Using Imitation Learning for Human Motion Control in a Virtual Simulation

Christoffer Akrin

Faculty of Health, Science and Technology

M.Sc. in Computer Engineering

Second Cycle, 30 ECTS

Supervisor: Assoc. Prof. Sebastian Herold

Examiner: Assoc. Prof. Tobias Pulls

Date: June 2022

Abstract

Test Automation is becoming a more vital part of the software development cycle, as it aims to lower the cost of testing and allow for higher test frequency. However, automating manual tests can be difficult as they tend to require complex human interaction. In this thesis, we aim to solve this by using Imitation Learning as a tool for automating manual software tests. The software under test consists of a virtual simulation, connected to a physical input device in the form of a sight. The sight can rotate on two axes, yaw and pitch, which require human motion control. Based on this, we use a Behavioral Cloning approach with a k-NN regressor trained on human demonstrations. Evaluation of model resemblance to the human is done by comparing the state path taken by the model and human. The model task performance is measured with a score based on the time taken to stabilize the sight pointing at a given object in the virtual world. The results show that a simple k-NN regression model using high-level states and actions, and with limited data, can imitate the human motion well. The model tends to be slightly faster than the human on the task while keeping realistic motion. It also shows signs of human errors, such as overshooting the object at higher angular velocities. Based on the results, we conclude that using Imitation Learning for Test Automation can be practical for specific tasks, where capturing human factors are of importance. However, further exploration is needed to identify the full potential of Imitation Learning in Test Automation.

Keywords— Imitation Learning, Behavioral Cloning, k-NN Regressor, Test Automation, Manual Testing, Virtual Simulation

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

AI	*Artificial Intelligence*
ML	*Machine Learning*
k-NN	*k-Nearest Neighbors*
IL	*Imitation Learning*
BC	*Behavioral Cloning*
DPL	*Direct Policy Learning*
IRL	*Inverse Reinforcement Learning*
CV	*Cross Validation*
RMSE	*Root Mean Square Error*
RRMSE	*Relative Root Mean Square Error*
SMA	*Simple Moving Average*

Chapter 1

Introduction

1.1 Motivation

During the last decade, the complexity and presence of software have grown tremendously, where most of our daily devices are driven by software. When developing such software, testing is a vital part of the development life cycle. This is mainly due to the pervasive nature of software, which requires sophisticated testing to ensure that it works as intended [42]. An absence of proper testing can have a detrimental effect on cost and—in certain cases—people. An example is the 2015 Spanish Airbus A4700M transport plane which improperly shut down due to a software error—leading to the death of four crew members [58].

Software testing is an expensive part of development [46]. Therefore, the software industry is moving towards test automation as an investment strategy [13]. With the increased use of agile practices in software development, the testing cycles are becoming more frequent and relying on more sophisticated test automation tools [32]. However, certain tests are much simpler to automate than others. Testing the underlying functionality of the software can easily be automated and integrated into the development cycle—as such tests do not require human interaction. Tests requiring human interaction, namely, manual tests are more difficult to automate. A common solution is to record the manual inputs given by the tester and later play them back to automate the test—known as record/playback. This is generally used for Graphical User Interfaces (GUIs), where the inputs may be a sequence of interactions with the available components, such as buttons and text fields [23].

1.2 Problem Formulation

The use of record/playback for automating manual tests is limited. For more complex systems, involving a more dynamic environment that may require human motion control and decision-making, the use of record/playback may become unfeasible. For example, a flight simulator with a physical cockpit requires fine motor control from the pilot to steer the aircraft. In this particular case, automated controls—known as autopilot—

exist to control the aircraft. However, during the testing of such systems, the behavior of the human pilot may be of interest—allowing system evaluation in a human-driven state. Using record/playback in this case may be impractical, as the state space of the simulation is vast.

1.3 Objective

Artificial Intelligence (AI) is a rapidly growing field with the intention of automating tasks and solving difficult problems [41]. Throughout history, the difficulty and complexity of such tasks and problems are increasing. However, AI is not a newly founded field in computer science. As early as in 1989, Pomerleau et al. developed a self-driving car that was capable of steering the vehicle correctly while driving along a road [47]. Although limited compared to today's standards, it is considered to be the earliest example of AI with the goal of imitating human behavior—also known as Imitation Learning (IL).

Today, IL is of growing interest and is used for a multitude of tasks that involve human behavior, such as assistive robots, video games, and self-driving cars. The idea behind IL is to create an agent that learns from expert demonstrations. The expert, which in most cases is a human, demonstrates what actions are optimal in a given situation. For example, with self-driving cars, the human can demonstrate how to steer the car correctly given the current state of the car. Such tasks are considered difficult to solve with AI as they consist of human motion control and complex decision-making [29, 62].

AI has been used for test automation regarding test validation and test generation [21]. However, for automating manual tests requiring human interaction, the use of IL has not been widely researched. The thesis aims to explore the use case of IL for automating such tests. Through human demonstrations of how to control the system under test, an IL agent can learn to imitate the human. This can potentially allow for automating manual tests in a dynamic environment while maintaining human factors.

In this thesis, the task consists of a virtual simulation with an input device in the form of a physical sight. In the virtual simulation, the sight is present in 3D space, and the goal is to rotate the sight towards a static object in the sky. The human user can rotate the physical sight horizontally and vertically, i.e., yaw and pitch. The virtual simulation is visible from two perspectives. The first perspective is through a scope on the sight—showing a zoomed-in view of the current rotation. The second perspective is a separate screen attached on top of the sight, which displays the virtual world from the perspective of the user.

1.4 Stakeholders

The thesis is done in partnership with Saab Dynamics AB. At Saab, testing is a vital part of the development cycle of their systems—for both hardware and software. Most systems involve human interaction, where the human stimulates the system with a given input device, such as a joystick. The interaction with a system can be done through

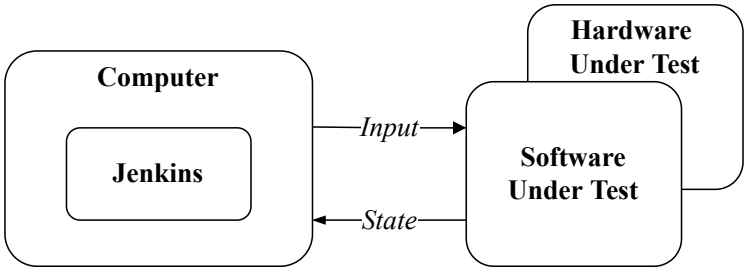


Figure 1.1: An overview of how test automation is done with simulation software running on top of real hardware. A computer running Jenkins executes tests by sending inputs to the software. When the test is finished, the state of the software is validated—resulting in a passed/failed test.

a virtual simulation, where the system is connected to a virtual world. This is used for training soldiers on the systems, and for testing the whole system under realistic circumstances. Testing the system is generally done manually as it partly may require complex human interaction.

For the system of interest, the manual tests are time-consuming and can take up to two days in total to do manually by one developer. The tests generally consist of the following parts:

- **GUI interaction:** Navigation in menus and simpler events, e.g., a button press.
- **Human motion control:** Finer controls that require human motion control and decision making.
- **Test validation:** After a manual test is finished, it is validated with a pass or fail.

As seen in Figure 1.1, test automation generally consists of connecting hardware running relevant software to a separate computer that executes tests with Jenkins[1]. As certain systems require human motion control, such test automation is not feasible without the actual input device. However, with an IL agent imitating the human motion control, automating such tests becomes possible without needing the actual input device.

Furthermore, the manual tests are not done as frequently as desired, due to their severe time consumption. The ability to automate such tests would save time in the long run while also allowing for more frequent test cycles. Higher frequency of the tests would increase the robustness of the system as faults can be detected earlier.

1.5 Methodology

The methodology consists of creating a prototype that can be evaluated by comparing it to the human. For creating a prototype, data collection is first necessary—where the

[1] https://www.jenkins.io/

human demonstrator is recorded while performing the relevant task. The collected data is then analyzed to gain an understanding of how the human behaves and what may be required from the IL agent to learn. This also involves the extraction and selection of informative features. From this, an IL approach is selected, optimized, and implemented. The implementation consists of a controller that communicates with the simulation and allows the IL agent to control the sight. Evaluation of the IL agent is done in terms of human resemblance and task performance. Task performance is measured by the time it takes to stabilize the sight at the object.

1.6 Delimitations

The thesis does not include a comparative study of different IL approaches. It mainly explores the potential for using IL in test automation, not what is considered best practices for IL. The choice of IL approach is motivated by flexibility and simplicity, which are considered important features for applicability in test automation.

1.7 Ethics and Sustainability

Ethical concern for AI is tremendous—with such rapid innovation in the field, it is difficult to predict the future possibility of AI [16].

According to Miriam C Buiten, regulators argue that many of the ethical issues with AI lie in a lack of transparency [17]. If it is always known what the rationale behind an agent's decision is, it can be regulated. However, knowing what AI agents base actions on is obscure [16]. AI agents are dynamic in their functionality, where constant learning changes the behavior of the agent. This leaves room for unwanted behavior and moral status of the AI agent, which can be harmful depending on the use case. For example, if AI was used to fully control an aircraft, it would be difficult for the developers to assure regulators of its safety.

A new topic in AI, namely, explainable AI, aims to solve the problem of obscurity [60]. However, this topic mainly concerns the growing use of deep learning, where Deep Neural Networks (DNNs) are used. DNNs suffer severely from the lack of rationale behind decisions, however, this is getting challenged by more recent research [40]. Nevertheless, this thesis does not use deep learning, instead, it relies on a simpler approach, which—according to DARPA—is considered easier to interpret and has high explainability [22].

The ability to automate tests has a positive effect on the sustainability of software—as it lowers the cost and time spent testing and maintaining software. Test automation also enables higher test coverage which results in more robust software. Faulty software is not economically sustainable for the industry, and not environmentally sustainable either as the lifespan of the device running the software is shortened—requiring more devices to be manufactured [33].

1.8 Outline

The thesis structure is as follows. Chapter 2 contains the relevant background knowledge needed for the thesis and the related work in the field of AI. Chapter 3 explains and motivates the methods used. Chapter 4 explains how human demonstrations were collected and how the IL agent is used to automate human control. Chapter 5 presents the experimental results for the IL agent behavior and performance. Chapter 6 discusses the results. Chapter 7 contains potential future work and final words.

Chapter 2

Background

2.1 Virtual Simulation

A computer simulation is software that represents a real-world model of a system. The goal is to approximate the real system as close as possible and provide realistic data that can be used to study the system [57].

A category in computer simulation, known as virtual simulation, is the use of a real physical system as input to a virtual world. Virtual worlds were first mainly used in video games for the mere purpose of entertainment. However, they are today also being used for training humans to use real systems, for example, military systems, and ambulance driving [8, 24, 54]. This enables the human to use similar controls to reality, resulting in a more immersive experience.

2.2 Software Testing

Software testing is the process of evaluating software with the goal of finding errors and bugs. It is also used to test quality factors of software, such as efficiency, security, and usability [51]. The software has a set of requirements that needs to be met, and is verified with the use of tests. A common approach for software testing is to divide tests in terms of visibility to the developer [35]. The developer may be completely aware of the underlying functionality of the software or not at all—changing the possible approach taken by the tester. More specifically, the visibility can be divided into three types:

- **White Box testing:** The internal implementation of the software is known and tested. White box testing is an effective method for testing software, however, it requires knowledge of how the software is designed to be effective.
- **Black Box testing:** The internal implementation of the software is hidden and testing involves the functionality of the application at the user level. For example, the usability of a GUI can be tested by a user's capability to perform a given task.

- **Grey Box testing:** The combination of white box testing and black box testing by allowing the tester to know the internal implementation of the software while doing user-level testing.

2.2.1 Testing Levels

Apart from describing the software visibility to the tester, the different types of software testing can be divided into three main levels of granularity:

1. **Unit Testing:** A white box approach where the tester focus on smaller units of the software [42]. Testing is done directly in code where, for example, a single function is tested with relevant parameters. As the tester knows how a function should behave, it can be properly verified with a set of unit tests.

2. **Integration Testing:** Unit testing is done on the lowest level, i.e, on single modules of software, and sometimes also called module testing. However, when the modules are integrated, additional testing is necessary to ensure that the modules coordinate properly together [44].

3. **System Testing:** After doing functional testing with unit testing and integration testing, the whole integrated system is tested to ensure it meets the specified requirements [42]. The software is tested on the original objectives to ensure that it works as initially intended.

2.2.2 Regression Testing

When a developer makes a change in the software, re-running tests is necessary to ensure that the modified software still works as intended [59]. If the modified software fails to behave properly, it is called a regression. Regression testing is therefore important, as it helps ensure that new bugs are found early.

2.3 Test Automation

Certain tests are done manually or automatically, for example, regression tests. The problem is that manually testing software can be difficult and time-consuming, therefore, the use of software that automates the testing process is common practice. However, different types of tests differ in automation complexity [23]. On the lowest level of testing, namely unit testing, the tests are very isolated and simple to automate. On the highest level, for example, Graphical User Interface (GUI) testing, the process of automation can be difficult. Automated GUI tests are sensitive to changes in the software and can become expensive to maintain [23].

Test automation of user interaction with a GUI aims to automate user behavior. In its simplest form, this can be done with a record/playback technique, where the user input to the GUI components is recorded and played back to automate the user [20]. Another simple technique is to stimulate the GUI with random events, known as a test

monkey, which can be useful for finding unknown bugs that may be difficult to find otherwise [37, 28]. However, test monkeys have no capability of planning and the end goal is generally comprised of crashing the system. More complex approaches are based on the planning capabilities of AI. With an AI-based planner, tests can be executed by giving an initial state, end state, and a set of actions; where the planner calculates a path to the end state before execution [15, 38].

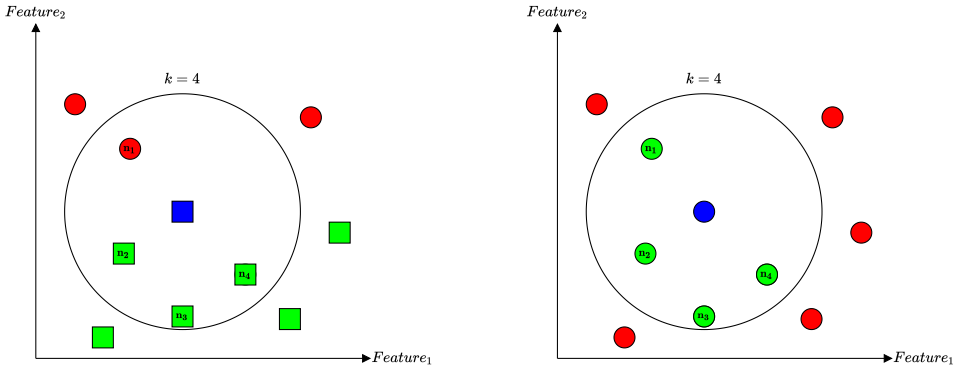
2.4 Machine Learning

As described by Mohri et al., Machine Learning (ML) is used to create a prediction algorithm from experience [39]. Given an input, the algorithm uses a model built on experience to predict the correct output. The experience consists of past information, where the quality and size of the information are crucial for the algorithm's capability to build an accurate model. The goal is for the model to predict an accurate output on unseen input—without explicit information on what the correct output should be.

2.4.1 Types of Machine Learning

Given a set of data points, $\mathbf{X} \in \{\mathbf{x_1} \dots \mathbf{x_k}\}$ where each one contains the vectorized information about the data point, i.e., a feature vector. Each feature vector maps to a corresponding output vector $\mathbf{Y} \in \{\mathbf{y_1} \dots \mathbf{y_k}\}$. The goal of an ML model is to learn to predict the output vector for a given—unseen—data point [27, 7]. A practical example is where a model is given a dataset containing images of dogs and needs to predict the breed. The feature vector would contain the pixel values of the image, while the output vector would contain the breed of the dog. This is an example of a supervised classification problem, where the model classifies the given data point to a label, i.e., the breed. Instead of predicting dog breeds, the model may want to predict the price of a house, given features such as the size of the house and the distance to the nearest school. This is known as a supervised regression problem, where the output consists of real numbers instead of labels.

One example of a supervised learning algorithm is k-nearest neighbors (k-NN). k-NN can be used as a classifier or regressor, as seen in Figure 2.1. For a classifier, it works by collecting the k nearest data points to a new sample and assigning it the most common label. In (a) of Figure 2.1, the labels consist of squares and circles, where the new sample—in blue—becomes a square as 3 out of 4 neighbors are labeled as squares. The process is similar for a regressor, except, the data points contain a real value instead of a label. The assigned value is calculated by taking the mean of the k nearest data points. This is seen in (b) of Figure 2.1, where the mean value for the 4 nearest neighbors becomes the new sample value—also known as interpolation. k-NN can also incorporate a weight function, for example, a distance weight function affects the influence of a neighbor depending on how far away it is from the sample [2].



(a) k-NN classifier.

(b) k-NN regressor.

Figure 2.1: An example of k-NN classifier and regressor with $k = 4$ and 2 features.

Reinforcement Learning

Reinforcement Learning (RL) is similar to supervised learning [56]. The key difference is that the model—commonly known as an agent—is interacting with an environment by applying actions. The agent is rewarded on how good an action is based on its current state in the environment. More specifically, the reward is a function that defines a set of rules for what is considered good or bad behavior from the agent. This creates a feedback loop where the agent exists in a current state, causes an action, receives a reward, and proceeds to a new state. Through exploration of the environment, the agent tries to maximize the reward.

Unsupervised Learning

The last learning method is unsupervised learning, where the model is given a set of data points, however, it is not given a reward or the output vectors [27]. This means that the model has to label the data points itself by finding relations between them. Clustering is a common example of unsupervised learning, where the model separates the dataset into groups with similar features [27].

2.4.2 Achieving a Satisfactory Model

To achieve a satisfactory model—that generalizes well on unseen data—there are multiple steps that need to be taken. First, the collected data has to be sufficient in terms of size and information, otherwise, it becomes impossible to create a model that can perform well. Second, the features given to the model from the dataset need to be well selected. Features directly extracted from the dataset are known as raw features. Using raw features directly to train a model may contain redundant and obscure information depending on the task and dataset. Instead, features can be designed to become more manageable by removing redundancy and obscureness—resulting in a better model [39].

Hyperparameter

Each ML algorithm provides a set of tunable parameters, known as hyperparameters, that can drastically change the performance of the model [39]. For example, the k-NN algorithm only has one hyperparameter, namely, the k value. Finding the optimal hyperparameters for a model can be done by using cross-validation, where the model performance is tested on unseen data. This is done by splitting the initial dataset into two parts, training data for training the model, and test data—e.g., 30 percent —that is used to test the model.

Fitting A Model

A problem with training a model is to make sure that it fits the training data properly [39]. A model that too closely represents the training data may fail to generalize well on new data, known as overfitting. Furthermore, a model that already struggles to perform well on the training data is underfitted.

A model's capability to generalize well on unseen data can be measured with three different errors [26]:

- **Bias:** A measure of the model's faulty assumptions. An underfitted model has a high bias as it misses relevant relations between the features.
- **Variance:** A measure of the model's sensitivity to variations in the training data. An overfitted model has high variance as it fits the random noise in the training data.
- **Irreducible error:** The error caused by noise in the training data that can not be reduced with a better model. Lowering this error requires better training data, for example, by removing outliers.

K-fold Cross-validation

To avoid overfitting, K-fold cross-validation can be used to find the hyperparameters that truly perform best on unseen data [12]. As seen in Figure 2.2, the training data is split into K different folds, where a model is trained with $K-1$ folds and tested with the remaining fold. This is done for K possible combinations. As each part of the training data is used for testing, overfitting the training data is avoided.

Loss Function

The performance of the model is quantified with a loss function. The loss function returns a real number—known as the loss—based on a set of model predictions. For example, in a k-NN regressor, the loss can be calculated by the squared distance between the predicted value and the actual value. Squaring the loss means that the loss increases quadratically as the distance is increased [45].

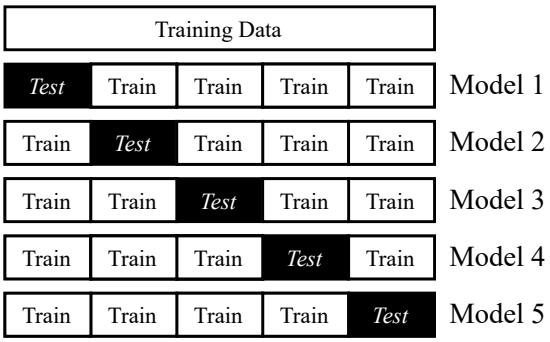


Figure 2.2: An illustration showing how a 5-fold CV works.

2.5 Imitation Learning

Imitation Learning (IL) is a part of ML where the objective of the learner is to imitate an expert's actions [29]. The expert is in most cases a human who performs the task several times to collect training data for the learner to train on. The learner creates a state-to-action mapping based on the demonstrators behavior, also known as a policy π. The goal is to learn a near-optimal policy that acts as similar as possible to the demonstrator.

The earliest application of IL dates back to 1989 when Pomerleau et al. built an autonomous land vehicle with a neural network, named ALVINN [47]. The neural network was trained on input from simulated road images (1200 images in total) and a laser range finder, with the corresponding direction that the vehicle should travel to follow the road. The feature vector consisted of the pixels of the road image (30x32 units) and the input retina for the range finder (8x32 units). The neural network classified a given input vector into 45 different turn directions, i.e., one for driving straight and 22 each for left and right turns, with different turning intensities.

2.5.1 Extraction and Selection of Features

To learn a near-optimal policy, the selected features need to comprise of an adequate and efficient representation of the observable state [29]. This can be done by simply using raw features from the data, such as, capturing image data from a video game and using the pixels as features [50]. However, some features need to be manually extracted and designed from the raw data to attain useful information about the task [29]. For example, when trying to imitate humanoid motion with a humanoid robot, extracting the position of body parts can be used as features for the learner [14, 30].

2.5.2 Environment

Extracted data for the learner are represented as state-action pairs, consisting of the current state of the expert and their current action. The environment that the expert is

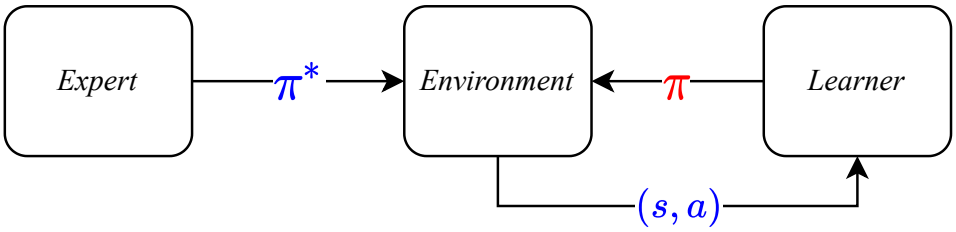


Figure 2.3: An illustration of how the expert policy π^* is observed by the learner by collecting the state-action pairs to create its own policy π.

demonstrating can be interpreted as a Markov decision process (MDP), defined as the following tuple:

$$MDP = (S, A, P_a(s, s'), R_a(s, s')) \tag{2.1}$$

Where S is the state-space, A is the action-space, $P_a(s, s')$ is the probability that action a in state s leads to entering state s' in the next step, and $R_a(s, s')$ is the reward received for entering s' through action a in state s. Time is the discrete step from one state to another [9]. The expert policy π^* is the mapping from state to actions to be taken in a given state, i.e., the behavior of the expert. For the learner policy π, the goal is to as closely as possible mimic π^* [29]. It is important to acknowledge that all the rewards $R_a(s, s')$ can be zero, meaning that there is no feedback given to the learner when performing the task [9].

2.5.3 Behavioral Cloning

Behavioral Cloning (BC) is the simplest form of IL [29]. The expert demonstrations are perceived as a supervised learning problem and generate state-action pairs (s, a) as data for the learner. This allows the learner to predict actions in a given situation based directly on demonstrations, as seen in Figure 2.3. The learner does not receive any reward when performing the task and is strictly dependent on how the environment and model are built—i.e., what features and actions are used. For its simplicity and efficiency, BC has proven to produce compelling results in certain tasks [25, 18]. However, due to its simplicity, it is not applicable to all IL problems. As described by Ross et al., BC can struggle severely with large state spaces that lack expert trajectories, and where a 1-step deviation from the policy can lead to detrimental error [50]. For example, if the learner is trying to steer a vehicle and ends up on the edge of the road that the expert has not visited, the learner is unable to behave accordingly by steering the vehicle back onto the road.

Direct Policy Learning (DPL) is an improved version of BC, where the learner has access to an interactive expert during training time [61]. The learner can query the expert, correcting the learner if the wrong action is taken. This creates an iterative loop that increases the ability to converge towards the expert policy over each iteration. DPL

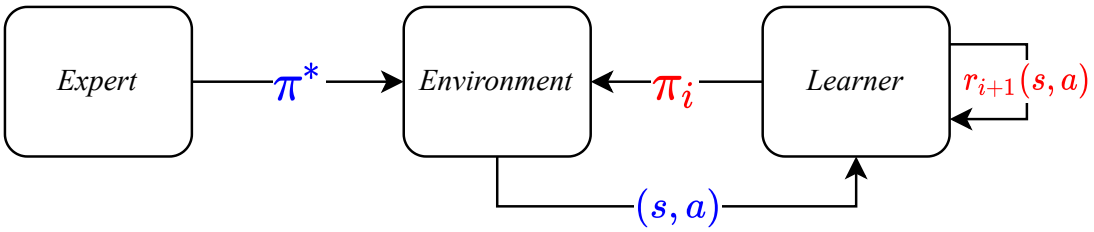


Figure 2.4: An extension of Figure 2.3, where the IRL learner iterates over i and learns a new reward function $r_i(s,a)$ until the policy π_i is satisfactory.

is advantageous to BC as it helps the learner to act properly when encountering unseen states. However, this requires an interactive expert to be available during training.

2.5.4 Reinforcement Based Imitation Learning

Another approach that differs from BC is Inverse Reinforcement Learning (IRL), where the learner is given a reward for each $P_a(s,s')$, i.e., the learner is rewarded for how adequate the action taken is for the current state. IRL is useful when directly mimicking the expert policy can be difficult and creating a reward function is easier to learn. As seen in Figure 2.4, the learner iterates the process of updating the parameters of the reward function, leading to a new policy for each iteration. The learner updates the reward function until the learner policy is satisfactory, i.e., the policy that maximizes the reward function [43, 5].

IRL is challenging to implement as finding the best reward function for mimicking the expert policy is ill-posed. Furthermore, reinforcement based learning becomes severely expensive to train at a larger scale, and the research in IRL lacks general theoretical guidance for the complexity and accuracy of different methods [6]. Another finding is that many different reward functions—even a reward of $r = 0$ for all behavior can realize the expert behavior [43]. This is due to the fact that most tasks in IL have a small action space, allowing for many reward functions to realize the expert behavior.

A more recent method by Reddy et al., namely Soft Q-Learning Imitation Learning (SQIL) [48], aims at giving the learner an incentive to trace back to the expert demonstrated trajectory upon exploring new states. This is achieved by defining a reward function that gives the agent a constant reward of $r = +1$ when the action matches the expert in a demonstrated state and a constant reward of $r = 0$ for all other behavior. This method removes the critical IRL problem of finding the correct reward function. SQIL is a proof-of-concept idea that proves that a simple imitation method can be as effective as more complex IRL methods.

2.6 Related Work

This section presents the most relevant work in IL for this paper.

Cardamone et al. [18] used BC to mimic a bot driving in The Open Racing Car Simulator (TORCS) [4]. More specifically, they compared two supervised methods, k-NN and Neuroevolution of augmenting topologies (NEAT) [53]. In their experiment, they showed that with a high-level defined environment—i.e., features and actions—BC can have impressive results. The high-level features were experimented with two approaches, namely, range finders that are ray-casted from the car and inform of the distance to the edge of the road, and lookahead sensors informing of the curvature of the road segments ahead—inspired by human driving behavior. Actions consisted of the desired velocity and trajectory of the vehicle. Compared to using low-level actions, such as gas and braking, the high-level actions simplified the task for the learner and improved performance significantly.

For future work, Cardamone et al. believe that further generalization of the environment can improve performance, such as, treating left and right turns the same, using better supervised methods, and training data from multiple different tracks. They also mention that cleaning up the training data by removing duplicates and less informative attributes can help remove noise and lower computational cost.

Renman, C [49] used k-NN classification to mimic human movement for AI in games. The classification algorithm used KD-tree to speed up the k-NN classifier. The task for the learner was to move to a certain goal while avoiding obstacles. Encouraging the learner to move to a specific goal required the environment to be split up into a discretized grid with each cell having a score, which increases the inclination of the learner to move towards certain cells. Similar to Cardamone et al., the sensors where rays cast from the player and indicate how far away an obstacle is in a certain direction. The actions were based on the trajectory of the player. The result were evaluated through a user study, inspired by the Turing test. The participants watched videos of the player moving without knowing if it is controlled by a human or AI, and had to judge how human-like the player was behaving.

Ross et al. [50] used BC and DPL to mimic human steering in a 3D racing game (*Super Tux Kart*). The feature set consisted of the current game image, and the actions consisted of steering the vehicle with a joystick—controlled with a continuous value between -1 and 1. They showed that the performance was significantly increased by using DPL compared to supervised learning with BC. The method used, namely Data Aggregation (DAgger), works by iterating the training where the learner can query the expert when exploring unseen states. The dataset collected for each training iteration is aggregated, forming a new policy at each iteration.

Chapter 3

Experimental Setup

3.1 Introduction

Software tests involving human interaction are a major challenge to automate. The record/playback approach for automating human interaction may be sufficient in a static environment, however, in a more dynamic environment where starting states and end states may constantly change, it becomes impractical to cover all scenarios with a record/playback approach.

IL is a proposed solution to the problem, where a learner acts as a record/playback approximation for any given set of start to end states. The learner is trained on human recordings and learns to find a generalized policy that resembles the human behavior sufficiently given unseen states. The thesis aims to answer the following research questions:

- **RQ1:** How can IL be used for automating human interaction in software tests where record/playback is impractical?
    - **RQ1.1:** How well can an IL model resemble human interaction with limited data?
    - **RQ1.2:** How feasible is the use of IL for test automation?

3.2 Overview of Experimental Setup

The approach taken for the experimental setup is summarized by the following steps:

1. **Analysis of the dataset:** The task provides a set of recordings representing the human policy π^* (See Section 2.5.2). Analysis of the recordings can give a better understanding of the human behavior and limitations. The approach taken for collecting recordings is described in 4.3.
2. **Feature extraction:** The data present in the recordings are considered the raw features. To enable a better model, extracting relevant features from the recordings

can help increase the mode performance. This is also where the environment is defined, i.e., what the state and action should contain.

3. **IL approach selection:** IL consists of several methods for creating a sub-optimal policy, as stated in the previous chapter. Here, a single method is chosen based on effectiveness and flexibility.

4. **Optimization of model parameters:** With a chosen model, tuning the parameters of the model is crucial to finding the best fit. The approach for parameter optimization is based on a loss function that measures how well the model imitates π^*.

5. **Evaluation of the optimized model:** Last, an explanation and motivation for how the optimized model is evaluated, in terms of task performance and human resemblance.

3.3 Description of Task to Automate

A user controls a physical sight in a virtual simulation. The simulation is in 3D space and contains a still object in the sky. The sight can be rotated on two axes, horizontally and vertically, i.e., yaw and pitch, and the goal is to rotate the sight so it points towards the object. When a developer makes a change in the implementation of the simulation, regression testing is carried out, where the developer tests the physical sight by rotating it to the object. This means that the updated software has to be implemented on a physical sight to be tested, which is an infeasible task, and therefore, the simulation can be executed on a separate computer without a physical sight. The sight is then rotated with a keyboard instead. However, with a keyboard, the rotation of the sight is controlled with a much simpler motion, which does not represent how the human would behave with a physical sight. To solve this, an IL model can learn to mimic the human motion of the physical sight, allowing for realistic behavior of the sight when doing regression tests. Furthermore, the IL model enables the possibility of test automation. When the developer makes a change in the implementation of the simulation, the IL model can help run the regression tests automatically and verify that the simulation still works as intended.

To train an IL model on the movement of the sight, a set of recordings R containing information on how the human rotates the sight towards the object are used as training data. Every frame—or timestep—in the recordings contain the information shown in Table 3.1. The frames are recorded in 60 Hz and can be interpreted as the raw features, containing the position of the sight $\mathbf{S}$ and object $\mathbf{O}$, and the sight yaw θ_s and pitch γ_s in miliradians (mrad). This means that the task involves rotating θ_s and γ_s to where the sight is pointing towards the object $\mathbf{O}$.

To understand how far away the sight rotation is from pointing at the object, the angular distance from pointing at the object to the current sight rotation can be calculated by taking the angular differences between the ray (θ_s, γ_s) starting from $\mathbf{S}$, and

$\mathbf{S} : (x_s, y_s, z_s)$	$\mathbf{O} : (x_o, y_o, z_o)$	θ_s	γ_s

Table 3.1: Raw features contained in each timestep in a recording.

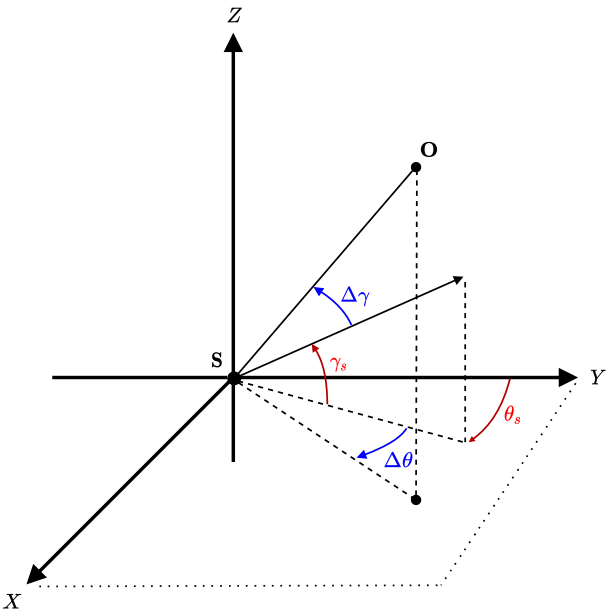


Figure 3.1: A graph depicting the angles of the sight ray, yaw θ_s and pitch γ_s, and the angular difference between the object and the sight (delta angles), $\Delta\theta$ and $\Delta\gamma$.

the angles of the line between $(\mathbf{S}, \mathbf{O})$. Shown in Figure 3.1, the angular differences—i.e., delta angles—are defined as $\Delta\theta$ for yaw and $\Delta\gamma$ for pitch.

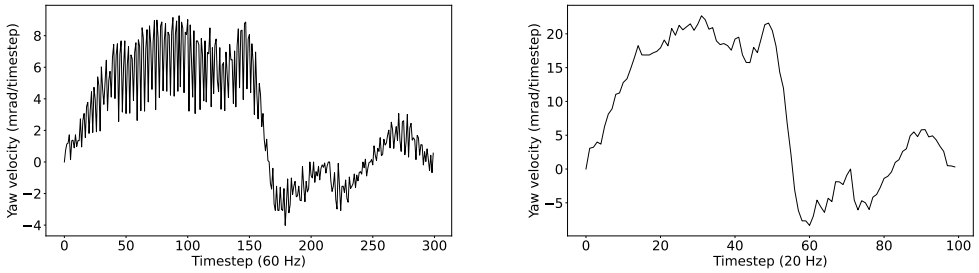
Detailed information about the recordings R is shown in Table 3.2. The recordings are short, meaning that the time to rotate the sight towards the object is a relatively fast task. The set of recordings is a small dataset and may be difficult to fit depending on the quality of the data and how difficult the motion is to imitate. An understanding of how the human rotates the sight is needed for answering these questions.

3.4 Analyzing the Human Behavior

As the human rotates the sight, the behavior can be analyzed by looking at the angular velocity. Taking the current sight angle from one frame and subtracting with the next frame angle results in a velocity in *mrad/timestep*. A part of a recording showing the velocity for the yaw is seen in Figure 3.2. Looking at (a) in Figure 3.2, plotting the velocity for all frames show a severe fluctuation between frames. This means that there are irregular changes in sight angles for each frame, for example, one frame may change the sight angle by 10 mrad, while the next frame change by 5 mrad. This is due to

Number of recordings	142
Total recording time	533.60 sec (∼9 min)
Mean recording time	∼3.74 sec
Longest recording time	9.55 sec
Shortest recording time	1.15 sec

Table 3.2: Information about the recordings in the dataset.



(a) Yaw velocity in 60 Hz.

(b) Yaw velocity in 20 Hz.

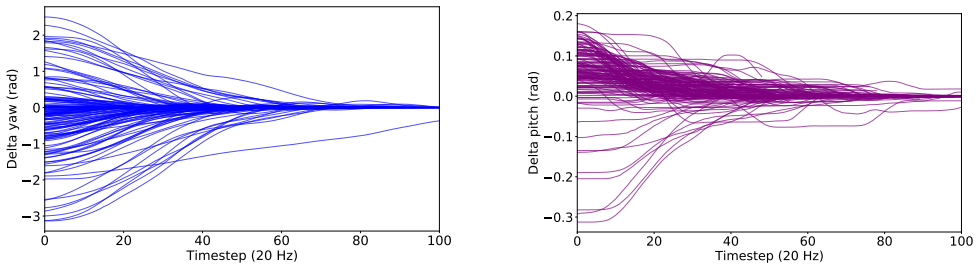
Figure 3.2: An example of the difference in plotting the sight yaw velocity with a sampling rate of 60 Hz and 20 Hz.

the recordings being in a lower frame rate than the actual update of the sight angles. For example, if the sight updates its angle in 90 Hz, while the recording is in 60 Hz, the change in sight angle for the recording will fluctuate. Two consecutive frames in the recording capture three updates of the sight, meaning one frame will have twice the change in velocity.

In (b) of Figure 3.2, the recording is downsampled with an integer factor of three—resulting in 20 Hz—which shows a much smoother velocity curve. Downsampling to 20 Hz is beneficial, as the dataset becomes a third in size without losing significant features. This makes the fitting and optimization of the model much faster.

To gain a better understanding of how the human moves the sight over time, plotting the delta angles $\Delta\theta, \Delta\gamma$ for each timestep will show how the human moves towards the object. As seen in Figure 3.3, it is clear that the movement differs in the yaw and pitch. The yaw movement, seen in (a), follows a smooth curve towards the object. The pitch movement, seen in (b), shows a much more drastic movement with more inconsistencies.

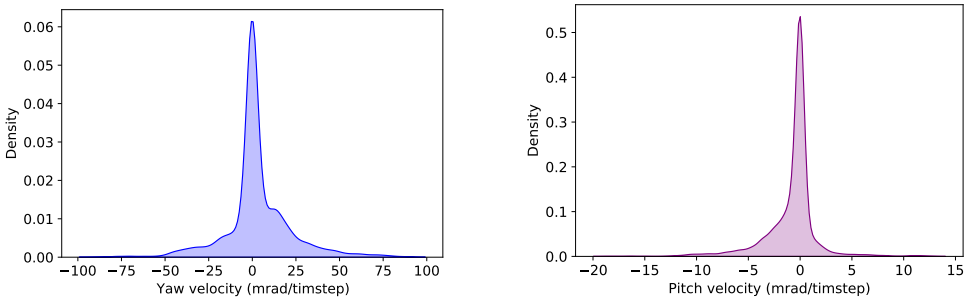
To understand the behavioral domain of the sight movement—i.e., what is a plausible movement for the yaw and pitch—the density for the angular velocities and accelerations are plotted in Figure 3.4 and Figure 3.5 respectively. Furthermore, the mean and standard deviation for the angular velocity and acceleration are shown in Table 3.3. The angular velocities in Figure 3.4 show that the yaw velocity is skewed towards positive values. The same can be observed for the pitch velocity, which is skewed towards negative values. The skewness can indicate two things, the recordings are biased toward a certain



(a) Delta yaw $\Delta\theta$ over time.

(b) Delta pitch $\Delta\gamma$ over time.

Figure 3.3: Plotting the delta angles $\Delta\theta, \Delta\gamma$ for the first 5 seconds for all recordings.



(a) Density for yaw velocity.

(b) Density for pitch velocity.

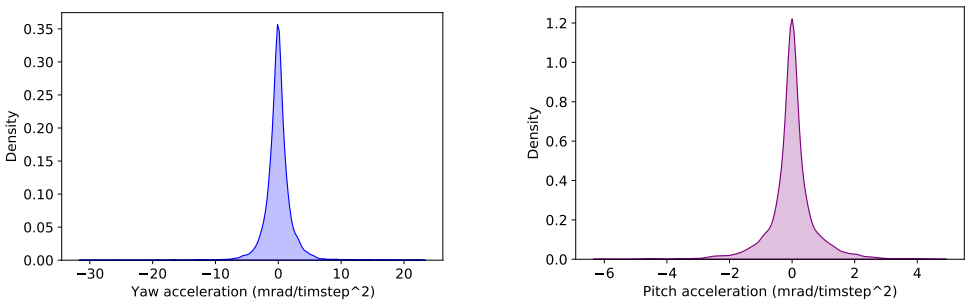
Figure 3.4: The velocity density for yaw and pitch for all recordings.

starting rotation, i.e., it is more common to start from one side of the object than the other. This bias is clearly visible in (b) of Figure 3.3, where almost all recordings start with a positive $\Delta\gamma$. The second possibility for skewness is that the movement behaves differently depending on which side the human starts on.

Looking at the angular acceleration in Figure 3.5 the skewness is gone for both yaw and pitch, and follows a normal distribution with a mean close to 0. The acceleration can better indicate how the human moves the sight, as it explains what forces were applied to the sight, instead of what velocities the sight rotates with. This shows that the applied forces are close to identical from either direction in yaw and pitch.

3.5 Feature Extraction From Recordings

The model only needs to know how it should change the angles θ_s, γ_s to point towards the object. Therefore, the raw features in Table 3.1 can be designed into fewer features. This can be done by extracting the delta angles $\Delta\theta, \Delta\gamma$, which encapsulate the important information from the raw features into a 2-dimensional state. With the delta angles, the model knows where it is in regard to the object.



(a) Density for yaw acceleration.

(b) Density for pitch acceleration.

Figure 3.5: The acceleration density for yaw and pitch for all recordings.

	μ	σ
v_θ $(mrad/timestep)$	3.058	18.943
v_γ $(mrad/timestep)$	-0.746	2.439
a_θ $(mrad/timestep^2)$	-0.002	2.196
a_γ $(mrad/timestep^2)$	-0.002	0.743

Table 3.3: The mean μ and standard deviation σ for the angular velocity and acceleration of yaw and pitch.

3.5.1 State

As visualized in Figure 3.1, the delta angles correspond to the angular difference between the current sight yaw θ_s and pitch γ_s, and the correct yaw θ_o and pitch γ_o needed to point towards the object. The calculation of $\Delta\theta, \Delta\gamma$ can be seen in Algorithm 1. Line 8 and 9 ensures that $\Delta\theta, \Delta\gamma \in (-\pi, \pi]$. This means that if the sight is rotated to the right of the object, $\Delta\theta$ is negative, and if the sight is rotated to the left of the object, $\Delta\theta$ is positive. Similarly for $\Delta\gamma$, which is positive if rotated below the object and negative if rotated above the object. It is necessary for the model to know the signed delta angle to rotate in the correct direction. This is possible with the use of the function *atan2* [1], which takes a point (y, x) as arguments, and calculates the angle between the ray from the origin to the point (x, y) and the positive x-axis. The returned angle is between $\theta \in (-\pi, \pi]$.

3.5.2 Action

The model needs to rotate the sight at a similar velocity as seen in the recordings. Velocity is a change in distance over time, and in the recordings, the velocity for yaw and pitch can be calculated by the difference between the angles for timestep t and $t+1$.

Predicting a velocity every timestep and applying it to the sight may not imitate the human policy well. After all, this is not how a human would control the sight. The sight

Algorithm 1 Calculating signed delta angles $\Delta\theta, \Delta\gamma$

function GetSignedDeltaAngles($\mathbf{O}, \mathbf{S}, \theta_s, \gamma_s$)
    $d_x, d_y, d_z \leftarrow \mathbf{O} - \mathbf{S}$
    $d_{xy} \leftarrow \sqrt{d_x^2 + d_y^2}$

    $\theta_o \leftarrow acos(d_y / d_{xy})$
    $\gamma_o \leftarrow atan(d_z / d_{xy})$

    $\Delta\theta \leftarrow atan2(sin(\theta_o - \theta_s), cos(\theta_o - \theta_s))$
    $\Delta\gamma \leftarrow atan2(sin(\gamma_o - \gamma_s), cos(\gamma_o - \gamma_s))$

    return $\Delta\theta, \Delta\gamma$
end function

is rotated by a force applied by the human, meaning that the human is controlling the sight through acceleration. Applying a force to the sight causes a trajectory that may not change for a certain time period. If the model corrects the velocity for every timestep, the trajectory can change unnaturally. The low standard deviation for acceleration in the dataset indicates that the velocity does not change very often, and when it changes, it usually changes slowly. It also indicates that acceleration would be hard to predict, as it contains little information.

3.5.3 The *lookahead* Parameter

To solve the previously mentioned problems for model actions, a third approach is proposed. The idea can be understood as a method for accelerating according to the model's future desired velocity. The model can look n timesteps in the future to base the next prediction on—i.e., *lookahead*. This is formally known as Simple Moving Average (SMA) [55], used in time series for calculating averages for a subset of data points in the time series. This filters out intense fluctuations that can happen between the discrete steps in a time series, without losing the future trend. In the relevant task, SMA can be used to calculate the average acceleration in the next *lookahead* timesteps:

$$\bar{\mathbf{a}}_{\mathbf{t+1}} = \frac{1}{n} \sum_{i=t+1}^{t+n} \mathbf{a_i} \tag{3.1}$$

Where $\bar{\mathbf{a}}_{\mathbf{t+1}}$ is the angular acceleration for the next timestep $t+1$ using SMA, n is the number of steps in the future, i.e., the *lookahead* value. However, the model is predicting the angular velocity $\mathbf{v_{t+n}}$ and therefore the following rewrite is needed:

$$\bar{\mathbf{a}}_{\mathbf{t+1}} = \frac{1}{n}(\mathbf{v_{t+n}} - \mathbf{v_t}) \tag{3.2}$$

This allows the model to control the acceleration for the next timestep implicitly by

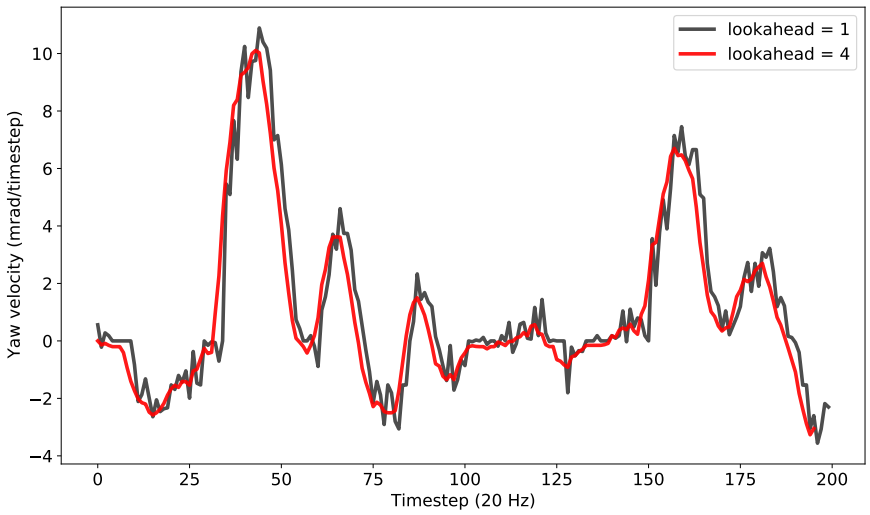


Figure 3.6: An example of how increasing the *lookahead* parameter changes the velocity curve over time.

predicting the future velocity. With $lookahead = 1$, the model is simply predicting the velocity for the next timestep $t + 1$, and the predicted velocity is achieved at the next timestep. With a $lookahead = 2$, the model predicts the velocity it wants for the next 2 timesteps ahead. This means that the achieved velocity for the next timestep $t + 1$ is:

$$\mathbf{v_{t+1}} = \mathbf{v_t} + \bar{\mathbf{a}}_{\mathbf{t+1}} \tag{3.3}$$

Figure 3.6 shows how increasing the *lookahead* changes the velocity curve over time. With $lookahead = 1$, the velocity is adjusted rigorously between timesteps. Simply increasing to $lookahead = 4$ shows a much smoother transition over time, where the sudden changes in velocity are filtered out. The proper value for *lookahead* is not trivial and depends on the task. Therefore, it is necessary to tune it to find the best value.

3.5.4 Extraction of State-Action Pairs: S, A

From the raw features in the recordings, the state-action pairs S, A are extracted according to Algorithm 2. Given the list of recordings R and a *lookahead*, the algorithm iterates over all recordings in R, downsampling it to 20 Hz by only keeping every third timestep, and iterates over each timestep. The delta angles $\Delta\theta, \Delta\gamma$ are calculated with Algorithm 1 for each timestep in the recording. The last loop populates S, A with corresponding angular distance and velocity, according to the *lookahead* parameter. The velocity is calculated by taking the difference in sight angle for timestep $i + lookahead$ and $i + lookahead - 1$. The use of $atan2$ ensures that the correct angle difference is calculated while keeping the correct sign.

Algorithm 2 Extraction of state-action pairs from recording(s)

function GetStatesAndActions($R, lookahead$)
  $S, A \leftarrow [\dots]$
  for each $recording \in R$ **do**
    $Downsample(recording)$ ▷ Keep every third timestep for 20 Hz.
    $D \leftarrow [\dots]$ ▷ Delta angles vectors $\Delta\theta, \Delta\gamma$.
    $\Theta_s \leftarrow [\dots]$ ▷ Angles for the sight θ_s.
    $\Gamma_s \leftarrow [\dots]$ ▷ Angles for the sight γ_s.
    for each $features_{raw} \in recording$ **do**
      $\mathbf{O}, \mathbf{S}, \theta_s, \gamma_s \leftarrow features_{raw}$
      $\mathbf{d} \leftarrow GetSignedDeltaAngles(\mathbf{O}, \mathbf{S}, \theta_s, \gamma_s)$
      $D.Append(\mathbf{d})$
      $\Theta_s.Append(\theta_s)$
      $\Gamma_s.Append(\gamma_s)$
    end for
    for $i \in [0, Length(D) - lookahead)$ **do**
      $S.Append(D[i])$
      $d\theta \leftarrow \Theta_s[i + lookahead] - \Theta_s[i + lookahead - 1]$
      $d\gamma \leftarrow \Gamma_s[i + lookahead] - \Gamma_s[i + lookahead - 1]$
      $v_\theta \leftarrow atan2(sin(d\theta), cos(d\theta))$
      $v_\gamma \leftarrow atan2(sin(d\gamma), cos(d\gamma))$
      $A.Append([v_\theta, v_\gamma])$
    end for
  end for
  return S, A
end function

3.6 Behavioral Cloning With k-NN Regressor

As stated in the previous chapter, there exist many different approaches in IL. In this particular task, the state and action are of a low dimension and a 1-step deviation from the human recording is inevitable, i.e., given an unseen starting state, the actions will not equal a human precisely. The model is supposed to approximate the human actions, and the goal is to find an acceptable approximation. It is also important that the model can be augmented with new recordings "on the fly", without much extra work needed.

Looking at previous research, k-NN has proven to be a promising supervised model for BC [18, 49, 29]. Therefore, BC with a k-NN regressor is suitable for the relevant task—designed with a high-level environment at a low dimension. The choice of k-NN is further motivated by the fact that it does not require any training, instead, the model is defined by the training set through interpolation of k nearest data points. This makes it easy to add new recordings to the dataset without needing much extra work.

3.6.1 Lowering Inference Time With K-D Tree

To lower the time complexity of the search time for k-NN, K-D Tree [10] algorithm was chosen. Given an average time complexity of $O(log(n))$ for search and performing well on lower dimensions compared to other algorithms [36], K-D Tree is fitting for the task.

3.6.2 Distance Weighted Neighbors

Using distance as a weight for the neighbors ensures that closer states have a larger impact on the next action. The state can be interpreted as a 2-dimensional position in space, with a corresponding action in that state. Hence, closer states to the current model state should have a larger impact on the prediction.

To calculate a new prediction with distance weighted neighbors, the weight function $w(\mathbf{s_i})$ calculates the inverse euclidean distance from the model state [52]. This means that closer states have a larger weight. Calculating the next action $\mathbf{a}$ for the model in state $\mathbf{s}$ can be done as following:

$$\mathbf{a} = \frac{\sum_{i=1}^{k} w(\mathbf{s_i})\mathbf{a_i}}{\sum_{i=1}^{k} w(\mathbf{s_i})} \tag{3.4}$$

where

$$w(\mathbf{s_i}) = \begin{cases} \dfrac{1}{||\mathbf{s} - \mathbf{s_i}||}, & \text{if } ||\mathbf{s} - \mathbf{s_i}|| \neq 0 \text{ for all } i \\ [\ ||\mathbf{s} - \mathbf{s_i}|| = 0\ ], & \text{if } ||\mathbf{s} - \mathbf{s_i}|| = 0 \text{ for some } i \end{cases} \tag{3.5}$$

Where the k closest neighbors are $S \in \{\mathbf{s_i} \dots \mathbf{s_k}\}$. If $||\mathbf{s} - \mathbf{s_i}|| = 0$ for some i, i.e, a distance of zero, the weight function $w(\mathbf{s_i})$ is 1 for states with a distance of zero, and 0 for all other states [3][1].

3.7 Parameter Optimization: k and *lookahead*

Having only two discrete parameters to tune, using grid search k-fold CV is feasible. The most common approach for testing the predictability of a model is to simply take single state-action pairs, show the model the state and make a prediction on what action to make. The loss can be calculated by the difference between the actual action and the model action.

A problem with this approach is that the time when an action is made does not influence the loss, i.e., the loss is not weighted relative to the time of occurrence. Instead, it would be preferred to calculate a loss based on the difference in the path taken by the human and model.

[1] An Iverson bracket [34]: Given a statement P, $[P] = 1$ if P is true, else $[P] = 0$.

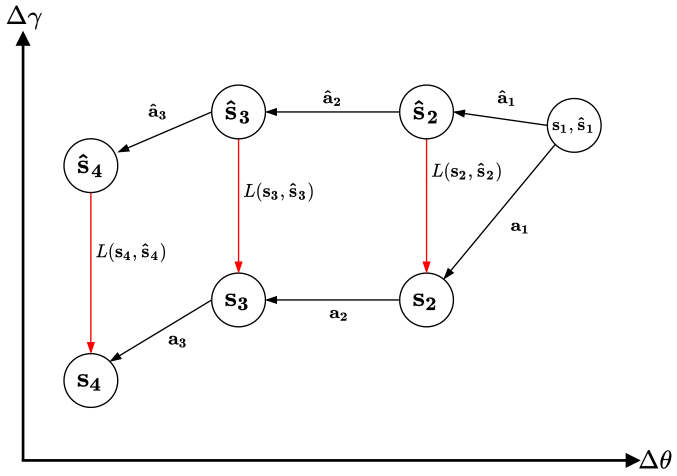


Figure 3.7: An illustration of loss calculation on a sequence of states. The model starts in the same state as the human $\hat{\mathbf{s}}_\mathbf{1} = \mathbf{s}_\mathbf{1}$. A loss $L(\mathbf{s}_t, \hat{\mathbf{s}}_t)$ is calculated for each timestep in a recording.

3.7.1 Deviation in Sequences of States

Given a recording with starting state s_1—containing delta angles $\Delta\theta\Delta\gamma$—and total timesteps T, a model would start at s_1 and make predictions for T timesteps. This results in a sequence of states visited by the model $\hat{S}$ that can be compared with the corresponding states S visited by the human in the recording. Interpreting the state as a position in 2-dimensional space allows a loss to be calculated based on the euclidean distance between state s_t and $\hat{s}_t$ for each timestep $t \in [1, T]$. An example is shown in Figure 3.7. The first action by the model $\hat{\mathbf{a}}_\mathbf{1}$ is severely different from the human action $\mathbf{a}_\mathbf{1}$, causing a deviation. In the future states, the model actions are similar to the human, however, due to initial error in $\hat{\mathbf{a}}_\mathbf{1}$, the loss $L(\mathbf{s}_t, \hat{\mathbf{s}}_t)$ is still present in the future states.

Gathering the visited states $\hat{S}$ by the model for a given set of recordings R is done with Algorithm 3. The algorithm iterates over each timestep t for each recording in R. At $t = 0$, the model start in the same state as the human in the recording. For every timestep in the recording, the model predicts the angular velocity it wants in *lookahead* timesteps, which is based on the state-action pairs from Algorithm 2. From the predicted angular velocity, acceleration for the current timestep can be calculated as shown in Equation 3.2.

Algorithm 3 Simulates model predictions on given recording(s)

function SIMULATEPREDICTIONS($model, R, lookahead$)
  $\hat{S}_{tot} \leftarrow [\dots]$ ▷ States encountered by the model through predictions.
  $\hat{A}_{tot} \leftarrow [\dots]$ ▷ Corresponding actions for each state.
  for each $recording \in R$ **do**
    $S, \_ \leftarrow GetStatesAndActions(recording, 1)$
    $\hat{S}, \hat{A} \leftarrow [\dots]$
    $\mathbf{d} \leftarrow S[0]$ ▷ Delta angles vector $\Delta\theta, \Delta\gamma$
    $\mathbf{v}, \mathbf{a} \leftarrow [0.0, 0.0]$
    for $Length(S)$ **do** ▷ Number of timesteps in recording.
      $\hat{S}.Append(\mathbf{d})$
      $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}$
      $\mathbf{d} \leftarrow \mathbf{d} - \mathbf{v}$

      $\mathbf{v_p} \leftarrow Inference(model, \mathbf{d})$ ▷ Get new velocity prediction.
      $\mathbf{a} \leftarrow (\mathbf{v_p} - \mathbf{v})/lookahead$
      $\hat{A}.Append(\mathbf{v_p})$
    end for
    $\hat{S}_{tot}.Append(\hat{S})$
    $\hat{A}_{tot}.Append(\hat{A})$
  end for

  return $\hat{S}_{tot}, \hat{A}_{tot}$
end function

3.7.2 Using Relative RMSE for Loss Calculation

Root-mean-square error (RMSE) is a common metric for measuring a models predictability [31]. It is defined as following:

$$RMSE(Y, \hat{Y}) = \sqrt{\frac{1}{|Y|}\sum_{i=1}^{|Y|}(y_i - \hat{y_i})^2} \tag{3.6}$$

Where Y is the set of actual values and $\hat{Y}$ is the respective set of predicted values. Squaring the error ensures that the error increases quadratically instead of linearly. It is important that larger errors have a higher impact than smaller errors. Finally, to receive an error in the same unit as $y, \hat{y}$, the square root is applied. RMSE does not calculate a relative error. With a relative error, a predicted value of, for example, $\hat{y} = 50$ and an actual value $y = 60$ will have the same error as $\hat{y} = 5$ and $y = 6$. This can be solved by

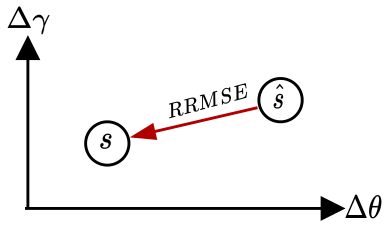


Figure 3.8: A visual representation of how the RRMSE is calculated for timestep t. The human is present in state s, while the model is present in state $\hat{s}$. RRMSE is calculated by the angular euclidean distance between $(s, \hat{s})$.

using the relative RMSE (RRMSE) instead:

$$RRMSE(Y, \hat{Y}) = \sqrt{\frac{\sum_{i=1}^{|Y|} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{|Y|} y_i^2}} \tag{3.7}$$

RRMSE results in a percentage error relative to the actual value and is important for the relevant task, where the error should consider the magnitude of the delta angles $\Delta\theta, \Delta\gamma$. For example, the error from deviating 10 mrad from an actual value of 100 mrad, is equal to the error from deviating 1 mrad from an actual value of 10 mrad.

The RRMSE needs to be calculated for both yaw and pitch to achieve a final loss. As seen in Figure 3.8, the final loss for a single timestep is calculated by taking the euclidean distance of the RRMSE in yaw and pitch between s and $\hat{s}$.

3.7.3 Grid Search K-fold CV for Parameters: $k, lookahead$

Finally, to measure the performance of a model with a given value for k and *lookahead*, Algorithm 3 and the loss calculation explained previously can be used. By fitting a model for each combination of the parameters—in a set range—the loss can be calculated with K-fold CV. A 5-fold CV is chosen, meaning that 80% of the recordings are used for fitting the model and 20% for testing. A low K also helps lower the computational cost.

The implementation of grid search K-fold CV can be seen in Algorithm 4. The function is embarrassingly parallel—as each combination of the parameters can be tested in parallel [11]. The loss matrix L, defined on Line 2, holds the loss for each combination of parameters. Shuffling the order of the recordings in R is done with a set seed and later split into K folds. Shuffling the recordings before doing k-fold CV is necessary, otherwise, the model may train on biased data that does not consist of data points from all types of scenarios. As each recording consists of a time series, it is important that the shuffling is not done on the individual data points for each recording. Instead, shuffling the whole recordings ensures that the model trains and is tested on non-biased data.

For each combination of parameters, iteration is done over each fold, where the test set is the current fold. For each test set, a model is fitted with the remaining folds and

a loss is calculated. After iterating over each fold, the final model loss is calculated by taking the mean loss from all folds. The mean loss is stored in the loss matrix L at the corresponding position.

3.8 Evaluating the Optimized Model

There are generally two main approaches for evaluating the performance of an IL model. First, a quantitative approach can be used where it is possible to model a score based on the task performance. If the model can achieve a similar score to the expert, it may be satisfactory. This approach may be enough of an evaluation criteria depending on the task. However, it does not evaluate how well the model resembles the expert. In the relevant task, the resemblance is vital to evaluate, as the automated tests need to behave similarly to a human tester. Evaluating the resemblance can be done in a qualitative approach by employing a Turing test, where demonstrations by the human expert and the model are shown to a group of people who do not know if they are observing the human or the model [29].

3.8.1 Evaluating Performance

For the relevant task, the model can be scored based on the time it takes to reach the object. It is important that the score is based on a stable sight movement at the object, for example, if the human moves the sight to the object and suddenly moves away from the object, it should result in a lower score. This can be solved by imposing a certain amount of time needed to be spent close to the object before a score can be calculated. Finding the time when the sight is stable at the object is done in Algorithm 5, where S is the sequence of states for the human or model, Δd is the minimum angular euclidean distance from the object required, and Δt is the number of consecutive timesteps needed. If the model or human fails to stabilize at the object during the limited time of the recording, *nil* is returned instead.

The time of stabilization does not consider how the model is moving to the object. A similar time to the human can be achieved with different behavior in regards to sight movement. The actual movement is a matter of resembling the velocity and acceleration of the human and is a vital metric to evaluate alongside the task performance.

3.8.2 Evaluating Resemblance

Evaluating the resemblance between the model behavior and the human is a matter of creating a sub-optimal policy that is satisfactory. Given a sequence of unseen states, the model should behave similarly to what would be expected from the human. Therefore, evaluating resemblance is possible by comparing the path taken to the object, and how the velocity of the model resembles the human.

Algorithm 4 Grid search K-fold cross-validation for parameters $k, lookahead$

function GridSearchCV($R, k_{folds}, k_{min}, k_{max}, lookahead_{max}, seed$)

L $\leftarrow$ $\underbrace{\left.\begin{bmatrix} 0.0 & \dots & 0.0 \\ \vdots & \ddots & \vdots \\ 0.0 & \dots & 0.0 \end{bmatrix}\right\}_{k_{max}-k_{min}+1}}_{lookahead_{max}}$ ▷ Initialize the loss matrix.

$Shuffle(R, seed)$ ▷ Shuffle the recordings order.
$F \leftarrow Split(R, k_{folds})$ ▷ Split the whole recordings into k folds.

for $k \in [k_{min}, k_{max}]$ **do**
  for $lookahead \in [1, lookahead_{max}]$ **do**
    $L_{folds} \leftarrow [\dots]$ ▷ Loss array used for storing loss from all folds.
    for $fold \in F$ **do**
      $R_{test} \leftarrow fold$ ▷ One fold for test set.
      $R_{train} \leftarrow \forall f(f \in F \wedge f \neq fold)$ ▷ Training on all other folds.

      $S, \_ \leftarrow GetStatesAndActions(R_{train}, lookahead)$
      $model \leftarrow KNeighboursRegressor(k)$
      $model.Fit(S, A)$

      $S, \_ \leftarrow GetStatesAndActions(R_{test}, 1)$
      $\hat{S}, \_ \leftarrow SimulatePredictions(model, R_{test}, lookahead)$
      $L_{folds}.Append(Loss(S, \hat{S}))$
    end for
  end for
  $loss \leftarrow Mean(L_{folds})$ ▷ Final loss calculated from all folds.
  $L[lookahead - 1, k - k_{min}] \leftarrow loss$ ▷ Matrix indexes starts from zero.
end for

return L
end function

function Loss($S, \hat{S}$)
  $\Delta\theta_{error} \leftarrow RRMSE(S_{\Delta\theta}, \hat{S}_{\Delta\theta})$
  $\Delta\gamma_{error} \leftarrow RRMSE(S_{\Delta\gamma}, \hat{S}_{\Delta\gamma})$
  return $\sqrt{\Delta\theta_{error}^2 + \Delta\gamma_{error}^2}$
end function

Algorithm 5 Returns the time of stabilization at the object

```
1: function GetTimeOfStabilization(S, Δd, Δt)
2:     t_c ← 0                                   ▷ Count the consecutive valid timesteps.
3:     for t ∈ [0, length(S)) do
4:         Δθ, Δγ ← S[t]
5:         if √(Δθ² + Δγ²) ≤ Δd then
6:             t_c ← t_c + 1
7:         else
8:             t_c ← 0
9:         end if
10:        if t_c = Δt then return t
11:    end for
12:    return nil
13: end function
```

Chapter 4

Implementation

4.1 Overview of Implementation

The simulation is built in C++ and runs on Linux OS with Ubuntu. Implementation is mainly done with Python and Scikit-learn for ML [45]. Scikit-learn is a lightweight and easy-to-use framework that suits the purpose well.

The physical sight runs the virtual simulation on a built-in computer. The user can view the virtual world through the scope of the sight, or through a separate attached monitor. Alternatively, the simulation can run on a separate computer with Ubuntu, which is used for the development of the simulation. This means that the sight is controlled with a keyboard and mouse instead.

The implementation consists of two main parts. First, if the physical sight is used by a human, recordings of the sight motion are captured and stored in files. Second, with a separate computer, the simulation is executed and the sight is controlled with the IL model described in Chapter 3. The model trains on the recordings and tries to imitate the physical motion of the sight.

Figure 4.1 shows an overview of the implementation. The test automation is done through a Python application, which communicates with the simulation and the model. Communication between the Python application and the simulation is done through two files, the state file, and the action file. The simulation writes the current state of the simulation to the state file, containing the position of the sight and object, and the current rotation of the sight. The Python application writes the desired angular velocities of the sight to the action file. For each frame—in 60 Hz—the simulation will apply the current angular velocities to the current sight rotation. For example, if the action file contains a yaw velocity of 1 $mrad/timestep$, the simulation will update the yaw angle of the sight by 1 mrad for each frame.

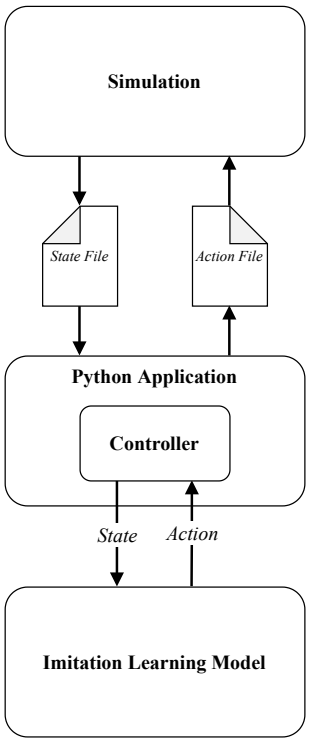


Figure 4.1: An overview of the implementation, showing how the communication is done with the simulation and the IL model.

4.2 The Simulation & Sight

When using the physical sight, as seen in Figure 4.2, the virtual world is seen through the scope of the sight and through a separate monitor attached to the sight. The separate monitor displays the virtual world from the perspective of the user, which is shown in Figure 4.3. Looking through the scope of the sight gives a zoomed-in view of the current sight rotation. The user sits on a seat and can rotate the sight horizontally and vertically, i.e., yaw and pitch. The user changes the yaw with their feet by rotating the whole sight—including the seat. The pitch is rotated up and down by holding two handles on each side of the sight. Both axes can be changed independently of each other. In other words, the yaw can be rotated without changing the pitch and vice versa. The state of the sight consists of an X, Y, and Z position in the virtual world and a current yaw and pitch rotation. The virtual world contains a stationary object in the sky, with a corresponding X, Y, and Z position. For regression testing, the user tries to rotate the sight towards the object as fast as possible. This involves precision and is prone to errors when rotating the sight, especially at higher velocities. If the user controls the sight with a keyboard instead, the precision needed is severely lower—causing an unrealistic motion of the sight.

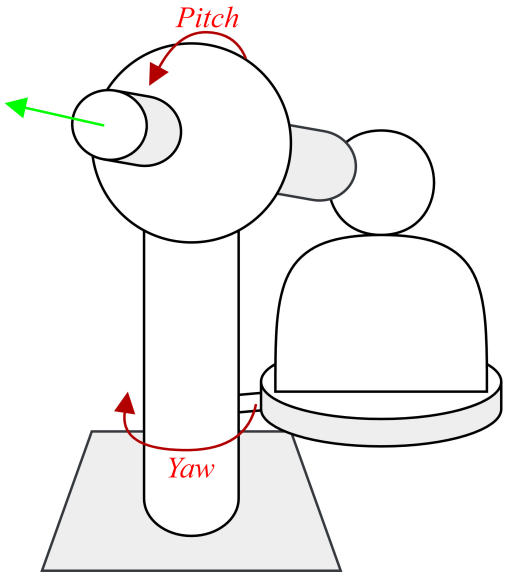


Figure 4.2: A crude illustration of the sight—showing how it can be rotated by a human.

4.3 Recording and Monitoring the Sight

The flowchart in Figure 4.4 shows how the state of the sight is monitored. The user starts by loading a scenario in the menu of the simulation. When starting a scenario, the virtual world is loaded where an object is present in the sky. The simulation is updating at 60 FPS, and for each frame, the current state of the sight is processed. The current state contains the raw features seen previously in Table 3.2. The sight state is overwritten to the state file, which is read by the Python application. The first decision asks if the simulation is being recorded, and if true, the current state of the sight is appended to the current recording file. If the simulation is not recording, the next decision decides if the simulation should start recording at this frame. If true, a new recording file is created and the current state is appended to it. If false, the simulation continues to the next frame.

A recording involves moving the sight to different starting rotations, starting a new recording, moving the sight to the object, and finally stopping the recording. To mitigate human bias in the recordings, two different users were recorded. Rotating the sight towards the object is highly dependent on the experience of the user. Therefore, it is desirable to have recordings of different users with different levels of experience. It is worth noting that the recordings are partly represented by an untrained user, where the amount of human error is severely increased. However, human error is desirable in the recordings, as the goal is for the model to learn human-like errors.

Small amounts of lag in the simulation that causes longer frame times than 1/60 seconds are present. If the frame time is longer than 1/60 seconds, it would affect the calculated angular velocity between frames. The recordings do not contain the current

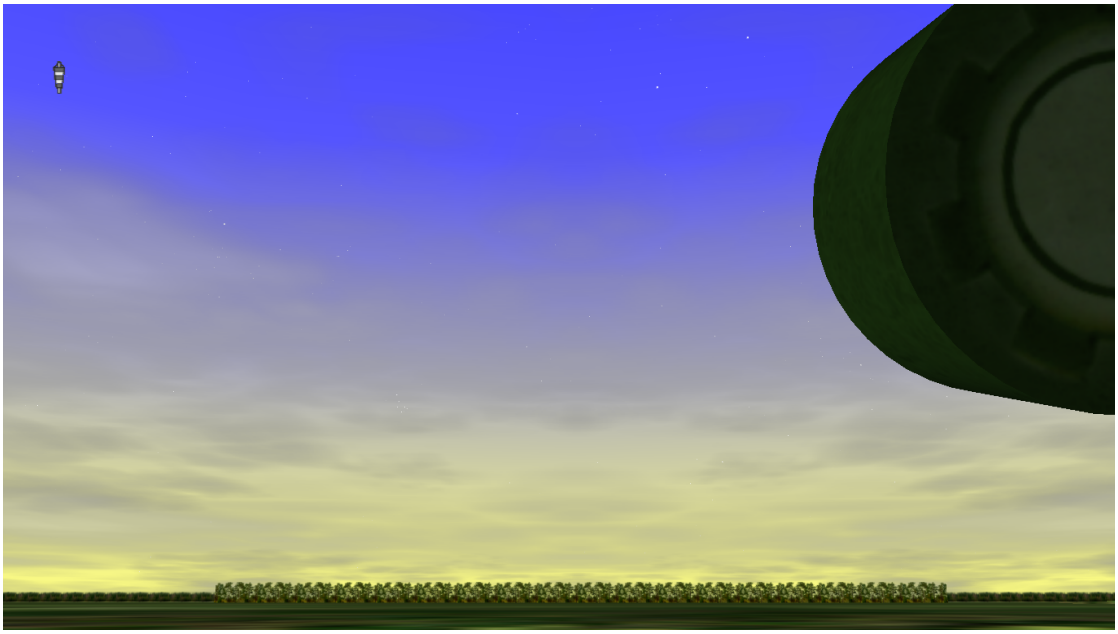


Figure 4.3: The virtual simulation from the perspective of the user, where the sight is slightly visible in the upper right corner.

elapsed time between each frame, instead, it is approximated to 1/60 seconds between every frame. However, the observed difference in frame time is minuscule—with an error of less than 3% between frames. Therefore, the frame time is not accounted for between frames and is assumed to always be 1/60 seconds.

4.4 Controller

The controller, shown in Algorithm 6, controls the communication between the model and the sight velocity. The controller queries the model every third frame, i.e., 20 Hz. The model is trained on state-action pairs at 20 Hz, where the *lookahead* value is timesteps in 20 Hz. As the simulation updates in 60 Hz, the controller has to ensure that the model prediction is translated to 60 Hz.

The loop in Algorithm 6 consist of two if statement blocks. The first code block is executed at 60 Hz and updates the angular velocities $\mathbf{v}$ with the current angular accelerations $\mathbf{a}$. The new $\mathbf{v}$ is written to the action file. The second code block, running at 20 Hz, handles the model communication. The current state from the state file is read and sent to the model for inference. The model returns the predicted angular velocities $\mathbf{v_p}$ that is desired for the next *lookahead* timesteps in 20 Hz. The new angular accelerations are calculated according to Line 12, where $\mathbf{a}$ is calculated according to the *lookahead* in 60 Hz.

The ternary condition is controlled by the boolean variable *running*, which is assigned

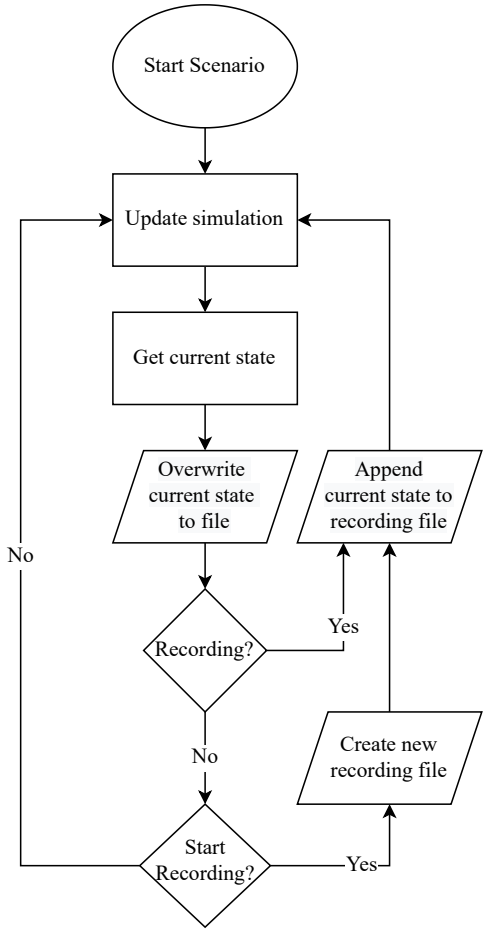


Figure 4.4: A flowchart showing how the simulation writes current state to files.

at Line 13. The boolean is assigned false if the euclidean angular distance to the object is less than 2 mrad and $||\mathbf{v}|| < 0.1$ mrad. The non-zero margins are necessary as entering the state with $\Delta\theta = \Delta\gamma = 0$ is not plausible. The distance margin of 2 mrad is considered to be satisfactory. It is also difficult for the regression model to achieve the action $||\mathbf{v}|| = 0$ close to the object. Therefore, a margin of $||\mathbf{v}|| < 0.1$ mrad is satisfactory.

Algorithm 6 Model control loop of sight

function ControlSight($model, lookahead, f_{sim} \leftarrow 60Hz, f_{model} \leftarrow 20Hz$)
    $\mathbf{v}, \mathbf{a} \leftarrow [0.0, 0.0]$
    while running **do**
        if next timestep in f_{sim} Hz **then**
            $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}$ ▷ Update current velocity.
            $WriteActionToFile(\mathbf{v})$
        end if
        if next timestep in f_{model} Hz **then**
            $\mathbf{O}, \mathbf{S}, \theta_s, \gamma_s \leftarrow ReadCurrentStateFromFile()$
            $\mathbf{d} \leftarrow GetSignedDeltaAngles(\mathbf{O}, \mathbf{S}, \theta_s, \gamma_s)$
            $\mathbf{v_p} \leftarrow Inference(model, \mathbf{d})$ ▷ Get new velocity prediction.
            $\mathbf{a} \leftarrow (\mathbf{v_p} - \mathbf{v})/(lookahead * (f_{sim}/f_{model}))$
            $running \leftarrow \neg(AtObject(\Delta\theta, \Delta\gamma)$ **and** $||\mathbf{v}|| < 0.1_{mrad})$
        end if
    end while
end function

function AtObject($\Delta\theta, \Delta\gamma$)
    return $\sqrt{\Delta\theta^2 + \Delta\gamma^2} < 2_{mrad}$ ▷ 2 mrad is the margin of error.
end function

Chapter 5

Experimental Results

5.1 Parameter Optimization Results

The loss matrix L received from Algorithm 4 is shown in Figure 5.1. The best parameters for the model are $lookahead = 4$ and $k = 24$ with a loss of $RRMSE \approx 0.557$. The mean loss for the delta yaw and pitch respectively are $RRMSE(\Delta\theta) \approx 0.279$ and $RRMSE(\Delta\gamma) \approx 0.497$.

The *lookahead* parameter in Figure 5.1 has a major impact on the loss. If $lookahead = 4$, the difference made by optimizing k is minimal, meaning that the unique case of $k = 1$ is not much worse than the optimal value of $k = 24$. It also shows that $lookahead = 1$ has the worst performance, as the predicted velocities of the model are instantly applied to the velocities of the sight.

5.2 Resemblance Results

In Figure 5.2 and Figure 5.3, the delta angles $\Delta\theta\Delta\gamma$ for the human and the model on the validation set are plotted. The plots provide an overall view of how the model behavior compares to the human. The $\Delta\theta$ in Figure 5.2 shows that the model resembles the human well, by using a similar path to the object. However, the $\Delta\gamma$ in Figure 5.3 shows less resemblance to the human. Learning the pitch movement is more difficult than the yaw movement—as it is more inconsistent across recordings. The sudden moments where the human decides to correct the pitch are difficult for the model to imitate. Instead, the model follows a smoother curve towards the object, which does not capture the motion of the pitch properly.

In Figure 5.4 and 5.5, the mean and standard deviation of the velocities are plotted over relative time in each recording in the validation set. This will give a better understanding of how the overall behavior of the model resembles the human. Comparing the yaw velocity in Figure 5.4, with a resulting $RRMSE(\Delta\theta) \approx 0.230$, the standard deviation shows that the model does not reach higher yaw velocities than the human, and keeps a relatively similar yaw velocity for most timesteps.

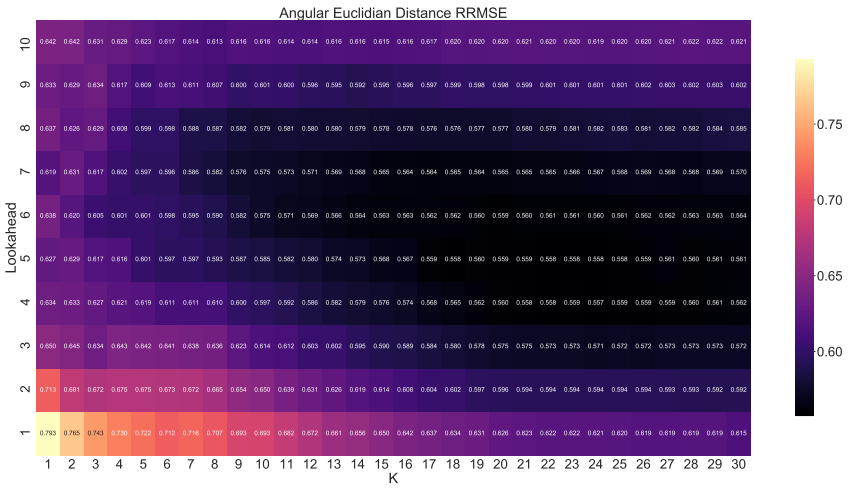
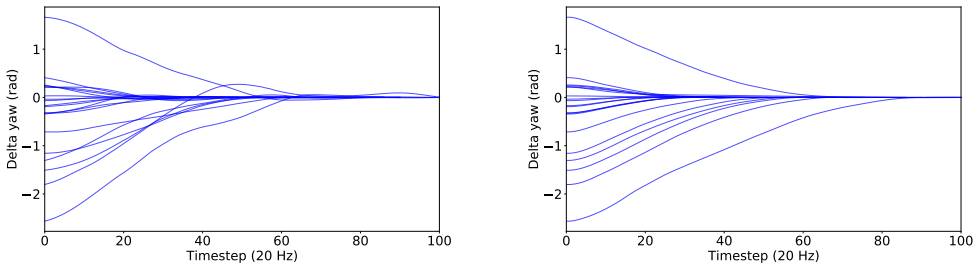


Figure 5.1: A color bar showing the model angular euclidean distance RRMSE with $S = (\Delta\theta, \Delta\gamma)$ and parameters $(k, lookahead)$. Generated with the returned loss matrix L in Algorithm 4.



(a) Human $\Delta\theta$.

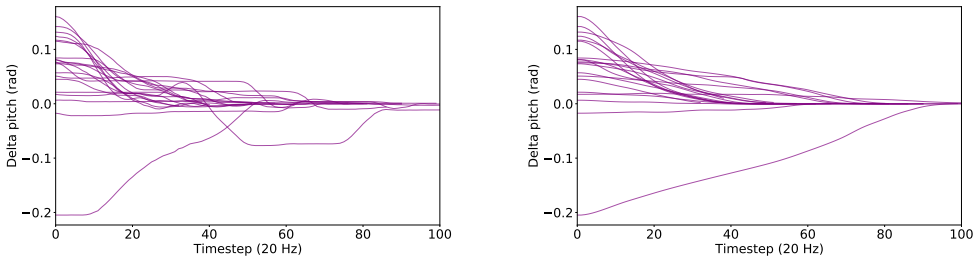
(b) Model $\Delta\theta$.

Figure 5.2: Plotting the delta yaw $\Delta\theta$ for human and model on the validation set.

For the pitch velocity in Figure 5.5, the model mean and standard deviation are less similar to the human, with a resulting $RRMSE(\Delta\gamma) \approx 0.471$. The human mean and standard deviation fluctuate more—making it difficult for the model to fit. It is worth noting that the human mean pitch velocity is almost one magnitude less than the yaw velocity for most timesteps, where more timesteps contain a near-zero velocity—leaving the model with less significant information.

5.3 Performance Results

Table 5.1 shows the time of stabilization for the human t_h and model t_m on the validation set with $\Delta t = 10$ and $\Delta d = 20$ $mrad$. Out of 20 recordings, the model is faster in 12 recordings, slower in 7, and equal in 1. This indicates that the model—overall—performs slightly better than the human on the validation set. Table 5.1 also shows the deviation



(a) Human $\Delta\gamma$. (b) Model $\Delta\gamma$.

Figure 5.3: Plotting the delta pitch $\Delta\gamma$ for human and model on the validation set.

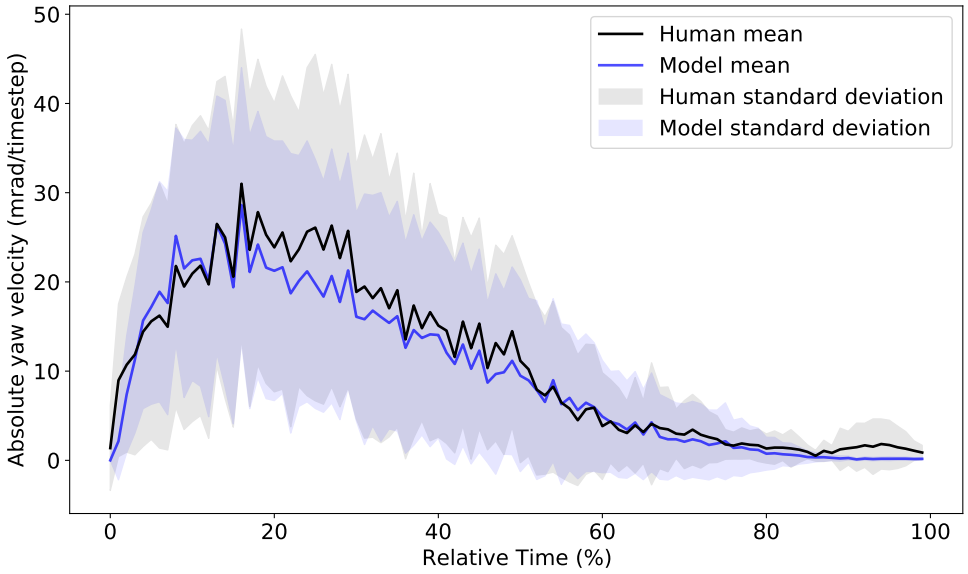


Figure 5.4: The mean and standard deviation for yaw velocity over relative time.

in time between the model and human, where $\frac{t_h - t_m}{t_h}$ indicates the percentage deviation from the human. This resulted in a mean deviation of $\mu \approx +4.9\%$ and a standard deviation of $\sigma \approx 16.7\%$.

The recordings $R_i = 4$ and $R_i = 10$ in Figure 5.6 show the edge cases where the difference of $\frac{t_h - t_m}{t_h}$ is the lowest and highest. The circle indicates the $\Delta d = 20$ $mrad$. In (a), showing the lowest difference, the starting point is above the object. The model takes a shorter path to the object, however, the longer time for stabilization compared to the human indicates that the model moves slower. This edge case is visible in (b) of Figure 5.3, where there is only one recording starting above the object, i.e., a negative $\Delta\gamma$. In (b) of Figure 5.6, the model was 30.8% faster than the human. The model takes a smoother path to the object and keeps a higher velocity.

The recordings $R_i = 9$ and $R_i = 13$ in Figure 5.7 shows two examples where the human does not manage to stabilize at the object, while the model does. The recording

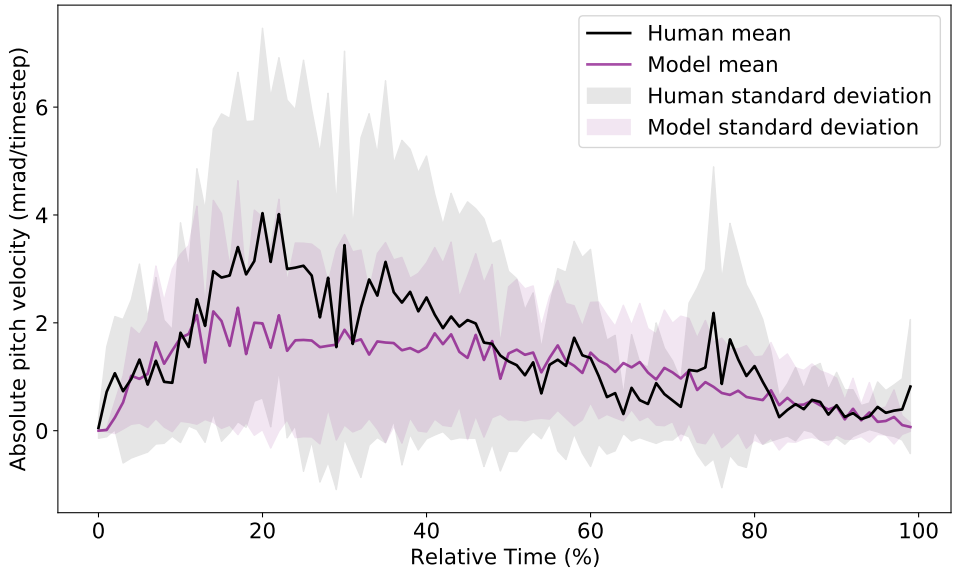
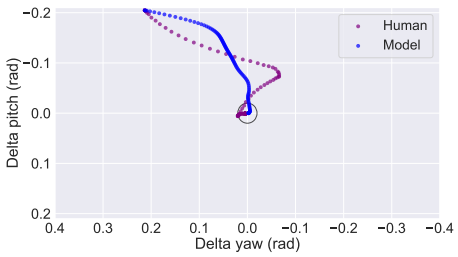
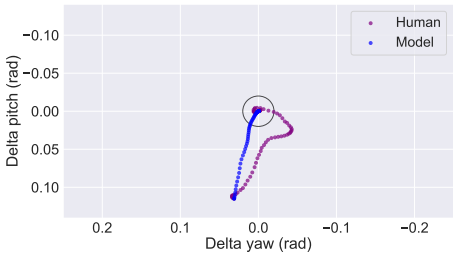


Figure 5.5: The mean and standard deviation for pitch velocity over relative time.

was simply stopped prematurely—meaning that the human did not spend 10 consecutive frames at the object by the end. Both examples show that the model takes a smoother path compared to the human. Furthermore, the model slightly overshoots the object while crossing it. In (b) of Figure 5.7 the human does a severe overshoot that is overexaggerated and therefore uncommon. The human overshoot is also visible in Figure 5.2 at timesteps 40–60, where there is a visible "bump" in one of the recordings—reaching a $\Delta\theta \approx 0.3$ rad.

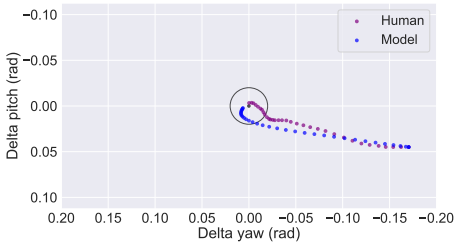


(a) $R_i = 4, T = 100$.

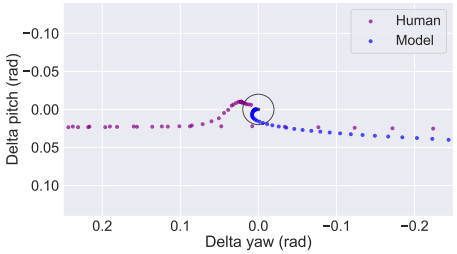


(b) $R_i = 10, T = 63$.

Figure 5.6: Plotting the states for the recordings with the lowest (a) and highest (b) differences for $\frac{t_h - t_m}{t_h}$ in Table 5.1. The circle has the radius of $\Delta d = 20$ mrad.



(a) $R_i = 9, T = 33$.



(b) $R_i = 13, T = 87$.

Figure 5.7: Plotting the states for the two recordings where human $t_h = nil$ and model $t_m \neq nil$ in Table 5.1. The circle has the radius of $\Delta d = 20$ mrad.

R_i	T	t_h	t_m	$t_h - t_m$	$\frac{t_h - t_m}{t_h}$
1	58	42	45	-3	-0.071
2	81	61	58	+3	+0.049
3	82	35	36	-1	-0.029
4	100	68	92	-24	-0.353
5	119	68	49	+19	+0.279
6	41	39	35	+4	+0.103
7	52	41	46	-5	-0.122
8	104	nil	94	NaN	NaN
9	33	nil	29	NaN	NaN
10	63	52	36	+16	+0.308
11	77	50	36	+14	+0.280
12	91	73	66	+7	+0.096
13	87	nil	74	NaN	NaN
14	52	44	36	+8	+0.182
15	91	62	63	-1	-0.016
16	105	87	76	+11	+0.126
17	50	41	45	-4	-0.098
18	39	nil	nil	NaN	NaN
19	40	39	36	+3	+0.077
20	67	43	44	-1	-0.023

Table 5.1: The time of stabilization t_h for human and t_m for model on all the recordings in the validation set. T is the total number of timesteps in the recording. The time is calculated with Algorithm 5, where $\Delta d = 20$ mrad and $\Delta t = 10$ (20 Hz), and has the unit $mrad/timestep$.

Chapter 6

Discussion

6.1 Human Resemblance

This section tries to answer **RQ1.1**, which asks the following: *How well can an IL model resemble human interaction with limited data?*.

The possibility for an IL model to resemble the human depends greatly on the task complexity. In the relevant task, the environment consists of a simple design, where the model is given a position in 2D space and predicts a 2D velocity vector. This is possible due to the high-level—and simple—design of the features and actions. Furthermore, the chosen actions ensure that a 1-step deviation from the human is not detrimental, which allows for the use of a BC approach—the simplest form of IL. The choice of BC is also reinforced by its proven effectiveness for learning motion [18, 49, 29]. It is also worth noting that a k-NN model does not require any training, as the dataset is the model. This allows for an easy process of augmenting new recordings to the model "on the fly", without needing to spend time re-training it.

6.1.1 Velocity Resemblance

There is a clear distinction between the yaw and pitch motion for the human. Therefore, it is important to discuss the two motions individually. The resulting RRMSE loss for the individual motions shows that the model fits the yaw motion much better than the pitch.

Figure 5.4, and 5.2 show resemblance in the yaw velocity mean and standard deviation. In the time range of 20%–50%, the model has a slightly lower mean and standard deviation. However, the model velocity is still following a similar pattern as the human during this time range. This may be caused by the model generally taking a smoother path towards the object—leading to lower velocities needed to reach the object. For example, in (a) of Figure 5.6, the human is taking a much longer path in the yaw to reach the object compared to the model.

The mean and standard deviation for human pitch velocity in Figure 5.5 changes more hectically than the yaw. At the time range 10%–40%, the model mean and standard

deviation is lower than the human. This is a similar phenomenon as seen with the yaw velocity previously. The model generally takes a smoother path towards the object, which is clearly seen in Figure 5.3.

The smoother path towards the target is bound to happen with the k-NN regressor. For example, if the model interpolates two neighbors with equal distance weight, where one neighbor has a 0 *mrad/timestep* velocity, and the other neighbor has a 5 *mrad/timestep* velocity, it will estimate a velocity of 2.5 *mrad/timestep*. The times taken for stabilization in Table 5.1 shows that the model—for most times—stabilizes at the object faster than the human. Therefore, the lower mean pitch velocity simply indicates a much smoother path than the human.

Through observations of the model controlling the sight in the simulation, it is difficult to conclude that the model does not resemble the human well—despite the stated differences. The pitch motion is small compared to the yaw, and therefore, it is hard to spot the difference between the human and the model in real-time. Nevertheless, the model tends to take a smoother path than the human, which is visible in Figures 5.6 and 5.7.

In Figure 5.7, the model overshoots the object slightly at the end—causing a sudden correction. This is a common human error that the model is imitating to an extent. This imitation may benefit from the *lookahead* parameter, which limits the model's capability to suddenly change the velocity of the sight. For example, with $lookahead = 1$, the model would follow a stricter path towards the object—resulting in unrealistic motion.

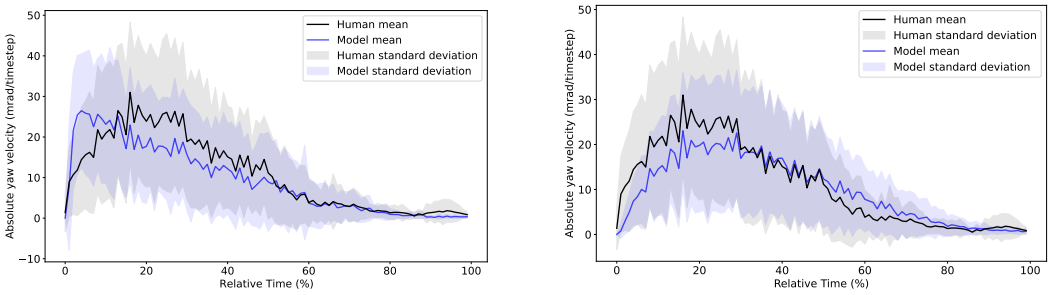
6.1.2 Task Score Performance

Table 5.1 shows that the model tends to be slightly faster than the human. The question is, can the model move faster than the human by an unrealistic amount? One problem is that the validation set is quite small and therefore is severely affected by anomalies. As seen in (b) of Figures 5.7 and 5.6, the human path is not always ideal. The model was severely faster than the human in these cases as the human is performing worse than on average.

6.1.3 The *lookahead* Parameter

In Figure 5.1, the impact on loss when tuning the *lookahead* parameter is noticeable. The main goal of the *lookahead* parameter was to disallow the model from adjusting the velocity too quickly. Figure 6.1 (a) shows how the model instantly increases the yaw velocity to a high value with $lookahead = 1$, which results in an unrealistic acceleration. This may be a limitation of only having the delta angles in the state, however, instead of adding more features, increasing the *lookahead* value solves this also. A too large value for *lookahead*, such as $lookahead = 10$ in (b) of Figure 6.1, results in a model that has lower initial yaw acceleration than the human—failing to reach a high enough velocity in the time range 0%–30%.

It is also worth noting that the *lookahead* parameter lowers the impact of the k parameter. In Figure 5.1, having a $lookahead = 1$ shows the most change in loss when



(a) $k = 24$ and $lookahead = 1$.

(b) $k = 24$ and $lookahead = 10$.

Figure 6.1: The mean and standard deviation for yaw velocity over relative time for $lookahead = 1$ and $lookahead = 10$.

increasing the k, however, with a $lookahead = 4$ the value of k has a much lower impact on the loss. This is most likely due to the fact that, with $lookahead > 1$ and any value for k, the next velocity in the next timestep is always an interpolation of the predicted velocity in $lookahead$ timesteps. Therefore, the significance of interpolating k nearest neighbors has a lower impact on the resulting velocity.

6.1.4 Improving the Human Resemblance

As stated by Cardamone et al., using high-level states and actions can significantly increase the performance of a BC model [18]. The choice of states and actions is therefore not trivial, and multiple approaches may result in a satisfactory model. The approach taken for the thesis revolved around keeping the environment as simple as possible while resulting in a satisfactory model. However, multiple proposals were made for the design of the environment.

Why not include the current velocity in the state?

It may be logical to include the current velocity of the sight in the state, as this would separate the states on more dimensions than only the distance from the object. Adding current velocity to the state was tested, and it showed no significant decrease in loss—approximately 0.5% higher RRMSE with $k = 21$ and $lookahead = 6$. It was observed in the simulation that, when including current velocity in the state, the model would overshoot a lot more, and take a longer time to stabilize at the object. The reason behind this is left for speculation, however, increasing the number of features will cause the recordings to be more separated from each other in the dataset, which causes the model to achieve larger deviations from the human demonstrator, and have a higher bias towards individual recordings. Therefore, when close to the object and having a high velocity, the model would imitate similar recordings more precisely—leading to larger errors. It is desired by the model to act as such, however, a larger dataset may be needed to prove better results.

The behavior of the sight motion has a clear impact on this as well. For example, when the human rotates the yaw of the sight, an initial force is given—causing an acceleration—and quickly resulting in a constant velocity. This means that achieving a constant velocity does not take many timesteps, and the constant velocity starting from, for example, π rad from the object is not much different from starting $\pi/2$ rad from the object.

Why not include the current time in the state?

Another possibility to improve the model performance is to include the current time in the state. This can be done by splitting the recordings into discrete time sequences. For example, states with $t \leq 10$ has a weight of $w(t) = 0$ and states with $t > 10$ has a weight of $w(t) = 0.1$. This would cause a small cut-off where the initial 10 timesteps of recordings do not interfere as much with the model performance after timestep 10. During testing, this showed a small decrease in loss—approximately 0.5% lower RRMSE with $k = 17$ and $lookahead = 6$.

Why is it difficult for the model to predict acceleration?

The difference in difficulty for predicting the velocity and acceleration may not be evident at first. However, due to the limitations of BC, a 1-step deviation from the demonstrator has to be tolerable. If the model predicts the acceleration, errors caused in each step may be compounded, leading to an increasing deviation that is difficult to recover from. If the model predicts the velocity instead, the error caused in one step may be corrected in the next step—avoiding an increasing deviation.

6.2 Using IL for Test Automation

This section tries to answer **RQ1.2**, which asked the following: *How feasible is the use of IL for test automation?*.

To create an IL model that resembles the human demonstrator well is a challenge, however, creating an IL model that can be useful for test automation is another challenge in itself. This mainly boils down to the complexity of the task. Implementing an IL model for test automation is useful if the task requires human interaction and is time-consuming to do manually. However, due to the involvement of human interaction, such tests are difficult to automate.

The proposed reason for using IL in test automation is to create a model of the human tester, which can properly interact with the environment no matter the current state. Record/playback is useful in simpler cases, such as automating manual GUI testing. However, in a more complex environment, it can be difficult and time-consuming to maintain such recordings. With an IL approach, the model is not limited to a single recording, instead, it estimates the correct action at any given state based on a set of recordings.

As stated in Chapter 1, the manual regression tests for the relevant task can take up to two days to do manually for one developer. Therefore, it is valuable to enable test automation, as it would severely decrease the time spent on manual tests—while also enabling the possibility to run the regression tests more frequently. This is the main reason behind the choice of such a simple IL approach with a limited dataset—as it lowers the potential cost of automation. However, ensuring that the IL model will behave properly is difficult to assure and depends greatly on the complexity of the task and environment. This can quickly become a costly model to implement as responsibility is increased. Therefore, it is convenient to keep the IL model small and have minimal responsibility. This becomes easier to integrate into a whole test automation process while allowing for better flexibility compared to record/playback.

6.3 Limitations of the BC Model

It may be desired for the model to perform a more difficult task, and due to the limitations of BC, the chosen approach may fail to work. A prominent example is to rotate the sight towards a moving object instead. The model may struggle to follow the object as it is not aware of the trajectory of the object. This was tested in the simulation and showed that the model would successfully move to the object, however, it would later struggle to follow the object at lower velocities.

The model may also need to handle more actions, for example, predicting when to press—or hold—a button while rotating the sight. Adding new actions require a new dataset and possibly new features to enable good model performance. The performance of the BC model depends greatly on how detrimental the deviation from the human is, therefore, successfully adding new actions depends greatly on the complexity and importance of the new action. An action that depends on a multitude of new features will severely affect the small dataset—causing a much larger deviation from the human. Furthermore, an action that requires high precision and where a small error has a detrimental impact, the BC model will most likely struggle to perform well.

6.4 Threats to Validity

This section will discuss threats to internal, external, and construct validity—as described by Crano et al. [19]. The set of methods used throughout the thesis may be limited and lack a proper reflection of the model performance. Furthermore, the results are severely affected by the dataset used.

6.4.1 Threats to Internal Validity

Internal validity is the level of confidence that the presented results are not affected by alternative explanations. In the relevant task, the collected data may affect the results in ways that may harm internal validity.

During data collection, the human demonstrator is aware of being recorded, which may lead to cognitive bias. The human demonstrator may change their behavior with the knowledge of being recorded. This may lead to a different dataset compared to what would be achieved during "normal" testing circumstances.

The small size of the dataset hinders the capability to validate the results. A small dataset was necessary, as it is collected manually and limiting the time spent collecting data was an important factor for real use. However, the small dataset implies a small validation set—20 recordings in total. The validation set is solely used to evaluate the resulting resemblance of the model, and if it does not represent the human well, it may jeopardize the validity of the model.

6.4.2 Threats to External Validity

External validity refers to how well the presented results translate to a different scenario. In the relevant task, the results are severely limited to a specific task, and may not produce similar results in a different task.

The translation from simulating the control of the sight in Algorithm 3 and controlling the real sight in the simulation may cause contradicting results. Algorithm 3 is a simplified version of the real simulation, and measuring model performance in the real simulation may cause different results.

The dataset was mainly collected by inexperienced testers. Collecting a dataset from experienced testers may result in different results, as it would consist of fewer inconsistencies. With an inexperienced tester, the initial recordings may differ from the final as the learning is increased. However, with an experienced tester, the learning process is already surpassed—leading to a more consistent motion and decision making. In other words, it would result in a dataset with less irreducible error, i.e., noise that can not be reduced by a better model.

6.4.3 Threats to Construct Validity

Construct validity concerns the level of validity in a measurements true construct of interest, or in other words, how well a measure reflects its true intention.

The used loss function is based on the model deviation from the human—in terms of euclidean distance for $\Delta\theta, \Delta\gamma$ at each timestep. This method may not result in a loss that fully reflects the model's resemblance to the human. For example, if the model is slightly slower than the human in the first few timesteps, and afterward follows a similar path as the human did—it would result in an unfair total loss for the model. This is coincidentally illustrated in Figure 3.7, where the initial mistake by the model is compounded across multiple timesteps, even if the next predictions are similar to the human.

Chapter 7

Conclusion

7.1 Future Work

Many approaches and improvements have been left unexplored in the thesis. Here, the most prominent ideas are presented.

The results show a large difference in the model resemblance for the individual motions, yaw and pitch. A potential solution to improve the pitch resemblance is to use two separate models for the motions—optimized separately.

The *lookahead* parameter uses SMA of future accelerations to calculate the interpolated acceleration for the next timestep. There are other more complex alternatives to SMA, for example, exponential moving average (EMA), where the impact of future accelerations decreases exponentially over time [55]. However, using EMA instead of SMA requires the model to predict all future *lookahead* velocities, as each future acceleration is needed to calculate EMA. This may be solved by integrating the EMA velocity calculation into the model action. The model would be trained on predicting the EMA velocity for timestep $t + 1$, instead of the actual velocity in *lookahead* timesteps.

The chosen BC approach was motivated by simplicity and proven effectiveness for controlling motion. However, another BC approach, namely, NEAT has also proven good results [18]. It would also be interesting to try SQIL, which has shown great performance [48], and may be possible to implement in the relevant task, as training can be done offline through a modified version of Algorithm 3.

It was intentional to use a small dataset to increase practicality for test automation. A larger dataset may show different results, for example, lower loss with more informative features, such as current velocity and time. The dataset would also benefit from consisting of recordings from more than two users, and possibly, more skilled users that have undergone significant training with the sight.

The used loss function was created with the intention of evaluating the model's capability to follow the same path as the human. However, as mentioned in 6.4.3, the loss function is limited as it strictly penalizes the model deviation from the human state in the same timestep. For example, if the model follows the same path as the human, however, it would start one timestep behind, the loss function would penalize the model

unfairly. Instead of mapping states over time, a nearest neighbor approach could be used. In this case, the loss would be calculated by the deviation from the nearest human state, instead of the human state at equal time.

Outliers in the dataset were not removed. This was done intentionally to avoid removing human factors in the dataset, even if certain recordings show overexaggerated behavior, for example, the human path in (b) of Figure 5.7. However, keeping such outliers in the small validation set severely affects the loss of the model. Removing outliers would result in more accurate loss, and therefore, is recommended.

7.2 Final Words

Manual software testing is a costly and time-consuming task of the development cycle. For complex systems with large state spaces requiring human motion control and decision-making, simpler test automation solutions, such as record/playback can quickly become impractical. The thesis aimed to solve this problem by applying IL for automating human behavior while limiting the approach through the lens of practicality. This meant a limited dataset and a simple IL approach to lower the cost of implementation.

The results presented a simple BC approach using a k-NN regressor that, to an extent, manages to resemble the human in unseen states. Through the use of high-level states and actions, the BC model's resemblance to the human was increased, without increasing model complexity. The chosen states and actions also ensured that deviations from the human do not have a detrimental impact on the model performance.

Finally, the thesis has managed to answer the research questions as follows:

- **RQ1.1:** *How well can an IL model resemble human interaction with limited data?*

  This depends greatly on the complexity of the task in question. A BC approach can quickly struggle to resemble the human as the complexity of the environment is increased. However, the results showed that by designing better features and actions the complexity of the model can be kept low while achieving resemblance to the human. For example, the *lookahead* parameter was designed specifically for the relevant task, which limited the model control of the sight by hindering sudden changes in velocity.

- **RQ1.2:** *How feasible is the use of IL for test automation?*

  Again, this mainly depends on the complexity of the task, and how much a company is willing to invest. However, if manual tests are severely time-consuming and can not be executed as frequently as desired, large investments into test automation can be rewarding. Therefore, using IL for test automation may be a plausible solution, as it can further extend a company's ability to automate, while also increasing the robustness of the software. It is important to note that the thesis focused mainly on how IL can be used in software testing—by showing capability to imitate human motion control with limited data. To further expand the answer, it would require more measurements, such as potential cost reduction

and time saved. It would also be necessary—in greater detail—to evaluate what kind of tests IL can be used for, and how it can be feasible for different test cases.

- **RQ1:** *How can IL be used for automating human interaction in software tests where record/playback is impractical?*

  The thesis has shown that a simple supervised learning approach with the k-NN algorithm can use a set of recordings to create a model that can generalize on unseen states. Given an unseen starting state, the model shows capability of approximating a new recording—similar to how a human recording would resemble.

Bibliography

[1] atan2 – Mathematical functions in Python. https://docs.python.org/3/library/math.html#math.atan2. Accessed: 2022-03-11.

[2] Nearest Neighbors. https://scikit-learn.org/stable/modules/neighbors.html#regression. Accessed: 2022-02-24.

[3] Scikit-learn Repository – Distance Weighted Neighbors. https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/neighbors/_base.py#L97. Accessed: 2022-04-25.

[4] TORCS. http://torcs.sourceforge.net/. Accessed: 2022-01-31.

[5] Abbeel, P., and Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning* (2004), p. 1.

[6] Arora, S., and Doshi, P. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence 297* (2021), 103500.

[7] Ayodele, T. O. Types of machine learning algorithms. *New advances in machine learning 3* (2010), 19–48.

[8] Bartle, R. A. *Designing virtual worlds*. New Riders, 2004.

[9] Bellman, R. A markovian decision process. *Journal of mathematics and mechanics* (1957), 679–684.

[10] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517.

[11] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *Journal of machine learning research 13*, 2 (2012).

[12] Berrar, D. Cross-validation. In *Encyclopedia of Bioinformatics and Computational Biology, Volume 1* (2019), Elsevier, pp. 542–545.

[13] Bertolino, A. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)* (2007), IEEE, pp. 85–103.

[14] Billard, A., and Matarić, M. J. Learning human arm movements by imitation:: Evaluation of a biologically inspired connectionist architecture. *Robotics and Autonomous Systems 37*, 2-3 (2001), 145–160.

[15] Borjesson, E., and Feldt, R. Automated system testing using visual gui testing tools: A comparative study in industry. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation* (2012), IEEE, pp. 350–359.

[16] Bostrom, N., and Yudkowsky, E. The ethics of artificial intelligence. *The Cambridge handbook of artificial intelligence 1* (2014), 316–334.

[17] Buiten, M. C. Towards intelligent regulation of artificial intelligence. *European Journal of Risk Regulation 10*, 1 (2019), 41–59.

[18] Cardamone, L., Loiacono, D., and Lanzi, P. L. Learning drivers for torcs through imitation using supervised methods. In *2009 IEEE symposium on computational intelligence and games* (2009), IEEE, pp. 148–155.

[19] Crano, W. D., Brewer, M. B., and Lac, A. *Principles and methods of social research*. Routledge, 2014.

[20] Dobslaw, F., Feldt, R., Michaëlsson, D., Haar, P., de Oliveira Neto, F. G., and Torkar, R. Estimating return on investment for gui test automation frameworks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (2019), IEEE, pp. 271–282.

[21] Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., and Guimaraes, M. P. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability 68*, 3 (2019), 1189–1212.

[22] DW, G. D. A. Darpas explainable artificial intelligence program. *AI Mag 40*, 2 (2019), 44.

[23] Fewster, M., and Graham, D. *Software test automation*. Addison-Wesley Reading, 1999.

[24] Fletcher, J. Education and training technology in the military. *science 323*, 5910 (2009), 72–75.

[25] Florence, P., Lynch, C., Zeng, A., Ramirez, O. A., Wahid, A., Downs, L., Wong, A., Lee, J., Mordatch, I., and Tompson, J. Implicit behavioral cloning. In *Conference on Robot Learning* (2022), PMLR, pp. 158–168.

[26] Géron, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2019.

[27] Ghahramani, Z. Unsupervised learning. In *Summer school on machine learning* (2003), Springer, pp. 72–112.

[28] Hu, C., and Neamtiu, I. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), pp. 77–83.

[29] Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR) 50*, 2 (2017), 1–35.

[30] Hwang, C.-L., Chen, B.-L., Syu, H.-T., Wang, C.-K., and Karkoub, M. Humanoid robot's visual imitation of 3-d motion of a human subject using neural-network-based inverse kinematics. *IEEE Systems Journal 10*, 2 (2014), 685–696.

[31] Hyndman, R. J., and Koehler, A. B. Another look at measures of forecast accuracy. *International journal of forecasting 22*, 4 (2006), 679–688.

[32] Hynninen, T., Kasurinen, J., Knutas, A., and Taipale, O. Software testing: Survey of the industry practices. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)* (2018), IEEE, pp. 1449–1454.

[33] Imran, A., and Kosar, T. Software sustainability: a systematic literature review and comprehensive analysis. *arXiv preprint arXiv:1910.06109* (2019).

[34] Iverson, K. E. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference* (1962), pp. 345–351.

[35] Jamil, M. A., Arif, M., Abubakar, N. S. A., and Ahmad, A. Software testing techniques: A literature review. In *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)* (2016), IEEE, pp. 177–182.

[36] Kibriya, A. M., and Frank, E. An empirical comparison of exact nearest neighbour algorithms. In *European conference on principles of data mining and knowledge discovery* (2007), Springer, pp. 140–151.

[37] Li, K., and Wu, M. *Effective GUI test automation: developing an automated GUI testing tool*. Sybex, 2005.

[38] Memon, A. M., Pollack, M. E., and Soffa, M. L. Hierarchical gui test case generation using automated planning. *IEEE transactions on software engineering 27*, 2 (2001), 144–155.

[39] Mohri, M., Rostamizadeh, A., and Talwalkar, A. *Foundations of machine learning*. MIT press, 2018.

[40] Montavon, G., Samek, W., and Müller, K.-R. Methods for interpreting and understanding deep neural networks. *Digital signal processing 73* (2018), 1–15.

[41] Müller, V. C., and Bostrom, N. Future progress in artificial intelligence: A survey of expert opinion. In *Fundamental issues of artificial intelligence*. Springer, 2016, pp. 555–572.

[42] Myers, G. J., Sandler, C., and Badgett, T. *The art of software testing*. John Wiley & Sons, 2011.

[43] Ng, A. Y., Russell, S. J., et al. Algorithms for inverse reinforcement learning. In *Icml* (2000), vol. 1, p. 2.

[44] Patton, R. *Software testing*. Pearson Education India, 2006.

[45] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[46] Planning, S. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* (2002), 1.

[47] Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems 1* (1988).

[48] Reddy, S., Dragan, A. D., and Levine, S. Sqil: Imitation learning via reinforcement learning with sparse rewards. *arXiv preprint arXiv:1905.11108* (2019).

[49] Renman, C. Creating human-like ai movement in games using imitation learning, 2017.

[50] Ross, S., Gordon, G., and Bagnell, D. 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* (2011), JMLR Workshop and Conference Proceedings, pp. 627–635.

[51] Sawant, A. A., Bari, P. H., and Chawan, P. Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA) 2*, 3 (2012), 980–986.

[52] Shepard, D. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference* (1968), pp. 517–524.

[53] Stanley, K. O., and Miikkulainen, R. Evolving neural networks through augmenting topologies. *Evolutionary computation 10*, 2 (2002), 99–127.

[54] Sterten, J., and Ogorodnyk, O. Application of modern educational methods through implementation of the ambulance simulator at a clinic laboratory (ntnu gjøvik). *Procedia CIRP 54* (2016), 41–46.

[55] Su, Y., Cui, C., and Qu, H. Self-attentive moving average for time series prediction. *Applied Sciences 12*, 7 (2022), 3602.

[56] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

[57] Winsberg, E. *Computer simulations in science*. (2013).

[58] Wolf, M. Embedded software in crisis. *Computer 49*, 1 (2016), 88–90.

[59] Wong, W. E., Horgan, J. R., London, S., and Agrawal, H. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering* (1997), IEEE, pp. 264–274.

[60] Xu, F., Uszkoreit, H., Du, Y., Fan, W., Zhao, D., and Zhu, J. Explainable ai: A brief survey on history, research areas, approaches and challenges. In *CCF international conference on natural language processing and Chinese computing* (2019), Springer, pp. 563–574.

[61] Yue, Y., and Le, H. M. Imitation Learning Tutorial. `https://sites.google.com/view/icml2018-imitation-learning/`. Accessed: 2022-01-27.

[62] Zhang, J., and Cho, K. Query-efficient imitation learning for end-to-end autonomous driving. *arXiv preprint arXiv:1605.06450* (2016).