Figure 3.2 for a visualization of the module.

The building of the individual layers of the NN and the class functions `apply_grads()` and `get_theta` remains the same as in the Deep Maximum Entropy IRL implementation Lu (2017) we started out with. `apply_grads` applies back-propagation and weight adjustment to the network and `get_theta` grants access to the network weights, e.g. for printing. All other code in `irl_cost`, has been created by us or changed beyond recognition.

`get_cost` and `get_cost_with_grads`, both passes the input state and action through the network to calculate the corresponding cost. However, `get_cost_with_grads` returns more than just the cost, c. This second function also calculates the

gradients of the cost with respect to the input, specifically c_x, c_u, c_{xx}, c_{uu} and c_{ux} (using the notation of this report), and returns it together with the cost. These gradients are needed by the `reworked_gps` module during policy optimizing. `get_cost_with_grads` is thus called from the `reworked_gps` module, while `get_cost` is called during training, from `train_cost_fc`. In addition to the aforementioned class functions, which are all directly related to the NN, an additional function was added to the class, `normalize_input`. This function normalizes the state-action pair before they are fed to the neural network and was created to facilitate learning, motivated by the fact that fixing the input to zero mean and unit variance is known to improve training speed (Ioffe and Szegedy; 2015). Given a deeper NN than we used for our testing, the extension to Batch Normalization (Ioffe and Szegedy; 2015) might be a good idea.

3.2.3 `reworked_gps` module

The `reworked_gps` module performs GPS as described in Section 2.4, optimizing a global policy $\pi_\theta(u_t|o_t)$ by simultaneously optimizing local Gaussian distributions, or linear-Gaussian controllers $p(u_t|x_t)$, for each state, under linear-Gaussian estimates of the dynamics $p(x_{t+1}|x_t, u_t)$. The optimization of $p(u_t|x_t)$, is performed by optimizing $p(\tau) = p(x_t)\Pi_{t=1}p(x_{t+1}|x_t, u_t)p(u_t|x_t)$ with respect to the expected cost and it is this trajectory distribution, $p(\tau)$, which is needed to perform the cost optimization step in GCL.

In order to optimize the cost function, `irl_cost` needs both the trajectory distribution, $p(\tau)$, and a set of sample trajectories generated from it. As illustrated in Figure 3.1, these are both calculated in `reworked_gps`. As we could not find any explicit calculation of $p(\tau)$ in the GCL implementation of Finn et al. (2016b), both the calculation of $p(\tau)$ and the sample trajectories were additions created by us.

The sample trajectories that are fed to cost optimization are generated by the use of the estimated dynamics and the Gaussian controller mentioned in Section 2.4. For each timestep in a trajectory, we have

$$u_t = \mathbf{K}_t x_t + \mathbf{k}_t + Cholesky(\mathbf{C}_t)v_t \tag{3.1}$$

$$x_{t+1} = f_{xt}x_t + f_{ut}u_t + f_{ct} + Cholesky(\mathbf{F}_t)w_t \tag{3.2}$$

where v_t and w_t are white noise variables, i.e. drawn from a Gaussian distribution of zero mean and unit variance, and are assumptions of the Gaussian models. The generated sample trajectories are collected in a list and sendt to `irl_cost`. $Cholesky()$ marks the Cholesky decomposition of the matrix. (Hofert; 2013)

In our program, $p(\tau)$ takes the form of a list of probability densities. Each entry in the list is the density of the corresponding entry in the list of trajectories. The probability density of each sample trajectory is calculated using

$$p(\tau_i) = pdf(x_0)\Pi_{t=1}^{T} pdf(u_{t-1}|x_{t-1}) pdf(x_t|x_{t-1}, u_{t-1}) \tag{3.3}$$

which is the same formula presented in Section 2.3.3 and 2.4, but with, what we feel, is a more intuitive notation. The notation $pdf()$ denotes, the probability density function (PDF) of the distribution governing its variables. Thus, $pdf(u_{t-1}|x_{t-1})$ denotes the the probability density of u_{t-1} using the PDF of the Gaussian distribution which parameterizes $p(u_{t-1}|x_{t-1})$. τ_i in Equation 3.3, denotes generated trajectory sample number i.

Because of problems with numerical underflow, the probability calculation, and following use of it in cost optimizing, was performed in the logarithmic domain.

During optimization of the policy, `reworked_gps` uses the cost function, the NN, of `irl_cost`. To achieve this, we added an interface between the two modules by referencing to the `GCL_cost_fc` in the inner loop of the GCL algorithm and a corresponding function to format the returned cost according to the module's needs. As portrayed in Figure 3.4, `reworked_gps` calls for the cost and its gradients from `irl_cost` multiple times, once for each timestep in each trajectory. In addition to this, it calls to `irl_cost` to normalize its state-action pair before feeding it to the NN, but this relationship was omitted in Figure 3.4 for the sake of simplicity.

3.2.4 `reworked_gcl` module

`reworked_gcl` combines the IRL of `irl_cost` and the RL of `reworked_gps` into one system. It is not especially clear from Finn et al. (2016a) how the alternation between the cost optimization and the policy optimization is performed. We identified and implemented two possibilities, v1 and v2, illustrated in Figure 3.3. In one loop of rGCL, v1 completes (possibly) several iterations of `reworked_gps` before

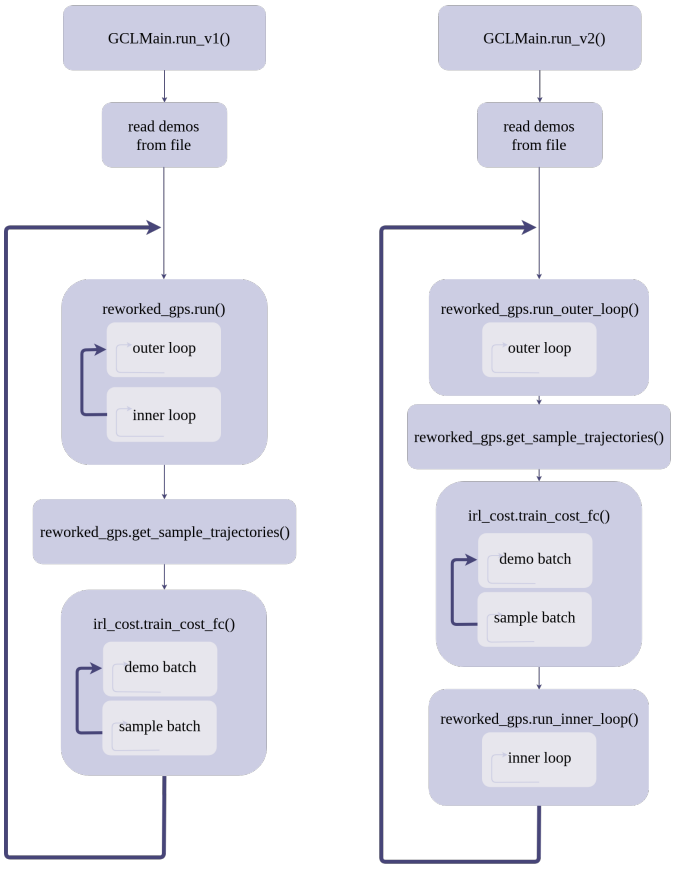

Figure 3.3: A simplified flowchart of rGCL, v1 (left) and v2 (right). The illustration was created using Draw.io.

updating the cost in `irl_cost`. Meanwhile, one iteration of rGCL v2 performs only the outer loop of the GPS algorithm, i.e. only running the current controller in the environment and estimating the dynamics, before the cost function is optimized. Afterward, the inner loop of the GPS algorithm, the actual optimization of the controller and policy, is performed. Thus, one iteration of v2 is shorter than one iteration of v1 as it contains one less nested loop. Implementing v2 required a larger change in the GPS interface than that of v1.

In GPS, the calculation of the step size (from KL-divergence), the deciding factor for how much to change the current controller, is performed as part of the inner loop of GPS. In rGCL v2 however, we decided to place this calculation as part of the outer loop. We did this because Finn et al. (2016a) mentioned they calculated this step based on the previous cost function. This is sensible because the controller

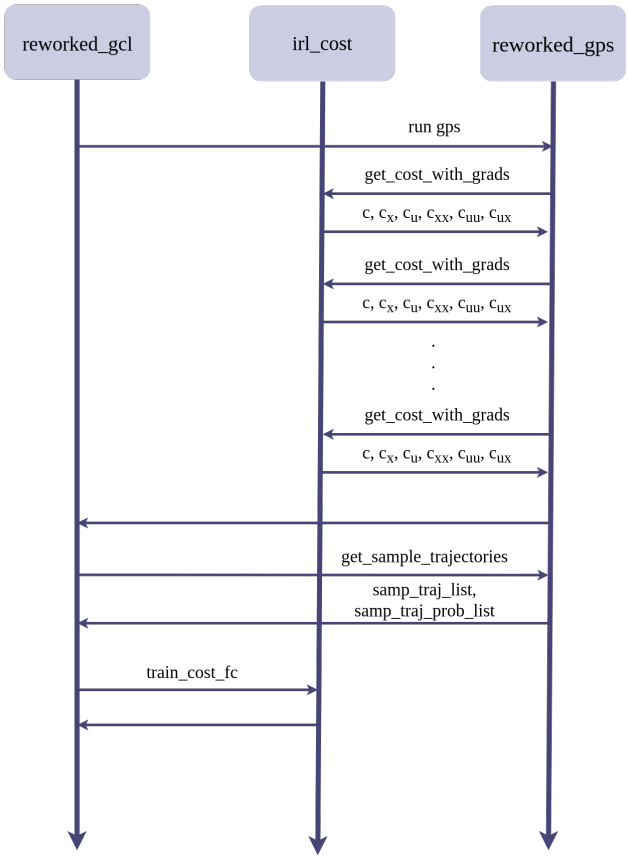

Figure 3.4: A simplified illustration of the interface of the three modules `reworked_gcl`, `irl_cost` and `reworked_gps` for rGCL v1. The illustration depicts the flow between the modules during one iteration of the `reworked_gcl` main loop. The flow is slightly different for rGCL v2.

which is run in the environment in the current iteration was updated using the previous cost function and will thus have a different expectation of cost than if it was updated based on the current cost function. In v1, such a change has not been made, meaning that when the `reworked_gps` loop starts, it uses the controller which was updated based on the previous cost function, while the step size is calculated based on the new cost function.

Chapter 4

Results

4.1 rGCL

In the following section we discuss the experiments performed with rGCL, the results and possible improvements.

4.1.1 Experiment setup

The GPS implementation (Finn et al.; 2016b), which we build upon, contains a few examples for testing. One of these is a Python box2d example, where a 2D, two-joint arm were to learn how to reach an upright position from a fixed initial position. We decided to use this simple example during our experiments with the rGCL implementation. Finn et al. (2016b) also contained a Graphical User Interface which was used during training. Figure 4.1 and 4.2 illustrates the experimental setup, with the 2D arm and the GUI respectively.

During training, a set of 10 demonstrations of the arm rising to the upright position was used as the IRL expert demonstrations. Before the rGCL loop, `irl_cost` performed one loop of cost optimizing using all the demos and no sample trajectories. This was performed as a pre-training step and was instated in an effort to initialize the weights of the cost function to prepare for the training. The demonstrations were performed by an agent who had learned the task using GPS (Finn et al.; 2016b), with a manually crafted cost function. In all of these demonstrations, the arm rose to its target position using the shortest route.

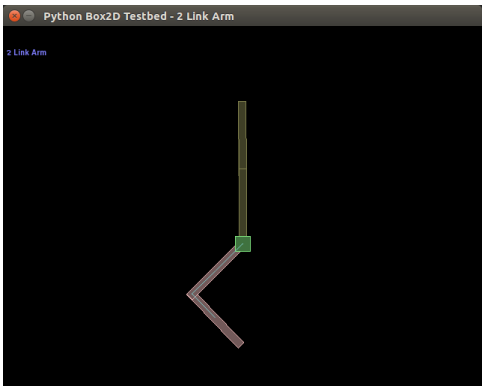

Figure 4.1: The two-joint arm in its initial position. The yellow outline represents the target position.

The specifications of the experiment are read into the rGCL program from a `hyperparamethers.py` file. Below is a summary of some of the parameters used in the experiments and a brief explanation of the reasoning behind them. However, because of limited time, we have not properly tuned the parameters.

Specifications for v1:

- 18 iterations of the rGCL loop
- 5 iterations of the complete GPS loop
- 5 trajectory samples taken in the outer loop of GPS
- 10 sample trajectories generated for cost optimization
- 5 trajectories in demo batch in `irl_cost`
- 10 trajectories in sample batch in `irl_cost`
- 1 iteration of the training loop in `irl_cost`
- Learning rate of `irl_cost` was 0.0001

Specifications for v2:

- 30 iterations of the rGCL loop
- 5 trajectory samples taken in the outer loop of GPS
- 10 sample trajectories generated for cost optimization

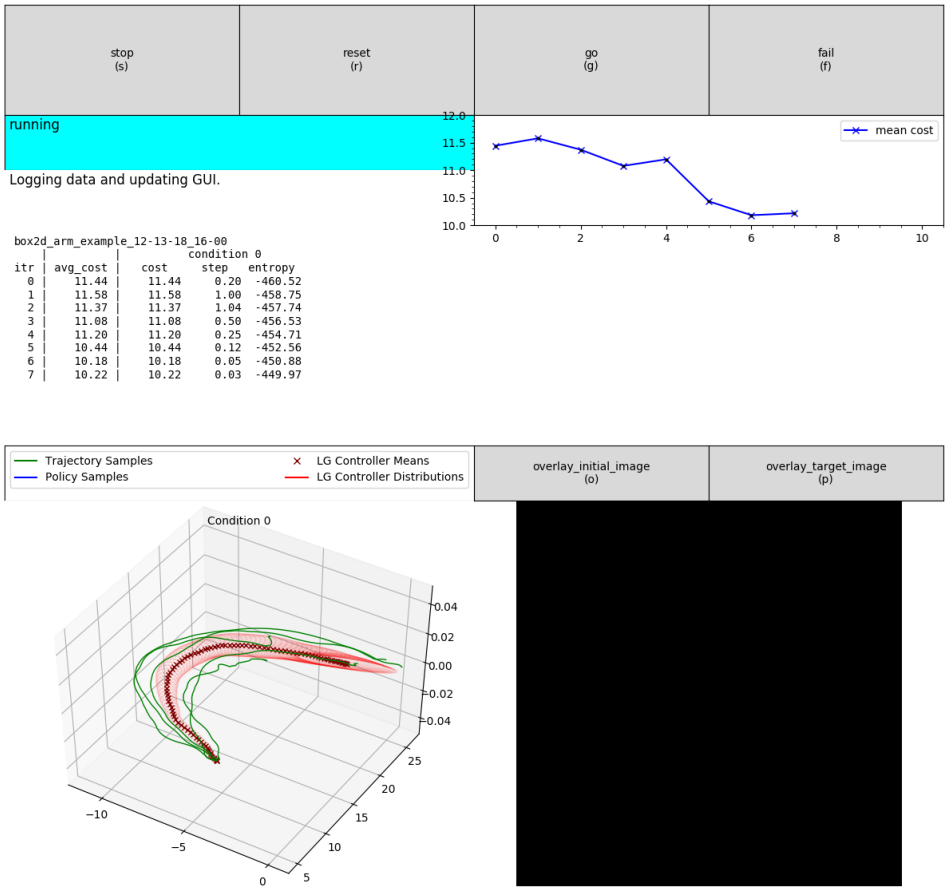

Figure 4.2: The Graphical User Interface (GUI) of the GPS part of the program. For each set of samples taken for the policy optimizing, the GUI presents the average cost, step-multiplier (based on KL-divergency) and entropy (the randomness of a distribution) in the top left corner. The mean cost is also expressed graphically in the top right corner. In the graph to the lower left, the linear-gaussian controller means and distribution is represented by the red crosses and lines respectively, while the sampled trajectories are represented by the green lines.

- 5 trajectories in demo batch in `irl_cost`
- 10 trajectories in sample batch in `irl_cost`
- 1 iteration of the training loop in `irl_cost`
- Learning rate of `irl_cost` was 0.0001

The number of samples collected through the outer GPS loop was chosen based on the default value present in the code. The more trajectories are collected, the

more likely they are to capture the dynamics. On the other hand, collecting the trajectories is the most time-consuming part of the training. Thus, a compromise was made.

We use a batch size for the generated sample trajectories twice the size as the batch size of the demonstrations. This was chosen in order to allow the cost function to experience a larger set of variations, as the demonstrations are quite similar. Similar reasoning lies behind the choice of a singular loop for cost optimization.

Backpropagation for the cost function is performed for every time step of every demonstrated or sampled trajectory in their respective batches. Because the length of each episode is 100 time steps, this means a total of 1500 back-propagations are performed per rGCL loop. In order to smooth the learning and allow the rGPS to guide it, we used a small learning rate. For both v1 and v2, a two-layered NN was used as a cost function.

The running time of one training session of rGCL v1 and v2 with the parameters above is approximately 40 minutes and 20 minutes respectively.

4.1.2 The training results

While the results of the training did not yield optimal solutions, rGCL shows notable promise. 19 tests were performed with v2, whereas 10 learned to reach an approximately upright position. In most of the tests, v2 showed a tendency towards the goal position, even though it did not always reach it. 13 tests were performed with v1, where 4 were learned to reach an approximately upright position. V1 thus performed more poorly than v2, however, when v1 did learn to rise to its target, it did so in a better way than v2.

Reaching the goal state

V1 had many unsuccessful runs, but when it did learn to rise, it did so rather well. It always picked the optimal route and 3 out of its 4 successful runs, it reached its goal pose and held it with a performance that rivaled that of the demonstrations themselves. Some runs started out good from the start and might come as a result of the pre-training, while others started out in the opposite direction and changed course midway through training. All successful runs were taken using the shortest path. Both v1 and v2 had a few runs where they learned to perform loops, rather

than to rise.

In contrast to v1, v2 had no fully optimal runs and alternated between taking the shortest run to the upright position and taking the long way around. V2 also had less of a collected distribution, i.e. with larger variance. However, even more than the 10 somewhat successful runs seemed to have grasped the fact that the goal state was in the upward direction.

V2 thus seems to have gained a more general notion of the task of rising, but lack the specifics of the goal state and how to get there, while v1 either performs the task better than v2 or much worse.

Cost and step values

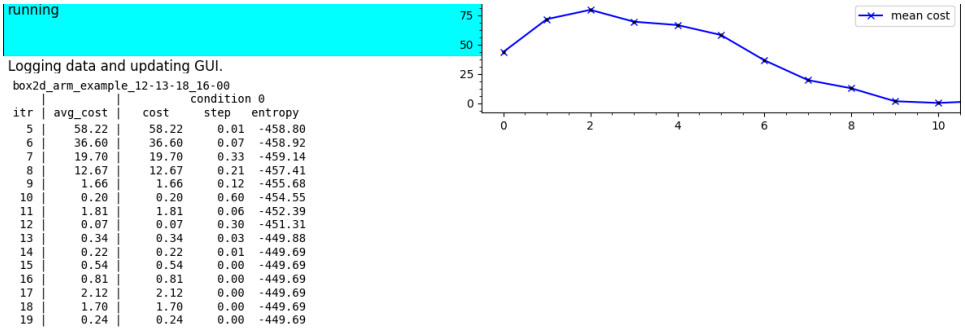

Figure 4.3: Crop out from the training GUI during run time with v2. Only the first two decimals of a number is shown, thus the printout may be misleading. E.g. the GPS code reveals that step is never at a complete zero, but rather at a minimum value.

Figure 4.3 shows an example of a print out during training with v2. The print out indicates that the cost is going down. However, we experienced that this is more due to the optimization of the cost function than the agent approaching an optimal performance with respect to the task. We can also see that the step in trajectory distribution is going down as the iterations increase to the point where it reaches a minimum step. With such low steps, the agent barely changes its trajectory distribution. This tendency was present in most runs with v2, and to a slightly lesser extent with v1. This is likely the reason why most of the successful runs happened before iteration 20.

Another concern is that judging from the printout from the cost optimization,

the cost function seemed to converge to weights which offered little distinction between suboptimal trajectories. This may be part of the reason why the step seems to disappear. The `reworked_gps` uses the gradients of the cost with respect to its input in addition to the cost itself. These are likely part of the reason why it was able to learn a somewhat correct behavior despite poorly trained costs.

4.1.3 Possible improvements

As previously stated, rGCL is currently a bare bones implementation. GCL as explained in Finn et al. (2016a) employ a number of tricks that we have not. One of the differences is the fact that they seem to grow their set of samples, by keeping those of older distributions, whereas we only use samples from the current iteration. The larger dataset might be useful as it could give the cost function a more complete picture.

The probability density of trajectories from previous distributions could be re-fitted to the current controller, or they could be kept the same as when they were sampled. However, based on the notation of $p(\tau)$ in Finn et al. (2016a), we view the first of the two as the more likely option. This way, the probability density of out-dated trajectories are made smaller and, considering the expression of the GCL gradients in Equation 2.13, they will have a larger impact on the updating of the NN weights. Equation 2.13 is repeated below.

$$\frac{d\mathcal{L}_{GCL}}{d\phi} = \frac{1}{N} \sum_{\tau_i \in \mathcal{D}_{demo}} \frac{dc_\phi}{d\phi}(\tau_i) - \frac{1}{Z} \sum_{\tau_j \in \mathcal{D}_{samp}} w_j \frac{dc_\phi}{d\phi}(\tau_j) \tag{4.1}$$

with $w_j = \frac{exp(-c_\phi(\tau_j)}{p(\tau_j)}$ and $Z = \sum_j w_j$.

As our optimization is a minimization problem, applying the positive gradients of the demonstrations will result in a smaller output (a smaller cost) for their states. Applying the negative gradients of the sampled trajectories, on the other hand, will cause a greater cost for their input. With a smaller $p(\tau_j)$, a trajectory is less likely to appear with the current distribution. This results in a smaller w_j and thus the trajectory makes a greater change in the network than those more likely to appear. In other words, the outdated trajectories, those considered wrong to the current distribution, will show the network what it should penalize, while the demonstrated trajectories show the network what it should not penalize.

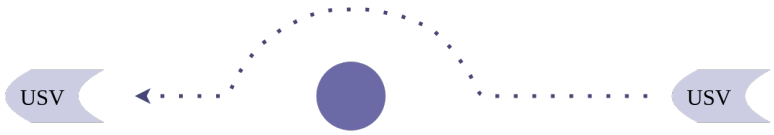

Figure 4.4: Approximate illustration of obstacle avoidance during expert demonstration. The dark circle represents the obstacle in question.

rGCL has no recollection of past trajectories. In a way, it relies solely on short-term memory. Because it only uses trajectories sampled from the current distribution to update its cost function, the trajectories will always be probable and most of the indication of what to penalize is lost. This might explain why the cost function seems to converge to low costs for most trajectories, even though they are quite different from the demonstrations.

The inclusion of older trajectories may not be all that hinders rGCL from better performance, but it could be an improvement. In addition, we point out that because of a lack of time, no tuning of the parameters was performed. Tuning of the parameters may also contribute to a better performance. This is especially true concerning the initialization of the weights in the cost function. Though the initial training seemed to boost the results somewhat, a better initialization and a deeper look into training strategies should be considered as the results of NNs heavily rely on their initialization and training scheme (Goodfellow et al.; 2016).

4.1.4 From state to observation in rGCL

The cost optimization uses the full state of the system. This does, however, not undermine GPS's learning of observation based policy as the cost function is used purely for training, not during testing. During testing, only the observation based policy is used, as in regular GPS, because the cost function itself is not used during testing. The difference is that the policy has been trained using a learned cost function, rather than a pre-made one.

4.2 Expert data collection

One of the goals of this project was to collect a set of expert USV demonstrations, to be used in the following master thesis. This was accomplished December

4th, by the use of the USV, Odin, courtesy of the Norwegian Defence Research Establishment. We traveled to Horten, Norway, where we performed sampling of manual obstacle avoidance in two data sets. 19 demonstrated episodes were collected in the first set and 47 in the second, making a total of 66 demonstrated episodes collected over the course of the day. During both of these two runs, the weather was dry and the water was relatively calm. This fact was beneficial to us as more challenging weather would introduce further challenges into an already difficult set of dynamics and task.

Each demonstration consisted of the USV being manually steered directly towards an obstacle, before swerving either right or left to avoid collision and then steering back to the previous course. An illustration of the procedure is presented in Figure 4.4. The obstacle being avoided was either a green or grey pole-shaped object, where half the demonstrations were performed on each. The initial positions and swerving point of the demonstrations varied. By varying the initial position, we hope to make the cost function invariant to the initial state.

During each demonstration, sensor data and output data was sampled in a ROS-bag file format. The sensor data collected included readings from Radar, LiDAR and Heading. Some problems occurred during the first set of demonstration episodes, hindering the LiDAR data from being sampled. Thus, only the second set contains LiDAR readings. Because the input size cannot change during training, this means that for the whole data set of 66 demonstrations to be used, the LiDAR readings from the second set cannot be used. Though, it might be more beneficial to simply use the second set. Whether 47 demonstrated episodes is enough to extract a good cost function remains to be seen. However, it is worth noting that Finn et al. (2016a) in their testing of the GCL approach on robotics tasks such as pouring from one cup to another, used 30 or fewer demonstrations. Thus, our set of demonstrations may be enough for the task of obstacle avoidance.

In addition to the previously mentioned sensor data, we were originally planning to sample camera data. Unfortunately, neither the USVs IR camera nor its visual camera was operational at the time.

Chapter 5

Conclusions and future work

Over the course of this project, we have accomplished three objectives. In our literature study in Chapter 2, we studied Reinforcement Learning (RL) and Inverse Reinforcement Learning (IRL) as well as an approach which combines the two in Guided Cost Learning (GCL)(Finn et al.; 2016a). This provided the necessary background for the next chapter of our project. In Chapter 3, we described our implementation of the GCL algorithm, where we combined Guided Cost Learning (Levine et al.; 2016; Finn et al.; 2016b) with cost optimization as described in Finn et al. (2016a). We called our implementation reworked GCL (rGCL) and tested it in a simple 2D environment, receiving results that were discussed in Chapter 4. The results from testing revealed rGCL is not yet able to learn optimal behavior from demonstrations. Yet, the results were promising as tendencies towards the target state were noticeable. In the discussion of the results, a set of possible improvements were suggested. The work put into the implementation and analysis of its results, provided a deeper insight into GCL, GPS and Neural Networks. Further results of the project come in the form of data collected from an USV. A data set of expert demonstrations from manual obstacle avoidance was collected for use in future work.

The project will continue in the following master thesis, where the focus will be to create a steering model for end-to-end control of a USV. As of now, we have not been working end-to-end as in e.g. the mapping of images to action parameters, and this extension will be a relevant topic. In addition, GCL, and by extension, rGCL, relies on the possibility to act in its environment during training. For an USV, such practice is impractical and possibly dangerous. Thus, an alternative way

of training should be explored, like the use of a simulator or a model.

References

(n.d.).

Baram, N., Anschel, O., Caspi, I. and Mannor, S. (2017). End-to-end differentiable adversarial imitation learning, *in* D. Precup and Y. W. Teh (eds), *Proceedings of the 34th International Conference on Machine Learning*, Vol. 70 of *Proceedings of Machine Learning Research*, PMLR, pp. 390–399.

Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L. D. and Muller, U. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car, *CoRR* **abs/1704.07911**.

Cheng, Y. and Zhang, W. (2018). Concise deep reinforcement learning obstacle avoidance for underactuated unmanned marine vessels, *Neurocomputing* **272**: 63–73.

Chi, L. and Mu, Y. (2017). Learning end-to-end autonomous steering model from spatial and temporal visual cues, *VSCC 2017 - Proceedings of the Workshop on Visual Analysis in Smart and Connected Communities, co-located with MM 2017*, pp. 9–16.

Finn, C., Levine, S. and Abbeel, P. (2016a). Guided cost learning: Deep inverse optimal control via policy optimization, *33rd International Conference on Machine Learning, ICML 2016*, Vol. 1, pp. 95–107.

Finn, C., Zhang, M., Fu, J., Tan, X., McCarthy, Z., Scharff, E. and Levine, S. (2016b). Guided policy search code implementation. Software available from rll.berkeley.edu/gps.
URL: *http://rll.berkeley.edu/gps*

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

Gosavi, A. (2009). Reinforcement learning: A tutorial survey and recent advances, *INFORMS Journal on Computing* **21**(2): 178–192.

Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning, *CoRR* .

Hofert, M. (2013). On sampling from the multivariate t distribution, *The R Journal* **5**: 130–137.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift, *32nd International Conference on Machine Learning, ICML 2015*, Vol. 1, pp. 448–456.

Jaynes, E. (1957). Information theory and statistical mechanics, *Physical Review* **108**(2).

Kober, J., Bagnell, J. A. and Peters, J. (2013). Reinforcement learning in robotics: A survey, *International Journal of Robotics Research* **32**(11): 1238–1274.

Levine, S. and Abbeel, P. (2014). Learning neural network policies with guided policy search under unknown dynamics, *Advances in Neural Information Processing Systems*, Vol. 2, pp. 1071–1079.

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

Levine, S. and Koltun, V. (2013). Guided policy search, *30th International Conference on Machine Learning, ICML 2013*, pp. 1038–1046.

Levine, S., Popović, Z. and Koltun, V. (2011). Nonlinear inverse reinforcement learning with gaussian processes, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011, NIPS 2011*.

Lu, Y. (2017). Implementation of inverse reinforcement learning (irl) algorithms in python/tensorflow. deep maxent, maxent, lpirl. Software available from https://github.com/yrlu/irl-imitation.
URL: *https://github.com/yrlu/irl-imitation*

Martinsen, A. B. and Lekkas, A. M. (2018). Straight-path following for underactuated marine vessels using deep reinforcement learning, *IFAC-PapersOnLine* **51**(29): 329–334.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015). Human-level control through deep reinforcement learning, *Nature* **518**(7540): 529–533.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview, *Neural Networks* **61**: 85–117.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search, *Nature* **529**(7587): 484–489.

Sutton, R. S. and Barto, A. G. (2017). *Reinforcement Learning: An Introduction, 2nd edition*, MIT Press Ltd.

Tensorflow home page (2017). [Online; accessed 13-December-2018].
URL: *https://www.tensorflow.org/*

Wang, H. and Banerjee, A. (2014). Bregman alternating direction method of multipliers, *Advances in Neural Information Processing Systems (NIPS)*.

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

Wulfmeier, M., Ondruska, P. and Posner, I. (2015). Maximum entropy deep inverse reinforcement learning, *CoRR* **abs/1507.04888**.

Wulfmeier, M., Wang, D. Z. and Posner, I. (2016). Watch this: Scalable cost-function learning for path planning in urban environments, *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2089–2095.

Zhang, M. (2017). Model-based reinforcement learning. [Online; posted 27-September-2017].
URL: *https://michaelrzhang.github.io/model-based-rl*

Zhifei, S. and Joo, E. M. (2012). A survey of inverse reinforcement learning techniques, *International Journal of Intelligent Computing and Cybernetics* **5**(3): 293–311.

Ziebart, B. D., Maas, A., Bagnell, J. A. and Dey, A. K. (2008). Maximum entropy inverse reinforcement learning, *Proceedings of the National Conference on Artificial Intelligence*, Vol. 3, pp. 1433–1438.