A MODULAR FRAMEWORK FOR TRAINING AUTONOMOUS SYSTEMS VIA HUMAN INTERACTION

A Thesis

by

RITWIK BERA

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	John Valasek
Committee Members,	Gregory Chamitoff
	Daniel Selva
	Dileep Kalathil
Head of Department,	Ivett A. Leyva

May 2022

Major Subject: Aerospace Engineering

ABSTRACT

Recent successes in the field of robot learning have combined reinforcement learning algorithms and deep neural networks. Yet, reinforcement learning has not been widely applied to robotics and real world scenarios. This can be attributed to the fact that current state-of-the-art, end-to-end reinforcement learning approaches usually require an impractically large number of data samples to converge to a satisfactory policy. They are often subject to catastrophic failures during training since they need to be able to 'explore' their state space. The environmental interactions require some finite time each and coupled with the risky exploration stage in RL, imitation learning is a much more feasible candidate for learning on hardware platforms. However, imitation learning's simplest form, behavior cloning, also requires a fairly large number of demonstration trajectories to learn an effective policy. Humans are able to accomplish the same real-world tasks with much fewer samples that may be provided in the form of demonstrations. Just a few partial demonstrations in the form of corrective action are often enough to achieve reliable performance. This suggests that humans implicitly utilize other modalities of information, such as eye-gaze, apart from just motor actuation data. This thesis investigates both the incorporation of such modalities in training data, as well as when and how this training data is collected, such as through corrective action, partial demonstrations etc. The need to perform multiple experiments across both hardware and simulation-based platforms with different observation and action-space configurations creates a need for a software library that is modular, extensible and universal in design. One of the primary contributions of this thesis is the development of a unique software library that enables single-operator, fault-tolerant operation of data collection experiments on vehicle platforms. It permits a safe rollout of machine learning models as controllers while keeping a human-in-the-loop. This thesis demonstrates how the usage of certain fundamental principles of program design and software architectural patterns results in a high level of common code execution across experimental configurations. The high level of common code execution is demonstrated through measurement of configuration-agnostic code coverage across different experimental runs and is shown to exceed 50%. This simplifies the setup phase for

any experiment involving machine learning and autonomous systems. The same software library is used to perform the experiments investigating effects of additional human sensory modalities in the training data as well as sampling schemes to collect data through interventions. Improvements in key metrics are such as task completion rate are observed when eye-gaze is incorporated as a training signal. Task completion rate is shown to increase by 13.7 percentage points with the incorporation of eye-gaze. Data sampling changes by shifting to intervention-guided data collection also results in higher task-completion rates (65 % as opposed to 35 % earlier). This improvement in task completion rate is accompanied with a lower expert data requirement. When compared to an intervention-free baseline, the intervention-guided data collection routine needs 27.1% fewer expert samples while having a more proficient policy.

ACKNOWLEDGMENTS

I would like to acknowledge and thank who contributed to the development of this research and the successful completion of this thesis. Dr. John Valasek, who has been an exceptional advisor and mentor not only on academic matters, but also on leadership and management, which I see as invaluable skills for my future career. My committee members, Dr. Gregory Chamitoff, Dr.Dileep Kalathil, and Dr. Daniel Selva for their guidance when grounding the research fundamentals and support when defining the research direction.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor John Valasek, Gregory Chamitoff, and Daniel Selva of the Department of Aerospace Engineering and Professor Dileep Kalathil of the Department of Electrical and Computer Engineering. The data analyses for Chapter 5 was conducted in consultation with Dr. Vinicius G. Goecks from the U.S. Army Research Laboratory.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was supported by a graduate research assistantship and a fellowship from the Department of Aerospace Engineering at Texas A&M University. The research was sponsored by the U.S. Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-20-2-0114. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

NOMENCLATURE

AAAI	Association for the Advancement of Artificial Intelligence
a_t	Action vector at time t
ANN	Artificial Neural Network
BCE	Binary Cross-Entropy
CE	Cross-Entropy
CNN	Convolutional Neural Network
CoL	Cycle-of-Learning for Autonomous Systems
BC	Behavior Cloning
CPU	Central Processing Unit
DAgger	Dataset Aggregation
Deep RL	Deep Reinforcement Learning
DL	Deep Learning
DOF	Degree-of-Freedom
FPV	First-Person View
HRI	Human-Robot Interaction
IL	Imitation Learning
IMU	Inertial Measurement Unit
FCN	Fully-Connected Network
GPU	Graphics Processing Unit
KL	Kullback-Leibler
$\mathcal{L}$	Loss function
LfD	Learning from Demonstrations

LfI	Learning from Interventions
MDP	Markov Decision Process
MLP	Multilayer Perceptron
MSE	Mean Squared Error
MTL	Multi Task Learning
o_t	Observation vector at time t
π	Policy
π_0	Initial Policy
POMDP	Partially Observable Markov Decision Process
ReLU	Rectified Linear Unit
RGB	Visible spectrum color code
RL	Reinforcement Learning
RMSProp	Root Mean Square Propagation
RNN	Recurrent Neural Network
s_t	State vector at time t
sUAS	Small Unmanned Air System
TAMU	Texas A&M University
Tanh	Hyperbolic Tangent
VAE	Variational Autoencoder

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Problem Overview

1.2 Introduction

Most real-world tasks that are desired to be performed by robots, quickly exceed the capability of designers to handcraft optimal or even successful policies due to the inability to foresee the amount of variability in the state-space of such systems. Hand-engineered control policies once designed can no longer adapt or improve themselves as they gain more experience while attempting the task. This makes them brittle and unusable for many sensorimotor tasks [41].

On the other hand, data-driven approaches and learning algorithms are well suited to solve high-level prediction and control problems in an information-rich world. Learning algorithms are able to learn directly from examples, to search for an underlying pattern.

Reinforcement learning (RL) [73] is a machine learning (ML) approach concerned with how intelligent agents ought to take actions in an interactive environment in order to maximize the notion of cumulative reward. It involves both taking random actions to understand environmental dynamics (exploration) and repeating actions known to be optimal through past experience (exploitation). It has long been shown to work on specific problems [14, 36] in specific scenarios [38] with well-designed objective functions (reward functions, in the RL literature) where interactions with the environment [47] have associated costs which are easily available.

However, in real-world robotic applications explicit reward functions are non-existent (unless previously designed), and interactions with the hardware are expensive and susceptible to catastrophic failures. It can even be infeasible to construct appropriate objective functions in many cases. This motivates leveraging human interaction to reduce the amount of high-risk interactions with the environment, and to safely shape the behavior of intelligent robotic agents.

Consequently, the proposed research centers around supervised learning-based approaches to human-in-the-loop learning. One of the most widely studied modes of human interaction studied is

teleoperation, which can be further differentiated into *Learning from Demonstrations* and *Learning from Interventions*.

- *Learning from Demonstrations* (LfD) [6, 52, 26] can be used to provide a more directed path to these intended behaviors by utilizing examples of humans performing the task. However, given that it is typically performed offline, it does not provide a mechanism for corrective or preventative inputs when the learned behavior results in undesirable or catastrophic outcomes, potentially due to unseen states.

- *Learning from Interventions* (LfI), where a human acts as an overseer while an agent is performing a task and periodically takes over control or intervenes when necessary, can provide a method to improve the agent policy while preventing or mitigating catastrophic behaviors [65].

A challenge in training and utilizing imitation learning agents is learning from as few demonstrations as possible to minimize the burden on end-users and allow for real-time learning, while resolving ambiguity in user intentions and avoiding model overfitting. [26, 32]. This can be resolved in two ways:

The first would be selecting informative features for the input space. Observations that a human would depend on to infer the right action would make for a good candidate to be a part of the input to the policy model. This feature selection can occur through a variety of methods, the most commonly used being *Recursive Feature Elimination*. The second would be to incorporate additional interaction modalities such as eye-gaze data (as investigated in this research) from the demonstrator (apart from the state-action pairs obtained from teleoperation). This is often enabled by training a model on multiple prediction tasks simultaneously so that the model implicitly learns the most relevant features that help in all the related prediction tasks. This approach is commonly known as *Multi-Task Learning* [16], which is utilized in this research in by concurrently training the model on two prediction tasks: behavior cloning and gaze prediction.

However, both these approaches require the ability to rapidly prototype different neural network

models, testing across multiple vehicular testbeds, sensor configurations and data pipelines. There exists a major need for a middleware that stitches together simulation environments, hardware APIs and machine learning libraries while providing for real-time human supervision and maintaining fail-safe safety overrides. The proposed research aims to fill this software gap.

A *framework* is an integrated collection of components that collaborate to produce a reusable architecture for a family of related applications. A *software framework* is a concrete or conceptual platform where common code with generic functionality can be selectively specialized or overridden by developers or users. Frameworks take the form of libraries, where a well-defined application program interface (API) is reusable anywhere within the software under development. A user can extend the framework but not modify the code. The purpose of software framework is to simplify the development environment, allowing developers to dedicate their efforts to the project requirements, rather than dealing with the framework's mundane, repetitive functions and libraries and the proposed research involves the development of such a framework for human-in-the-loop learning for robotic tasks.

1.3 Relevant Literature

1.3.1 Software Frameworks for Learning-based Robotic Systems

A number of popular tools exist that are purpose-built for either research in robotics or research in machine learning. On the machine learning side, this includes automatic differentiation libraries like *PyTorch* [53] and visualization tools like *Tensorboard*. On the robotics side this includes middleware like ROS [56] that enable and regulate the operation of an autonomous system itself and simulation engines like Gazebo [37] and AirSim [68]. However there exists a vacuum in the space that stitches these two research domains together. A middleware framework that enables agile experimentation and interfaces with black-box simulation engines and machine learning libraries and operates seamlessly across environments and platforms is missing from the landscape.

1.3.2 Multi Task Learning

Multi-task learning (MTL) comes in many guises: joint learning; learning to learn; and learning with auxiliary tasks are only some names that have been used to refer to it. Generally as soon as more than one loss function is being optimized it is effectively a case of doing multi-task learning (in contrast to single-task learning). Even if only optimizing one loss is the typical case, chances are there is an auxiliary task that will help improve upon the main task. Caruana [15] summarizes the goal of MTL succinctly: "MTL improves generalization by leveraging the domain-specific information contained in the training signals of related tasks". Multi-task learning has another advantage on inference since it allows for structured predictions (predictions that take structural biases of the problem at hand into account) with only a single forward pass being required [71]. This reduces the risk of overfitting.

Multi-task learning helps the model focus its attention on input features that actually matter since optimizing multiple objectives simultaneously forces the model to learn relevant features that are invariant across each objective [16, 3, 62].

There are various reasons that warrant the use of MTL. Machine learning models generally require a large volume of data for training. However, this often result in ending up with many tasks for which the individual datasets are insufficiently sized to achieve good results. In this case, if some of these tasks are related, e.g. predicting many diseases and outcomes from a patient's profile, one can merge the features and labels into a single larger training dataset, so that it is possible to take advantage of shared information from related tasks to build a sufficiently large dataset.

MTL also improves the generalization of the model. Using MTL, the information learned from related tasks improves the model's ability to learn a useful representation of the data, which reduces overfitting and enhances generalization. MTL can also reduce training time because instead of investing time training many models on multiple tasks, a single model is trained.

MTL is crucial in some cases, such as when the model will be deployed in an environment with limited computational power. Since machine learning models often have many parameters that need to be stored in memory, for applications where the computational power is limited (e.g.

edge devices), it is preferable to have a single MTL network with some shared parameters, as opposed to multiple single-task models doing related tasks. For example, in self-driving cars, multiple perception tasks need to be executed in real-time, including object detection and depth estimation. Having multiple neural networks doing these tasks individually requires computational power that might not be available. Instead, using a single model trained with MTL reduces the compute hardware's requirements and speeds up inference.

However, multi-task learning requires that the tasks that are jointly learnt be carefully chosen. Tasks that are related can help incorporate relevant structural biases associated with the overall problem and guide the optimization process to a general and well-fit solution. On the other hand if tasks that are unrelated are jointly learnt, the optimization process may be distracted from the intended goal and lead to a suboptimal solution.

1.3.3 Learning from Demonstrations

An example of prior work that attempts to augment learning from demonstrations with additional human interaction is the Dataset Aggregation *(DAgger)* algorithm [60]. DAgger is an iterative algorithm that consists of two policies: a primary agent policy that is used for direct control of a system, and a reference policy that is used to generate additional labels to fine-tune the primary policy towards optimal behavior. As opposed to 'on-the-fly' interventions, the reference policy's actions are not taken but are instead aggregated and used as additional labels to re-train the primary policy for the next iteration. A further extension of the work applied *Dagger* to collision avoidance [61]. There are some drawbacks to this approach. Because the human observer is never in direct control of the policy, safety is not guaranteed since the agent has the potential to visit previously unseen states, which could cause catastrophic failures [61]. Additionally, the subsequent labeling by the human can be suboptimal in terms of the amount of data recorded; perhaps due to recording and annotating more data in 'easy' states (in the observation space) than is needed to learn an optimal policy and lacking in annotations/action-labels for the more 'critical states'.

1.3.4 Data Augmentation for Imitation Learning

An example of prior work that attempts to augment learning from demonstrations with additional human interaction is the Dataset Aggregation *(DAgger)* algorithm [60]. DAgger is an iterative algorithm that consists of two policies: a primary agent policy that is used for direct control of a system, and a reference policy that is used to generate additional labels to fine-tune the primary policy towards optimal behavior. As opposed to 'on-the-fly' interventions, the reference policy's actions are not taken but are instead aggregated and used as additional labels to re-train the primary policy for the next iteration. A further extension of the work applied *Dagger* to collision avoidance [61]. There are some drawbacks to this approach. Because the human observer is never in direct control of the policy, safety is not guaranteed since the agent has the potential to visit previously unseen states, which could cause catastrophic failures [61]. Additionally, the subsequent labeling by the human can be suboptimal in terms of the amount of data recorded; perhaps due to recording and annotating more data in 'easy' states (in the observation space) than is needed to learn an optimal policy and lacking in annotations/action-labels for the more 'critical states'.

1.3.5 Visual Attention and Imitation Learning

Saran et al. highlight the importance of understanding gaze behavior for use in imitation learning. The authors characterized gaze behavior and its relation to human intention while demonstrating object manipulation tasks to robots via kinesthetic teaching and video demonstrations. For this type of goal-oriented task, Saran et al. show that users mostly fixate at goal-related objects, and that gaze information could resolve ambiguities during subtask classification, reflecting the user internal reward function. In their study the authors highlight that gaze behavior for teaching could vary for more complex tasks that involve search and planning, such as the the one presented in this proposal.

Augmenting imitation learning with eye gaze data has been shown to improve policy performance when compared to unaugmented imitation learning [79] and improve policy generalization [43] in visuomotor tasks, such as playing Atari games and simulated driving. [76] also proposed a periphery-fovea multi-resolution driving model to predict human attention and improve accuracy in

a fixed driving dataset.

Attention Guided Imitation Learning [79], or AGIL, collected human gaze and joystick demonstrations while playing Atari games and presented a two-step training procedure to incorporate gaze data in imitation learning. Training order is first a gaze prediction network modeled as a human-like foveation system, then a policy network using the gaze predictions represented as attention heatmaps.

1.4 Research Objectives

The proposed research aims to investigate ways of augmenting behavior cloning through incorporation of either additional types of sensory input or differing modalities of teleoperation. In order to meet the requirements of such flexible experimentation, this research aims to develop a software-based solution specifically designed for such tasks. The solution is a cross-platform software framework and API optimized for experimental research in human-in-the-loop learning with minimal coding overhead needed for the end user. When human eye gaze data is post-processed (for example, through filtering or label correction) to have a high signal-to-noise ratio (SNR), it can be used to train low-dimensional representations of high-dimensional visual observations, which can be then used as input to a policy. The generation of efficient low-dimensional representations that capture the task context also depends on the type of models trained to generate those representations. Models based on Graph Neural Networks are better suited to generate scene representations where visual attention belongs to a discrete set of objects in the scene whereas convolutional attention networks are more suited in scenarios where visual attention is less structured and is more like a spatial-blob.

This research includes two case studies that investigate the effect of training with teleoperation data collected in the form of interventions and scene representation features during policy rollouts. The first case study investigates how human eye gaze can be incorporated into existing imitation learning architectures to improve performance when learning from expert demonstrations. This includes learning scene representation from gaze data during training, which, due to the risk of policies easily overfitting to the signal noise (notwithstanding the high compute requirements),

prevents us from training a reliable policy in real-time. In order for the improvements to be apparent when the policy is trained in real-time, the second case study uses fixed and domain-invariant scene representations instead of learnt scene representations. As a result, we validate improvements provided by interactive learning on hardware by abstracting the scene representation part out of the problem and using just AprilTag detections as scene representation. This provides a reliable high SNR input and makes real-time learning feasible. In the future, the AprilTags can also be replaced by a well developed task-specific scene representation model trained with a reliable gaze-based feedback signal in an offline learning setting.

1.5 Research Contributions

The contributions of this research are:

1. Development and evaluation of a novel multi-head policy model trained on a dataset augmented with eye-gaze data, using an asynchronous quadrotor navigation task with high-dimensional state and action spaces in a high-fidelity, photo-realistic simulation environment. The proposed model is based in the concept of multi-task learning and leverages training feedback from passive sensory data to improve behavior cloning.

2. A novel modular and extensible software framework that facilitates research in human-in-the-loop machine learning by providing for agile experimentation across vehicle, sensor and environment platforms. The framework is provided as an installable Python library that can be setup for different experimental configurations through a command-line interface (CLI).

3. A hardware demonstration using the Parrot Anafi vehicle platform that showcases the effectiveness of training with a combination of demonstration and interventions over just demonstrations. The goal is to show that for the same number of human interaction episodes, timely interventions provide a stronger training signal than pure demonstrations as they effectively cover gaps in the model's understanding of the task.

1.6 Organization

This thesis is organized as follows. Chapter 2 details Behavior Cloning and Imitation Learning in discrete and continuous spaces and how these algorithms are used to replicate human behavior. Chapter 3 introduces and details one of the core contributions of this work: a software library built from scratch and specifically engineered to support modular experimentation for machine learning on autonomous vehicles in real-time. Chapter 4 introduces a theoretical contribution of this thesis: Gaze-Informed Multi-Objective Imitation Learning from Human Demonstrations. It introduces a mechanism based on Multi-Task Learning (MTL), to incorporate additional sensory or input modalities to make behavior cloning more sample-efficient and less prone to overfitting. Chapter 5 explains how to efficiently combine human demonstrations and interventions on learning algorithms to increase task performance while using fewer human samples. The main study case is learning from human interaction autonomous landing controllers for unmanned aerial systems.

Conclusions from the various applications and recommendations for future work are summarized in Chapter 6.

2. DATA-DRIVEN SHAPING OF AGENT BEHAVIOR

This chapter introduces the elementary basics of machine learning and targets supervised learning in particular. The fundamentals are built upon to show how autonomous agents can learn to imitate demonstrated behavior by framing the learning problem as a sequential decision making problem. Beyond covering the important concepts in machine learning, this chapter also introduces various mathematical formulations of the sequential decision making problem for intelligent systems and their respective pros and cons.

2.1 Background

Machine Learning is a field of research that aims to use statistical techniques and models to discover underlying patterns in data [7]. The available data is often split into different sets, namely the *training set*, *testing set* and the *validation set*. The training set includes the set of data points used to learn from to fit the model/function [12]. The validation set is used to repeatedly evaluate prediction performance of a model. This helps in determining various hyperparameters during the training phase so that the trained model is be able to *generalize* to new unseen inputs which form the test set [12]. The type of machine learning is dependent on the form of the available data and can be any of: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. In a problem set up for supervised learning, the training set contains data inputs and desired outputs. During the training process, model's parameters are incrementally updated in order to produce outputs closer to the desired ones given the same input data. In problems where input-output pairs are readily available, supervised learning is primarily used.

In a problem requiring unsupervised learning no labels are provided and just input data points are known. Unsupervised learning is often used in problems where the goal is to either uncover the fundamental structure of the distribution of data or to understand the spatial spread of the different families of data points present in the overall set. This is often used in problems requiring grouping of clustering of data without having access to explicit labels. The distinction between

supervised learning and unsupervised learning is not a hard line and middle-of-the-road techniques like semi-supervised learning exist that combine a large amount of unlabeled data with a small amount of labeled data to identify the right predictions for each of the data points in the identified clusters [12].

Reinforcement learning is a branch of machine learning that does not operate on the basis of a fixed training set. Training data is collected through a trial-and-error driven interaction of an untrained model with an environment that provides data points and a reward signal to learn from. This is often useful in scenarios where training data might not be explicitly provided but can be cheaply obtained over time through interaction.

2.2 Supervised Learning

The supervised learning problem is a function fitting problem that attempt to fit to a given set of input-output pairs. [7] Mathematically this given set of input-output pairs (also known as training set) can be represented as:

$$\mathcal{D} = \{(x^n, y^n), n = 1, \ldots, N\},$$

The goal of the problem is to learn a functional transformation between the input x and output y such that, given an unseen input x' (not present in $\mathcal{D}$), the predicted y' output is close to its underlying true value. This is of course assuming that the pair (x', y') belongs to the same underlying data distribution as $\mathcal{D}$.

If the output y can be only one of a finite set of discrete outcomes, the supervised learning problem is called *classification*. On the other hand, if the output y is a continuous variable, the supervised learning problem is called *regression*. This distinction is important to define the loss function used to compute the misfit between true and predicted model, which guides the optimization process used to learn the model.

2.2.1 Modelling and Fitting Predictors

In the context of machine learning, a *model* is the mathematical representation of any underlying function or distribution it aims to approximate. This underlying distribution is also the distribution

that the training set is sampled from. Appropriate sampling is necessary so that the model learns to to generalize to unseen data samples from the same distribution and accurately predict the distribution values/function outputs. Sampling from only a few select regions of the state-space might lead to learning incorrect or 'overfit' models. Models, just like any other function are differentiated based on their structure. The number of layers, type of activation units etc. define the structure of a model while the actual values of the layer weights uniquely define a neural network. The structure is chosen based on certain assumptions that help to accurately model a problem while not making things overly complex. These assumptions include but are not limited to factors such as the degree of non-linearity in the training data and the level of noise in the collected data among other things. It is important to note that neural networks always include activate functions that help them model non-linearities since a stack of neural network layers without activations is just a linear function which can only model linear data.

2.2.1.1 Structure of Neural Networks

Deep neural networks have an extremely large number of parameters compared to the traditional statistical models [12, 48, 27]. Neural networks are of multiple distinct (as well as hybrid) types. The simplest and most commonly studied class of neural networks are called feedforward neural networks, or multilayer perceptrons (MLPs) [27]. Feedforward neural networks are essentially a series of functional transformations which can be understood as a chain of alternating linear transforms and non-linear activation functions. Neural networks are said to be *universal approximators* due to their approximation properties, which are able to approximate any function given a sufficiently large network capacity [25, 21, 30, 72, 20, 34, 39, 31, 58]. The universal approximation property however does not guarantee whether the model can be learned or generalized to a satisfactory level.

2.2.1.2 Learning for Neural Networks

Neural networks are trained by sequentially and incrementally updating the weights of each of the constituent layers with the aim of pushing the the network output $\hat{y}_i$ towards the target value or label y_i available for the corresponding input data point. Training progress is evaluated through a

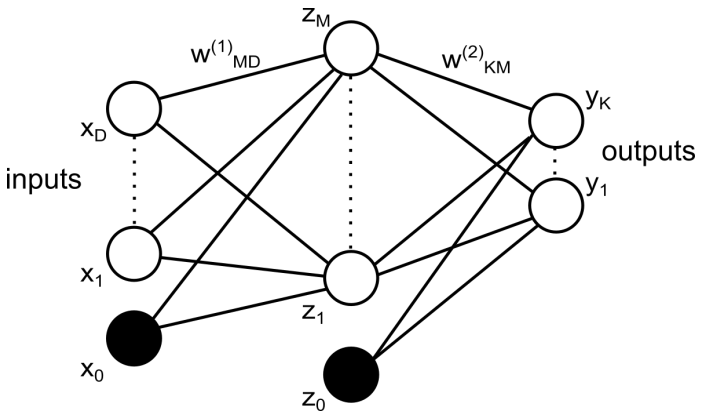

Figure 2.1: A neural network with D inputs, one hidden layer with M units, and K outputs. Reprinted from [12].

loss function, generally represented by $\mathcal{L}(\cdot)$, which is different for discrete spaces (classification) in continuous (regression) cases. Common cost functions that exist for classification tasks are Cross-Entropy and Hinge loss. On the other hand, for regression some commonly used loss functions include Mean Squared Error (MSE), Mean Absolute Error (MAE), and Huber loss. The Cross-Entropy (CE) loss, written as

$$\mathcal{L}_{CE}(y_i, \hat{y}_i) = -\frac{1}{N}\sum_{i}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij}),$$

The cross-entropy loss is useful for behavior cloning setups where the action space is finite and can be represented as a discretely-valued random vector.

For regression problems, Mean Squared Error (MSE) loss computes the average of the square of the errors between predicted $\hat{y}_i$ and true y_i value, which results in an arithmetic mean-unbiased estimator:

$$\mathcal{L}_{MSE}(y_i, \hat{y}_i) = \frac{1}{N}\sum_{i}^{N}(y_i - \hat{y}_i)^2.$$

The MSE loss is commonly used in behavior cloning setups where the output can be represented as a continuous random vector. This can be a general random vector and the individual components of the vector need not be independent and identically distributed.

2.3 Decision Making for Autonomous Systems

Research problems in trajectory generation and sequential decision making are formalized in one of two ways: The first includes formalization as a *Partially Observable Markov Decision Process* (POMDP) [11] (similar to any other decision-making problem). A POMDP $\mathcal{M} = \{\mathcal{S}, \mathcal{O}, \mathcal{E}, \mathcal{A}, \mathcal{T}, r\}$ is characterized by various vector spaces including its state-space $\mathcal{S}$, an observation-space $\mathcal{O}$ and an action-space $\mathcal{A}$. Also included are various functional tranforms including an emission probability, $\mathcal{E}$, that controls the probability of observing $\boldsymbol{o} \in \mathcal{O}$ given an underlying state $\boldsymbol{s} \in \mathcal{S}$, a transition function $\mathcal{T}$ (which defines the probability of transitioning from a given state to another state), and the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This mathematical formulation is graphically shown in Figure 2.2. This way is used to model problems where perfect state information might not be present and noisy sensor data is used to infer state. The second way includes formalization as a *Fully Observable Markov Decision Process* (known simply as a *Markov Decision Process* (MDP)), the underlying states are directly known. This eliminates observation vectors and simplifies the process notation to $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$. This way is a simpler but less realistic approach since perfect state information is rarely known in real-world problems.

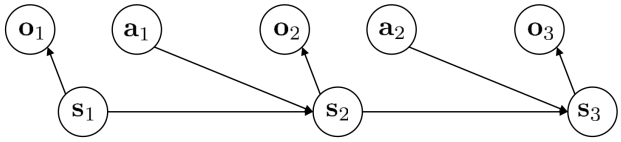

Figure 2.2: Partially-Observable Markov Decision Process diagram. Reprinted from [40].

2.4 Behavior Cloning

Imitation Learning is a principled approach to learn from demonstrated behaviors by utilizing examples from an expert policy performing the task. Imitation learning frequently starts from a random untrained policy and attempts to converge to a more stable policy. This expert policy can be any of the following: another agent trained with reinforcement learning, a human, or another

hard-coded controller. The expert simply needs to be able to provide expert trajectories that can be used as a training signal.

Behavior Cloning (BC) is a subdomain of Imitation Learning where state-action pairs from the demonstration data are directly used in an attempt to mimic the demonstrated behavior. This is done by training a model to generate policy outputs/actions/control commands as similar as possible to the ones seen in the expert control sequences, given the same observations as inputs as seen by the demonstrating agent. In BC, a policy π is trained in order to fit to a subset $\mathcal{D}$ of visited states to actions taken at those states during a expert demonstration of the task over a time-horizon spanning T time steps.

When supervision is provided through human demonstrations, the human is sub-consciously trying to maximize a goal metric for a given task. In BC, this goal metric happens to be the likelihood that a particular action would be taken at a given state if the expert were performing the task. This can be seen in Equation 2.1 where $\pi_E(\boldsymbol{a}_t^E|\boldsymbol{s}_t)$ represents the expert policy, in which the optimal action $\boldsymbol{a}^E$ is taken at state $\boldsymbol{s}$ for every time step t. By maximizing log-likelihood instead of likelihood, this equation ties in the optimization objectives in both reinforcement learning (reward functions) to behavior cloning as both optimize a metric over the temporal length of an episodic trajectory.

$$\max_{\boldsymbol{a}_0,\ldots,\boldsymbol{a}_T} \sum_{t=0}^{T} r_t(\boldsymbol{s}_t, \boldsymbol{a}_t) = \sum_{t=0}^{T} \log p(\pi_E(\boldsymbol{a}_t^E|\boldsymbol{s}_t)) \tag{2.1}$$

The above equation can be further simplified by assuming the probability distribution of actions to be taken at any state is a Gaussian distribution centred around the expert action with a unit variance. This reduces the log-likelihood objective to a mean-squared error term. Mathematically, in a setting where the expert policy is represented by π_E, the learnt policy is represented by $\hat{\pi}$, and the parameterized form of the policy model is represented by π_θ, the policy learnt through behavior cloning can be represented as below:

$$\hat{\pi} = \underset{\theta}{\operatorname{argmin}} \sum_{t=0}^{T} ||\pi_\theta(\boldsymbol{s}_t) - \pi_E(\boldsymbol{s}_t)||^2 \tag{2.2}$$

The above equation illustrates how behavior cloning can be reduced to a least squares regression problem, which is also a subset of supervised learning.

3. A SCALABLE SOFTWARE FRAMEWORK FOR MACHINE LEARNING ACROSS VEHICLE PLATFORMS

3.1 Problem Definition and Defined Requirements

The software framework that forms the core contribution of this work was born out of a need to develop a reliable software package that can be readily used for cross-platform machine learning research on autonomous vehicles. The core requirements were:

- **Modularity**: Modularity was the core driving principle behind the initial development process. Machine learning experiments on autonomous vehicle testbeds require multiple components to be configured. These range from vehicle interfaces, environments to loss functions, data resampling functions and input-output configurations. Modularity also enables platform independence since systems do not need to have be able to support all available functionality. Dependencies can be included or excluded based on the functionality of the library that is actually utilized for an experiment.

- **Extensibility**: A major focus behind the software architecture design was to prevent the codebase from being too rigid and fragile so that new functionality can be added over time without needing significant code refactoring. Extensibility also requires code design to be intuitive enough so that new module implementations can be added by external developers without minimal overhead and no intervention needed by the original code authors.

- **Portability**: The core software module that controls the vehicle operation and experimentation routine is built with redundancies and safety overrides to allow for hardware-in-the-loop experiments. The addition of this functionality should not affect experiments in simulation and should be seamless compatible across different vehicle types and operational environments (simulation or real).

3.2 Software Architecture

The entire software framework is built upon 3 primary categories of components: Rollout, Training and Other. The Other category includes components that are required during both the evaluation (by the Simulator (defined below)) and training phases of an experiment.

The decoupling between rollout and training-related components ensures that different machine learning frameworks can be used in conjunction with the rollout components. For example: RLlib [42], a popular library that provides abstractions for reinforcement learning can utilize the in-built Gym Environments and the compatible *Vehicle* and *Sensor* classes directly and train policies. This would not use either the library's internal *Simulator* or the *Task* class, which is purpose-built for supervised and unsupervised learning workflows.

3.2.1 Rollout Modules

All the class components defined in this section are essential to performing any sort of functionality that requires a vehicle (simulated or physical). The core types of component classes/modules developed as a part of this library are:

- **Sensor**: Classes of this type interface with actual sensor hardware itself and perform all data reading, caching and processing operations and return measurements in human-readable formats. Currently implementations for the *Tobii5C EyeTracker*, IMU and RGB sensors onboard the *Parrot Anafi* and the simulated IMU and GPS sensors in *Airsim* are provided out-of-the box.

- **Vehicle**: Classes of this type interface with either the physical vehicle itself or any intermediary API that is provided by the vehicle's original equipment manufacturer (OEM). Currently the *Parrot Anafi* and the *Parrot ARDrone* (in *AirSim*) are supported out-of-the-box by the developed software library.

- **Environment**: An *Environment* class defines the actual scenario and visuomotor task the vehicle is expected to perform. Programmatic conditions related to task termination are

defined in the class. This class also houses any custom routine that needs to be executed between episodes such as rebooting the vehicle's systems etc.

- **Controller**: Derived classes of the *Controller* module are responsible for commanding the vehicle's actuation. They are responsible for running inference on the policy model and translating the model output to an action representation that can be consumed by a compatible *Vehicle* class.

- **RolloutRunner** The *RolloutRunner* is responsible for the execution of the entire experiment and handles both autonomous policy rollouts and teleoperation for demonstration collection purposes. The *RolloutRunner* also handles all auxiliary functionality such as vehicle repositioning between episodes, self-diagnosed and user-initiated emergency shutdowns, immediate control transfer on user interventions and data logging/saving. The execution flow of this module is discussed in further detail in the succeeding section.

3.2.2 Execution Routine for Safe Human-in-the-Loop Learning

The component of this software framework that executes the actual vehicle motion and data collection is the *RolloutRunner*. The execution of this module follows a Finite State Machine pattern which is illustrated in Fig. 3.1. This module activates the vehicle, initializes the entire sensor suite and allows for the entire experiment to be safely performed with just a single operator. Once the user initiates the experiment, the vehicle takes off and starts executing the action commands. The action commands can either come through teleoperation or through a neural network policy that continuously performs inference in the background. In going with an implicitly safe design, the NN policy is only actively controlling the vehicle when a deadman's trigger is continually engaged. Any disengagement of the trigger reverts the control of the vehicle back to teleoperation and it stays idle (hover mode for multirotor UAS) till the operator actively resumes providing input.

The termination of an episode can be triggered in two ways. The first is user-provided termination signal that is mapped to a button on the teleoperation control. The second is automatic episode termination based on a hard-coded termination condition that can be specified by overriding the

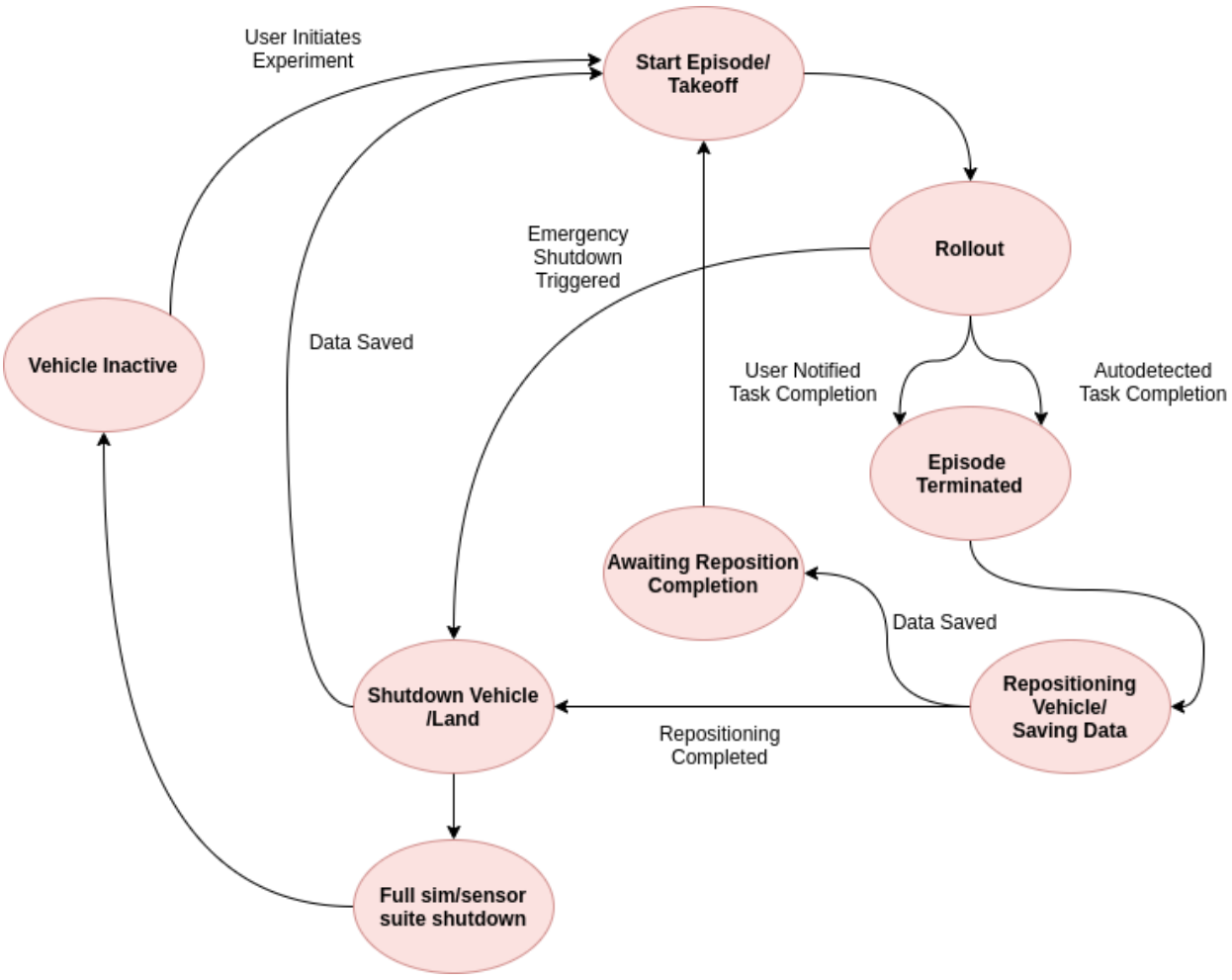

Figure 3.1: Finite State Machine representation of the experimental workflow

task_done method in the *Environment* class being used.

At each episode termination, data is saved and the vehicle shifts to "repositioning mode" which allows to user to remotely move the vehicle to a new start location for the next episode. If at the end of the repositioning phase, the data saving process is still pending completion, the vehicle goes into *sleep* mode (lands) to conserve power else it resumes flight from that same hovering state. This saves an extra takeoff-land cycle for UAS. At any point in the experiment, a single-click "emergency shutdown" option can be utilized that will safely land the vehicle at the current spot and shutdown all systems (including cutting off propulsive power).

3.2.3 Design for Safety and Fault-Tolerant Vehicle Operation

Safe and interactive machine learning on autonomous vehicle platforms requires several functionally important operations to occur in parallel. These requirements range from needing to ensure asynchronous data ingestion from various sensors linked to the vehicle platform or even the operator (such as an eye-gaze tracker), to continuously polling joysticks and other input devices for priority requests such as emergency shutdown.

Multithreading is often used to implement such parallelism, however it comes at a price. It is critical to ensure that different threads can access the same resources without exposing erroneous behavior or producing unpredictable results. This programming methodology is known as “thread-safety” and is implemented through locks that guard access to a shared resource, such as a variable that describes a state in the library’s state machine.

The entire core of this library is written in Python for readability and thus is limited by the restrictions of the Global Interpreter Lock [9]. The Python Global Interpreter Lock or GIL, is a lock that allows only one thread to hold the control of the Python interpreter. This means that only one thread can be in a state of execution at any point in time. The impact of the GIL isn’t visible to developers who execute single-threaded programs, but it can be a performance bottleneck in CPU-bound and multi-threaded code.Some of the code that is a part of external dependencies is written in C++ and invoked through Python bindings. These native C++ functions can be split into two types:

- **Non-blocking**: These are functions that explicitly release the GIL allowing for time-consuming operations to occur in a separate CPU core while not blocking the flow of execution of the caller program in the library. Use cases include modules that are suited for asynchronous operation such as ingesting data from sensors or logging non-critical warnings.

- **Blocking**: These are functions (including those provided through external dependencies) that do not release the Python GIL, thus forcing the Python runtime to wait for their execution to finish. Since some of the states in the state machine are guarded by a lock that can only be acquired once the blocking thread is released, this occasionally results in other Python threads from being unable to immediately trigger changes in the core state machine.

Multithreading is utilized in this work in implementing various functionality for safe human-in-the-loop learning. The most prominent use-cases are described below.

- **Override threads**:

  Override functionality is implemented in a simple thread so that mission-critical commands that need to be executed immediately are not blocked due to a set of instructions awaiting execution in the main thread. In the event the main thread's execution is stuck due to a time-intensive operation, the override thread can acquire the GIL to execute any priority commands. An exception to this is when the main thread's execution is stuck while it attempts to modify a shared resource such as the associated Environment object.

- **Sensor threads**:

  Sensor modules asynchronously process and return measurements/observations and are implemented as independent Python threads and are of two-type:

  - *Non-persistent*:

    These are threads that do not persistently run since the requests for measurement data in these cases is externally handled outside the library (eg. by the Airsim engine). In these

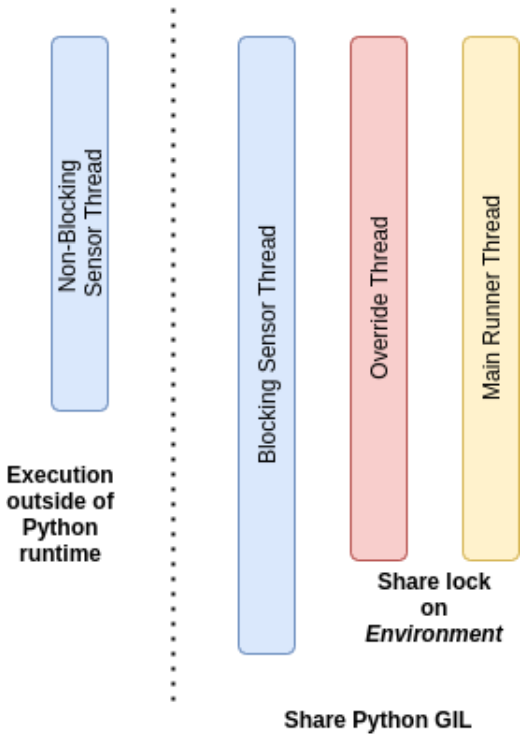

Figure 3.2: Operational threads and the bottlenecks affecting the various threads

cases the threads terminate immediately on starting and do not occupy any CPU cores. They only exist to ensure compliance with the Liskov Substitution Principle [50].

– *Persistent*:

  These have Python code being persistently run (such as the *AnafiRGB* class), which depending on their implementation might run into some kind of bottleneck. At the very least, the Python interpreter's GIL will pose a bottleneck (if not explicitly released in C++ code). Additional sources of a bottleneck could include, for example, a user-specified lock on a state, put in place to ensure mutation by only one thread at a time.

An overview of the various threads operational at any time along with the constraint relations is shown in Fig. 3.2

3.3 Design Principles for Software Architecture

The software framework developed has its entire design philosophy based on the *SOLID* principles of program design. SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable [45, 70, 17]. Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for agile development. The name *SOLID* is derived from the first letter of the names of the constituent principles. The principles that form it are titled:

- Single Responsibility Principle
- Open-closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

All of these 5 principles are extensively used in the library's architectural framework and are discussed in the following subsections:

3.3.1 Single Responsibility Principle

As the name implies, the *Single Responsibility Principle* requires each module or class to have responsibility for one and only one purpose. This principle enables the writing of software with high cohesion and robustness since singular functionality ensures that it's implementation is not tied to the structure of the remaining codebase. It also ensures that individual modules are easily unit-testable, swappable and allow for partial codebase upgrades. By decoupling functionality, the complexity of code upgrades reduces from $\mathcal{O}(SV)$ to $\mathcal{O}(S)$ where S is the number of *Sensor* modules to be modified and V is the number of *Vehicle* modules that utilize those to-be-updated modules.

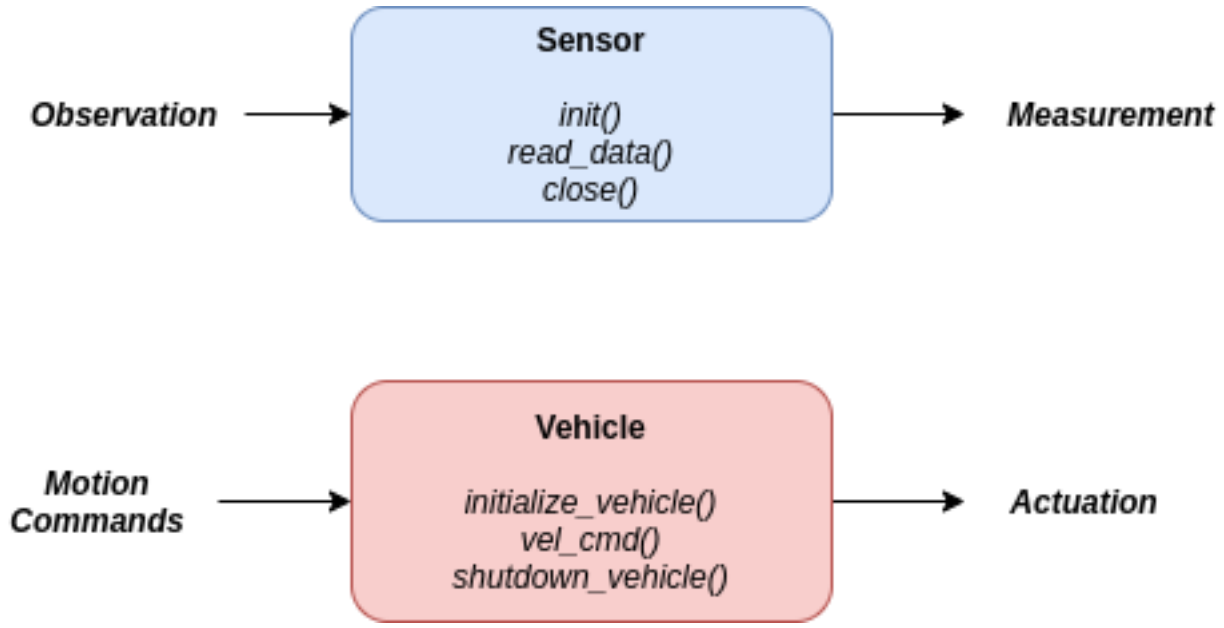

Figure 3.3: Class design demonstrating *Single Responsibility Principle*

For example, Figure 3.3 shows how a *Sensor* class only returns a measurement given an observation and does not implement any additional logic. Similarly a *Vehicle* class only actuates the vehicle (simulated or real). Since a *Sensor* class does not perform any data processing and thus would not need modification if the data processing pipeline was desired to be altered. Essentially the modules in the software framework can be thought of as independent but non-identically distributed random variables.

3.3.2 Open-closed Principle

The *Open-closed Principle* [44] dictates that the constituent software modules are open for extension but closed for modification. This allows new functionality to be added without changing existing source code.

In practice, the best way to achieve this is polymorphism, particularly with the use of abstract interfaces. Using abstract interfaces allows for extending a class and specializing its behavior without changing the interface specification. This principle allows for reusable and maintainable software.

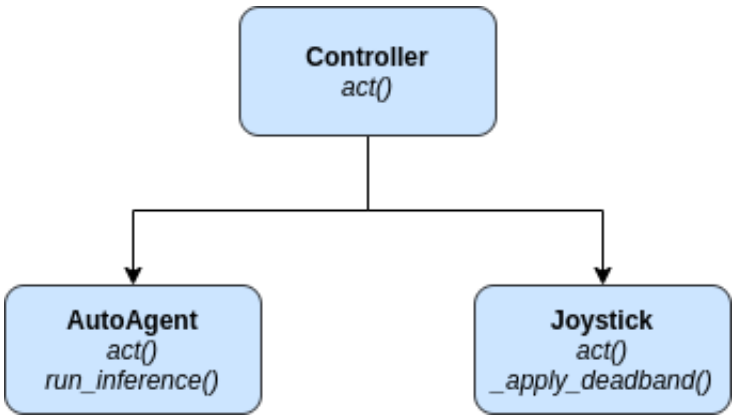

Figure 3.4: Class design demonstrating *Interface Segregation Principle*

The core modules that interface with the actual vehicle and sensors respectively are called *Vehicle* and *Sensor* classes. These classes are primitive interfaces that only encapsulate the essential and bare-basic functionality that is not to be modified. They provide external developers the ability to build their own custom implementations for their hardware-specific needs by just defining a subset of member functions instead of needing to define the entire implementation, which is locked.

3.3.3 Liskov Substitution Principle

The *Liskov Substitution Principle* [50] states, "If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness).".

This principle requires subclasses to not only satisfy the syntactic expectations but also the behavioral ones, of its parents. To view things from a different perspective, as a user of a class, one should be able to utilize any of its children that may be passed to me without caring about which particular child is being called. This requires ensuring that the arguments as well as all the return values of the children are consistent. Thus *EyeTracker* class (child class of *Sensor*) or a new child of *EyeTracker* (say *Tobii5EyeTracker*) should interact with the rest of the codebase in an identical fashion.

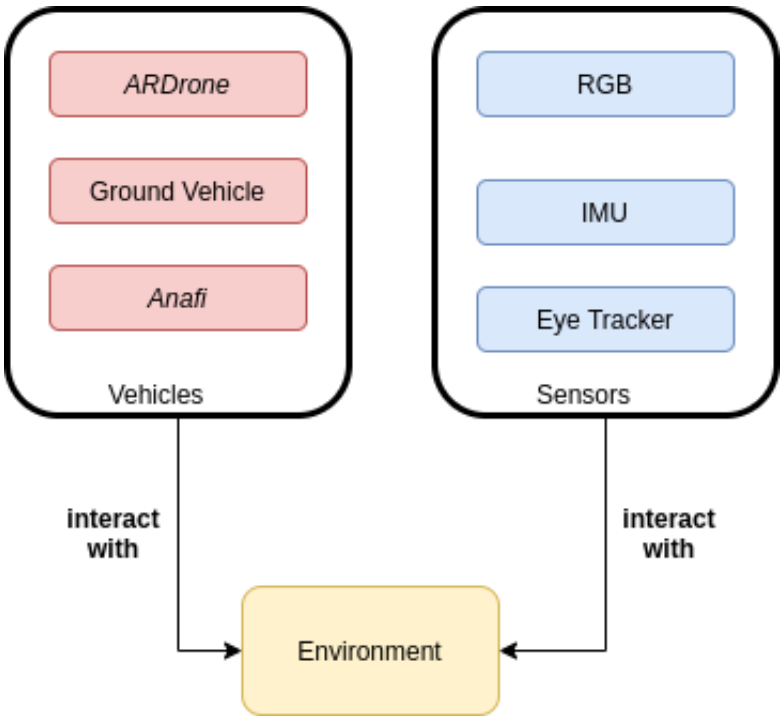

Figure 3.5: Class design demonstrating *Dependency Inversion Principle*

3.3.4 Interface Segregation Principle

The *Interface Segregation Principle* (ISP) [49] states that "no client should be forced to depend on methods it does not use". ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

Fig. 3.4 illustrates how the interface segregation principle is put to use in this software framework. Both *Joystick* and *AutoAgent* modules command the vehicle and derive from a common base class. However the former operates on human input while the latter operates by running inference on a trained NN. As a result, both implement methods that are need-specific and are invoked through *act()*, a function common to the all modules that share the same base type (*Controller*). The base interface is kept free from any unnecessary functionality that is not used by all modules of its type.

3.3.5 Dependency Inversion Principle

The *Dependency Inversion Principle* (DIP) states that: "High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces)". As an example of what this means, in context of this work, the dependency inversion principle requires that the core implementations of the simulation and environment classes be totally independent of the type of vehicle and sensors they control. Fig. 3.5 shows how an *Environment* class has its implementation independent of the type of *Sensor* and *Vehicle* classes that interact with it. This ensures that the base Gym environment is out-of-the-box compatible with not just aerial vehicles but also ground vehicles etc. in any sensor-payload configuration.

3.4 Structural Design Patterns Used

3.4.1 Facade

The *Facade* pattern is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. The core intent is to provide an intuitive and limited interface to a complex subsystem which contains lots of moving parts. A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that downstream client programs really care about. This is closely tied to the concept of abstraction [24] in object-oriented programming. Fig. 3.6 is an example of how a user can run entire simulation experiments and train policy models by relying on just two function calls. Essentially, an entire experimental setup can be configured in a JSON file (including vehicle parameters, machine-learning related parameters etc.) and the library's internals handle the rest without the user needing to know what happens behind the scenes.

3.4.2 Proxy

The *Proxy* pattern is a structural design pattern that lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object. For vehicle rollouts in simulation where no human is required to be present in the loop, forcing a dependency on a *Joystick* module

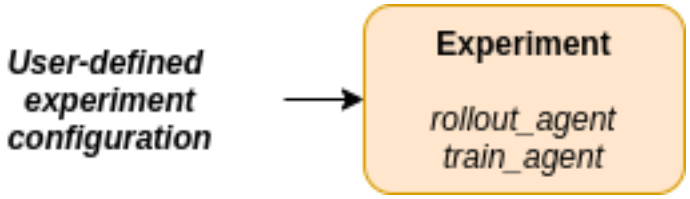

Figure 3.6: Class design demonstrating abstraction of all internal functioning to make only user-need functions visible.

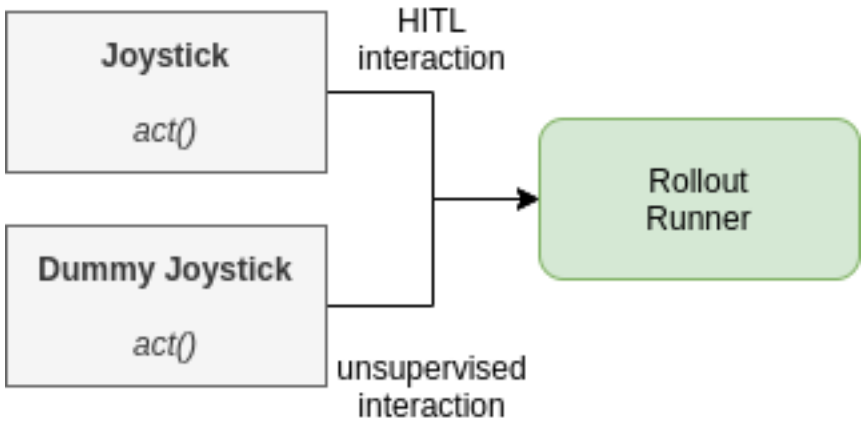

Figure 3.7: *DummyJoystick* as a proxy for regular joystick-teleoperation

would be bad design and is thus substituted by a proxy (*DummyJoystick*) module that follows the Liskov Substitution Principle and exposes an exactly similar interface to the other modules.

The proxy pattern is also extremely useful for performing integration tests on the mid-to-high level modules of the library. Essentially, expensive function calls and object creation requiring unrelated dependencies can be avoided during integration testing of modules by using proxy objects where possible.

3.4.3 Decorator

The *Decorator* pattern is a structural design pattern that enables attaching new behaviors to objects by placing these objects inside special wrapper objects that contain the original behaviors [33]. Computationally intensive function calls, such as those those involve large matrix computations (during deep learning training) are decorated with a caching mechanism that enable the results of

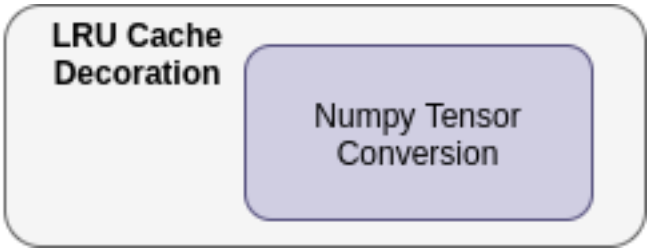

Figure 3.8: Caching of expensive and repeated matrix computations

these computations to be reused while maintaining the same user-facing interface.

3.5 Behavioral Design Patterns Used

3.5.1 Visitor

The *Visitor* pattern abstracts away algorithms from the objects on which they operate. This is useful if the objects are closed to any modification which prevents polymorphism from being an option for implementing object-specific behavior. The *Visitor* pattern is primarily used in this work to execute graceful fault handling. Message-passing and event handling with the hardware platform used in this experiment happens through asynchronous coroutines. A coroutine is a concept similar to a thread in traditional concurrent programming, but is based on cooperative multitasking. Cooperative multitasking implies that the switching between different execution contexts is done by the coroutines themselves rather than the operating system or user-specified handling (as is the case in multi-threading). Since these message objects are closed to modification and have sealed definitions within the vehicle software's API, the visitor pattern allows to have a single stateful event handler that depending on the type of message received and state of the program, performs custom operations that ensure safe operation and provide mission assurance.

3.5.2 Chain of Responsibility

The *Chain of Responsibility* pattern is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

An example of the use of such a design pattern in the library is how the specific of just the

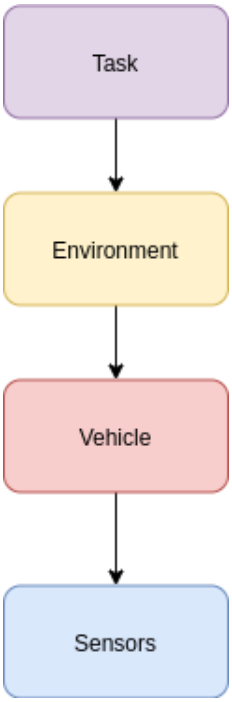

Figure 3.9: Chain of responsibility between various component modules

name of the *Task* class, ensures that the correct set of sensors, vehicles and the Gym environment is initialized. This is illustrated in Figure 3.9.

3.5.3 State

The *State* pattern is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. The State pattern is closely related to the concept of a Finite-State Machine. The main idea is that, at any given moment, there's a finite number of states which a program can be in. Within any unique state, the program behaves differently, and the program can be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to certain other states. These switching rules, called transitions, are also finite and predetermined.

The state design pattern is extensively used in the *Rollout Runner* class to decide next course of action for the vehicle depending on mission state. The state design pattern is also used in the design of the *Vehicle* class. Depending on the state of the vehicle hardware (critical faults, degraded

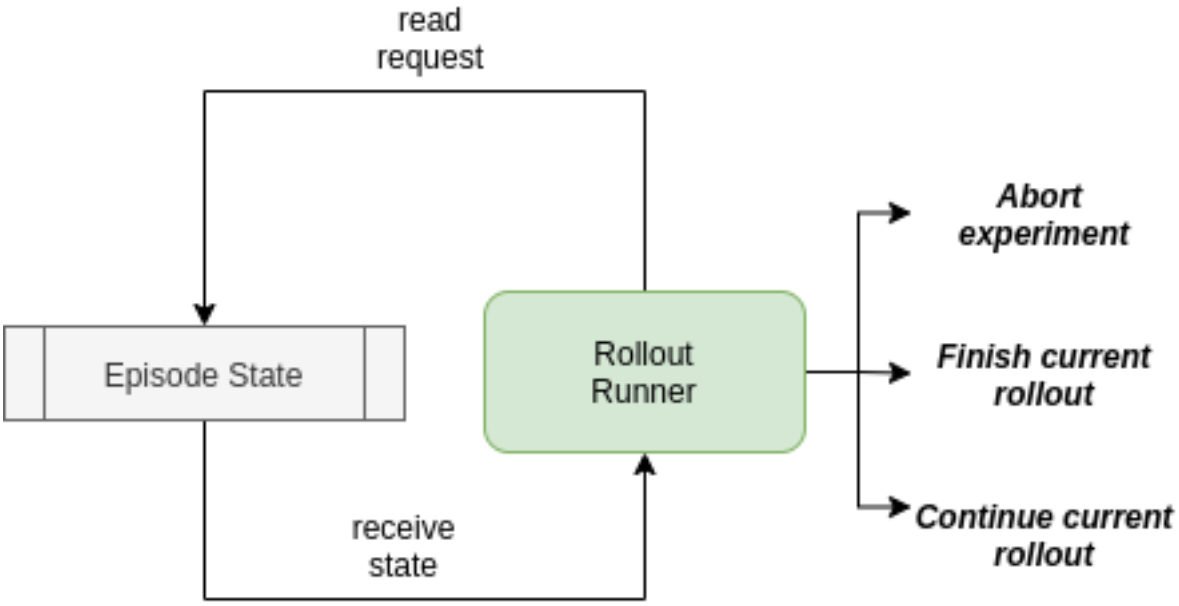

Figure 3.10: Effect of mutable state variables on the behavior of the *Rollout Runner* class responsible for performing the rollouts

sensing etc.), the vehicle can dispatch different actuation commands. This reduces the need for operator decision-making and intervention in such scenarios.

3.5.4 Lazy Loading

The entire library has a modular architecture with various modules in each module class having their own set of external and internal dependencies that might need to be installed and loaded for usage. Since only one of the available modules within any module class may be needed for a set of experiments (such as doing a set of experiments with the Parrot Anafi), lazy loading of modules ensures that the Python runtime doesn't throw up dependency errors while attempting to initialize unnecessary modules. This also lowers startup times while loading the entire library. In this work, lazy loading has been implemented through a "virtual proxy" pattern where references to modules are present wherever needed and the actual object required gets created only when one of its attributes is accessed for the first time.

Code Coverage Summary (measured in number of Python statements)			
Experiment	Shared Code	Vehicle-Specific Code	% Shared Code Execution
Anafi (Physical)	94+32+14+88	80	64.91
ARDrone (Simulated)	94+32+14+88	60	73.68

Table 3.1: Shared code execution across different vehicle platforms and different operating environments

3.6 Results

In order to measure the modularity of the resulting codebase, code coverage experiments were conducted wherein a Parrot Anafi was rolled out using a NN policy on hardware and a simulated Parrot ARDrone was rolled out in Airsim using a NN policy. The Python instructions executed were logged and the amount of common code that was executed in both the above runs was measured.

The first run involving rolling out a NN policy on a physical Parrot Anafi resulted in execution of 308 lines of code of which 64.91% was shared code that was also executed during rolling out a simulated Parrot ARDrone in Airsim. The percentage of shared code execution for it was even higher due to lesser amount of vehicle-specific code needed. The shared pieces of code were across the base *Environment*, base *Vehicle* and *RolloutRunner* modules.

Also, all code executed that was specific to either run was attributable to differences in the vehicle platform being used and was entirely contained in the modules designated to the specific vehicle platform used.

3.7 Summary

The work described in this chapter highlights how usage of *SOLID* design principles in a codebase's design results in the majority proportion of the code, executed during an experimental run, being common code (shared across platforms and experimental configurations). This work also highlights the development of a state machine that regulates the execution of the software to enable automated and safe evaluation of machine-learning based policies on actual hardware.

4. CASE STUDY: LEARNING FROM GAZE *

4.1 Problem Definition and Background

In the field of human-robot interaction, learning from demonstrations (LfD), also referred to as imitation learning, is widely used to rapidly train artificial agents to mimic the demonstrator via supervised learning [6, 52]. LfD using human-generated data has been widely studied and successfully applied to multiple domains such as self-driving cars [19], robot manipulation [57], and navigation [69].

While LfD is a simple and straightforward approach for teaching intelligent behavior, it still suffers from sample complexity issues when learning a behavior policy directly from images, i.e. mapping what the robot's camera sees to what action it should take. The majority of the work on LfD utilizes only behavioral information from the demonstrator, i.e. what actions were taken, and ignores other information such as the eye gaze of the demonstrator [6]. Eye gaze is an especially useful signal, as it can give valuable insight towards where the demonstrator is allocating their visual attention [23], and leveraging such information has the potential to improve agent performance when using LfD.

Eye gaze is an unobtrusive input signal that can be easily collected with the help of widely available eye tracking hardware [55]. Eye gaze data comes at almost no additional cost when teleoperating robots to collect expert demonstration data, as the human operator is able to do the task naturally as before. Eye gaze acts as an important signal in guiding our actions and filtering relevant parts of the environment that are perceived [66], and as such, measuring eye gaze gives us an indication of visual attention that can be leveraged when training AI agents.

Previous works that have attempted to utilize eye-gaze for imitation learning (such as [79]) have only done so in simple, synchronous environments, limiting their applicability to real-world domains. This work utilizes multi-objective learning for gaze-informed imitation learning of real-

*Adapted with permission from "Gaze-Informed Multi-Objective Imitation Learning from Human Demonstrations"[10], by Ritwik Bera, Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich, available under a *Creative Commons* license at *https://arxiv.org/abs/2102.13008*

world, asynchronous robotics tasks, such as autonomous quadrotor navigation, which have proven difficult for traditional imitation learning techniques [59, 13].

This work attempts to frame the problem of visual quadrotor navigation as a supervised multi-objective learning problem wherein our model has an input space comprised of onboard inertial measurements, RGB and depth images, and attempts to predict both the demonstrator's actions as well as the the demonstrator's eye gaze while performing a given task, as shown in Figure 4.2. The benefit behind using a multi-objective optimization, framework is two-fold. First and foremost, if a task is noisy or data is limited and high-dimensional, it can be difficult for a model to differentiate between relevant and irrelevant features. Multi-objective learning helps the model focus its attention on features that actually matter since optimizing multiple objectives simultaneously forces the model to learn relevant features that are invariant across each objective [16, 3, 62]. Finally, a multi-objective learning framework, as proposed by this work, enables eye gaze prediction loss to act as a regularizer by introducing an inductive bias. As such, it reduces the risk of overfitting as well as the Rademacher complexity of the model, i.e. its ability to fit random noise [8].

The main contributions of this work are:

- A novel multi-objective learning architecture to learn from multimodal signals from human demonstrations, from both actions and eye gaze data.
- Demonstration of the approach using an asynchronous, real-world quadrotor navigation task with high-dimensional state and action spaces and a high-fidelity, photorealisitic simulator.
- Quantitative evaluation showing that the gaze-augmented imitation learning model is able to significantly outperform a baseline implementation, a behavior cloning model that does not utilize gaze information, in terms of task completion.

4.2 Related Work

4.2.1 Understanding Gaze Behavior

Saran et al. [64] highlights the importance of understanding gaze behavior for use in imitation learning. The authors characterized gaze behavior and its relation to human intention while

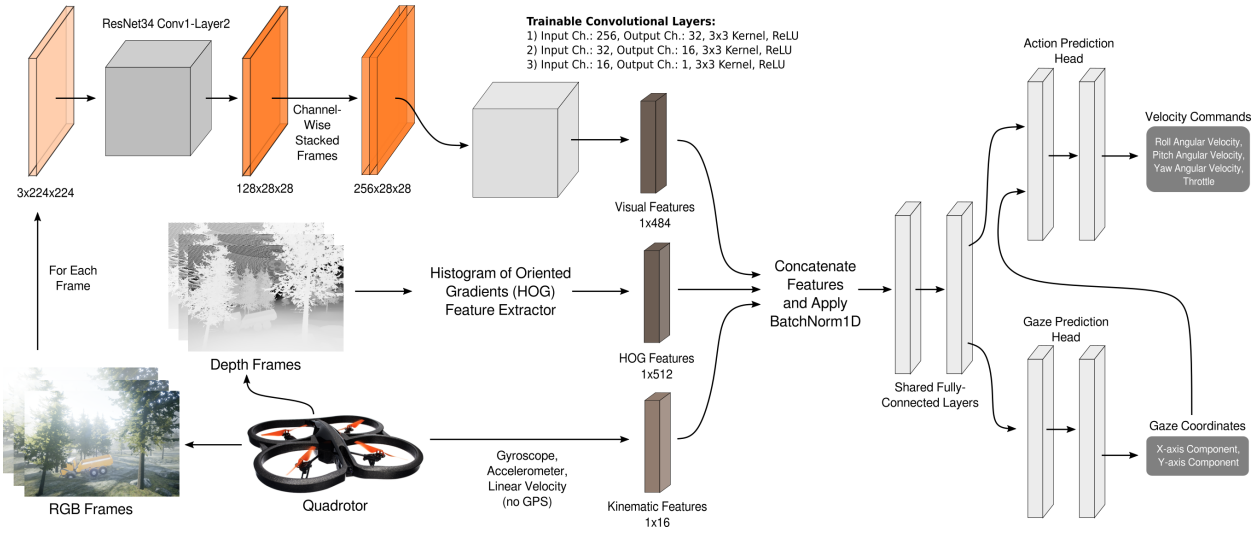

Figure 4.1: Model architecture illustrating how quadrotor onboard sensor data and cameras frames are processed, how features are combined in a shared representation backbone, and how multiple outputs, i.e. the gaze and action prediction networks, are performed via independent model heads.

demonstrating object manipulation tasks to robots via kinesthetic teaching and video demonstrations. For this type of goal-oriented task, Saran et al. show that users mostly fixate at goal-related objects, and that gaze information could resolve ambiguities during subtask classification, reflecting the user internal reward function. In their study the authors highlight that gaze behavior for teaching could vary for more complex tasks that involve search and planning, such as the the one presented in this study.

4.2.2 Gaze-Augmented Imitation Learning

Augmenting imitation learning with eye gaze data has been shown to improve policy performance when compared to unaugmented imitation learning [79] and improve policy generalization [43] in visuomotor tasks, such as playing Atari games and simulated driving. Xia et al. [76] also proposed a periphery-fovea multi-resolution driving model to predict human attention and improve accuracy in a fixed driving dataset.

Attention Guided Imitation Learning [79], or AGIL, collected human gaze and joystick demonstrations while playing Atari games and presented a two-step training procedure to incorporate gaze data in imitation learning, training first a gaze prediction network modeled as a human-like foveation

system, then a policy network using the gaze predictions represented as attention heatmaps. The reported gains in policy performance vary from 3.4 to 1143.8%, depending on the game complexity and the number of sub-tasks the player has to attend to, which illustrates how the benefit of a multimodal learning approach is tied to the desired task to be solved. However, to eliminate effects of human reaction time and fatigue, eye-gaze was collected in a synchronous fashion in which the environment only advanced to the next state once the human took an action, and game time was limited to 15 minutes, followed by a 20 minutes rest period. This highlights the challenges of human-in-the-loop machine learning approaches and real-time human data collection. This limits the applicability of attention guided techniques to only relatively simple domains where the environment can easily be stopped and started to synchronously line up with human input, and will not work on real-world environments such as quadrotor navigation.

Liu et al. proposed two approaches to incorporate gaze into imitation learning: 1) using gaze information to modulate input images, as opposed to using gaze as an additional policy input; and 2) using gaze to control dropout rates at the convolution layers during training. Both approaches use a pre-trained gaze prediction network before incorporating it into imitation learning. Evaluated on a simulated driving task, Liu et al. [43] showed that both approaches reduced generalization error when compared to imitation learning without gaze, with the second approach (using gaze to control dropout rates during training) yielding about 24% error reduction compared to 17% of the first approach when predicting steering wheel angles on unseen driving tasks.

4.3 Methods

A novel multi-objective model is defined via supervised learning to predict actions to accomplish a given task and the location of eye-gaze of a human operating the quadrotor through a first-person camera view. The training is accomplished by collecting a set of human demonstrations while recording the observation space of the quadrotor (i.e. cameras and IMU sensors) as well as the demonstrator's eye-gaze position and their input actions while performing a quadrotor search and navigation task.

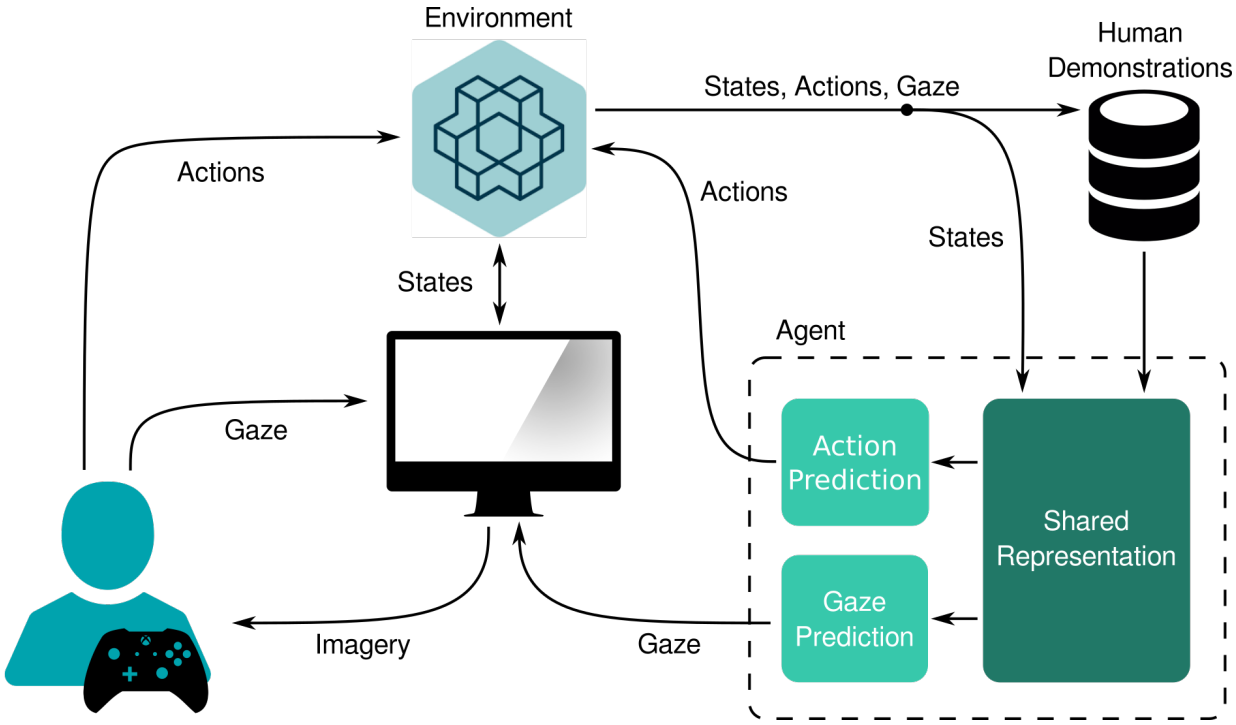

Figure 4.2: Block diagram of the proposed multi-objective learning architecture to learn from multimodal human demonstration signals, i.e. actions and eye gaze data.

4.3.1 Model Architecture

An end-to-end supervised learning approach is taken to solve the visual search and navigation problem, aiming to avoid making any structural assumptions with regard to either the task being performed by the quadrotor, or the environment the quadrotor is expected to operate in. The model architecture is illustrated in Figure 4.1 and explained in the following paragraphs. The input space of the model is composed of three modalities:

- The quadrotor has access to a constant stream of RGB frames, 704 by 480 pixel resolution, captured by its onboard forward-facing camera, which are stored in an image buffer. Each frame is passed through a truncated version of a pre-trained *ResNet34* [28], which includes only its first three convolutional blocks, generating a feature map of dimensions 128x28x28 (channel, height, width) that are stored in an intermediate buffer. Given the photorealistic scenes rendered by *Microsoft AirSim*'s [67] *Unreal Engine* based simulator, *ResNet* was

chosen as the pre-trained feature extractor as it was trained with real-world images. At each time instant *t*, the feature map corresponding to time instant *t* is concatenated with the feature map corresponding to time instant *t-8*, across the channel dimension, generating a 256x28x28 dimensional tensor. The concatenated feature maps are passed through a sequence of trainable convolution layers, as described in Figure 4.1, forming an 484 dimensional vector of *visual features* and the first input to the model.

- The quadrotor also has access to depth images, which are generated by the simulated onboard camera from which the RGB frames are obtained and, consequently, sharing the same geometric reference frame. The depth images, 704 by 480 pixel resolution, are passed through a *Histogram of Oriented Gradients* [22] feature extractor that generates a 512 dimensional feature vector. This forms the second input to the model, *HOG Features* in Figure 4.1.

- The quadrotor also outputs filtered data from its simulated gyroscope and accelerometer, providing the model access to the vehicle's angular orientation, velocity, and acceleration, and linear velocity and acceleration, altitude, but no absolute position such as given by a GPS device, for a total of 16 features, as shown by *Kinematic Features* in Figure 4.1. The model is intentionally not provided with absolute position information in order to force it to be reliant on visual cues to performing the task instead of memorizing GPS information.

The resulting *Visual*, *HOG*, and *Kinematic Features* are concatenated, batch-normalized, and then passed through two fully connected layers with 128 output units each and ReLU activation function, forming the shared backbone of the proposed multi-headed model. The intermediate feature vector is then passed through two separate output heads, the *Action Prediction Head* and the *Gaze Prediction Head*, as seen in the right side of Figure 4.1. Each head has a hidden layer with 32 units and an output layer. The *Action Prediction Head* outputs 4 scalar values bounded between -1 and 1 representing joystick values that control the quadrotor linear velocities associated with movement in the x, y, and z directions as well as the yaw. The *Gaze Prediction Head* outputs 2 scalar values bounded between 0 and 1 representing normalized eye gaze coordinates in the

quadrotor camera's frame of reference.

4.3.2 Training Objective

The loss function used in the training routine is a linear combination of a *Gaze Prediction Loss*, $\mathcal{L}_{GP}$, and a *Behavior Cloning Loss*, $\mathcal{L}_{BC}$:

$$\mathcal{L}(\theta) = \lambda_1 * \text{Gaze Prediction Loss}(\mathcal{L}_{GP}) + \lambda_2 * \text{Behavior Cloning Loss}(\mathcal{L}_{BC}), \quad (4.1)$$

where θ represents the set of trainable model parameters and λ_1 and λ_2 are hyperparameters controlling the contribution of each loss component, defined as the mean-squared error between ground truth and predicted values:

$$\mathcal{L}_{GP} = \left\| \left(\pi_{gaze} - g^t \right) \right\|^2,$$
$$\mathcal{L}_{BC} = \left\| \left(\pi_{action} - a^t \right) \right\|^2,$$

where g_t is a vector representing the true x, y coordinates of the eye gaze in the camera frame at instant t, and a_t is the vector representing the expert joystick actions components for forward velocity, lateral velocity, yaw angular velocity, and throttle taken at that same time instant t. π_{gaze} and π_{action} are the outputs of the *Gaze Prediction* and *Action Prediction Heads*, respectively.

4.3.3 Task Description

The experiments were conducted in simulated environments rendered by the *Unreal Engine* using the *Microsoft AirSim* [67] plugin as an interface to receive data and send commands to the quadrotor, a simulated *Parrot AR.Drone* vehicle. The task consisted of a search and navigate task where the drone must seek out a target vehicle and then navigate towards it in a photo-realistic cluttered forest simulation environment, as seen in Figure 4.3. This environment emulates sun glare and presents trees and uneven rocky terrain as obstacles for navigation and visual identification of the target vehicle.

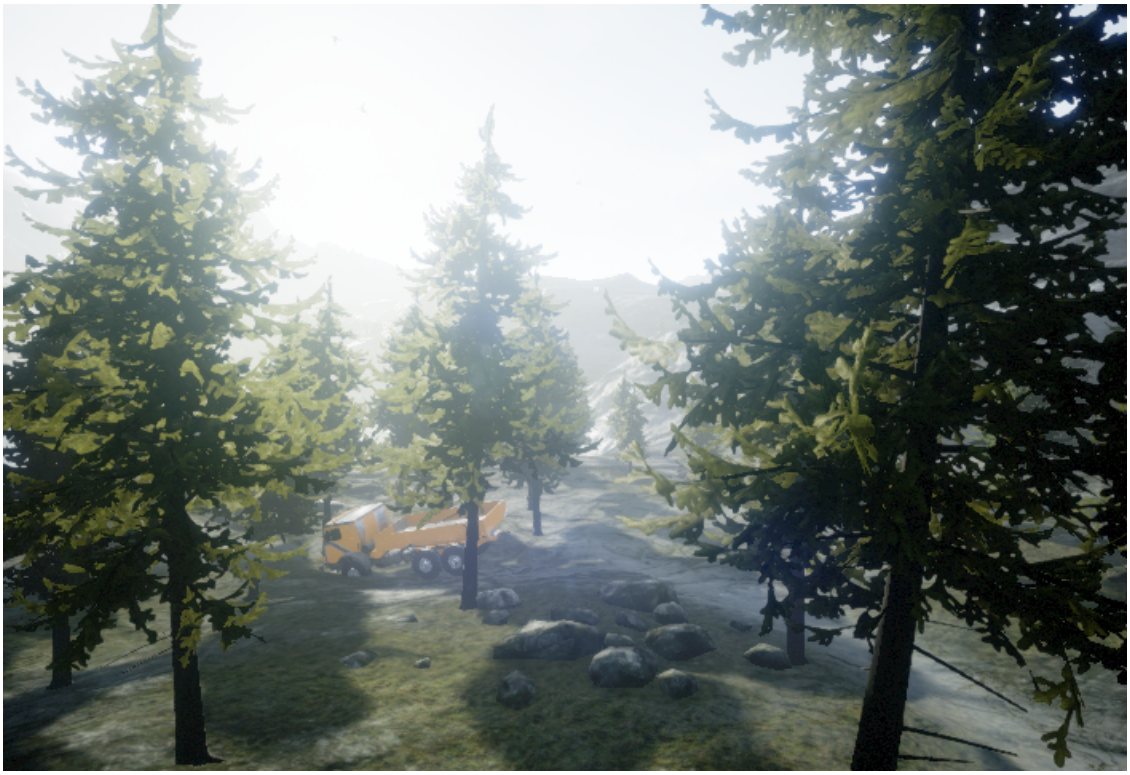

Figure 4.3: First-person view from the quadrotor illustrating the target vehicle (yellow truck) to be found and navigated to, and the realistic cluttered forest simulation environment using Microsoft AirSim [67].

4.3.4 Data Collection Procedure

The task was presented to the demonstrator on a 23.8 inches display, 1920x1080 pixel resolution, and eye gaze data collection was conducted using a screen-mounted eye tracker positioned at a distance of 61cm from the demonstrator's eye. Before the data collection, the height of the demonstrator's chair was adjusted in order to position their head at the optimal location for gaze tracking. The eye tracker sensor was calibrated according to a 8-point manufacturer-provided software calibration procedure. The demonstrator avoided moving the chair and minimized torso movements during data collection, while moving the eye naturally. The demonstrator was also given time to acclimate to the task until they judged for themselves that they were confident in performing it. To perform the task, the demonstrator was only given access to the first-person view of the quadrotor and no additional information about their current location or the target current

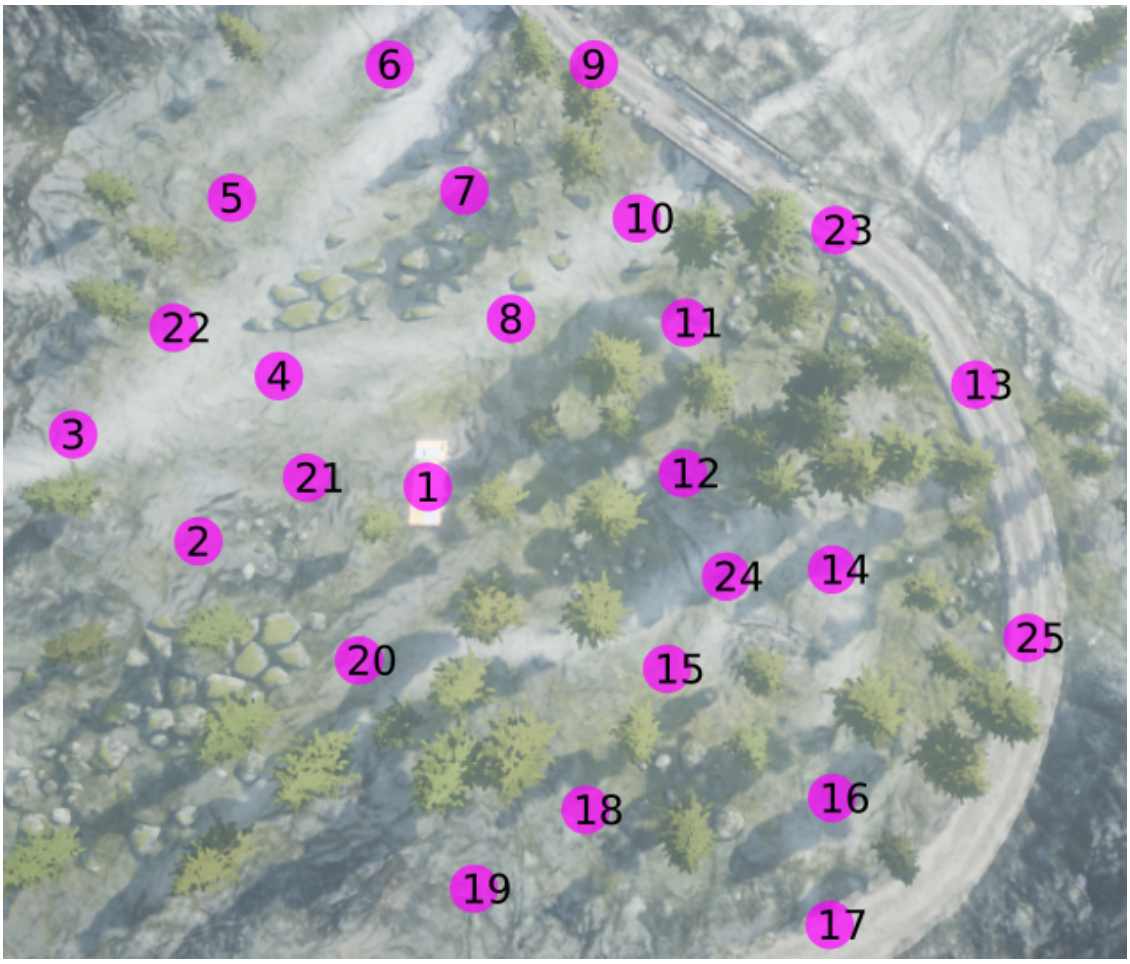

Figure 4.4: Bird's-eye view of possible spawn locations of the quadrotor and the target vehicle in the simulated forest environment.

location in the map. Using a Xbox One joystick, the demonstrator was able to control quadrotor throttle and yaw rate using the left joystick and forward and lateral velocity commands using the right stick, as is standard for aerial vehicles. The complete data collection setup is shown in Figure 4.5.

Training data is collected with both initial locations for the quadrotor and target vehicle randomly sampled without replacement from a set of 25 possible locations covering a pre-defined area of the map, as shown in Figure 4.4. This prevents invalid initial locations, such as inside the ground or trees, while still covering the desired task area. Initial quadrotor heading is also uniformly sampled from 0 to 360 degrees. In total, 200 human demonstration trajectories representing unique quadrotor

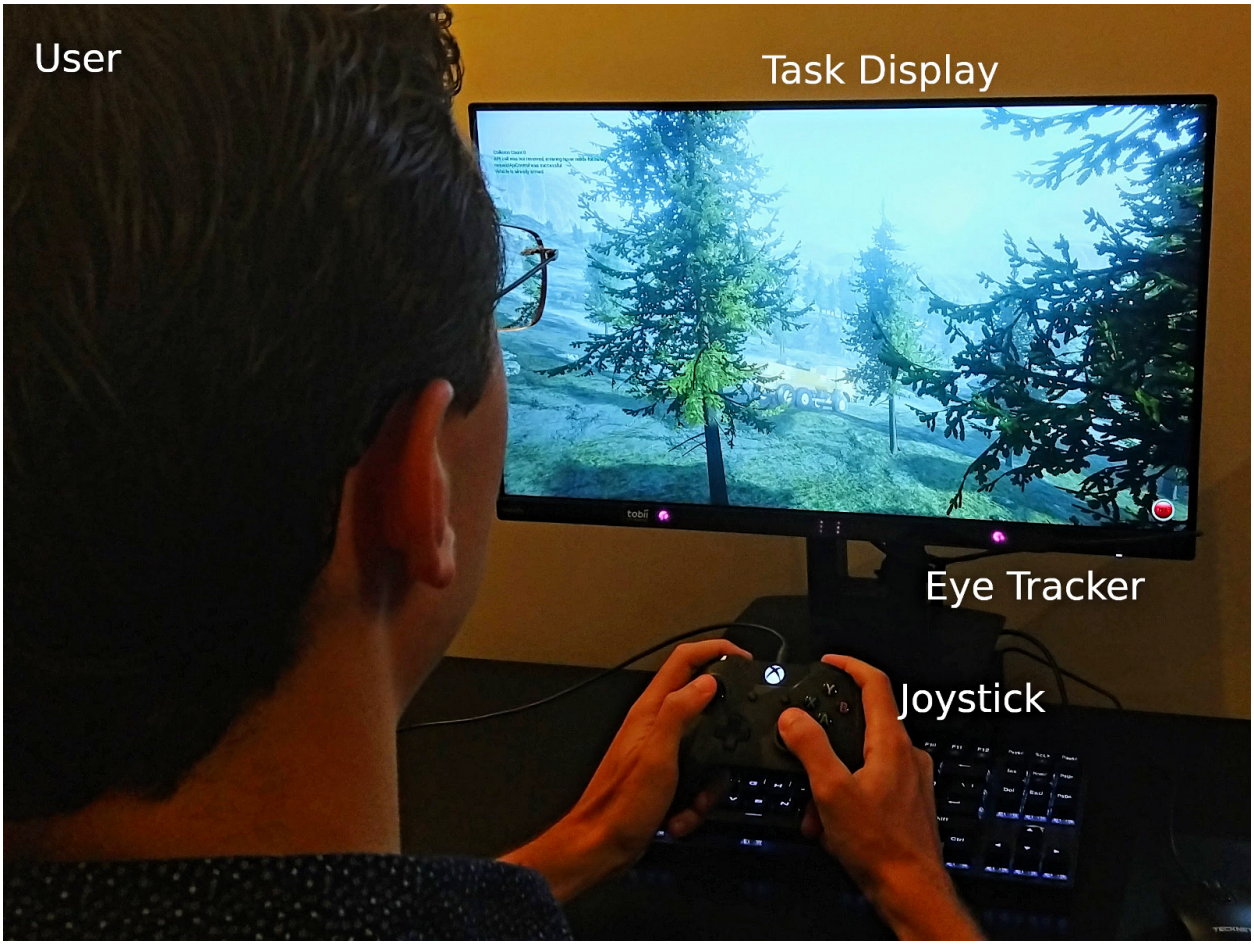

Figure 4.5: Data collection setup used to record human gaze and joystick data illustrating the relative positioning between user, task display, joystick, and eye tracker, as noted in the figure. User is given only the first-person view of quadrotor while performing the visual navigation task.

and vehicle location pairs (from a total number of $25 \times 24 = 600$ pairs) were collected to form an expert dataset for model training and testing. This includes RGB and depth images, the quadrotor's inertial sensor readings, and eye gaze coordinates.

4.3.5 Experimental Setup

The entire dataset of 200 trajectories is split into a training set of 180 trajectories and a test set of 20 trajectories, for each experimental run. During exploratory data analysis on the collected dataset, it was found that the yaw-axis actions were mostly centered around zero with a very low variance. To ensure that the agent is trained with enough data on how to properly turn and not just

fly straight, weighted oversampling was employed in the training routine.

Evaluation of each trained model is performed by conducting rollouts in the environment as follows. A *configuration* is defined as a pair consisting of an initial spawn location for the quadrotor, and a target location, where the target vehicle is spawned, that is, the truck. Since there are 25 different starting and ending locations, there are a total of 600 unique configurations. For the experiment, 200 unique configurations were chosen and 200 demonstration trajectories (1 for each configuration) collected. The 200 configurations were then split into a training set and a testing set, following a 90/10 split. This split proportion was chosen to maximize the utilization of the collected data for training while still having a sufficiently sized set of trajectories (for offline validation) that are representative of the distribution of the flight trajectories that may be encountered. Expert demonstrations were collected that accomplish the task for each of those configurations. 180 trajectories (corresponding to the 180 training configurations) were used to train the model, and the remaining 20 configurations were used to test the model (i.e. performing rollouts in the environment).

Owing to the stochastic nature of the simulation dynamics and sensor sampling frequency, 5 rollouts are conducted for each configuration to obtain more accurate estimates of the evaluation metrics, explained in the next section. Since test trajectory provides one unique configuration, a total of 100 rollouts are conducted while evaluating each model: 5 rollouts per configuration, for 20 different configurations. Since the test configurations remain the same for all models, evaluation consistency is ensured in each experimental run. All models are evaluated for 6 random seeds under identical conditions, including random seed value.

In terms of the hardware infrastructure, all experiments were run in a single machine running Ubuntu 18.04 LTS OS equipped with an *Intel Core* i9-7900X CPU, 128 Gb of RAM, and *NVIDIA RTX* 2080 Ti used only for Microsoft AirSim. In terms of the software infrastructure, the proposed model was implemented in *Python* v3.6.9 and *PyTorch* [54] v1.5.0, data handling using *Numpy* [74] v1.18.4 and *Pandas* [75] v1.0.4, data storage using *HDF5* via *h5py* v2.10.0, data versioning control via *DVC* v1.6.0, and experiment monitoring using *MLflow* [77] v1.10.0. Ran-

dom seeds were set in *Python*, *PyTorch*, and *NumPy* using the methods *random.seed(seed_value)*, *torch.manual_seed(seed_value)*, and *numpy.random.seed(seed_value)*, respectively. With respect to hyperparameters, the loss function is optimized using the *Adam* [35] optimizer with a learning rate of 0.0003 for 20 epochs and batch size of 128 data samples. Hyperparameters for model size, such as number of layers and units, are described in the "Model Architecture" section above. Hyperparameter tuning was done manually and changes in network size had minor impact in the proposed approach performance.

4.3.6 Metrics

In real-world robotics based applications such as the one pursued in this work, standard machine learning metrics such as validation losses, and others that quantify how well a model has been trained with the given data, tend to not translate into actual high performance models that can successfully satisfy end-user requirements. To that effect, this research employs certain custom metrics to evaluate the performance of our model when deployed to a new test scenario and help demonstrate the effectiveness of using eye gaze as an additional supervision modality. The core metrics for comparison between the proposed model and baselines include:

- *Task Completion Rate*: this is evaluated by dividing the number of successful completions of the task by the total number of rollouts performed with that model. A successful task completion is defined as the quadrotor finding the target vehicle and intercepting it within a 5 meter radius.

- *Collision Rate*: this is evaluated by dividing the number of collisions during a rollout with the total number of rollouts performed with that model.

- *Success weighted by (normalized inverse) Path Length (SPL)*: as defined in [5], is evaluated for episodes with successful completion by dividing the shortest-path distance from the agent's starting position to the target location with the distance actually taken by the agent to reach the goal. In this work, it is assumed the shortest-path distance is a straight line from the initial quadrotor location to the target vehicle location, irrespective of obstacles.

Metric	Vanilla BC	Gaze BC
Test Completion Rate (%)	16.83 ± 2.84	***30.5** $\pm$ **4.47**
Collision Rate (%)	52.5 ± 6.21	**50.83** $\pm$ **5.97**
Weighted Path Length (SPL)	0.08 ± 0.02	**0.14** $\pm$ **0.03**

Table 4.1: Metric comparison in terms of mean and standard error between the proposed method, gaze-augmented imitation learning *Gaze BC*, and the main imitation learning baseline with no gaze information, *Vanilla BC*. Asterisk (*) indicates statistical significance.

4.4 Results

The proposed gaze-augmented imitation learning model, denoted *Gaze BC* in this section, is compared against a standard imitation learning model with no gaze information, denoted *Vanilla BC* in this section. Note that, when training our proposed gaze-augmented imitation learning model, the values of both λ_1 and λ_2 in Equation 4.1 are set at 1.0, while for training the baseline imitation learning model with no gaze information, λ_1 is set to 0.0 while λ_2 stays at 1.0.

In terms of policy performance, Table 4.1 provides a summary of the relevant metrics. Task Completion Rate can be seen to be higher for the *Gaze BC* model by approximately 13.7 percentage points. A *Wilcoxon signed-rank test*, a non-parametric statistical hypothesis test, was performed using the task completion rate values (one metric value per seed) for both models, to evaluate the statistical significance of the obtained results. A p-value of 0.0312 was obtained which shows that our *Gaze BC* model outperforms the *Vanilla BC* model in statistically significant terms.

Collision rate remains roughly the same for both the *Gaze BC* and the *Vanilla BC* model. This can be attributed to the fact that both models have access to the same observation space (*visual features*, *HOG features* and *kinematic features*) and thus both models have the information needed for collision avoidance.

SPL for the *Gaze BC* model can be seen to be 1.75x the SPL for the *Vanilla BC* model. This shows that whenever the models successfully accomplish the task, the *Gaze BC* model, on average, takes a significantly shorter path to reach the goal than the *Vanilla BC* model. This result again illustrates the superior performance of a policy trained with eye-gaze data.

Figure 4.6: *"Motion leading"* gaze pattern.

Figure 4.7: *"Target fixation"* gaze pattern.

Figure 4.8: *"Saccade"* gaze pattern.

Figure 4.9: *"Obstacle fixation"* gaze pattern.

Visualization of sequence of three frames illustrating characteristic gaze patterns observed in the human demonstrations, showed as magenta circles in the figure, which were also learned by the proposed model, showed as cyan circles (best viewed in color). The Larger, less transparent circle illustrates the current gaze observation and the smaller, more transparent circles represent gaze (and gaze prediction) from previous time-steps.

4.5 Discussion

This work introduces a novel imitation learning architecture to learn concurrently from human actions and eye gaze to solve tasks where gaze information provides important context. Specifically, the proposed method was applied to a visual search and navigation task, in which an unmanned quadrotor was trained to search for and navigate to a target vehicle in an asynchronous, photorealistic environment. When compared to a baseline imitation learning architecture, results show that the proposed gaze augmented imitation learning model is able to learn policies that achieve significantly higher task completion rates, with more efficient paths while simultaneously learning to predict human visual attention.

The closest related work to ours is Attention Guided Imitation Learning [79], or AGIL, which presented a two-step training process to train a gaze prediction network then a policy to perform imitation learning on Atari games, as explained in the *Related Work* section. There was no direct comparison to AGIL due to fact that AGIL currently operates on synchronous environments which can be paused, allowing for the game states to be synchronized with the human data. The data collection process for AGIL requires the advancing of game states one step at a time after the human performs each action. This is feasible when working with simple environments such as Atari where the game engine can be paused, however the aim was to develop an approach that could be translated to real-world applications running asynchronously in real time. Moreover, AGIL's architecture does not include any pretrained feature extractor, which significantly increases data collection requirements when dealing with vision-based robotics tasks like the one featured in this work.

With respect to the gaze prediction performance, four distinct gaze patterns observed during the

human demonstrations were also learned by the *Gaze Prediction Head* of the proposed model, as seen in Figure(s) 4.6, 4.8, 4.7, 4.9. These patterns were:

1. *"Motion leading"* gaze pattern where gaze attends to the sides of the images followed by yaw motion in the same direction, as illustrated in Figure 4.6. This pattern is mostly observed at the beginning of the episode when the target vehicle is not in the field-of-view of the agent and is seen towards the beginning of the episodes as it is a behavior picked up from human flying.

2. *"Target fixation"* gaze pattern where gaze is fixated at target during the final approach, as illustrated in Figure 4.7. In this pattern gaze is fixed at the top of the target vehicle, independent of the current motion of the agent. Consistency between actual eye-gaze and predicted gaze is a result of the distinct visual features of a unique target in the camera frame.

3. *"Saccade"* gaze pattern where gaze rapidly switches fixation between nearby obstacles [63], as illustrated in Figure 4.8. This pattern is characteristic when there are multiple obstacles between the current agent location and the target vehicle.

4. *"Obstacle fixation"* gaze pattern where gaze attends to nearby obstacles when navigating to the target, as learned from multiple demonstrations, which might differ from the current obstacle user is attending to at the moment, as illustrated in Figure 4.9. This pattern is mostly observed when the quadrotor is in close distance to an potential obstacle.

This illustrates how the *Gaze Prediction Head* of the proposed model was able to capture and replicate similar visual attention demonstrated by the user when performing the task. The gaze patterns also confirm the hypothesis that the agent learnt representations of the visual scene that were not simply 'instructing' control to an eye-gaze notified location but learning control that was conditioned on both the area of attention and the type of object being attended to.

With respect to the selection of loss weighting parameters, it is to be noted that overweighting of the gaze prediction task results in the inability to learn an effective control policy in the same training

time (number of epochs) due to over-focusing on gaze prediction performance. It is important to note that training for a sufficiently large number of epochs while eventually lead to both the behavior cloning and gaze prediction losses to converge to their lowest possible values. However, when training for a fixed number of epochs, it is critical to ensure that the gradient updates for the model lower both the behavior cloning and gaze prediction losses at a similar scale to avoid any one task from dominating the other in the training process.

4.5.1 Limitations and Future Work

A limitation of the current work is not being able to complete the proposed task with a higher rate. This can be attributed to one simple reason, as in all end-to-end learning approaches: the limited training dataset. This issue is accentuated in human-in-the-loop machine learning tasks, such as the one used in this work, since human-generated data is scarce. For this work the only data available was a set of 200 human-generated trajectories and believe that task completion rate could increase by simply training the model with more data. This additional data could come from more independent trajectories, by labelling agent-generated trajectory and aggregating to the original dataset [59], or by tasking the human to overseer the agent and perform interventions when the agent's policy is close to fail, also aggregating this intervention data to the original dataset [26]. In this case, the ability of a human trainer to enact timely interventions may be greatly improved by the gaze predictions the model generates, which would highlight instances where this artificial gaze deviates from what they expect, potentially indicating subsequent undesirable behavior and providing some level of model interpretability or explainability.

Another possible avenue for future work is the addition of a one-hot encoded task specifier in the input space of the model, which could aid in training models that learn, like humans, to adapt their attention and behavior according to the task at hand. Furthermore, the integration of additional human input modalities to the proposed approach, such as natural language, could similarly be used to condition the model to perform multiple different tasks. The combination of gaze and natural language would enable humans to more naturally interact with learning agents, and enable those agents to disambiguate demonstrator behavior and attention dynamics that are otherwise

ambiguous because they are pursuant to distinct tasks and goals that could be grounded to language commands. The ability to use eye-gaze data to leverage a human's visual attention opens the door to adapting this research to learning unified policies that can generalize across multiple context- and goal-dependent, tasks.

5. CASE STUDY: LEARNING FROM HUMAN INTERACTION ON A HARDWARE PLATFORM

5.1 Problem Definition and Background

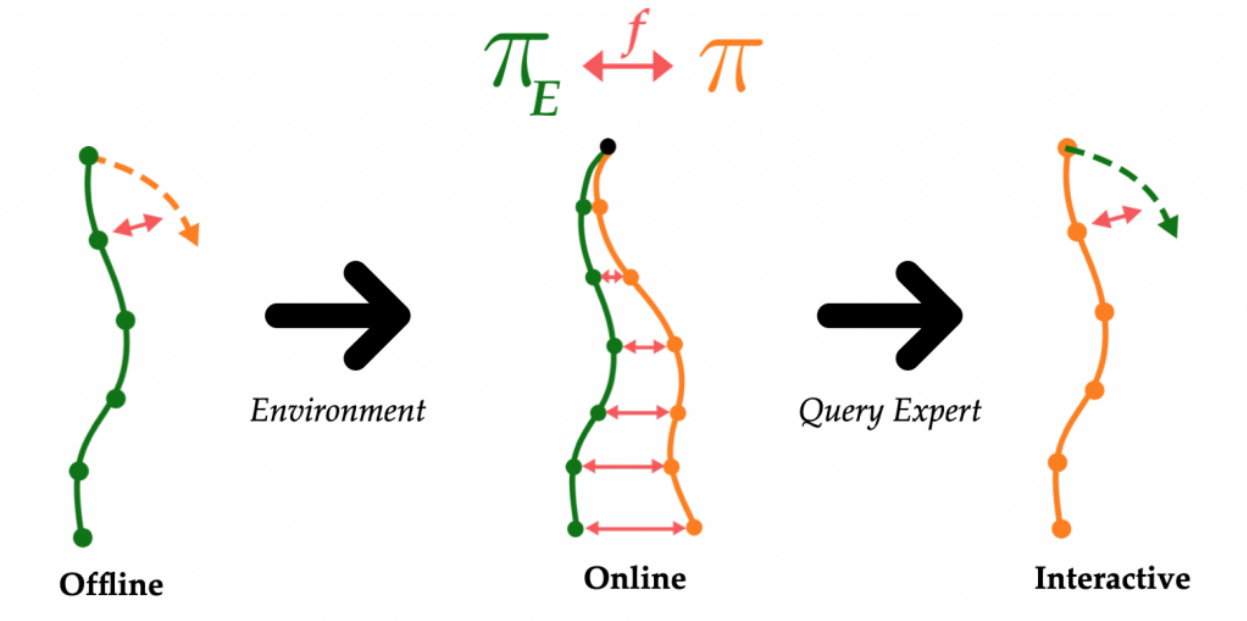

Figure 5.1: An overview of imitation learning paradigms. Reprinted from [2]

Approaches for real-time machine learning on actual vehicles, that depend on interaction with the world (such as reinforcement learning) have a major limitation associated with them which is that the agent needs to have the chance to try (and fail) many times. This is a tough constraint when operating in the real-world when safety is a concern and catastrophic failure is not an option. This is also quite impossible in general in real life where each interaction takes time (in contrast to simulation). This is where imitation learning comes into the picture as it allows for offline and non-interactive learning from a saved log of expert actions. This work attempts to demonstrate a combination of offline and safe interactive learning on hardware.

There are primarily three approaches to imitation learning:

- **Offline**: This approach involves an expert operator execute the trajectories with the states/observations and actions being simultaneously recorded. The recorded data is then used by a supervised learning algorithm to regress from states to actions. Divergence between learner and expert actions on states from expert demonstrations is measured and utlized by the supervised learning algorithm. This is also known as behavior cloning and does not require the physical involvement of the vehicle during the training process.

- **Online**: This approach involves the vehicle autonomously executing a trajectory (using a learner policy model) from the same start location as a demonstration trajectory. The delta between the learner and expert trajectories is measured and the policy model is trained to minimize this delta. Generative Adversarial Imitation Learning (GAIL) [29] is an algorithm that uses a discriminator network to measure this delta.

- **Interactive**: The interactive approach is a version of the offline approach but attempts to cover the gaps in the policy model's knowledge of the data distribution by seeking interventions in areas of poor model performance. An intervention essentially auto-labels the regions of the input space where the policy model performs poorly. This approach essentially follows the below routine:

  - Step 0: Start with an empty dataset $\mathcal{D}$
  - Step 1: Record the vehicle executing a sample trajectory.
  - Step 2: Seek intervention where model performs poorly. Append this data to $\mathcal{D}$.
  - Step 3: Regress over data in $\mathcal{D}$.
  - Step 4: Go back to 1

  Divergence between learner and expert actions but on states from learner rollouts is measured. This approach is known as DAgger [46].

5.2 Implementation

5.2.1 Model Design

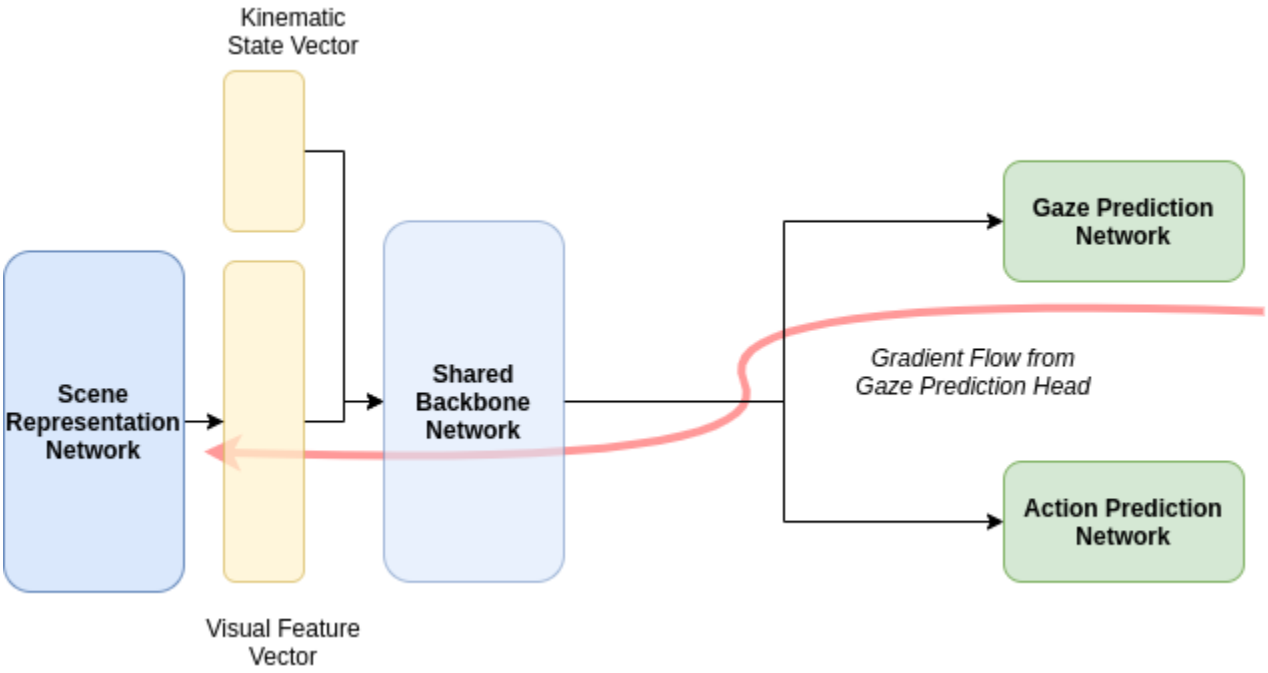

Figure 5.2: Backpropagated gradients from the Gaze Prediction Head update the trainable weights of the Scene Representation Network

A simplified version of the core model architecture used in the work discussed in Chapter 4 is shown in Fig. 5.2. Backpropagated gradients that flow from the Gaze Prediction Network update the weights in the trainable layers of the Scene Representation Network. This results in the scene representation network learning a representation of the visual scene that is specific to the task being performed by the expert demonstrator. In order to simplify the architecture and allow for "unit-testing" of human-in-the-loop learning, the scene representation network's vector output is substituted with *AprilTag*(s)[51] detection vectors that essentially represent the spatial location of the tag in the image plane. This offers the advantage of demonstrating learning from visual attention without needing to rely on noisy eye-gaze signal data from the operator. This model is shown in Fig.

5.3.

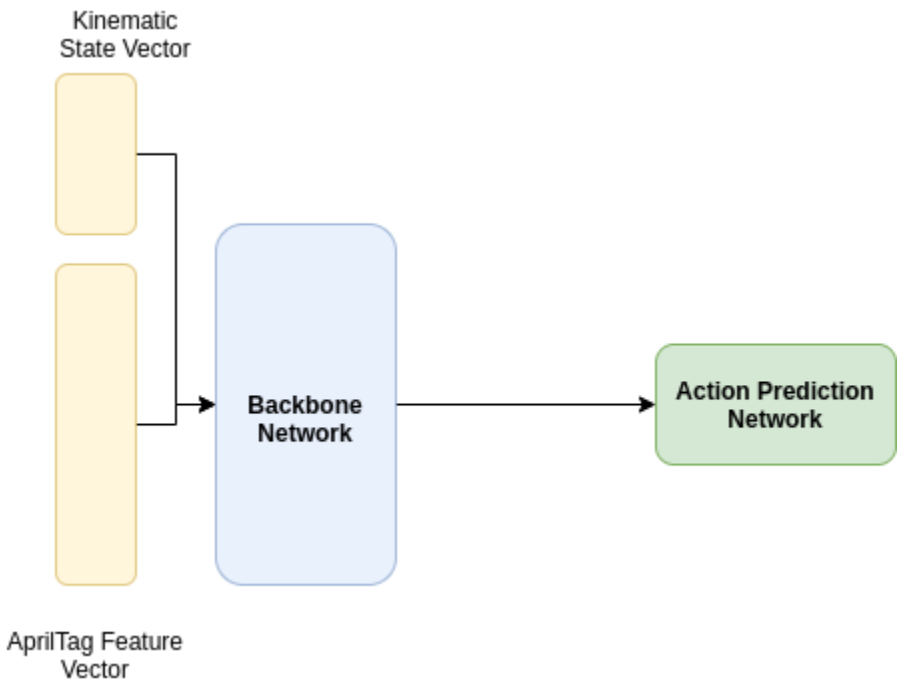

Figure 5.3: Reduced model representation

In order to prevent overfitting to values in the kinematic state vector, the model relies on only the AprilTag feature vector as input. This improves the policy model's ability to generalize to new initial locations. Another advantage of using only AprilTag features as the input to the model means that policies can be trained in simulation and rolled out directly in the real physical environment since AprilTag features are domain invariant and do not differ between simulation and reality. AprilTag(s) were also chosen for their quick detection times and minimal need for any parameter tuning for the detection process itself.

5.2.2 Training Objective

The training objective for the policy model is based on the general loss function formulation for multi-objective optimization.

Alternatively, MTL can be formulated as multi-objective optimization: optimizing a collection

of possibly conflicting objectives. This is the approach taken in this work. The multi-objective optimization formulation of MTL is specified using a vector-valued loss L.

The goal of multi-objective optimization is achieving Pareto optimality, defined mathematically as:

$$\min_{\theta_{sh},\theta_1,...\theta_T} \mathbb{L}(\theta_{sh},\theta_1,...\theta_T) = \min_{\theta_{sh},\theta_1,...\theta_T} \mathbb{L}(\theta_{sh},\theta_1) + \mathbb{L}(\theta_{sh},\theta_2) + ...$$

In the specific case of gaze-augmented behavior cloning, the above loss function is expressed as:

$$\mathcal{L}(\theta) = \lambda_1 * \mathcal{L}_{GP} + \lambda_2 * \mathcal{L}_{BC}$$
$$\mathcal{L}_{GP} = \left\| \left(\pi_{gaze} - g^t\right) \right\|^2$$
$$\mathcal{L}_{BC} = \left\| \left(\pi_{action} - a^t\right) \right\|^2$$

In the case of learning with demonstrations and interventions combined, λ_1 is set to zero. Since interventions are essentially partial demonstrations, the mathematical formulation is identical to one for standard behavior cloning.

5.2.3 Experimental Configuration and Hardware Setup

The experiments were conducted using a *Parrot Anafi* [4] in an indoor environment where a yellow cupboard was marked as an object of interest using an AprilTag. The task involves flying to the object of interest purely visually, from a set of unique initial locations. The set of unique locations are chosen while ensuring that the object of interest is within the the initial field of view.

For the experiment involving learning with interventions, the input to the policy model consists of the camera frame coordinates of the detected AprilTag. The episode ends in successful task completion when the AprilTag's projected area in the camera's frame crosses a threshold value, as this serves as a metric of proximity to the target.

For the experiment involving learning from eye-gaze, the input to the policy model consists of the raw camera frame image. The policy model consists of a series of pretrained layers on the input side. These are essentially the *MobileNet* layers except the output layer. The episode ends in

Algorithm 1 Human-in-the-Loop Policy Learning

1: **procedure** MAIN
2: Instantiate and initialize control policy π
3: Record expert dataset from human demos $\mathcal{D}_H$
4: Define termination criteria $\mathbb{T}$
5: **while** termination criteria is **false do**
6: Read observation o
7: Run policy model (π) inference
8: Sample action $a_\pi \sim \pi$
9: **if** Human Override detected (i_H) **then**:
10: Perform human action a_H, record any passive human modality i_H / a_H
11: Add o, i_H to $\mathcal{D}_H$
12: **else**
13: Execute a_π
14: **if** End of Episode **then**
15: Run *Update Policy* procedure (line 16)
16: **procedure** UPDATE POLICY
17: Initialize $loss$ threshold $loss_{TH}$
18: Load human dataset $\mathcal{D}_H$
19: **if** New Samples **then**:
20: **while** $loss < loss_{TH}$ or n epochs **do**
21: Sample N samples o, i from $\mathcal{D}_H$
22: Sample $\hat{a} \sim \pi$
23: Compute $loss(\hat{a}, a_H, o)$
24: Update agent policy π

successful task completion when the pixel coverage of the yellow cupboard exceeds one-third of the total number of pixels in the camera frame.

Task completion is automatically detected and once the detection triggers in, any autonomous control command gets overriden and set to idling values for safety. Manual control of the vehicle is available to allow for repositioning before the next episode. An episode/rollout that involves teleoperation for any portion of the executed trajectory is termed as a *human interaction episode*. The overall setup is illustrated in Figure 5.4.

Demonstrations and rollouts are collected in a manner such that each trajectory has a different initial configuration (x, y and θ_z). A top-down spatial overview with illustrative examples for a training and evaluation trajectory is presented in Fig. 5.4. Different randomized initial yaw positions are used for the same spatial location. The initial state configuration sets for training and evaluation are kept non-overlapping to prevent overfitting models from inaccurately influencing metrics such as task completion rate as they would perform better when used in same regions of the state space as the training data. Moreover, multi-task models (such as the one developed in this research) are implicitly regularized due to the multiple objectives they are trained on. They will tend to perform worse than the non-regularized models when evaluated in the training set itself and their generalization capabilities wouldn't be accurately reflected.

For the experiment investigating the effect of sampling data through interventions as well (as opposed to just demonstrations), a set of initial demonstrations are provided. Following this a policy trained from scratch on these initial seed demonstrations is rolled out while keeping the human-in-the-loop. A manual intervention is permissible at any stage and gets recorded to disk. At the end of each episode where an intervention occurred, the policy is tuned again by training a randomly initialized model on the aggregate set of human-collected samples so far.

The AprilTag feature vector is prepared by extracting the x, y coordinates of the corners of the AprilTag. These coordinates are then normalized using the height and width of the image frame the

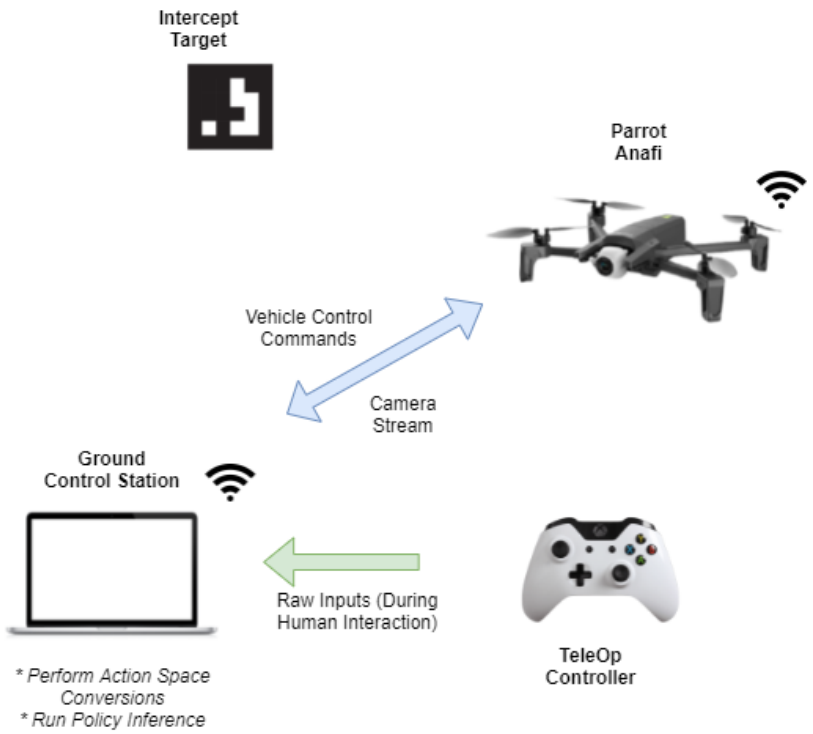

Figure 5.4: An overview of the setup used for the experimental operations

AprilTag detector operates on. Thus, the visual feature vector can be expressed as:

$$f_t = [x_1\ y_1\ x_2\ y_2\ x_3\ y_3\ x_4\ y_4]^T$$

5.2.4 Eye Tracker Calibration

Calibration is the process whereby the geometric characteristics of the human operator's eyes are estimated as the basis for a fully-customized and accurate gaze point calculation.

Before an eye tracking recording is started, the user is taken through a calibration procedure. During this procedure, the eye tracker measures characteristics of the user's eyes and uses them together with an internal and anatomical correct, 3D model of the human eye to calculate the requisite gaze data. This model includes information about shapes, light refraction and reflection properties of the different parts of the eyes (e.g. cornea, placement of the fovea, etc.). During the

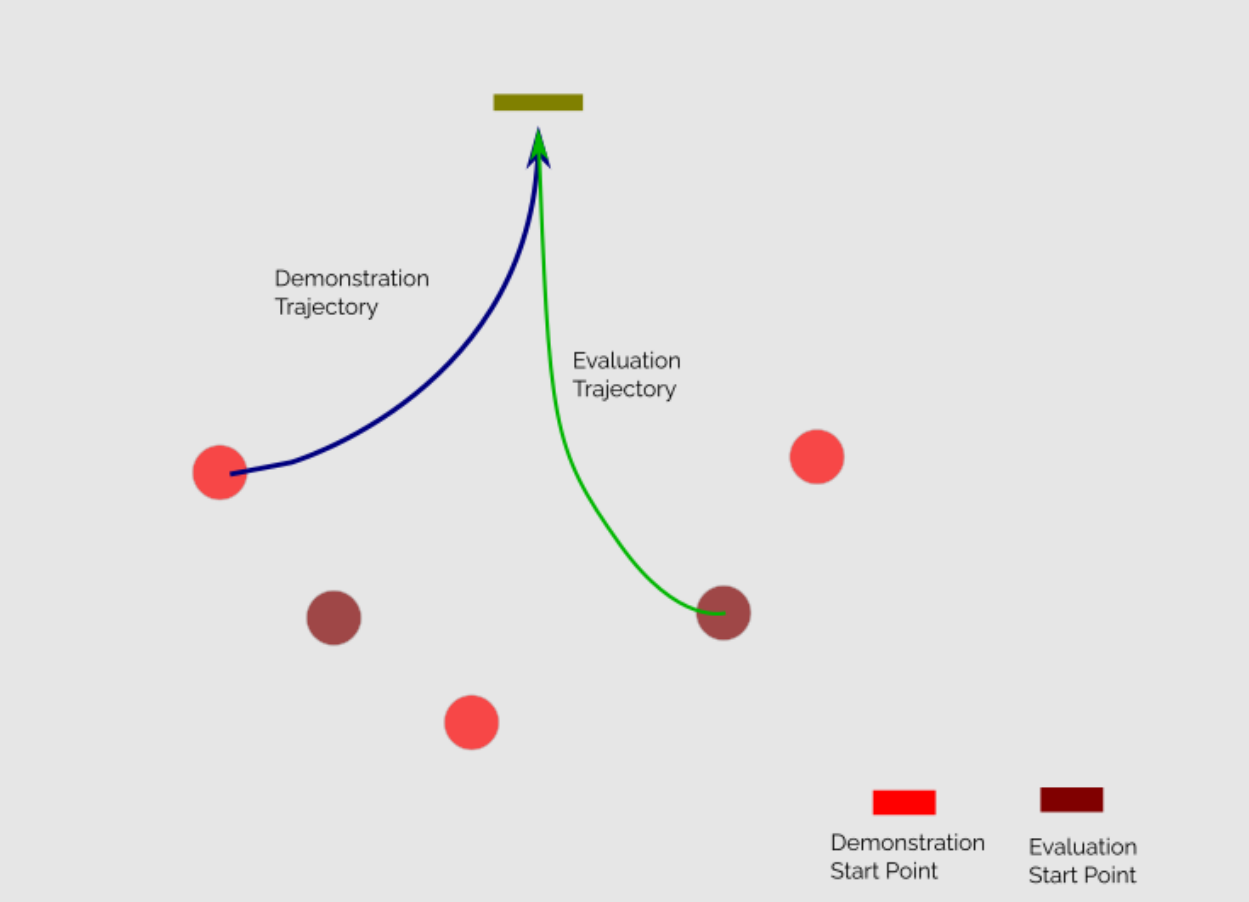

Figure 5.5: An overview of the spatial setup for the hardware experiments. The non-overlapping sets of starting configurations between the training and evaluation sets is highlighted.

calibration the user is asked to look at specific points on the screen. These specific dots are also known as calibration dots. During this period several images of the human operator's eyes are collected and analyzed. The resulting information is then taken into account along with the eye model and the gaze coordinates (abcissa and ordinate in the screen frame) for each image sample are calculated.

When the procedure is finished the quality of the calibration is illustrated by green lines of varying length. The length of each line represents the offset between each sampled gaze point and the center of the calibration dot. Large offsets (long green lines) can be caused by various factors such as, the user not actually focusing on the point, the user being distracted during the calibration or the eye tracker not being set up correctly. However, the user does not have to keep the head completely still during calibration as long as the focus of the user's eyes is kept on the moving dots. The dependence of the calibration procedure on the geometric and other related characteristics means that the calibration process needs to be redone whenever the human operator is switched or changes in seating position occur.

There are multiple ways to calculate the gaze coordinates and the two primary ways supported by the *Tobii* eye-tracking hardware are the light and dark pupil methods [1]. The bright pupil eye tracking method is the one where an illuminator is placed close to the optical axis of the imaging device causing the pupil to appear lit up (similar to the same phenomenon that causes red eyes in photos), and dark pupil eye tracking, where an illuminator is placed away from the optical axis causing the pupil to appear darker than the iris. During the calibration both the light and dark pupil methods are tested to identify the most suitable for the current light conditions and the user's eye characteristics.

5.2.5 Hardware Platform Selection

The Parrot Anafi was chosen as the hardware platform of choice for these experiments due to high-fidelity video streaming capabilities coupled with a highly stable hovering mode. A highly stable hovering mode is essential in ensuring that the human demonstrations can be provided intermittently. Intermittent teleoperation reduces the cognitive workload on the human operator

while ensuring there is enough time provided to the human operator to properly perceive the whole observation (image frame). This is necessitated by the fact that the central vision cone of a human is minimal and continuous teleoperation would lead to parts of the image frame being missed due to poor peripheral vision. Intermittent teleoperation is suited for policy models that are built for reactive control and are Markovian in nature.

The Parrot Anafi's hover stabilization works as follow: while hovering, the vehicle's vertical camera captures a reference frame. It is then compared to subsequent shots taken at a frequency of 15Hz. The algorithm calculates the camera movement that would minimize the reprojection error between the reference photo and more recent one. This movement is then used as an instruction for the autopilot. This method ensures that the Anafi is stable within a 1.5 cm radius-sphere at 1 m height. The algorithm also allows for yaw stabilization and contributes to the overall image stabilization performance.

The other mission-critical aspect of the vehicle is it's video streaming capability. Low-latency and tear-free streaming of first-person video is critical for a controller that relies solely on visual observations to generate commands to pilot the vehicle. The high-definition video stream is designed to minimize the impact of packet losses and to dilute any spatial errors. The algorithm combines slice-encoding and a periodic partial refresh of the frame at a high frequency. The video streaming algorithm encodes images as 45 slices of 16 pixels height each. It then refreshes all those slices in batches of 5 slices each, every 3 images. The refresh is complete every 29 images.

Lastly, the Parrot Anafi interfaces with the Python runtime on the Ground Control Station (GCS) computer through a message-passing mechanism. This message-passing mechanism not only controls the vehicle but also provides hardware debugging information in real-time on request. This allows the building of highly robust software systems for hardware operation that can adjust execution routines to handle hardware failures or other exceptions that degrade mission capability. This prevents the need to specifically train human operators to handle such circumstances. This exception handling is discussed in detail in Chapter 3.

5.2.6 Model Design

The policy model is a feed-forward neural network with two hidden layers having 16 and 8 units each. The output is a 3-dimensional vector. The components of this 3-dimensional vector are linear velocity along x-axis, linear velocity along y-axis and rotational velocity along z-axis. The input to the model (the observation space) is a scene representation vector, that is essentially an 8-dimensional vector consisting of the x,y coordinates of the 4 corners of the AprilTag marking the target for interception.

Exponential Linear Units (ELUs) [18] in contrast to ReLUs have negative values which allows them to push mean unit activations closer to zero like batch normalization but with lower computational complexity. Mean shifts toward zero speed up learning by bringing the normal gradient closer to the unit natural gradient because of a reduced bias shift effect. Contrary to other forms of normalization such as input normalization and batch normalization, normalization through activation functions such as ELUs is implicit and the propagation through multiple layers of the neural networks still results in the outputs converging towards a zero mean and unit variance distribution.

Deep learning is known to be effective at discovering underlying patterns from massive amounts of data, even with noisy outputs used as ground truth values. However, when extremely limited data is present, neural networks memorize the samples and learn to model the noise instead of just the underlying signal [78]. This defeats the purpose behind pattern recognition. The alternative is to provide as accurate an output signal as possible. Hence, to counter the issue of the model fitting to the noise, the following two steps are taken:

- **Appropriate Model Complexity**: The neural network policy is designed to keep the total number of trainable parameters lesser than the number of data points. Overfit models that learn to model the noise seen in the outputs usually do so because the number of unique data points seen during training are lesser or the same as the number of trainable parameters (analogous to solvable unknowns in a system of equations). Hence, it is important to ensure that the complexity of a model (in terms of number of trainable parameters) is at least the

same order of magnitude as the number of training samples available.

- **Removing Stochasticity from Ground Truth Data**: The raw human piloting data obtained is post-processed using the following transformation:

$$a_{expert} = \text{sgn}(a_{expert\_raw}) * GV$$

The gain value (GV) essentially controls the magnitude of the respective component velocities and is represented as a fraction of the maximum velocity permissible for that component of the action. The gain value is set to 0.5 for the purposes of this research. The backward propagated gradient with a mean-square error loss is essentially the difference between the predicted and ground-truth values. Since, the neural networks are initialized with near-zero weights, setting a low GV will result in low magnitude gradients being backpropagated thus significantly slowing down training. Despite the demonstration actions (used as a training signal) assuming only a finite set of values, MSE is used as the training objective since a mean squared error loss is well suited for underlying data distributions that are continuous and normal or similar.

Removing stochasticity in such a way, ensures that not only is the output data fed to the model during training is as close as possible to the output signal but also is resilient to any outlier data. Outliers in training data can significantly the trained model's prediction performance since model's trained with mean square error losses converge to the average of the outputs paired with a given input.

The outputs of the policy model (allowed to have continuous values) are normalized into a unit vector so that neural network determines just directional movement of the vehicle and the speed at which it executes the commanded trajectory can be hard-coded based on preferences related to safety, completion time constraints etc.

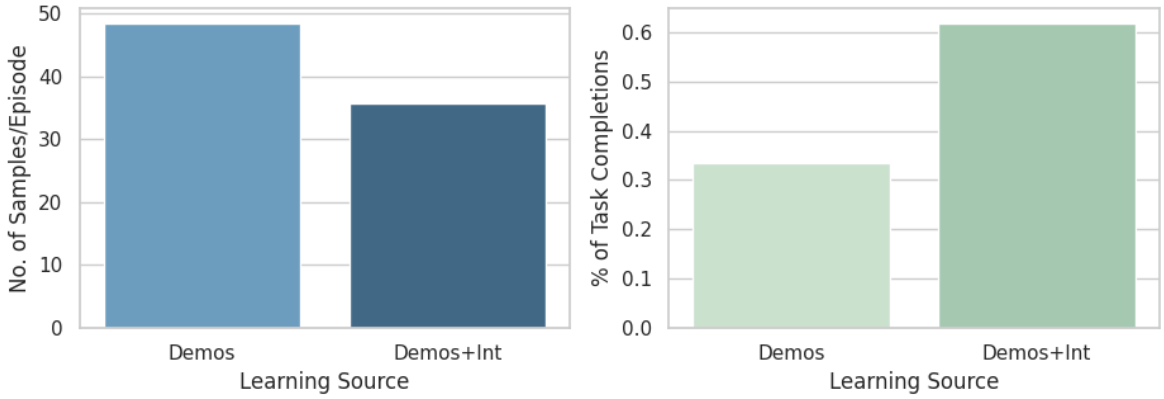

Figure 5.6: Task Completion Rates and Teleoperation Input Required compared for models trained on demos only and demos+interventions

5.3 Results

Fig. 5.6 shows the comparison between two models. Both are trained with 4 human interaction episodes but utilize different methods of sampling human data. The former contains 4 demonstration episodes while the latter consists of 2 demonstration episodes and 2 intervention episodes. Both models were evaluated over a set of 20 rollouts each. The results shown in Fig. 5.6 confirm the hypothesis that learning from both demonstrations and interventions results in lesser data requirement than the scenario in which we learn from demonstrations alone, for the same number of episodes of human interaction. With this context the higher task completion rate exhibited by the control policy trained with both demonstrations and interventions is even more significant. A two-proportion z-test conducted on the task completion rate values for both the models yielded a p-value of 0.029. This result further supports the hypothesis that a hybrid learning strategy inherently seeks more useful training data from the human observer (similar to active learning).

Fig. 5.7 shows the comparison between a policy model trained using gaze regularization and another trained without. Both models are evaluated over a set of 40 rollouts each. The plot shows how using eye-gaze as a training signal (under an appropriate loss weighting scheme) results in

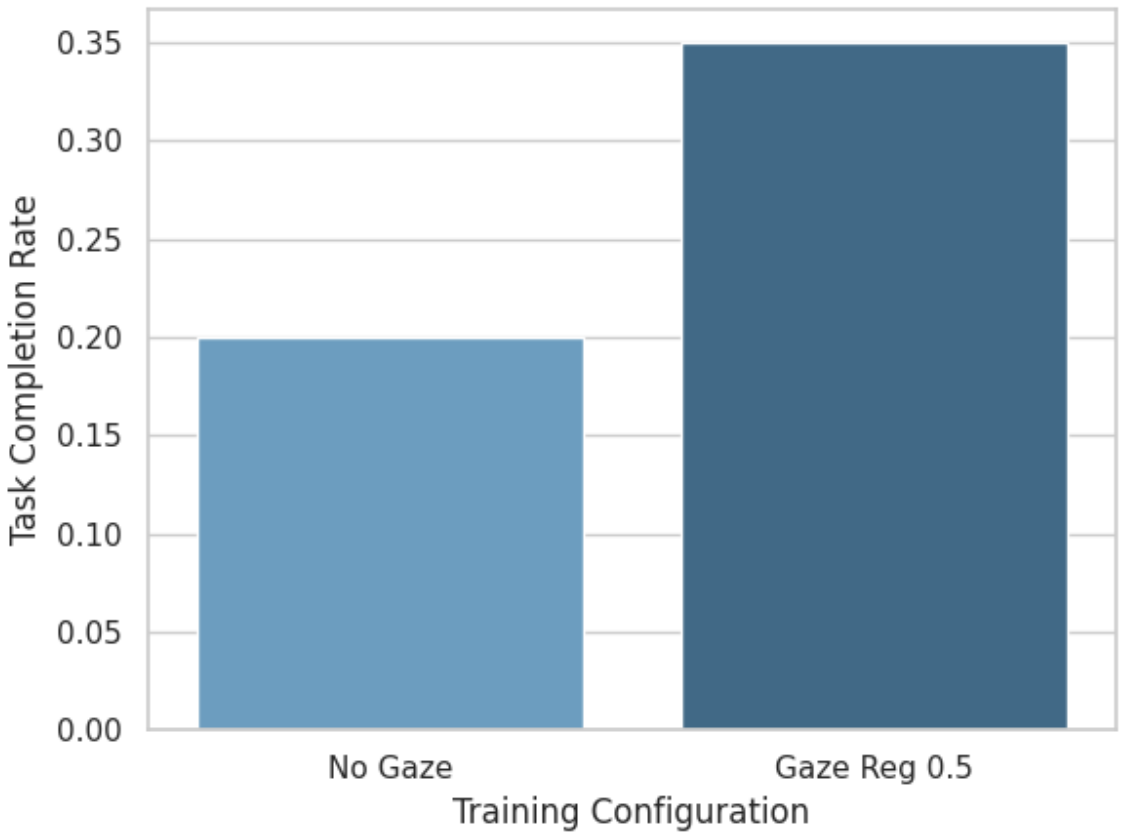

Figure 5.7: Task completion rates for policy models trained with different degrees of gaze regularization. Task completion rates are calculated by averaging binary success indicators over the total number of rollouts (40) for each policy model.

higher task completion rate (0.35 vs 0.25). A two-proportion z-test conducted on the task completion rate values for both the models yielded a p-value of 0.067. This demonstrates the effectiveness of visual attention data in learning representations of image data that help in visual servoing.

5.4 Discussion

Besides resulting in a higher task completion rate with a lesser degree of piloting input (as seen in the results above), learning from interventions also has another qualitative advantage. When implementing the human-demonstration driven learning in real world environments, catastrophic failures may be seriously damaging to the autonomous agent, and thus unacceptable. Having a human observer capable of intervening provides a mechanism to prevent this inadmissible outcome. Further, alternative techniques that attempt to enforce a similar safety override through hand-engineered code might implicitly limit the exploration of the state space, yielding a less general or less capable policy. Hand-engineering safety overrides might also not be exhaustive in their coverage of various scenarios. In conjunction with eye-gaze, learning from interventions could help teach agents control policies that are more visually aware while trying to avoid ending up in failure states.

6. CONCLUSIONS

The following conclusions are made based on the results presented in this thesis:

1. Learning from additional human sensory modalities in addition to learning from demonstrations leads to higher task completion and performance in tasks that require joint perception and reasoning. As seen in the work described in Chapter 4, an increase of 13.7 percentage points in task completion was observed when the policy models learnt from both human eye-gaze data and teleoperation commands as opposed to teleoperation commands only.

   Policies trained with eye-gaze achieve a more direct routing to the goal in visual navigation tasks as is evident in the Success-weighted Path Length metric (0.14 for gaze-regularized vs 0.08 unregularized).

   The two results in combination demonstrate the significant role played by visual attention in guiding directional control of an agent. This is especially important for visual navigation in unstructured and cluttered environments.

2. A hybrid mix of demonstrations and interventions (total interaction episodes equally split between the two) has a higher task completion rate (65% vs 35%).

   The hybrid mix also requires a lesser number of human samples (27.1% lower) while achieving higher task completion indicating interventions provide data points that help fit a more general (that fits better across the whole input space) model. This illustrates how interventions provide a richer feedback signal to the agent and the data samples collected during interventions effectively cover regions of the state space where the control policy model's fit is poor at that instant.

3. Learning from demonstrations using eye-gaze as an additional training signal is demonstrated to result in a higher task completion rate (0.35 as opposed to 0.25 for standard behavior cloning) given the same number of demonstrations (10) on a *Go To Target* task using a Parrot

Anafi and relying on just pure visual observations. This reinforces the idea of visual attention data being helpful in learning representations of image data that are well suited for visual servoing.

4. Development of a codebase in compliance with the *SOLID* principles of program design gives the ability to run experiments with varied configurations while resulting in execution of majority amount ($> 50\%$) of shared (configuration-agnostic) code. This consequently results in the ability to add functionality (such as support for a new vehicle or sensor platform) without the overhead of needing to write boilerplate code. Another advantage of a large amount of shared code execution is the ability to write plugins that universally work with different kinds of experimental configurations.

7. RECOMMENDATIONS

Several recommendations are made based on the research in this thesis:

1. A step towards enhancing the feasibility of eye-gaze as human interaction modality would be to investigate filtering techniques to make eye-gaze a more reliable training signal. Inattentiveness and saccades corrupt the training signal provided through eye gaze input and thus make any gradient feedback obtained from a gaze prediction task useless.

   Designing hand-engineered rejection mechanisms to filter out inattentiveness coupled with noise filtering to smooth out the training signal could make learning better and improve reliability of the eye-gaze training data by making sure it models the signal instead of noise.

2. Learning from visual attention requires perception networks that are specifically designed to operate on ultra high-dimensional input and have a finite output space. This output space is a discrete set containing spatial locations. Using neural networks better suited to model visual attention could help in gaining policy convergence in a much more data efficient manner making learning from gaze amenable to real-time learning.

   Networks that are well suited for this kind of problem include Graph Neural Networks and Graph Convolutional Neural Networks. These networks have a significantly lower set of connections and parameters and are thus more sample-efficient. The structure of the said neural networks models forces gaze predictions to be more spatially aware and entity-focused rather than being a generic non-parametric distribution over the entire image observation. This could help in gaining policy convergence in a much more data efficient manner making learning from gaze amenable to real-time learning.

3. A core requirement is the need to minimize the need to collect data in the real world and validate policies on physical vehicles. Incorporation of any reinforcement-learning based technique in the policy training process would also require having a fairly realistic simulated

environment that transfers well to the real world.

This can be enabled by incorporating research in domain adaptation and *sim2real* techniques focused on making control policies transferable from simulation to real-world environments. *Sim2Real* refers to a concept of transferring robot skills acquired in simulation to the real robotic system. *Sim2Real* draws its appeal from the fact that it is cheaper, safer and more informative to perform experiments in simulation than in the real world. Further investigation could cover advances in imitation from observations and transfer learning in simulation.

4. Sample-efficient behavior cloning would require transitioning away from neural networks towards models that fit better with fewer data points provided. This can lead to better policy maturity with lesser human piloting needed.

   One way forward to accomplish this would be to use Gaussian Processes for regression problems. Gaussian Processes are extremely sample-efficient and through the *Expected Improvement* metric, provide a good understanding of the existing gaps in a model's knowledge. Also, the continuous and differentiable nature of Gaussian Process functions ensures that nominal variations in the input space do not lead to large variations in the output space.

5. Another avenue of improvement would be the incorporation of active learning techniques for more efficient sampling of data while filling in the gaps in the policy model's understanding of the state space. This would alleviate the cognitive workload imposed on a human operator throughout a vehicle rollout by not relying solely on their judgement to decide when to intervene.

   Uncertainty quantification of neural network predictions can help in identifying regions of the state-space where a neural network's predictions have high variance/uncertainty. Interventions in such regions even when catastrophic failure is not imminent can improve the policy's performance.

6. The approach towards multimodal learning in this work is still based on behavior cloning and is not immune to its flaws such as overfitting. Moreover, a supervised learning approach

implies that policy performance will always be limited by the quality of the demonstration provided.

More robust approaches to imitation learning such as Generative Adversarial Imitation Learning could be explored as the skeletal framework to which additional modalities can be added in a manner similar to the one described in this work.

REFERENCES

[1] What happens during the eye tracker calibration. https://www.tobiipro.com/learn-and-support/learn/eye-tracking-essentials/what-happens-during-the-eye-tracker-calibration/. Accessed: 2021-08-18.

[2] A unifying, game-theoretic framework for imitation learning. https://blog.ml.cmu.edu/2021/07/30/a-unifying-game-theoretic-framework-for-imitation-learning/. Accessed: 2021-08-18.

[3] Yaser S Abu-Mostafa. Learning from hints in neural networks. *Journal of complexity*, 6(2): 192–198, 1990.

[4] Evan Ackerman. Parrot's new drone reclaims aniche: The anafi marks the company's return to theconsumerspace - [resources review]. *IEEE Spectrum*, 55(9):21–21, 2018. doi: 10.1109/MSPEC.2018.8449041.

[5] Peter Anderson, Angel X. Chang, Devendra Singh Chaplot, Alexey Dosovitskiy, Saurabh Gupta, Vladlen Koltun, Jana Kosecka, Jitendra Malik, Roozbeh Mottaghi, Manolis Savva, and Amir Roshan Zamir. On evaluation of embodied navigation agents. *CoRR*, abs/1807.06757, 2018. URL http://arxiv.org/abs/1807.06757.

[6] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[7] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, USA, 2012. ISBN 0521518148.

[8] Jonathan Baxter. A model of inductive bias learning. *Journal of artificial intelligence research*, 12:149–198, 2000.

[9] David Beazley. Understanding the python gil. In *PyCON Python Conference. Atlanta, Georgia*, 2010.

[10] Ritwik Bera, Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. Gaze-informed multi-objective imitation learning from human demonstrations, 2021.

[11] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 2nd edition, 2000. ISBN 1886529094.

[12] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[13] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. URL `http://arxiv.org/abs/1604.07316`.

[14] Han Cai, Kan Ren, Weinan Zhang, Kleanthis Malialis, Jun Wang, Yong Yu, and Defeng Guo. Real-time bidding by reinforcement learning in display advertising. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 661–670, 2017.

[15] Rich Caruana. Multitask learning. Technical report, Carnegie Mellon University, 1997.

[16] Richard Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[17] Elena Chebanyuk and Krassimir Markov. An approach to class diagrams verification according to solid design principles. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 435–441. IEEE, 2016.

[18] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[19] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE International*

Conference on Computer Vision, pages 9329–9338, 2019.

[20] N. E. Cotter. The Stone-Weierstrass Theorem and its Application to Neural Networks. *IEEE Transactions on Neural Networks*, 1(4):290–295, Dec 1990. ISSN 1941-0093. doi: 10.1109/72.80265.

[21] G. Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989. ISSN 0932-4194. doi: 10.1007/BF02551274.

[22] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, volume 1, pages 886–893. IEEE, 2005.

[23] Anup Doshi and Mohan M Trivedi. Head and eye gaze dynamics during visual attention shifts in complex environments. *Journal of vision*, 12(2):9–9, 2012.

[24] Amnon H Eden, Yoram Hirshfeld, and Rick Kazman. Abstraction classes in software design. *IEE Proceedings-Software*, 153(4):163–182, 2006.

[25] Ken-Ichi Funahashi. On the Approximate Realization of Continuous Mappings by Neural Networks. *Neural Networks*, 2(3):183 – 192, 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90003-8.

[26] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. Efficiently combining human demonstrations and interventions for safe training of autonomous systems in real-time. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 2462–2470. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33012462.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[29] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29:4565–4573, 2016.

[30] K. Hornik, M. Stinchcombe, and H. White. Multilayer Feedforward Networks Are Universal Approximators. *Neural Networks*, 2(5):359–366, July 1989. ISSN 0893-6080.

[31] Kurt Hornik. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, 4(2):251 – 257, 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90009-T.

[32] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[33] Bilal Hussein and Aref Mehanna. A design pattern approach to improve the structure and implementation of the decorator design pattern. *Research Journal of Applied Sciences, Engineering and Technology*, 13(5):416–421, 2016.

[34] Yoshifusa Ito. Representation of Functions by Superpositions of a Step or Sigmoid Function and Their Applications to Neural Network Theory. *Neural Networks*, 4(3):385–394, June 1991. ISSN 0893-6080. doi: 10.1016/0893-6080(91)90075-G.

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[37] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[38] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[39] Vladik Ya. Kreinovich. Arbitrary Nonlinearity is Sufficient to Represent All Functions by Neural Networks: A theorem. *Neural Networks*, 4(3):381 – 383, 1991. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(91)90074-F.

[40] Sergey Levine. UC Berkeley CS 294: Deep Reinforcement Learning, Lecture Notes, 2018. Last visited on 2019/09/01.

[41] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep

visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[42] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[43] Congcong Liu, Yuying Chen, Lei Tai, Ming Liu, and Bertram E Shi. Utilizing Eye Gaze to Enhance the Generalization of Imitation Networks to Unseen Environments. *CoRR*, abs/1907.0, 2019. URL `http://arxiv.org/abs/1907.04728`.

[44] Robert C Martin. The open-closed principle. *More C++ gems*, 19(96):9, 1996.

[45] Robert C Martin. Getting a solid start. *Robert C. Martin-objectmentor. com*, 2013.

[46] Kunal Menda, Katherine Driggs-Campbell, and Mykel J. Kochenderfer. Ensembledagger: A bayesian approach to safe imitation learning, 2018.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[48] Michael A Nielsen. *Neural Networks and Deep Learning*, volume 25. Determination press San Francisco, CA, USA:, 2015.

[49] Matthias Noback. The interface segregation principle. In *Principles of Package Design*, pages 55–66. Springer, 2018.

[50] Matthias Noback. The liskov substitution principle. In *Principles of Package Design*, pages 31–53. Springer, 2018.

[51] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*, pages 3400–3407. IEEE, 2011.

[52] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J. Andrew Bagnell, Pieter Abbeel, and Jan Peters. An algorithmic perspective on imitation learning. *Foundations and Trends® in Robotics*, 7(1-2):1–179, 2018. ISSN 1935-8253. doi: 10.1561/2300000053. URL `http://dx.doi.org/10.1561/2300000053`.

[53] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito,

Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[55] Alex Poole and Linden J Ball. Eye tracking in hci and usability research. In *Encyclopedia of human computer interaction*, pages 211–219. IGI Global, 2006.

[56] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[57] Rouhollah Rahmatizadeh, Pooya Abolghasemi, Ladislau Bölöni, and Sergey Levine. Vision-based multi-task manipulation for inexpensive robots using end-to-end learning from demonstration. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3758–3765. IEEE, 2018.

[58] Brian D. Ripley and N. L. Hjort. *Pattern Recognition and Neural Networks*. Cambridge University Press, USA, 1st edition, 1995. ISBN 0521460867. doi: 10.5555/546466.

[59] Stephane Ross, Geoffrey Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages

627–635, Fort Lauderdale, FL, USA, 2011. PMLR. URL http://proceedings.mlr.press/v15/ross11a.html.

[60] Stéphane Ross, Geoffrey Gordon, and Drew 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*, pages 627–635, 2011.

[61] Stéphane Ross, Narek Melik-Barkhudarov, Kumar Shaurya Shankar, Andreas Wendel, Debadeepta Dey, J Andrew Bagnell, and Martial Hebert. Learning monocular reactive uav control in cluttered natural environments. In *2013 IEEE international conference on robotics and automation*, pages 1765–1772. IEEE, 2013.

[62] Sebastian Ruder. An overview of multi-task learning in deep neural networks, 2017.

[63] Dario D Salvucci and Joseph H Goldberg. Identifying fixations and saccades in eye-tracking protocols. In *Proceedings of the 2000 symposium on Eye tracking research & applications*, pages 71–78, 2000.

[64] Akanksha Saran, Ruohan Zhang, Elaine Schaertl Short, and Scott Niekum. Efficiently guiding imitation learning algorithms with human gaze. *arXiv preprint arXiv:2002.12500*, 2020.

[65] William Saunders, Girish Sastry, Andreas Stuhlmueller, and Owain Evans. Trial without error: Towards safe reinforcement learning via human intervention. *arXiv preprint arXiv:1707.05173*, 2017.

[66] Alexander C. Schütz, Doris I. Braun, and Karl R. Gegenfurtner. Eye movements and perception: A selective review. *Journal of Vision*, 11(5):9–9, 09 2011. ISSN 1534-7362. doi: 10.1167/11.5.9. URL https://doi.org/10.1167/11.5.9.

[67] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. AirSim: High-fidelity visual and physical simulation for autonomous vehicles. In Marco Hutter and Roland Siegwart, editors, *Field and Service Robotics*, pages 621–635, Cham, 2018. Springer International Publishing. ISBN 978-3-319-67361-5. doi: 10.1007/978-3-319-67361-5_40.

[68] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635.

Springer, 2018.

[69] David Silver, J Andrew Bagnell, and Anthony Stentz. Learning from demonstration for autonomous navigation in complex unstructured terrain. *The International Journal of Robotics Research*, 29(12):1565–1592, 2010.

[70] Harmeet Singh and Syed Imtiyaz Hassan. Effect of solid design principles on quality of software: An empirical assessment. *International Journal of Scientific & Engineering Research*, 6 (4), 2015.

[71] Trevor Standley, Amir R Zamir, Dawn Chen, Leonidas Guibas, Jitendra Malik, and Silvio Savarese. Which tasks should be learned together in multi-task learning? *arXiv preprint arXiv:1905.07553*, 2019.

[72] Stinchcombe and White. Universal Approximation Using Feedforward Networks with Non-Sigmoid Hidden Layer Activation Functions. In *International 1989 Joint Conference on Neural Networks*, pages 613–617 vol.1, 1989. doi: 10.1109/IJCNN.1989.118640.

[73] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[74] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[75] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[76] Ye Xia, Jinkyu Kim, John Canny, Karl Zipser, Teresa Canas-Bajo, and David Whitney. Periphery-fovea multi-resolution driving model guided by human attention. In *The IEEE Winter Conference on Applications of Computer Vision*, pages 1767–1775, 2020.

[77] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. 2018.

[78] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understand-

ing deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64 (3):107–115, 2021.

[79] Ruohan Zhang, Zhuode Liu, Luxin Zhang, Jake A Whritner, Karl S Muller, Mary M Hayhoe, and Dana H Ballard. AGIL: Learning Attention from Human for Visuomotor Tasks. *CoRR*, abs/1806.0, 2018. URL http://arxiv.org/abs/1806.03960.