IMITATION LEARNING FOR AUTONOMOUS QUADROTOR FLIGHT

Xiatao Sun

A THESIS

in

Robotics

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Master of Science in Engineering

2023

Vijay Kumar

Supervisor of Thesis

M. Ani Hsieh

Graduate Group Chair

2023

Xiatao Sun

IMITATION LEARNING FOR AUTONOMOUS QUADROTOR FLIGHT

2023

Xiatao Sun

ACKNOWLEDGEMENT

First, I would especially like to thank Professor Vijay Kumar for his invaluable guidance and support on this project. I am grateful for his advice and encouragement, which have been not only helpful and constructive for this thesis and my past work but also enlightening for my future work and goal of becoming a roboticist.

Moreover, I would like to thank Yuwei Wu for her insightful suggestions on my research direction and diligent help with implementations. Her guidance and recommendations on recent literature allowed me to quickly grasp the essence and prerequisite knowledge of this project. Additionally, her understanding and expertise in trajectory planning and control provided me with countless valuable suggestions for working with existing control stacks, trajectory trackers, and representations.

Finally, this work would not have been possible without what I have learned from courses and research projects in the past two years. Therefore, I would like to thank all professors whom I have learned from and worked with during the past two years. I would also like to thank my parents for supporting me in pursuing a Degree of Master of Science in Engineering in Robotics.

ABSTRACT

With the rising popularity of machine learning and deep learning, recently, learning-based methods have gained increasing attention and are employed in various control and planning for quadrotors. However, existing works on learning-based planners for autonomous quadrotor flight mostly utilize reinforcement learning, which is inefficient due to its nature of trial and error in large state-action spaces and is hard to transfer to real-world scenarios since the reward function relies heavily on heuristics and has the risk of being exploited by the learner.

Compared with reinforcement learning, imitation learning has better sample efficiency thanks to having access to an expert who can provide demonstrations. Learned policies from imitation learning can avoid blind exploration and have a more stable performance during inference. Therefore, imitation learning is often considered as an alternative to reinforcement learning.

Although imitation learning has a strong assumption of requiring an expert, in the context of autonomous quadrotor flight, this assumption can be easily fulfilled due to the enormous amount of existing traditional optimization-based planners. Hence, planning policies from imitation learning are promising for improving sample efficiency and stability for autonomous quadrotor flight. Besides serving as a substitute for reinforcement learning, the increasing demand for computational resources brought by novel optimization-based planners also calls for knowledge distillation techniques that are suitable for quadrotors, which imitation learning can be counted as one of them.

However, few existing works of imitation learning pipelines are based on the Robot Operating System and have the potential to transfer to physical onboard platforms. Most existing imitation learning pipelines for quadrotors are unable to transfer to real-world scenarios and can only work in simulators. Therefore, in this work, we propose an imitation learning pipeline based on the Robot Operating System and a complementary simulation environment for training and validation. We provide versatile data collection and training components and show that our proposed imitation learning pipeline can effectively learn from traditional planners.

TABLE OF CONTENTS

ACKNOWLEDGEMENT iii

ABSTRACT iv

LIST OF TABLES vii

LIST OF ILLUSTRATIONS viii

CHAPTER 1 : INTRODUCTION 1

1.1 Background 1

1.2 Motivation 2

1.3 Contribution 3

CHAPTER 2 : RELATED WORK 4

2.1 Imitation Learning 4

2.2 Motion Planning for Quadrotors 6

2.3 Simulators for Quadrotors 8

CHAPTER 3 : METHODOLOGY 10

3.1 Overview 10

3.2 Initialization 11

3.3 Calibration 11

3.4 Simulation 11

3.5 Controller 13

3.6 Expert 13

3.7 Network Architecture 14

3.8 Data Collectors 15

3.9 Training 16

3.10 Inference 17

CHAPTER 4 : EXPERIMENTS AND ANALYSIS 19
4.1 PID Controller Response 19
4.2 Loss Functions 20
4.3 Scheduler 21
4.4 Data Collection 22
4.5 B-Spline Inference 25

CHAPTER 5 : LIMITATION AND FUTURE WORK 27

CHAPTER 6 : CONCLUSION 29

BIBLIOGRAPHY 30

LIST OF TABLES

TABLE 3.1 Structure of the post-processing network for depth embedding, the state backbone, and the planning network. 15
TABLE 3.2 Parameters for the PyTorch Scheduler. 17
TABLE 3.3 Parameters for *CustomCosineDecayRestarts*. 17

TABLE 4.1 Evaluation of the 12 B-splines from the inference of the learned policy 26

LIST OF ILLUSTRATIONS

FIGURE 3.1 Overview of the proposed imitation learning pipeline. 10
FIGURE 3.2 The simulation environment developed with Unreal Engine 4. 11
FIGURE 3.3 Visualization of the voxel map generated from the Unreal environment. 12
FIGURE 3.4 Visulization of the quadrotor model used in simulation. 13

FIGURE 4.1 Plots of PID responses for position control. 19
FIGURE 4.2 Plots of training loss and validation loss when training with MSE loss and SSE loss. 20
FIGURE 4.3 Plots of training loss and validation loss when training with our implementation of the cosine decay scheduler with restarts and PyTorch scheduler. 21
FIGURE 4.4 Plots of training loss and validation loss when training on datasets collected by the Python data collector with different sizes. 22
FIGURE 4.5 Plots of training loss and validation loss when training on datasets collected by the C++ data collector with different sizes. 24
FIGURE 4.6 Visualization of 12 expert-labeled B-splines and B-splines from the inference of the learned policy. 25

CHAPTER 1

INTRODUCTION

1.1. Background

Learning-based methods have been gaining increasing popularity for autonomous quadrotor flight thanks to the rapid advancement of machine learning, deep learning, and hardware for parallel computing. Among all learning-based techniques, Reinforcement Learning (RL) receives the most attention due to its compatibility with robot learning and the availability of numerous simulation environments for training.

However, RL heavily relies on carefully designed reward functions, which are usually based on heuristics. Such reward functions might be exposed to the risk of reward hacking, which allows the agent to leverage the flaws in reward functions and maximize its rewards in malicious ways [1]. Even if a reward function is carefully designed, RL may still suffer from reward sparsity and require an enormous amount of training steps in order to converge to a satisfying policy, which results in low sample efficiency and a considerable amount of training time [2]. This issue is further amplified, especially in the scenario of autonomous quadrotor flight, where the exploration of RL policy has to be performed in the high-dimensional state and action space of the quadrotor [3].

Moreover, although RL has been widely used for autonomous quadrotor flight, its application is relatively niche and can only be used for training an end-to-end policy for planning from the beginning. But besides training an end-to-end policy from scratch, the recent development in traditional optimization-based planners is also in need of learning-based techniques to reduce the computational complexity in order to deploy onboard [4]. Therefore, a new learning pipeline is generally desired to improve the sample efficiency when training an end-to-end policy and reduce the computational overhead of sophisticated planners.

1.2. Motivation

Imitation Learning (IL) is usually considered as an alternative to RL. Compared with RL, it has considerably higher sample efficiency but assumes having access to an expert [5]. However, this assumption can be easily satisfied in the scenario of autonomous quadrotor flight, as traditional motion planning has been well-studied, and numerous planners have been proposed [6]. Existing planners can be easily used as the expert for IL after slight modification.

Additionally, thanks to the paradigm of IL, it can also be employed to learn a low-cost policy from a high-cost sophisticated planner in order to reduce its computational complexity and execute onboard [4]. Since the learned policy can be represented using a neural network, the computational complexity and representation power of the learned policy can be easily adjusted as needed.

However, existing IL pipelines for autonomous quadrotor flight are usually based on OpenAI gym [7] or interact with gym-like interfaces, which are designed for testing or evaluation of learning algorithms rather than training an end-to-end policy that is possible to deploy on a physical quadrotor. Such learning pipelines and simulations typically use oversimplified environments that are without obstacles and collisions and do not take the dynamics of the quadrotor into consideration or just rely on an extremely simple dynamics model [8]. Due to utilization and reliance on the gym or gym-like interfaces, existing IL pipelines and simulations are also unable to leverage most existing planners and help reduce the computational complexity for sophisticated planners without heavy modification of the planners since they are usually developed with Robot Operating System (ROS) [9] and C++ rather than gym and Python.

To bridge the gap mentioned above, in this work, we propose a high-fidelity and versatile IL pipeline and a complementary simulation environment for autonomous quadrotor flight that are designed based on ROS while leveraging collision simulation and advanced dynamics from Unreal Engine (UE) 4 [10] and AirSim [11].

1.3. Contribution

The contribution of this thesis can be summarized as follows:

- Unlike most previous works that require training in a gym-like environment, this work provides an IL pipeline that can work with ROS. Thus, the fidelity of simulation during training can be improved, and the learned policy is possible to transfer to real-world scenarios. The code can be found at **here** [1].

- This work extends the current open-sourced ecosystem of AirSim. Consequently, it is also able to leverage miscellaneous existing works developed for the AirSim platform.

- In addition to the simulation environments provided by AirSim, we develop an additional simulation environment with UE 4 to assist IL specifically. The complementary simulation environment for IL can be found at **here** [2].

- The proposed IL pipeline is fully compatible with the existing software of Kumar Robotics [12]. Therefore, the learned policy can be easily integrated with existing controllers and trackers from Kumar Robotics.

- A neural network architecture for quadrotor trajectory planning is also proposed along with the IL pipeline.

- Extensive experiments and validations are performed using the proposed IL pipeline and the complementary simulation environment. All learned policies and experimental data can be found at **here** [3].

[1] If the embedded hyperlink is not accessible or clickable due to scenarios like a printed copy, here is the explicit link: https://github.com/M4D-SC1ENTIST/learning_perception_aware

[2] https://drive.google.com/drive/u/1/folders/1WgnixhRqEVkvfYKvTTJ8FzJnzPMjl9Ox

[3] https://penno365-my.sharepoint.com/:f:/g/personal/sxt_upenn_edu/EvSiGF1tdgVMsbk7V763MGoBosW_a031GdZ_zDlltAKJjQ?e=3c0BVQ

CHAPTER 2

RELATED WORK

2.1. Imitation Learning

2.1.1. Behavioral Cloning

Imitation learning, as the name suggests, refers to learning a policy that behaves similarly to an expert. In typical settings of IL, the learner is assumed to have access to one or more experts or demonstrations from experts. The simplest IL algorithm is Behavioral Cloning (BC) [13], which can be regarded as an application of supervised learning. In BC, the demonstrations, which are usually state and action pairs, are collected from experts. During the training process, states and actions are treated as inputs and outputs of the neural network, respectively. Therefore, BC directly applies supervised learning to find patterns from demonstrations from experts.

2.1.2. Direct Policy Learning

Although BC is simple, it provides a foundation and paradigm for further research in the field of IL. Based on the notion from BC, two branches of IL emerge, which are Direct Policy Learning (DPL) and Inverse Reinforcement Learning (IRL) respectively [14]. DPL, as the name suggests, focuses on directly learning a policy from provided demonstrations. It aims at solving the problem of compounding error brought by BC. Since it is hard to collect demonstrations from the expert for every state in many scenarios, especially when the state space is continuous, unseen states exist when performing inference using the learned policy. Therefore, during inference, the learned policy will inevitably deviate from the trajectory of the expert due to unseen states. As the number of unseen states encountered increases, the errors between the trajectory from the learned policy and the expert accumulate and increase as well, which leads to the ultimate failure of the learned policy [15].

To tackle the problem of error accumulation, Data Aggregation (DAgger) is proposed as the foundational algorithm in the branch of DPL [16]. Instead of purely relying on the expert to provide

states and actions for demonstrations, DAgger uses the learner and the expert to provide states and actions for demonstrations, respectively. For every rollout, DAgger first performs inference using the learned policy of the current rollout and collects the states generated by this policy. After the rollout terminates, DAgger uses the expert to label these collected states. Then, the collected states and actions are used to train a policy for the next rollout. In this way, DAgger can ultimately train a policy with demonstrations that cover the states that the learner most likely encounters.

Based on DAgger, two branches of DPL are then proposed for improving the efficiency of data collection and deployment in different scenarios. The first branch is human-gated methods, such as Human-Gated DAgger (HG-DAgger) [17]. Unlike DAgger, which separates the data collection of states and actions to the learner and expert, HG-DAgger first uses the learned policy for inference, and once the expert senses that the learned policy enters undesired states, it intervenes, takes control, and provides demonstrations while guiding the agent back to desired states. Based on the idea of gating function by human, Expert Intervention Learning (EIL) is proposed [18]. It uses two additional loss functions and three partitioned datasets and aims to solve the problem of covariate shift in IL with feedback [19]. Besides human-gated methods, another branch of DPL is robot-gated methods. As the name suggests, instead of letting the human (expert) decide when to intervene, robot-gated methods let the learner decide when to hand over the control of the agent to the expert. Therefore, robot-gated methods usually rely on some heuristic parameters for deciding when to hand over the control. For example, ThriftyDAgger uses risk and novelty to decide when the expert takes control of the agent [20].

2.1.3. Inverse Reinforcement Learning

Unlike DPL, which directly learns a policy, IRL learns a reward function, and then use RL to finally learn a policy [21]. Similar to a lot of other inverse problems, the primary issue with IRL is that the solution is often non-unique. More than one reward function can be derived from a given dataset of demonstrations. To tackle on this problem, Maximum Entropy (MaxEnt) IRL is proposed by leveraging the maximum entropy principle on feature matching [22]. However, the scenarios that MaxEnt can be deployed into are model-based and highly idealized. Therefore, Guided Cost

Learning is proposed in order to generalize it to more realistic model-free scenarios [23]. It uses MaxEnt as the foundation and use a neural network as the reward function in order to be used in model-free settings. Similar to Guided Cost Learning, Generative Adversarial Imitation Learning (GAIL) is proposed as an IRL algorithm that can work in model-free scenarios by introducing Generative Adversarial Network (GAN) into IRL [24].

2.2. Motion Planning for Quadrotors

2.2.1. Traditional Approaches

Traditional motion planning methods can be divided into the front end and the back end [25]. The front end is usually responsible for path-finding in discretized space, and the back end is usually for optimizing the trajectory to be continuous. The front-end methods can be categorized into sampling-based and searching-based methods.

Two primary approaches for sampling-based methods are Rapidly-Exploring Random Tree (RRT) [26] and Probabilistic Roadmap (PRM) [27]. RRT searches a path by randomly sampling points and connecting sampled points if possible in free space. Based on RRT, RRT* is proposed to optimize RRT by searching for the shortest path [28]. It finds the shortest path by recording the cost of traveling between each vertices and reconnecting to the neighbor with lowest cost if possible. To better perform kinodynamic planning with underactuated systems, LQR-RRT* is then proposed by adding a heuristics of linear quadratic regulator [29]. Compared to RRT, PRM samples random points and connect it with nearby configuration using a local planner after verifying the sampled points is in the free space. Based on PRM, numerous variant has been proposed as well, including visibility-based PRM [27], obstacle-based PRM [30], and Lazy PRM [31].

Searching-based methods usually require discretization of the configuration space and treat it as a graph search problem. Thus, the foundational methods for searching-based pathfinding include Breadth-First Search (BFS), Depth-First Search (DFS), and Dijkstra's algorithm [32]. Among the three foundational methods, Dijkstra's algorithm can find the shortest path quickly, whereas DFS and BFS have different priorities and properties when performing searching.

After the front-end pathfinding process, the back-end trajectory optimization is then performed in order to parametrize the trajectory in time for achievable kinodynamics [25]. The optimization methods can be divided into hard-constrained methods and soft-constrained methods. For hard-constrained methods, the constraints imposed on the variables have to be satisfied. In order to reduce the computational overhead with state space, the minimum-snap trajectory generation algorithm is proposed to leverage the efficiency of differential flatness [33]. For soft-constrained methods, the constraints are implemented by penalization rather than enforcing for guarantee. One example of soft-constrained methods is Covariant Hamiltonian Optimization for Motion Planning (CHOMP), which minimizes the obstacle potential [34].

Within the paradigm of front-end pathfinding and back-end trajectory optimization mentioned above, traditional motion planning methods have been well-studied, and various planners have been open-sourced for future research, which can be used as experts when performing imitation learning. In this work, we primarily use Fast Planner as our expert planner [35], which is developed with kinodynamic path searching and B-spline optimization [36].

2.2.2. Learning-based Approaches

Learning-based methods for quadrotor motion planning have been gaining increasing popularity recently. Most previous research mainly uses RL or IL to train a policy for specific scenarios, transfer planning policy to platforms with different sensing modalities, or decrease the computational cost to be able to run onboard.

RL is primarily used for training an end-to-end planning policy in a specific scenario or with a specific type of sensor. Camci *et al.* use RL to train an end-to-end planner that takes depth images as input and outputs motion primitive sequences [37]. Kim *et al.* train a planner to navigate in an environment with static obstacles for goal-reaching tasks [38]. Song *et al.* propose an RL-based approach for training a policy that can perform autonomous drone racing [3]. Kaufmann *et al.* provide a benchmark comparison for using Proximal Policy Optimization (PPO) to train policy for quadrotor agile flight with different input and output modalities [39].

Unlike RL, IL is primarily used for training a policy to remove the need for privileged information or reduce computational overhead. Loquercio *et al.* use IL to train an end-to-end policy that is able to run onboard from an expert that requires access to privileged information. Tordesillas *et al.* use IL to train a policy with a reduced computational cost for jointly optimized perception-aware planning [40].

2.3. Simulators for Quadrotors

Despite the rising popularity of learning-based methods for quadrotor motion planning, it still requires further research compared with the well-studied traditional methods, and the development of better simulators can assist in further research.

2.3.1. AirSim

AirSim is a simulation platform developed by Microsoft Research [11]. AirSim is comprised of a set of Python and C++ APIs, simulation environments developed with Unreal Engine 4, and a ROS bridge. It can achieve high fidelity both visually and physically and is compatible with a variety of flight controllers, including PX4 and ArduPilot. However, it is hard to do IL on it since most existing planners are developed with C++ due to consideration of efficiency, and AirSim does not provide a set of gym-like interfaces due to using Unreal Engine 4, which makes it extremely hard to control the precise time step and the training status.

2.3.2. Flightmare

Flightmare is a quadrotor simulator developed by the UZH Robotics and Perception Group [41]. It contains a rendering engine developed with Unity and a physics simulation. It is designed to be modular and provides APIs and gym-like interfaces for training. However, due to its modularity, the physics engine and rendering are decoupled, making it sometimes have mismatches between visual input and output from the expert due to latency. Also, since it does not have collision detection in the rendering engine, the agent may fly into obstacles visually but be detected to collide physically, which further amplifies the issue of discrepancy in demonstrations.

2.3.3. Other simulators for learning

There are some universal simulators for the general usage of IL and RL, such as MuJoCo[42] and Drake[43]. However, most of them are designed for testing and validation of learning, control, or planning algorithms instead of training a policy for use in real-world scenarios. They either lack high-fidelity rendering or physics. Using them in the scenario of training policies for quadrotors requires heavy modification of either rendering or physics simulation.

CHAPTER 3

METHODOLOGY

3.1. Overview

The proposed IL pipeline, as shown in Figure 3.1, is primarily comprised of an expert, data collectors, training pipelines, neural networks, an inference pipeline, a controller, AirSim simulator, and ROS bridge to AirSim. The following sections explain the proposed IL pipeline in detail.

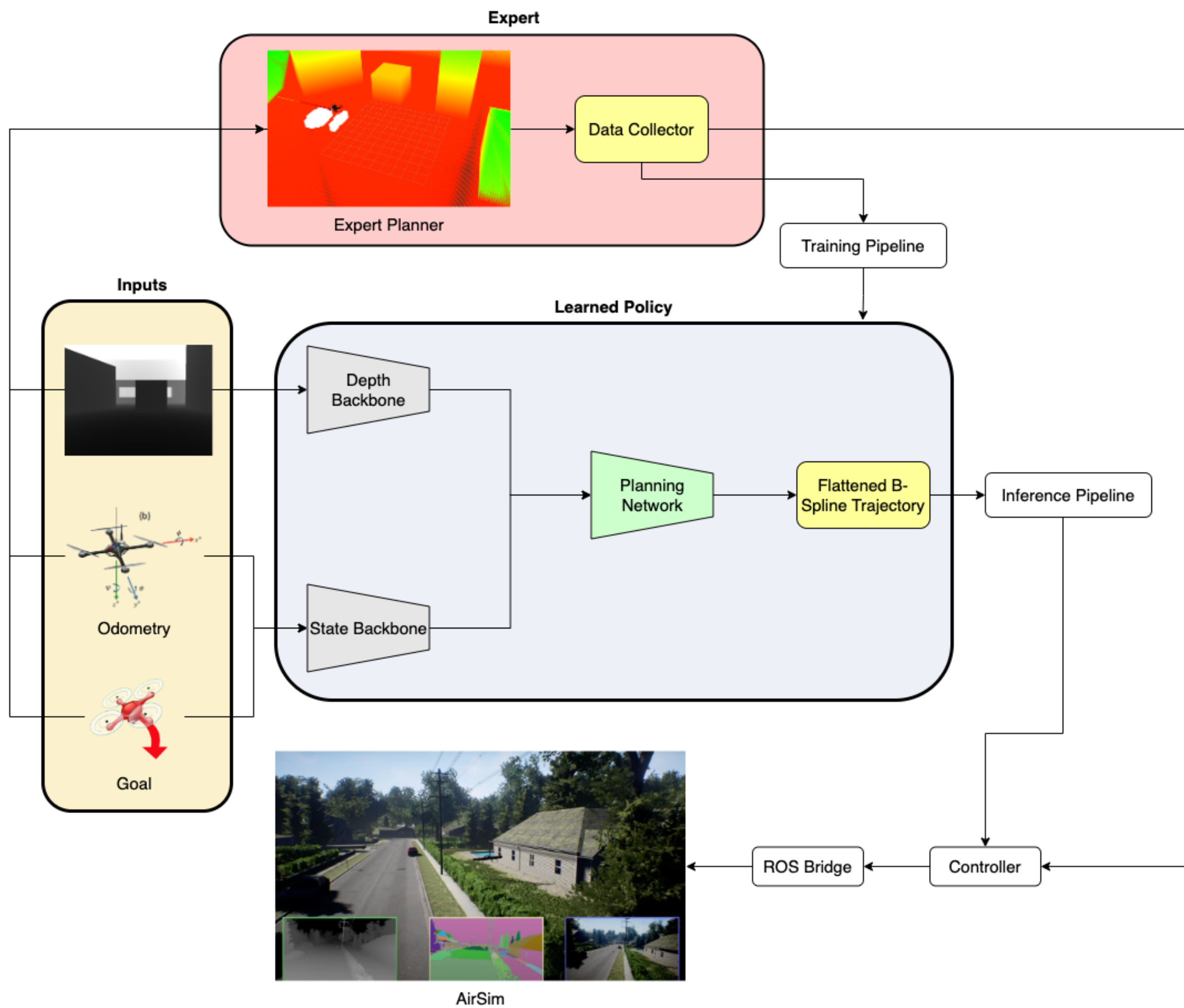

Figure 3.1: Overview of the proposed imitation learning pipeline.

3.2. Initialization

To initialize the IL pipeline, the user can execute the *imitation_learning_manager.py* script. It takes a YAML configuration file as input and manages the IL processes based on information provided in this YAML file. This script controls the initialization, operation, and termination of the Unreal Engine environment, the expert planner, data collectors, the training pipeline, and so on via signal handler based on the POSIX standard.

3.3. Calibration

Since the starting point and unit have discrepancies between the Unreal Engine environment and ROS, the *set_z_offset* node is developed to automatically check the gap between the starting point in Unreal Engine and ROS, and perform calibration and unit transformation in order to make the positions between Unreal Engine and ROS match with each other.

3.4. Simulation

The simulation environment is developed with Unreal Editor 4.25.4 with an AirSim plugin for communication with the ROS bridge. The environment is shown in Figure 3.2.

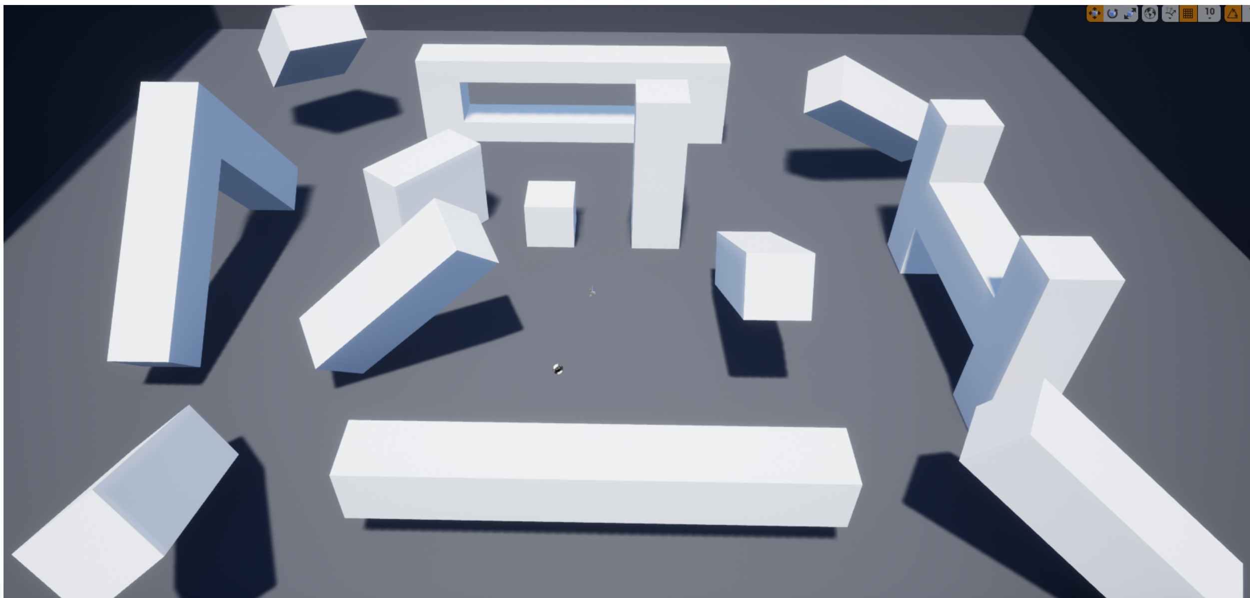

Figure 3.2: The simulation environment developed with Unreal Engine 4.

The environment is cluttered with blocks with various positions, scales, and orientations to better train and evaluate the learned policy. The simulation uses the East-North-Up (ENU) coordinate system. During the initialization process, a voxel map of the environment is also constructed by retrieving the properties of game objects in the Unreal environment via the AirSim plugin. A visualization of the constructed voxel map is shown in Figure 3.3.

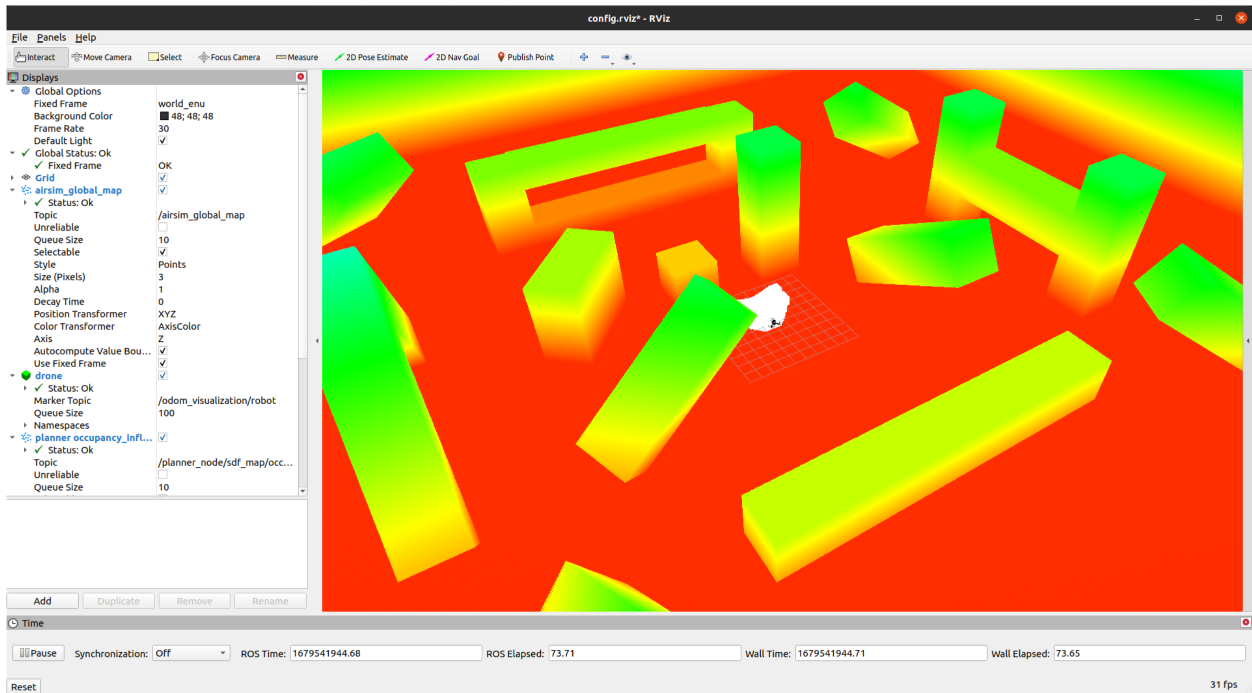

Figure 3.3: Visualization of the voxel map generated from the Unreal environment.

In the simulation environment, to reduce the sim2real gap, we use a quadrotor model that can both visually and physically simulate Falcon250V, which is the most recent quadrotor platform developed in Kumar Lab. The quadrotor model has a dimension of $0.25m \times 0.25m \times 0.25m$, which matches the dimension of the real quadrotor platform. The visualization of the quadrotor model is shown in Figure 3.4.

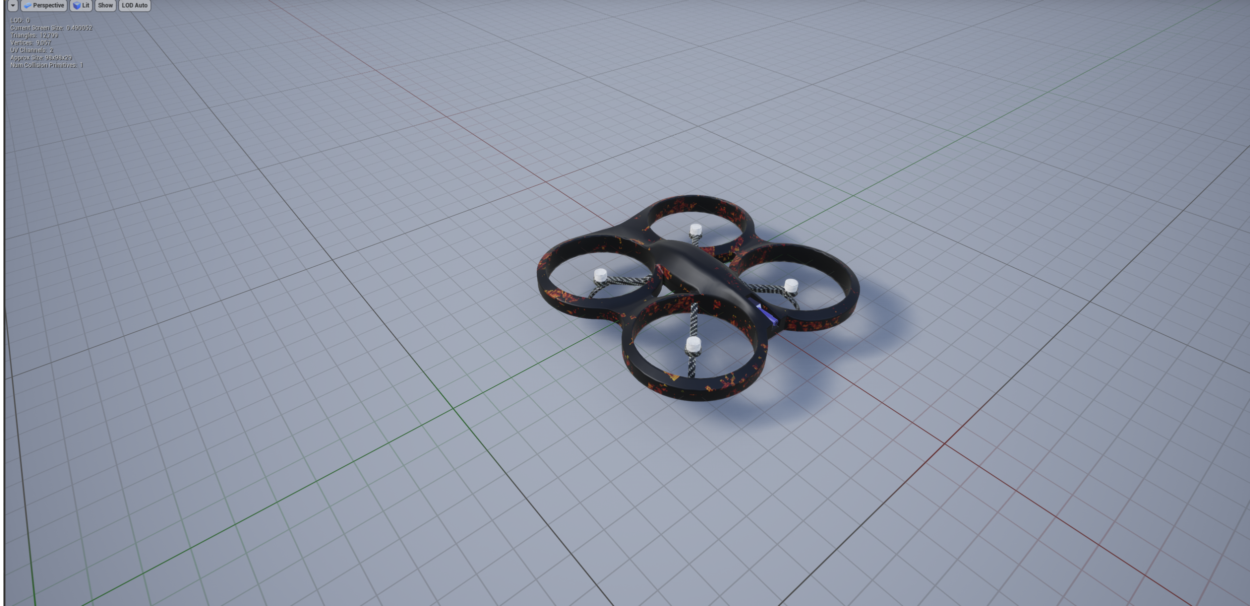

Figure 3.4: Visulization of the quadrotor model used in simulation.

3.5. Controller

In this work, we use *kr_mav_control* [44], which is a PID controller with various interfaces and trackers from Kumar Robotics [12], to control the quadrotor in the AirSim environment. Specifically, we use the B-spline tracker, which is an action server, to process the incoming planned trajectories. The controller is tuned using the Ziegler–Nichols method [45]. To connect with the ROS bridge of the Unreal environment and send the control command to the quadrotor in the simulation, we choose to use the mavros interface and register a callback in the AirSim ROS wrapper. In this way, we can control the quadrotor with roll, pitch, yaw rate, and throttle.

Although in this work we choose to use the PID controller, we can always switch to an MPC controller as long as it is compatible with ROS services, actions, and messages of Kumar Robotics.

3.6. Expert

We use the Fast Planner [35] as the expert planner in this work. Fast Planner is a quadrotor trajectory planner with traditional front-end and back-end architecture. The front end performs kinodynamic A* to search for a viable path, whereas the back end performs optimization to get a

B-spline trajectory. It takes a depth image and odometry as input and plans locally.

Despite using a local planner as the expert planner, the proposed IL pipeline also gives the option to use a global planner like the motion primitive library [46] as the expert planner since it already provides the global voxel map that can be used as input for global planners.

3.7. Network Architecture

As shown in Figure 3.1, the proposed neural network architecture consists of three components, which are two backbones for processing depth image and state information respectively, and one planning network to output a flattened B-spline trajectory.

The depth backbone is essentially a MobileNet V3 [47] with fine-tuning for our quadrotor flight scenario. The initial weight of the MobileNet V3 model is trained on the ImageNet dataset [48]. MobileNet V3 is designed to perform image classification tasks. Therefore, the output dimension of MobileNet V3 is an array with a length of 1000. Because we only want to use MobileNet V3 as a feature extractor, we design a post-processing network that is essentially a Multi-Layer Perceptron (MLP) with three layers. The post-processing network takes in the output of the depth backbone and outputs an array with a length of 32.

The state backbone is another 3-layer MLP that takes in a flattened concatenation of current position, current velocity, current orientation, and current goal as input. The current position, velocity, and orientation are obtained from the odometry message, while the current goal is generated by the training manager. The output dimension of the state backbone is also an array with a length of 32.

The extracted features from the depth backbone and the state backbone are then passed into the planning network, which is another 3-layer MLP. After processing the depth and state embeddings, the planning network outputs either a flattened B-spline or a flattened coefficient matrix.

The number of hidden units of the three MLPs can be found in Table 3.1. All networks use ReLU as the activation function.

Table 3.1: Structure of the post-processing network for depth embedding, the state backbone, and the planning network.

Network	Number of hidden units
Post-processing network for depth	[128, 64, 32]
State backbone	[64, 32, 32]
Planning network	[64, 128, 128]

3.8. Data Collectors

Two data collectors are developed with Python and C++ separately in our proposed IL pipeline. Each one serves different purposes and is designed with different objectives in mind. The Python data collector is developed with the concept of inheritance. It intercepts messages from the expert planner and processes them based on the modification in the inherited callback function in the child class. In this way, a variety of interactive imitation learning algorithms, such as DAgger [16], HG-DAgger [17], EIL [18], and MEGA-DAgger [49], can be implemented in the pipeline by overriding the callback function. Since in Python, the dataset can be directly saved as npz files, loading the demonstration data during training is also faster when using the Python data collector.

However, since the Python data collector relies on intercepting the message from the expert planner to achieve interactive learning, it suffers from the latency caused by network communication between different ROS nodes. Therefore, it can cause mismatches between observations and actions in the demonstration dataset if the CPU is not powerful enough or the expert is computationally expensive. Consequently, we also develop a data collector with C++ that directly collects the demonstration data right after the expert planner finishes calculating the B-spline trajectory. The C++ data collector is a class in the *cpp_data_collector* package and needs to be included in the expert planner when using it. However, interactive imitation learning algorithms cannot be used with the C++ data collector as there is no way to switch between the expert and the learner. The C++ data collector can only be used with Behavioral Cloning for learning a planning policy. Therefore, when choosing between the types of data collectors to use, the user needs to balance versatility and efficiency. A comparison between the two data collectors is provided in the Experiments and Analysis chapter.

3.9. Training

Regardless of which data collector is used, the training is always performed in Python with PyTorch [50]. However, the specific script to use for training is different for the two data collectors. If the user decides to use the Python data collector, *imitation_learning_manager.py* should also be used correspondingly. This learning manager initializes the *ImitationLearnerBase* class or its child, which can be inherited from the *ImitationLearnerBase* class with interactive learning functionality as illustrated in the previous section.

When using the C++ data collector, the user should use the *train_on_cpp_datasets.py* script under the *training_only* folder since the C++ data collector only supports learning using Behavioral Cloning, which is essentially supervised learning. To improve the efficiency of loading the dataset, the user can use the *cpp_dataset_postprocessing.py* script to convert the collected dataset from C++ to npz files. However, this can significantly increase the size of the dataset both loaded in memory and saved on disk due to the internal data structure of Numpy. Therefore, the user needs to balance the tradeoff between efficiency during training and data loading.

Currently, both training pipelines have two options for the loss function, which are Mean Squared Errors (MSE) loss and Sum [51] of Squared Errors (SSE) loss [52]. The MSE loss is calculated by taking the mean square error, which is the squared L2 norm, for the difference between each element in the flattened expert-labeled B-spline and the B-spline from inference, and then calculating the mean of these differences. The function is shown as follows:

$$L_{MSE}(a_e, a_l) = \frac{\sum_{i=1}^{n}(a_{e_i} - a_{l_i})^2}{n}, \tag{3.1}$$

where a_e denotes the flattened B-spline from the expert planner, a_l denotes the flattened B-spline from the learned policy, n denotes the total number of elements in the flattened B-spline, a_{e_i} denotes the current element in the flattened B-spline from the expert planner, and a_{l_i} denotes the current

element in the flattened B-spline from the learned policy.

The function of the SSE loss is shown as follows:

$$L_{SSE}(a_e, a_l) = \sum_{i=1}^{n} (a_{e_i} - a_{l_i})^2. \tag{3.2}$$

To make the training smoother while reducing the possibility of overfitting the training dataset, a learning rate scheduler can be applied to the training process. The learning rate scheduler can dynamically adjust the learning rate between epochs. In this work, although we mainly implement and use a cosine decay scheduler with restarts, which was proposed by Loshchilov *et al.* [53], we also use the PyTorch scheduler for comparison in the proposed IL pipeline. We implement the cosine decay scheduler with restarts as the class *CustomCosineDecayRestarts*. The reason for developing *CustomCosineDecayRestarts* is that the one that comes from PyTorch is not flexible enough for us to tinker with some parameters we want in the scheduler. Table 3.2 and Table 3.3 show parameters for the PyTorch scheduler and *CustomCosineDecayRestarts* respectively.

Table 3.2: Parameters for the PyTorch Scheduler.

Parameter	Value
Number of iterations for the first restart (T_0)	500
Factor for increasing (T_i)	1
Minimum learning rate (η_{min})	1×10^5

Table 3.3: Parameters for *CustomCosineDecayRestarts*.

Parameter	Value
First decay steps	50000
Multiplier for deriving the number of iterations	1.5
Minimum learning rate over the initial learning rate	0.01

3.10. Inference

The current inference pipeline is developed with Python to make inferences with PyTorch models easier. It involves pre-processing of depth image and state information, forward passing the

processed depth image and state array to the network, reconstructing the flattened B-spline representation to ROS message, and publishing the message to the controller. However, we notice that the Python inference pipeline creates lag when planning with the learned policy. Therefore, similar to data collection, a C++ inference pipeline with LibTorch and TorchScript model is under development.

CHAPTER 4

EXPERIMENTS AND ANALYSIS

4.1. PID Controller Response

The expert is one of the foundational elements in IL. Executing the actions from the expert properly is crucial to provide correct demonstrations to the learner. In order to effectively learn from an expert planner, the controller needs to be able to responsively execute the incoming position command. During the development, we use the Ziegler–Nichols method [45] to tune the control gains of *kr_mav_control* [44]. Combined with the *set_z_offset* node, we are able to achieve responsive control based on actions from the expert. Figure 4.1 shows the visualization of responses of the PID controller with PlotJuggler [54] when flying to the goal position $(3.0, 3.0, 3.0)$.

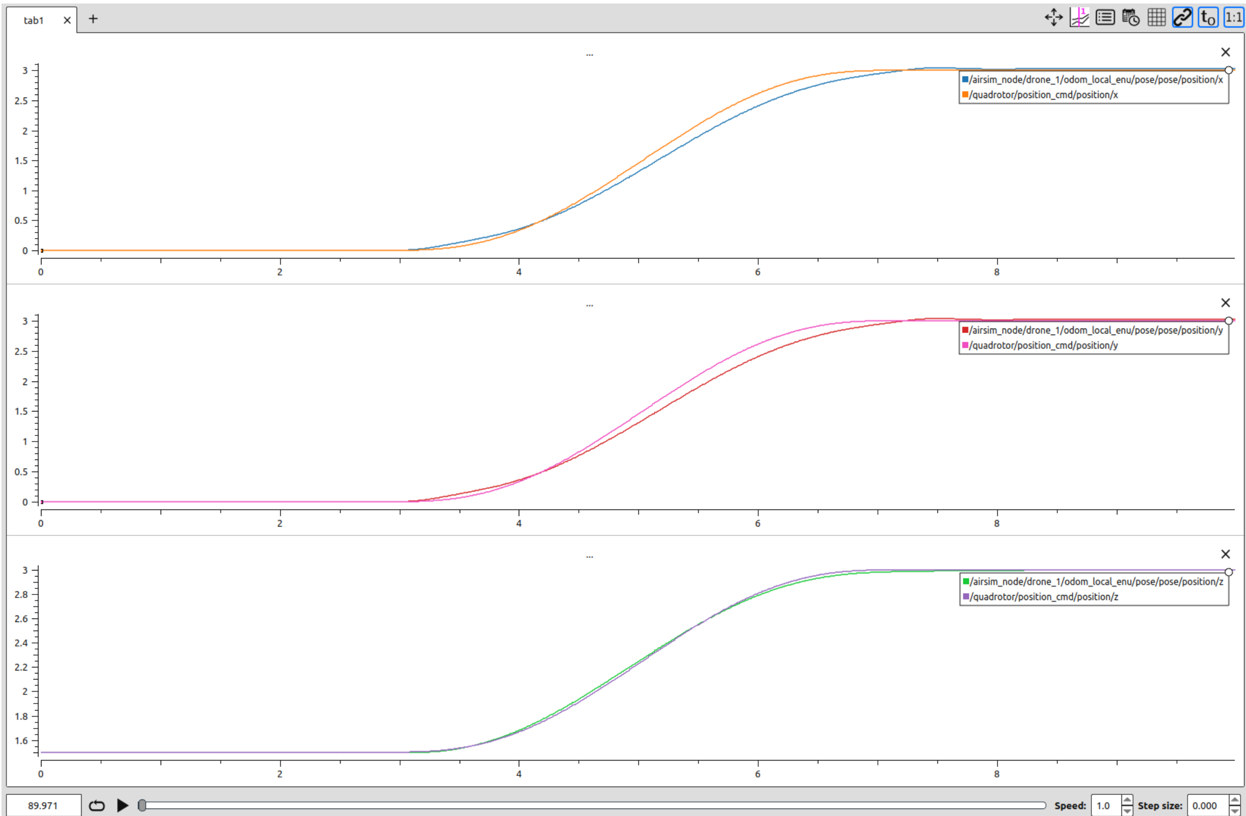

Figure 4.1: Plots of PID responses for position control.

4.2. Loss Functions

As mentioned in the Methodology chapter, two loss functions, which are MSE loss and SSE loss, are available during training. To test and compare the training performance between MSE loss and SSE loss, we perform two training trials on the same dataset with different loss functions. To fairly compare the performance of the two loss functions, both trials are performed with the Python data collection pipeline and an identical dataset with a size of 5.95 GB. The training and validation datasets are split in a 9:1 manner. Figure 4.2 shows the training loss and validation loss with respect to the number of epochs during training sessions with the two different loss functions.

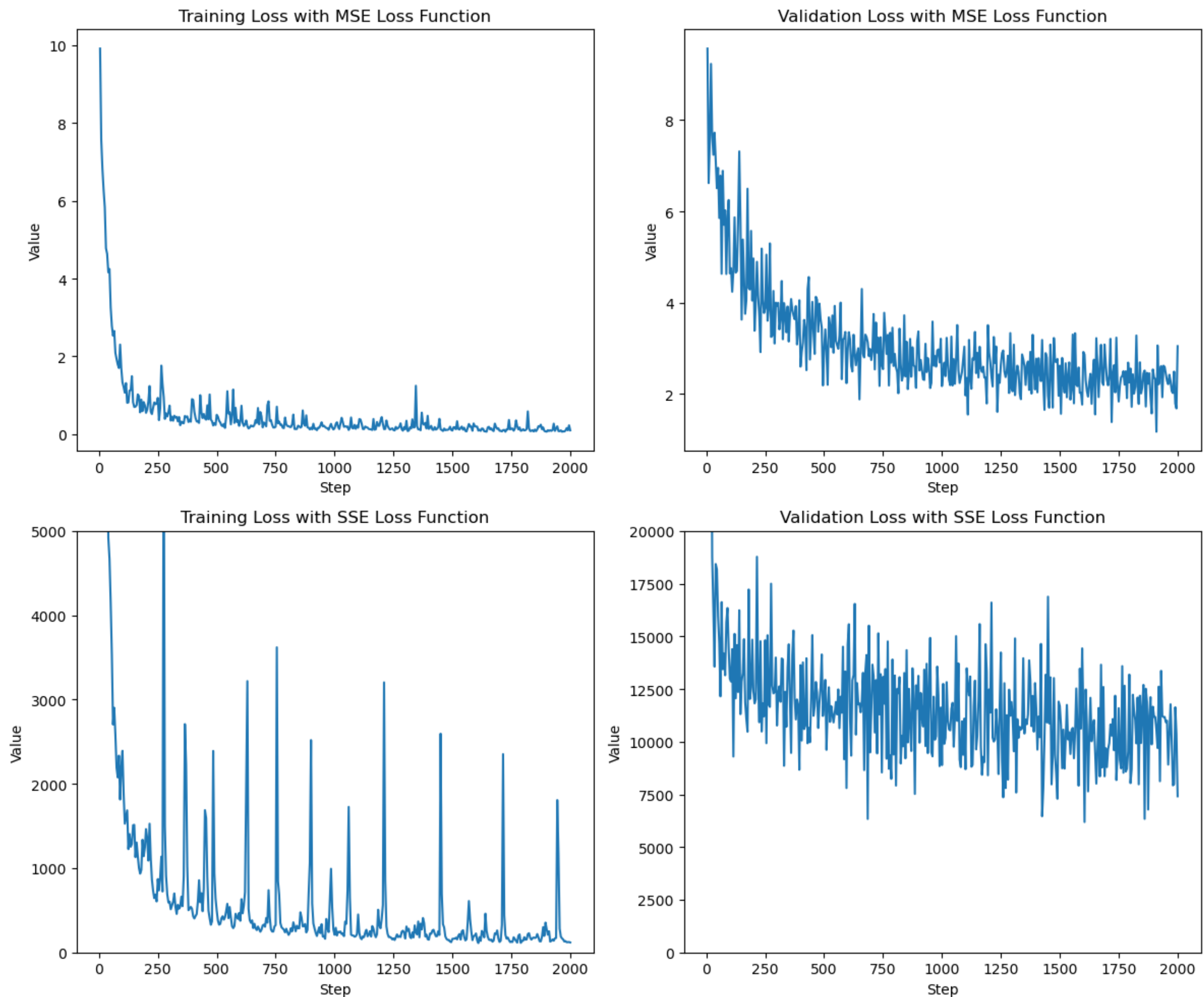

Figure 4.2: Plots of training loss and validation loss when training with MSE loss and SSE loss.

As the plots in Figure 4.2 show, overall, using the MSE loss can have more stable performance during training and can converge to a desired policy faster. Therefore, for all upcoming experiments and development, we choose to use the MSE loss.

4.3. Scheduler

Considering IL is extremely prone to overfitting, in the Methodology chapter, we mention that in order to reduce overfitting, we use schedulers for the learning rate during training. To better control the scheduling, instead of using the PyTorch scheduler implementation, we implement a cosine decay scheduler with restarts based on [53].

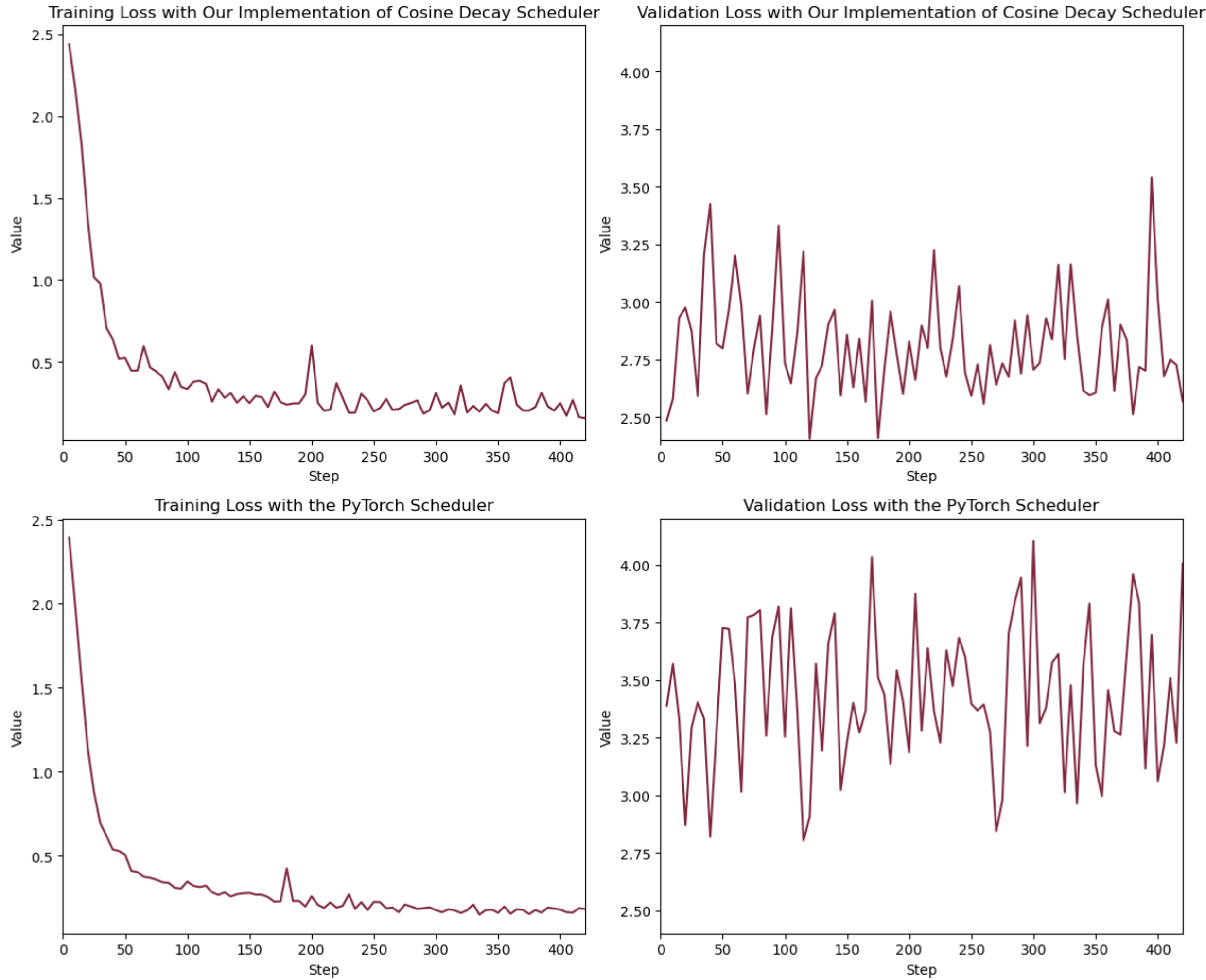

Figure 4.3: Plots of training loss and validation loss when training with our implementation of the cosine decay scheduler with restarts and PyTorch scheduler.

To test the performance of our custom implementation of the cosine decay scheduler with restarts, we perform experiments using our implementation and the PyTorch implementation on the same dataset used in the previous section. Figure 4.3 shows the experimental results. As suggested by the plots, although our custom implementation and the PyTorch implementation have comparable performance on training loss, the policy learned with our scheduler has a lower validation loss compared with the PyTorch scheduler, which indicates that the learned policy with our scheduler is less likely to be overfitted.

4.4. Data Collection

4.4.1. Python Data Collector

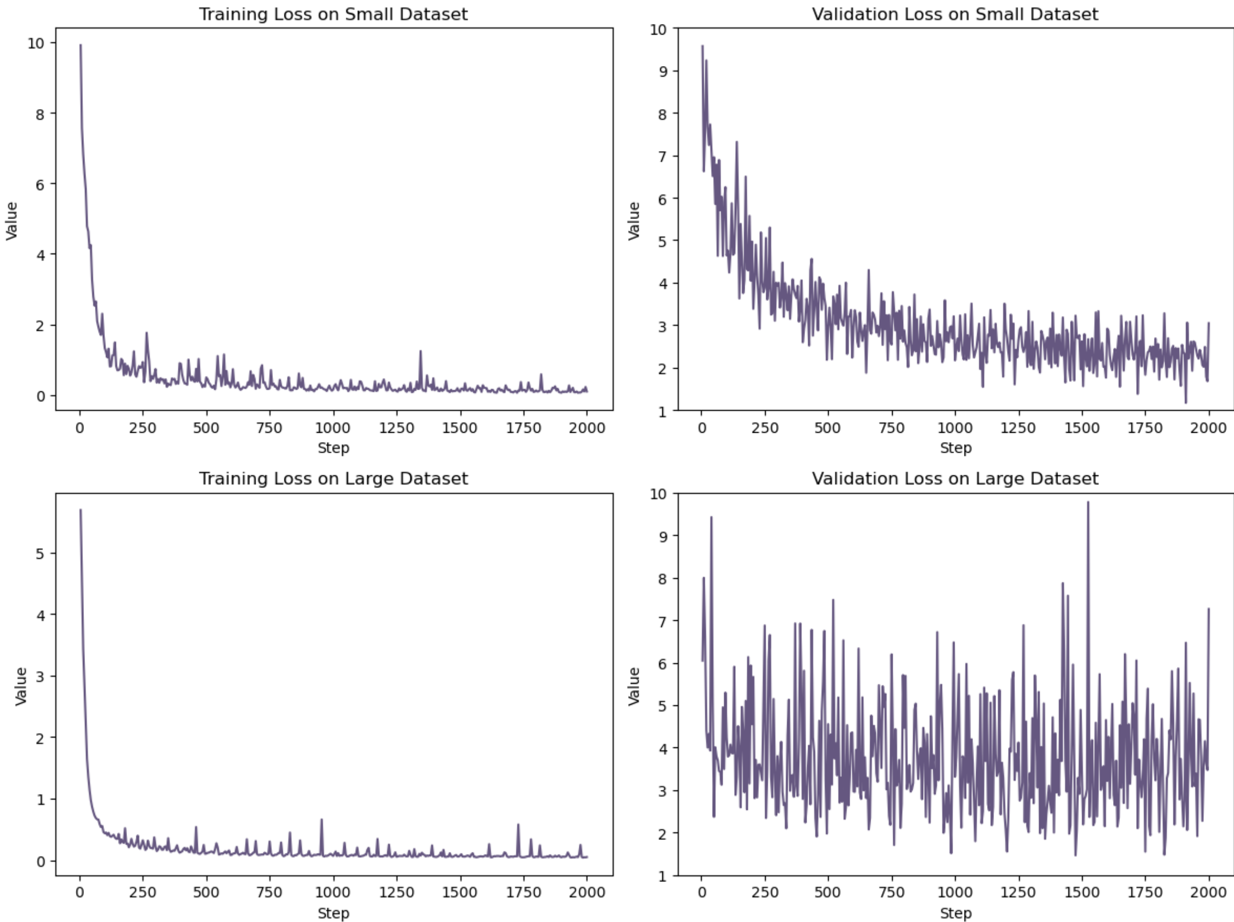

Figure 4.4: Plots of training loss and validation loss when training on datasets collected by the Python data collector with different sizes.

To be able to learn interactively while leveraging the efficiency of data loading with Numpy arrays in Python, initially, we develop a data collector with Python. We use this data collector to gather two datasets with sizes of 5.95 GB and 25.6 GB, respectively, to understand the effect of dataset size in our scenario.

As Figure 4.4 shows, although both training loss and validation loss look alright when training on the small dataset, the validation loss becomes extremely noisy when training on the large dataset. Combined with the smooth training loss on the large dataset, this indicates that despite using our implemented scheduler, the learned policy still overfits the training dataset and cannot generalize well on the validation dataset.

4.4.2. C++ Data Collector

We suspect that the undesired training performance on the large dataset is due to computational overhead in Python. At the beginning of data collection, since the dataset is small, saving demonstrations to the dataset does not burden the CPU. However, as the dataset becomes larger and larger, more and more intense computation is required to save the dataset. This issue is further amplified by the expert planner, which is the Fast Planner, and the simulation environment, which is in Unreal Engine 4, since they are all CPU-intensive applications.

Therefore, in order to reduce the latency as much as possible, we develop another data collector with C++ that prioritizes computation efficiency and synchronization of observations and actions while sacrificing interactive learning and ease of training with Python and Numpy.

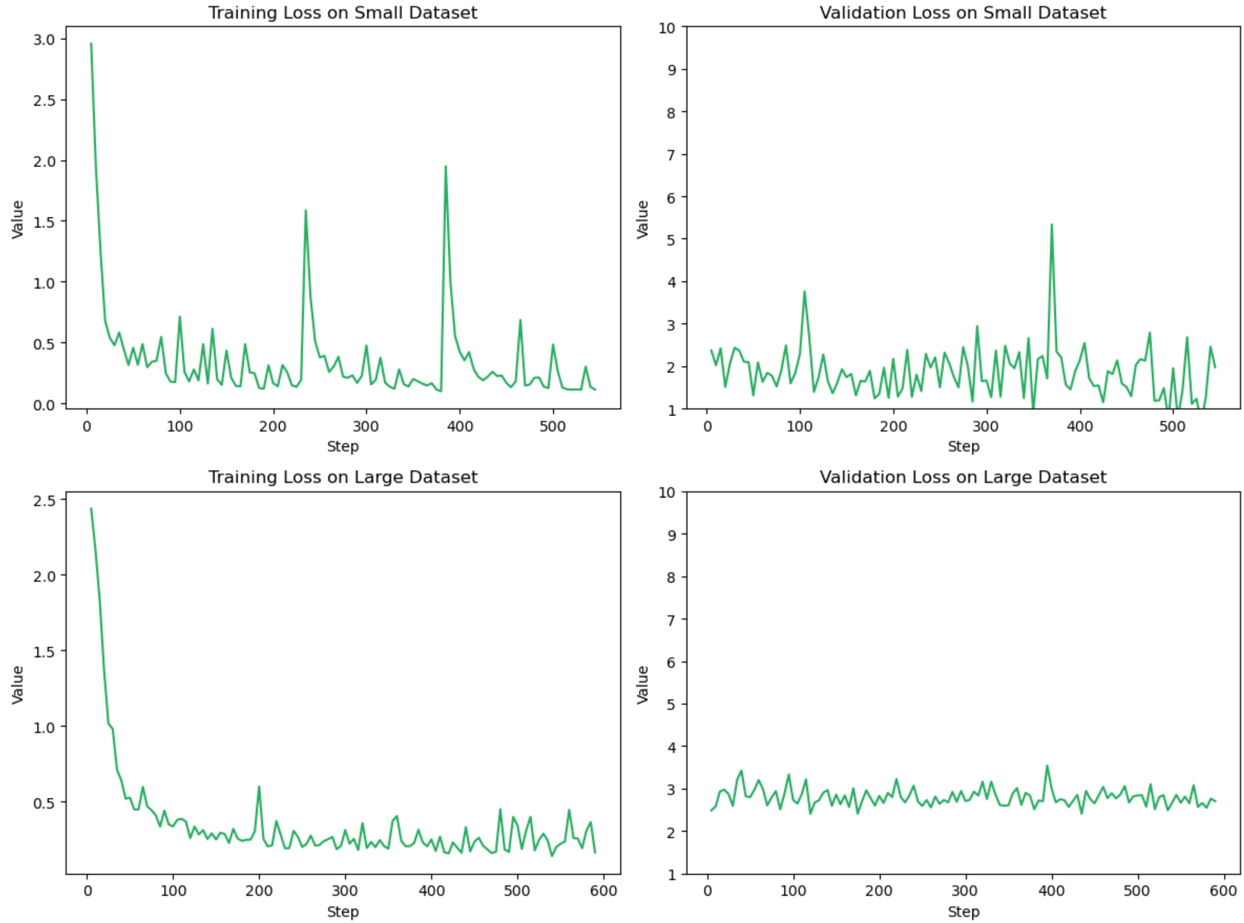

Figure 4.5: Plots of training loss and validation loss when training on datasets collected by the C++ data collector with different sizes.

We also gather a small dataset and a large dataset with the same sizes as the small and large datasets gathered by the Python data collector, respectively, to compare the improved synchronization between observations and actions of demonstrations from the expert. The results are visualized in Figure 4.5.

As Figure 4.5 shows, in comparison with Figure 4.4, the C++ data collector indeed results in lower validation loss. The training loss is also decreased when using the C++ data collector. This also verifies our previous suspicion that the cause of high validation loss is due to latency when collecting the demonstrations.

So far, our previous experiments primarily test and verify the best selections of the loss function, scheduler, and data collector during training. In the next section, the result of the final learned policy using the combination of the best choices from our previous experiments is shown and analyzed.

4.5. B-Spline Inference

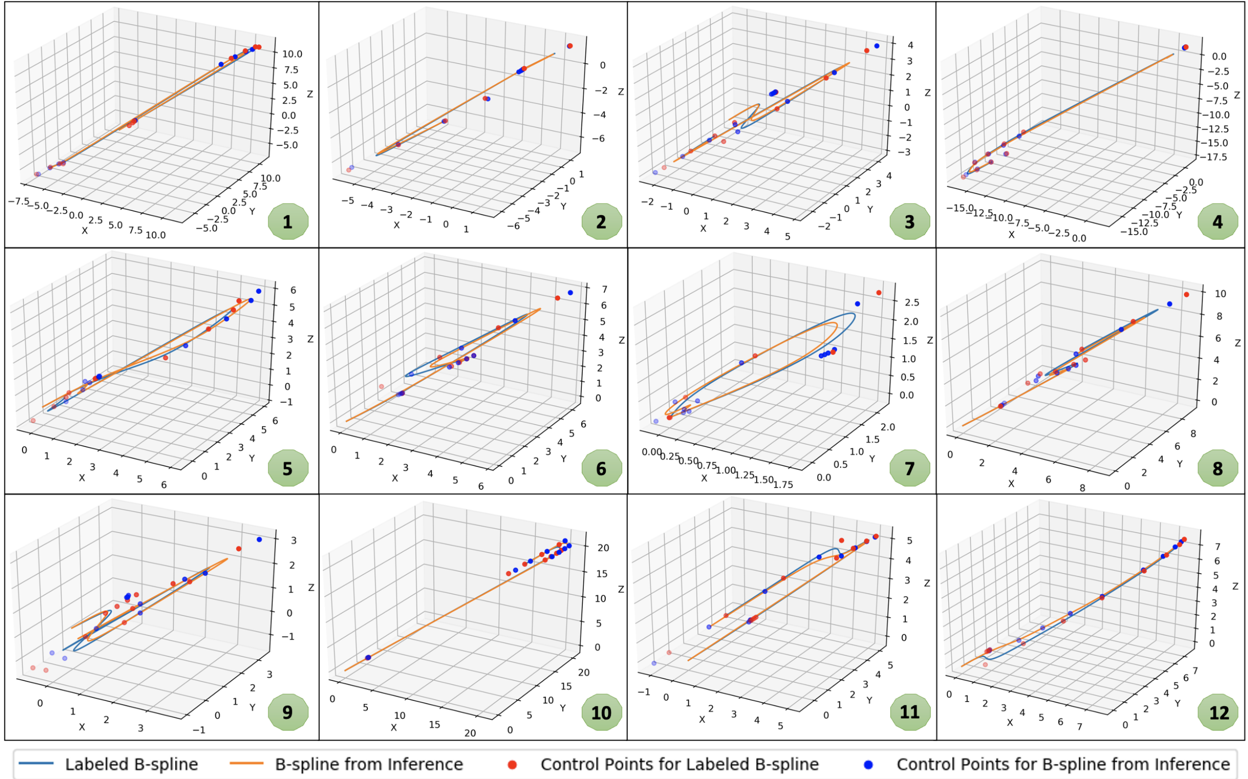

Figure 4.6: Visualization of 12 expert-labeled B-splines and B-splines from the inference of the learned policy.

To train the final policy, we use the MSE loss, our custom implementation of the cosine decay scheduler with restarts, and the C++ data collector. After training the final policy, we use the learned policy to perform several inferences and compare it with the expert both visually and statistically. In this thesis, the trajectories from inference in 12 trials are visualized with the trajectories from the expert under the same states and the same observations correspondingly, as shown in Figure 4.6.

To better compare the expert and the learner, instead of directly visualizing them in Rviz, we collect

Table 4.1: Evaluation of the 12 B-splines from the inference of the learned policy

Metrics	B-Spline Index					
	1	2	3	4	5	6
MSE Loss	0.45622	0.01187	0.06174	0.03680	0.18018	7.65659
Cosine Similarity	0.99299	0.99948	0.99492	0.99976	0.98994	0.75042
Bhattacharyya Distance	0.00763	0.00023	0.00238	0.00090	0.01232	0.07510
	7	8	9	10	11	12
MSE Loss	0.01363	0.06715	8.59117	0.15382	7.09543	0.12988
Cosine Similarity	0.99559	0.99873	0.90676	0.99995	0.71222	0.99465
Bhattacharyya Distance	0.02013	0.00541	0.04357	0.00010	0.13116	0.00587

the B-splines from the learned policy and the expert with Matplotlib in Python. As we can see in the plots, the B-splines from the inference of the learned policy are closely aligned with the B-splines from the expert, which indicates the learned policy can successfully learn from the expert.

Besides verifying and validating the learned policy visually, we also verify and validate it statistically. Since the essence of IL is to replicate behavior as closely as possible, we use several metrics to reflect the closeness of B-splines from the learned policy and the expert. As shown in Table 4.1, we use the MSE loss, the cosine similarity, and the Bhattacharyya distance to quantify the performance of the learned policy. The MSE loss, cosine similarity, and Bhattacharyya distance have average values of 2.03787, 0.9446175, and 0.0254 respectively. Overall, all metrics consistently show that the learned policy is good at mimicking the behaviors of the expert.

CHAPTER 5

LIMITATION AND FUTURE WORK

Within our proposed IL pipeline, so far we have developed data collectors in both Python and C++, a learning program in Python, and an inference program in C++. We also demonstrate that currently, the proposed IL pipeline can effectively learn from the Fast Planner, which is the expert we use throughout the experiment.

Although training in C++ would not be necessary, performing inference in C++ can also have improved performance in theory and should be implemented as well in our proposed IL pipeline. However, developing an IL pipeline from scratch with ROS, Unreal Engine 4, AirSim, etc., was time-consuming due to the tremendous amount of moving parts in the system. Thus, we do not have enough time for developing the C++ inference program. As a result, we leave developing the C++ inference program as one of the future directions of this project.

Also, as mentioned in the Introduction and Related Work section, one potential use case of IL for quadrotor motion planning is to learn from a computationally intensive expert for a low-cost learned policy that can be deployed onboard. We are interested in learning a low-cost policy from a jointly optimized perception-aware planner and deploying such a policy to a physical quadrotor in real-world scenarios. Although currently the learned policy is validated to be able to work in simulation, we have not performed any sim2real procedure and tested it in real-world scenarios. Therefore, in order to learn such a low-cost perception-aware policy and deploy it on a physical quadrotor, two future steps are figuring out a sim2real transfer pipeline for using the learned policy on a real quadrotor and combining the proposed IL pipeline with a perception-aware planner that we are currently working on.

Last but not least, since RL shares a lot of similarities in terms of assumptions and training setup with IL, incorporating several state-of-the-art RL algorithms, such as Proximal Policy Optimization (PPO) [55] and Soft Actor-Critic (SAC) [56], into the proposed IL pipeline would also be a good

future direction. Considering RL is frequently used for comparison with IL in existing literature, integrating RL into the proposed pipeline could also facilitate such comparisons in future work.

CHAPTER 6

CONCLUSION

In this work, we introduce an innovative IL pipeline that effortlessly melds with Kumar Robotics' existing software stack, harnessing the power of ROS and Unreal Engine 4. With modularity at its core, our IL pipeline implementation allows users to mix and match expert components, data collectors, schedulers, and loss functions based on their unique needs. To ensure a robust evaluation of each component and the entire pipeline, we've crafted a simulation environment using Unreal Engine 4, employing the Fast Planner as our expert and conducting comprehensive experiments.

Our findings reveal that, within this specific Fast Planner scenario, the optimal performance is achieved through the synergy of our C++ data collector, MSE loss, and an ingenious implementation of the cosine decay scheduler with restarts. Consequently, the learned policy flawlessly mirrors the Fast Planner's behavior.

Despite time constraints that prevented us from developing a C++ inference program, incorporating a perception-aware planner, and integrating RL algorithms into the IL pipeline, we envision a promising future. Our aspirations include enhancing efficiency with a C++ inference program, incorporating a perception-aware planner to bridge the sim2real gap for real-world quadrotor deployment, and blending in RL algorithms for a more comprehensive comparison between IL and RL methodologies.

BIBLIOGRAPHY

[1] L. Sanneman and J. A. Shah, "An empirical study of reward explanations with human-robot interaction applications," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 8956–8963, 2022.

[2] J. Eschmann, "Reward function design in reinforcement learning," *Reinforcement Learning Algorithms: Analysis and Applications*, pp. 25–33, 2021.

[3] Y. Song, M. Steinweg, E. Kaufmann, and D. Scaramuzza, "Autonomous drone racing with deep reinforcement learning," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 1205–1212.

[4] J. Tordesillas and J. P. How, "Deep-panther: Learning-based perception-aware trajectory planner in dynamic environments," *IEEE Robotics and Automation Letters*, 2023.

[5] X. Sun, M. Zhou, Z. Zhuang, S. Yang, J. Betz, and R. Mangharam, "A benchmark comparison of imitation learning-based control policies for autonomous racing," *arXiv preprint arXiv:2209.15073*, 2022.

[6] L. Quan, L. Han, B. Zhou, S. Shen, and F. Gao, "Survey of uav motion planning," *IET Cyber-systems and Robotics*, vol. 2, no. 1, pp. 14–21, 2020.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[8] ngc92, "Quadgym," 2017. [Online]. Available: https://github.com/ngc92/quadgym

[9] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org

[10] Epic Games, "Unreal engine." [Online]. Available: https://www.unrealengine.com

[11] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics: Results of the 11th International Conference*. Springer, 2018, pp. 621–635.

[12] Kumar Robotics, "Offical github organization for kumar robotics," 2023. [Online]. Available: https://github.com/KumarRobotics

[13] C. Sammut, *Behavioral Cloning*. Boston, MA: Springer US, 2010, pp. 93–97. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_69

[14] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning

methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[15] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010, pp. 661–668.

[16] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 627–635.

[17] M. Kelly, C. Sidrane, K. Driggs-Campbell, and M. J. Kochenderfer, "Hg-dagger: Interactive imitation learning with human experts," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 8077–8083.

[18] J. Spencer, S. Choudhury, M. Barnes, M. Schmittle, M. Chiang, P. Ramadge, and S. Srinivasa, "Expert intervention learning: An online framework for robot learning from explicit and implicit human feedback," *Autonomous Robots*, pp. 1–15, 2022.

[19] J. Spencer, S. Choudhury, A. Venkatraman, B. Ziebart, and J. A. Bagnell, "Feedback in imitation learning: The three regimes of covariate shift," *arXiv preprint arXiv:2102.02872*, 2021.

[20] R. Hoque, A. Balakrishna, E. Novoseller, A. Wilcox, D. S. Brown, and K. Goldberg, "Thriftydagger: Budget-aware novelty and risk gating for interactive imitation learning," *arXiv preprint arXiv:2109.08273*, 2021.

[21] S. Arora and P. Doshi, "A survey of inverse reinforcement learning: Challenges, methods and progress," *Artificial Intelligence*, vol. 297, p. 103500, 2021.

[22] N. Aghasadeghi and T. Bretl, "Maximum entropy inverse reinforcement learning in continuous state spaces with path integrals," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011, pp. 1561–1566.

[23] C. Finn, S. Levine, and P. Abbeel, "Guided cost learning: Deep inverse optimal control via policy optimization," in *International conference on machine learning*. PMLR, 2016, pp. 49–58.

[24] J. Ho and S. Ermon, "Generative adversarial imitation learning," *Advances in neural information processing systems*, vol. 29, 2016.

[25] L. Quan, L. Han, B. Zhou, S. Shen, and F. Gao, "Survey of uav motion planning," *IET Cyber-Systems and Robotics*, 2020.

[26] S. M. LaValle, "Rapidly-exploring random trees : a new tool for path planning," *The annual research report*, 1998.

[27] C. Nissoux, T. Simeon, and J.-P. Laumond, "Visibility based probabilistic roadmaps," in *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients (Cat. No.99CH36289)*, vol. 3, 1999, pp. 1316–1321 vol.3.

[28] S. Karaman, M. R. Walter, A. Perez, E. Frazzoli, and S. Teller, "Anytime motion planning using the rrt," in *2011 IEEE international conference on robotics and automation.* IEEE, 2011, pp. 1478–1483.

[29] A. Perez, R. Platt, G. Konidaris, L. Kaelbling, and T. Lozano-Perez, "Lqr-rrt*: Optimal sampling-based motion planning with automatically derived extension heuristics," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 2537–2542.

[30] N. M. Amato, O. B. Bayazit, L. K. Dale, C. Jones, and D. Vallejo, "Obprm: An obstacle-based prm for 3d workspaces," in *Proc. Int. Workshop on Algorithmic Foundations of Robotics (WAFR)*, 1998, pp. 155–168.

[31] R. Bohlin and L. Kavraki, "Path planning using lazy prm," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 1, 2000, pp. 521–528 vol.1.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[33] D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.

[34] M. Zucker, N. Ratliff, A. D. Dragan, M. Pivtoraiko, M. Klingensmith, C. M. Dellin, J. A. Bagnell, and S. S. Srinivasa, "Chomp: Covariant hamiltonian optimization for motion planning," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1164–1193, 2013.

[35] HKUST-Aerial-Robotics, "Fast-planner," 2022. [Online]. Available: https://github.com/HKUST-Aerial-Robotics/Fast-Planner

[36] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen, "Robust and efficient quadrotor trajectory generation for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3529–3536, 2019.

[37] E. Camci, D. Campolo, and E. Kayacan, "Deep reinforcement learning for motion planning of quadrotors using raw depth images," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–7.

[38] S. Kim, J. Park, J.-K. Yun, and J. Seo, "Motion planning by reinforcement learning for an unmanned aerial vehicle in virtual open space with static obstacles," in *2020 20th International Conference on Control, Automation and Systems (ICCAS)*, 2020, pp. 784–787.

[39] E. Kaufmann, L. Bauersfeld, and D. Scaramuzza, "A benchmark comparison of learned control policies for agile quadrotor flight," in *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 10 504–10 510.

[40] J. Tordesillas and J. P. How, "Deep-panther: Learning-based perception-aware trajectory planner in dynamic environments," *IEEE Robotics and Automation Letters*, vol. 8, no. 3, pp. 1399–1406, 2023.

[41] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Conference on Robot Learning*, 2020.

[42] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033.

[43] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," 2019. [Online]. Available: https://drake.mit.edu

[44] Kumar Robotics, "kr_mav_control," 2022. [Online]. Available: https://github.com/KumarRobotics/kr_mav_control

[45] V. V. Patel, "Ziegler-nichols tuning method: Understanding the pid controller," *Resonance*, vol. 25, no. 10, pp. 1385–1397, 2020.

[46] S. Liu, N. Atanasov, K. Mohta, and V. Kumar, "Search-based motion planning for quadrotors using linear quadratic minimum time control," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 2872–2879.

[47] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[48] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[49] X. Sun, S. Yang, and R. Mangharam, "Mega-dagger: Imitation learning with multiple imperfect experts," *arXiv preprint arXiv:2303.00638*, 2023.

[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[51] C. Sammut and G. I. Webb, Eds., *Mean Squared Error*. Boston, MA: Springer US, 2010, pp. 653–653. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_528

[52] D. L. Mohr, W. J. Wilson, and R. J. Freund, "Chapter 6 - inferences for two or more means," in *Statistical Methods (Fourth Edition)*, fourth edition ed., D. L. Mohr, W. J. Wilson, and R. J. Freund, Eds. Academic Press, 2022, pp. 243–299. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128230435000060

[53] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," *arXiv preprint arXiv:1608.03983*, 2016.

[54] D. Faconti, "Plotjuggler," 2023. [Online]. Available: https://github.com/facontidavide/PlotJuggler

[55] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[56] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.