University of Cincinnati

Date: 11/2/2022

I, Pavan M Saranguhewa, hereby submit this original work as part of the requirements for the degree of Master of Science in Electrical Engineering.

It is entitled:

Pinball: Using Machine Learning Based Control in Real-Time, Cyber-Physical System

Student's name: **Pavan M Saranguhewa**

This work and its defense approved by:

Committee chair: Zachariah Fuchs, Ph.D.

Committee member: John Gallagher, Ph.D.

Committee member: Ali Minai, Ph.D.

UNIVERSITY OF Cincinnati

44162

Pinball: Using Machine Learning Based Control in Real-Time, Cyber-Physical System

A thesis submitted to the
Graduate School
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Master of Science

in the Department of Electrical Engineering and Computer Science
of the College of Engineering and Applied Science
by

Pavan Saranguhewa
BS University of Ruhuna, Sri Lanka

Committee Chair: Zachariah E. Fuchs Ph.D.
Date: November 10, 2022

ABSTRACT

Saranguhewa, Pavan. M.S., Department of Electrical Engineering, University of Cincinnati, November 2022. *Pinball: Using Machine Learning Based Control in Real-Time, Cyber-Physical System.*

Applied Machine Learning on real-time Cyber Physical Systems (CPS) brings several new challenges to Machine Learning (ML) based control. CPS are subjected to environmental changes, noise, hardware limitations and tightly coupled time constraints, which make real-time control a non-trivial task. This thesis work focus on studying applicability of ML based control in real-time CPS using physical pinball machines as sandboxes.

A simulator framework to evaluate ML algorithms in a virtual setting and a real-world framework to evaluate ML algorithms in physical pinball machines are developed. Both frameworks provide visual information and extracted features for the ML agent, and actuates the system according to ML agent control signals. The real-world framework utilizes a real-time state tracker, hardware based synchronizer, and a non-invasive system actuation method to realize the abstracted framework. We discuss the development of the simulation framework and the real-world framework. Subsequently, we move into the application of model-free ML, where we experiment with reinforcement learning under different perception models and modular learning. Finally, we discuss the application of model-based ML where we experiment with Model Predictive Control (MPC) with Deep Neural Networks (DNN) and Support Vector Regression (SVR), on selected primitive goals in the system. Each technique is statistically evaluated and results are presented. The evaluation results showed that ML based MPC was able to reach up to 96% accuracy in the selected shot aiming scenario.

Acknowledgment

There are many people who helped me in numerous ways in reaching this milestone of my education, whom I must thank.

First, I must express my gratitude to my professor and academic advisor, Dr. Zachariah Fuchs, for the kind support and guidance. His insights on scientific problem solving and hands-on training inspired me and also shaped the outcome of this research. I would also like to thank my thesis committee members, Dr. Ali Minai and Dr. John Gallagher for finding time to serve in my thesis committee in their busy schedule. Also, I would like to thank my lab members, Michael Ikuru, Brian Swanson and Pavlos Androulakakis, who have been supportive by all means.

Then I must mention my loving wife, parents, brother, and all of my extended family who have always been there for me. Finally, I have to thank all my friends and everyone else, who helped and inspired me throughout this journey.

Contents

List of Figures

List of Tables

Introduction

Cyber-Physical-Systems (CPS) are systems which combine computer-based software and physical, engineered or natural system hardware to realize an intelligent application. The software components are utilized to monitor, control, coordinate and integrate the associated physical system [1]. A real-time CPS has tight constraints on the wall time for sensing, computation, communication and actuation relative to the physical time of the system. Examples for real-time CPS would be autonomous vehicles that have to make critical maneuvers in real-time, smart grids which have to regulate the system in real-time and aircraft anti-missile defense systems that neutralize attacks in real-time. With the global advancement to Industry 4.0 and IIoT, more and more systems have advanced into this category making the study of control strategies for real-time CPS crucial.

1.1 Motivation

Applications of real-time CPS are commonly associated with safety critical requirements. If we consider the same examples introduced above, autonomous driving has to react to events in the environment in real-time to avoid collisions and ensure safety. Smart grids have to respond quickly to changes in the system to avoid interruptions and hazards to it's subscribers as well as the safety of the grid, while the defense systems have to detect, track, and neutralize threats in real-time to safeguard the aircraft.

Recently, sophisticated Machine Learning (ML) methods such as AlphaFold[2], AlphaGo[3] and Dall-E[4] have achieved impressive performance and it is tempting to adopt similar so-

phisticated ML methods in other domains, including CPS. Though researchers are now looking for explainable AI and causal machine learning techniques, state-of-the-art ML methods are still black box models. Application of such techniques in real-world systems such as CPS, raise concerns about reliability and safety. Formal verification of the ML techniques prior to use in safety-critical systems is crucial, but the data requirements for ML methods limit the applicability, due to the cost of data generation in CPS in general. To address this issue, there are many simulation models developed for Model Based Testing (MBT) on CPS[5, 6, 7]. But the results will only be as good as the model's ability to emulate the real world. Hence, hardware based, real-time CPS sandboxes are essential to testing and validating ML control techniques.

Physical pinball machines are real-time physical systems with non-linear dynamics. A game of pinball generally consists of a single or multiple balls, two flippers, bouncers, fixed targets, responsive targets, ramps and rails. The combination of these components creates a fast-paced, chaotic physical environment. When making a shot with the flippers, the direction of the ball changes with time (in a resolution of milliseconds) based on the linear and angular momentum of the ball and random noise in the environment. The scoring system in a game of pinball extends from primitive goals like shooting particular targets to complex goals like making sequence of shots, certain types of shots, and certain game modes. Points accumulate till certain number of lives are used per each game, where each life ends when the ball is drained. This physical system, augmented with a framework consisting of a perception system, an actuation system, and a computation resource, create a real-time CPS.

Precise timing and time synchronization are essential for a real-time application. Most physical processes are time-dependent and their interactions have to abide by the time requirements of the process. Constraints on the combined wall time for sensing the system, computing the control instructions, communication between subsystems and actuation have to be met for safe and reliable operation of the system in real-time. For a CPS with

sub-modules, researchers assume an oracle provided time theoretically [8], but in reality, explicit time synchronization between each module and the physical process is required. The developed pinball framework includes explicit time synchronization to allow accurate real-time control.

In this work, we construct a framework applicable over commercial pinball machines in a generalized manner with no alternations to the machine as a real-time CPS sandbox. In parallel, a simulator sandbox is created to pre-validate the ML techniques. We test the applicability of selected ML techniques on both the simulation and the real-world frameworks, then use Principal Component Analysis (PCA) based plots to visualize the models. Finally, we present the performance results of the ML models in both simulation and physical environments.

1.2 Related work

There have been simulation-based, hardware-based and hybrid sandboxes for CPS testing pertaining to different domains including smart grids, automotive, transportation, industrial automation, health-care and robotics. Literature shows that simulation-based and hybrid CPS sandboxes are more common than hardware-based sandboxes, since they are easier to build, highly customizable, scalable, and are cost efficient comparing to hardware-based ones. From the previous studies focused on simulation based and hybrid testbeds for CPS, [9] presented a simulation-based evaluation sandbox for transportation networks in which the simulation environment is easily customizable. A ROS-based CPS simulation environment for testing control methods on groups of ground robots was presented in [10], while [11] presented a simulation environment for testing scheduling control methods in a smart grid. [12] presented a hybrid system, which combines hardware and simulated network properties in their CPS sandbox.

Examples of hardware-based CPS sandboxes are the topics of [13, 14, 15, 16, 17]. A CPS sandbox to test control algorithms and communication security on a low-cost ground

bot is presented in [13] and a hardware-based real-time CPS sandbox to test power system security and control applicable on computer clusters and networks is presented in [14]. An inexpensive UAV CPS sandbox consisting of small drones and a software framework to test control algorithms is presented in [15]. A CPS sandbox for testing cloud communication over smart grid consisting of physical hardware is presented in [16], and a multi-vehicle transportation system environment for testing traffic control methods is presented in [17].

When it comes to experiments involving the game of pinball, there are multiple work done related to both simulated and physical games. Simulator-based work, such as [18], focused on applying reinforcement learning methods on pinball. Moreover, there have been previous work done on functionally similar games like billiards [19] and angry-birds [20], where it is advantageous for the player to strategically aim shots. Few work has been done using real pinball machines [21, 22]. [21] created an AI that activated the flippers whenever the ball came near them and assessed the duration of time during which the agent was able to keep the ball from draining. It also presented the results of a preliminary experiment done on shot aiming, which yielded around a 50% success rate under a constrained pinball playfield. Our work in [22] used a parametric switched mode system and a polynomial fit based model predictive controller for shot aiming in a simplified custom pinball playfield.

1.3 Thesis Organization

Section 2 describes the build of the pinball simulation framework using Unity and the extended features for experimentation. Section 3 describes the experiments done with the simulation framework using model free control methods and their results. Section 4 describes the design and construction of a generalized real-world framework around physical pinball machines, including the features and functions of each component of the system. Section 5 describes the experiments done on the real-world system using model-based control and the results obtained. Finally, the conclusion of our work and possible future directions are presented in Section 6.

Simulation Framework

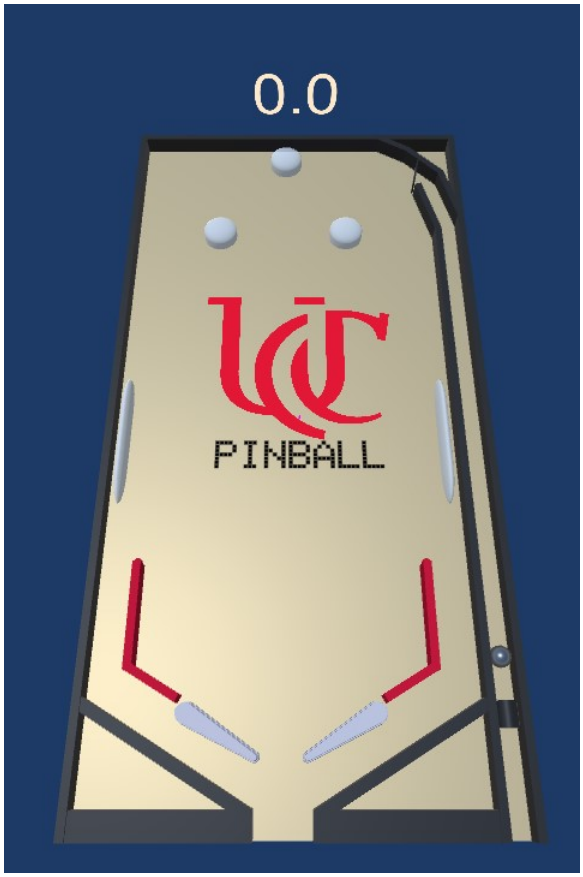


Figure 2.1: Pinball simulator

To construct the simulation environment, we created a virtual pinball game from scratch using the Unity game engine. Unity allows for the use of custom 3D models in the game with high customization of material and physical properties of every element. Unity also contains a physics engine which can simulate gravity to emulate the effect of the slanted playfield in the game. Figure 2.1 shows the pinball game created with Unity. The simulation game consists of a ball plunger connected to a ramp to shoot the ball into

the playfield, two flippers, two side-bouncers and three bouncer-targets covering the main components of a typical pinball game. Proportions of the playfield were made approximately equal to the standard commercial pinball machine, 'Indiana Jones' by Stern Inc. The ball diameter was set to 27mm and the mass was set to 80g to approximately match the size and mass of a real pinball. The angle of the pinball playfield, θ, was set to 7°and gravitational acceleration was set to $9.8ms^{-2}$. The game score is displayed on top of the game window. Scoring rules are customizable as required.

The game is controllable with keystrokes and a joystick in the manual control mode. Control commands include the game start button, reset button, pull the plunger button and left/right flipper buttons. Also, the game could be controlled using internal API calls, which we used in the context of learning agents. The game elements could be customized, such as the ball starting location, game speed, etc. In the simulation environment, the agents are provided with two methods to read the game information: visual information and ball position information. Visual information is passed to the agent from a virtual camera which is mounted on top of the playfield in which the region of capture can be adjusted. As the ball position information, the agent receives the location of the ball in Cartesian coordinates in the plane of the playfield. The agent is also able to read the current score of the game at any given moment. Depending on the learning and control method we choose, the agent can use any combination of the available information channels to read the state of the game.

The next aspect of this framework is the ability to collect training and testing data. As mentioned before, the agent can control the game similar to a human player, and has the ability to play consecutive games. To leverage the capabilities of the simulator, we introduce two options that can drastically speed up the data collection process: speed-up game time and parallel game instances. Unity allows the user to scale up or down the clock used in the game relative to the wall time. By manipulating this feature, we provided an Application Programming Interface (API) handle to the agent to change the game clock scale as desired. This allows the agent to speed up the game up to 100x from the real-time.

The upper limit had to be 100x as inaccuracies of the physics engine started to emerge at speeds above that. Other feature is the parallel game instances. Leveraging the features of the Unity environment, we allowed the agent to spawn multiple game instances to run independently in the available processors for a single data collection task. The games streamed data into the agents' sample database asynchronously till the required number of games samples were collected.

Agents were programmed with *C#*, which has the Unity 's native support. An agent is allowed to define any required customizations to the game configuration through API calls. It can then specify the number of training and test samples, mode of training from online and offline, and the training heuristics to monitor. For neural network based online learning agents, readily available optimization functions from Unity-ML package were used for training the networks. We attempted to use already available pinball game simulation engines: Future Pinball and Visual Pinball, to simulate the game and build a framework.[23][24] Future Pinball is a free software which allows for the design of custom playfields using 3D models of the components. We are able to modify the characteristics of the game such as material properties and the physics properties. Then it simulates the game on the computer screen with support for keyboard control. On the other hand, Visual Pinball consists of a game emulator component named PinMAME (Pinball Multi-Arcade-Machine-Emulator), which allows emulation of ROMs of real pinball machines along with a simulation capability similar to that of Future Pinball. Visual Pinball is mainly used to simulate real-life pinball machines with replicated virtual playfields and the emulated ROMs. Visual Pinball is supported by a large community consisting of players and playfield designers.

Even though we could extract the visual features of the live playfield via capturing the screen of these game simulators, the only way to control the game was via virtual keyboard commands, as there was not an API to directly communicate with the game engine. That was unreliable in the long run with the various interruptions from the operating system. The other issue was that we could not run simulation sessions in parallel or speed up the

game simulation as it was not supported. Interruptions to virtual keyboard commands were preventable using an isolated environment for the simulations. Yet the un-scalability of these simulators to run in parallel or at faster clock speeds, made them unsuitable for our work at the moment.

Simulator experiments: Model free learning

We can model the game of pinball as a Finite-Horizon Markov Decision Process (MDP) and then apply reinforcement learning methods to let an agent learn to play the game. Q-Learning is a well-studied algorithm for finding the optimal policy over MDPs. For systems with a larger number of states where tabular Q-Learning is not practical, Deep-Q-Learning is used where the Q values are approximated with a neural network function. Deep-Q-Learning has shown good performance in learning agents over simulated game environments [25].

3.1 Pinball as a Markov Decision Process

The game of pinball consists of different states and actions. Also, depending on the actions taken in each state, a final reward (score) will be received. Pinball games are finite and deterministic for the most part, except for little noise components present in the system. That allows us to assume the game of pinball to be an MDP. We model the state space of pinball as (a) image pixels and (b) derived features. More details on each state space representation will be presented in Sections 3.4 and 3.5, respectively.

While in play, a player's only input on the game is through the flippers. There are four distinct actions a player can take on any state of the game, which are shown in Table 3.1. The game ends when the player loses the ball from the playfield which makes the horizon

Table 3.1: Action Space

Action	Left flipper	Right flipper
1	Up	Down
2	Down	Up
3	Down	Down
4	Up	Up

of the game process indefinite yet finite. Though the game dynamics are continuous, we can assume the player observes the game state at a certain sampling rate, hence at discrete time steps. Next, we will discuss how the pinball MDP is defined based on these details.

In the following description, we use the terms s_t - state, a_t - action at time step t, and r_t - reward received for taking action a_t in state s_t. The horizon of the Markov chain is variable and depends on the length of each game. For each state and action pair (s, a), a Q-value is defined as the expected sum of present and discounted future rewards following a defined policy π. The discount factor for future rewards is γ $(0 < \gamma < 1)$. Q-value is defined as in Equation 3.1 [26].

$$Q(s, a) = E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi] \tag{3.1}$$

An optimal policy π^* can be formed by following the actions for each state that corresponds to maximum $Q^*(s, a)$, which can be calculated through Bellman Equation,

$$Q^*(s, a) = \max_{\pi} E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s, a_t = a, \pi]. \tag{3.2}$$

To learn this optimal policy on selected gameplay scenarios without using an expert model, we explore the use of Q-Learning, which is a model-free, temporal difference based learning method.

3.2 Optimal policy via Deep-Q-Learning

In Q-Learning, a database of Q-values for each (s, a) pair has to be maintained. Q-Learning algorithm updates the Q values database with each step in the process $(s_t, a_t, s_{t+1}, r_{t+1})$. The update rule for Q-Learning is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{3.3}$$

where α is the learning rate. With enough samples to sweep the entire space of state action pairs, (3.3) is able to converge the Q values [27]. However, as the state space grows in size, the required number of samples increases, which renders the direct use of Q-Learning impractical. Instead, an approximation function to estimate Q values can be used. Learning agents based on Deep-Q-Learning are extensively studied in the literature [28]. Following the work [29], we use Deep-Q-Learning as a non-linear function approximation method, which trains a Deep Neural Network (DNN) to predict Q-values. Weights θ, of DNN are updated as,

$$y_q(\theta) \leftarrow y_q(s_t, a_t, \theta^-) + \alpha[r_{t+1} + \gamma \max_a y_q(s_{t+1}, a, \theta^-) - y_q(s_t, a_t, \theta^-)] \tag{3.4}$$

which is adapted from 3.3. Here, θ^- represents the previous weights of the network. In generating training data, we follow an $\varepsilon - greedy$ policy, where action selection in each state is made according to the policy derived from the current state of DNN with a probability of $1 - \varepsilon$ and randomly otherwise. We make use of the experience replay technique used in [29] to reduce the effect of sample bias.

3.3 Experiment setup

A constrained game configuration was created to test Deep-Q-Learning using the simulator. One of the three targets was assigned points which the player could score by hitting it. The duration of each game episode was set to 300 frames. For each game, the ball start location was fixed at a point above the left flipper. Little noise is added to the simulation engine. The goal of the agent is to score many points as possible by directing the ball toward the selected target without overusing flippers or losing the ball.

During testing, we identified the following hyper-parameters which seemed to have a considerable impact on the overall learning process. Though an extensive study was not carried out on hyper-parameter tuning, they were adjusted through trial and error.

- **Reward and Penalty values**: a reward for flippers hitting the ball, a penalty for each activation of a flipper, and a penalty for losing the ball has to be assigned relative to the fixed reward for hitting the target.

- **Discount factor(γ) of Bellman equation**: $\gamma = 1$ accounts for all future rewards and $\gamma = 0$ discards all future rewards. Optimal γ lets the agent learn, accounting for both short-term and long term rewards under a balance.

- **Neural network hyperparameters**: the architecture of the neural network directly affects it's accuracy and efficiency[30].

- **Training parameters**: learning rate, batch size, and replay buffer size have to be adjusted to avoid convergence issues.

- **Exploration and exploitation**: a good balance between exploration and exploitation at each stage of the learning process leads to a better outcome.

3.4 Image-based DQN

Figure 3.1: Pinball simulator with the viewport of the virtual camera (middle-left)

We placed a virtual camera of 84 x 84 pixel image size on top of the simulator playfield, which captures the area around flippers. Figure 3.1 shows the simulator window with the view of the image captured by the virtual camera during a training session. The number at the top of the playfield is the cumulative game rewards. Color images captured through the virtual camera are converted to 8-bit grayscale images. Since static images do not embed dynamic features of the game, a stack of four consecutive images (as shown in Figure 3.2) is used as the input to the DNN. Overall, this results in a $256 \times 84 \times 84 \times 4 = 7,225,344$ sized state space.

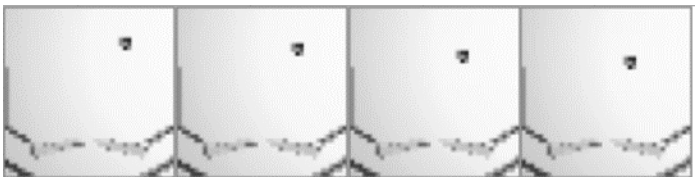


Figure 3.2: A sample input to the DCN where four consecutive frames are stacked

A Deep Convolution Network (DCN) is initialized arbitrarily to use as the Q value estimator following the architecture in [25]. They used an input layer of 84 x 84 x 4 where the input is then passed through 3 convolution layers and 2 fully connected layers. The corresponding action for the max value produced in the final layer is considered the optimal action. Hyperparameters were adjusted by trial and error to find a configuration that gives the maximum hit/miss ratio for the target, minimizing flipper overuse.

Table 3.2: Reward Scheme

Action	Reward
Swinging a flipper	-0.9
Flipper hitting the ball	+1.0
Ball hitting the target	+10.0

Table 3.3: Network configuration

Layer	Layer type	Output shape	Parameter count
1	Input	[4,84,84]	0
2	2D Convolution	[32,20,20]	8,224
3	ReLU activation	[32,20,20]	0
4	2D Convolution	[64, 9, 9]	32,832
5	ReLU activation	[64, 9, 9]	0
6	2D Convolution	[64, 7, 7]	36,928
7	ReLU activation	[64, 7, 7]	0
8	Linear	[512]	1,606,144
9	ReLU activation	[512]	0
10	Linear	[4]	2,052

The selected network structure is shown in Table 3.3, and the assigned reward scheme is shown in Table 3.2. The discount factor, γ, is set to 0.99. The training was done with

a batch size of 32 games, replay memory of 100,000 games, and a learning rate of 10^{-6} over 2×10^6 game samples. Exploration and exploitation balance was achieved using the equation,

$$A_t = \begin{cases} \underset{a}{arg\,max}\ Q_t(a) & \text{with probability } (1-\varepsilon) \\ a \sim Uniform(\{a_1 ... a_k\}) & \text{with probability } \varepsilon \end{cases} \tag{3.5}$$

where,

$$\varepsilon = \varepsilon_{final} + (\varepsilon_{start} - \varepsilon_{final}) \times \exp \frac{-game\ count}{\varepsilon_{decay}} \tag{3.6}$$

and ε_{decay} is tuned empirically.

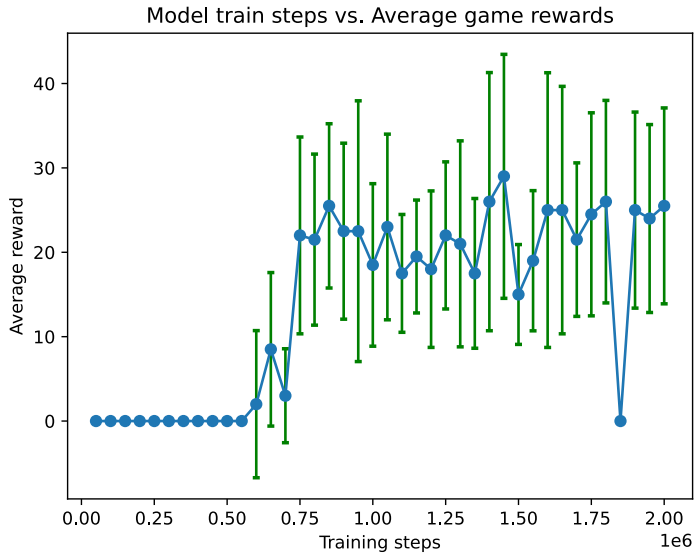


Figure 3.3: Image-based DQN evaluation results

Figure 3.3 shows the variation of game scores evaluated on 50 games, with model save points corresponding to the incrementing number of samples. The agent learned to shoot the target at around 0.75×10^6 training samples. Though the game score improvement was slightly slow from that point onward, we observed that the agent had reduced the overuse of flippers at later training steps. We concluded that the outlier at 1.85×10^6 was a result of a corrupted model save-point as PyTorch was not able to read that model back from the disk.

3.5 Motion feature based DQN

To explore how feature extraction would change the performance, we selected a set of motion parameters: ball position (x, y), ball velocity $(\dot{x}, \dot{y})$ and flipper states (up or down) to use as input features to the agent. The same reward scheme from Section 3.4 was used. Instead of the Deep Convolution Network used in image-based method, a deep neural network is used. The structure of the network is shown in Table 3.4. For training, a batch size of 32, a replay memory of 50,000, a sample size of 5×10^5 games and a learning rate of 10^{-6} were used.

Table 3.4: DNN configuration

Layer	Layer type	Output shape
1	Input	6
2	Linear	512
3	ReLU activation	512
4	Linear	512
5	ReLU activation	512
6	Linear	4

We are able to calculate the error between the ball position and the target location based on the ball position information that is extracted. We repeated the experiment on 3 different configurations to evaluate the agent performance with additional incentives. An additional incentive reward is added to the regular reward scheme. There are three incentive reward schemes:

1. **Original:** No additional incentive.

2. **Field-based:** Additional incentive based on,

$$Incentive = \frac{C}{distance^2 + 1} \tag{3.7}$$

3. **Horizontal line based:** Add an incentive only when the ball passes through the horizontal level of the target as,

$$Incentive = \begin{cases} \dfrac{C}{distance + 1} & \text{if ball crosses the horizontal line} \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

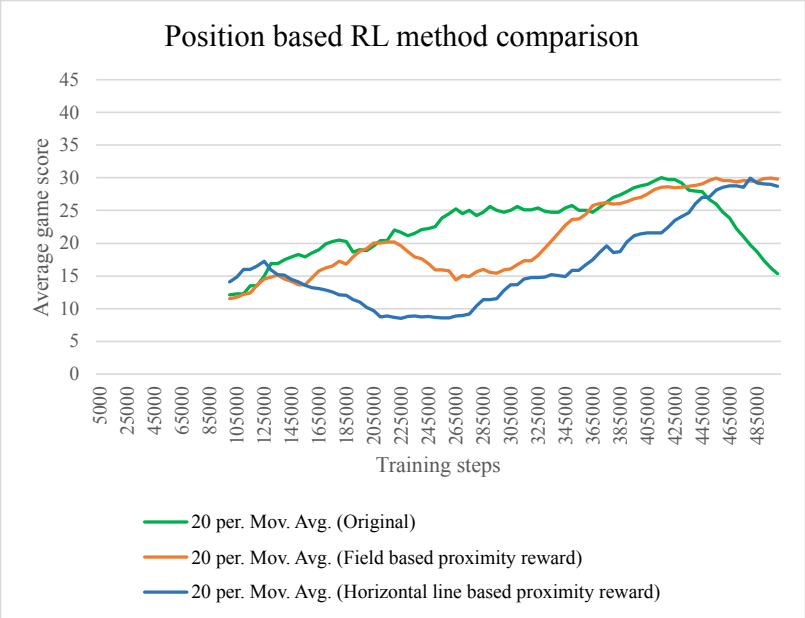


Figure 3.4: Performance comparison of motion feature based methods

C was fine-tuned with a few trials. Figure 3.4 shows the result comparison between the three incentive reward scenarios. All three methods were able to train the agent to score well. Looking at the scores, we see that the addition of proximity-based reward has brought some stability to the trained agent, whereas the agent who used the original method degraded in performance closer to the end, which we can assume to be a result of over-training.

Comparing the average scores recorded for motion feature based learning agent (from Figure 3.3) to image-based learning agent (from Figure 3.4), we can clearly see that motion feature based agent was able to reach the performance level of the image based learning agent using only $1/4^{th}$ of the training samples.

3.6 Modular Reinforcement Learning

In this approach, we studied how the use of a combination of Modular Reinforcement Learning (MRL) agents, which are trained individually, affect the overall performance. MRL breaks down a complex goal into a series of simplified tasks, which allows for the reuse of modular components as a baseline for new learning problems [31, 32]. It also paves the way for a hierarchical learning agent, which can use the trained sub-models as sub-routines in the high-level controller. In this section, we evaluate if decomposing the goal into a series of sub-tasks improves the performance.

For this experiment, we increased the difficulty of the game by allowing the target location to be random in the playfield and letting the ball to start from a random location. We conducted this experiment using the motion feature based learning method due to its sample efficiency as shown in Section 3.5. We train two modular agents: one who trains to capture the ball in cradled position and one who trains to shoot from the cradled position. To compare the performance, a third agent is trained to shoot the falling ball directly. Next two sections describe how the individual agents performed and a comparison between the modular learning agents and direct agent, respectively.

3.6.1 Trained agents

3.6.1.1 Learning to cradle the ball

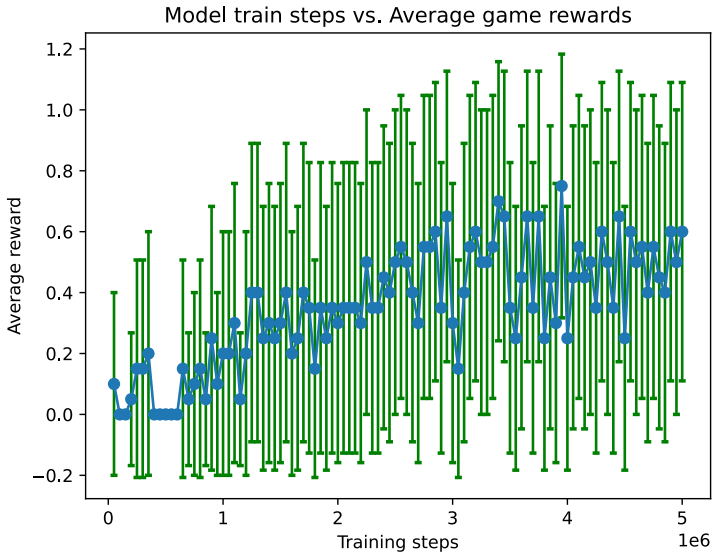


Figure 3.5: Performance of the agent learning to catch the ball in either of the flippers.

This agent is trained to learn how to capture the ball from either flippers within 4 seconds of gameplay. This agent is also the first sub-module of the modular scheme. Figure 3.5 shows the performance of the agent. Rewards were assigned as in Table 3.5 to promote capturing the ball in cradled position with the least number of flips. Initially, the agent tried to capture the ball by overusing the flippers and eventually reduced the flipper overusing behavior slowly as the training progressed.

Table 3.5: Reward Scheme

Action	Reward
Got the ball to cradled position	+1.0
Each flip	-0.1

3.6.1.2 Learning to shoot from the cradled position

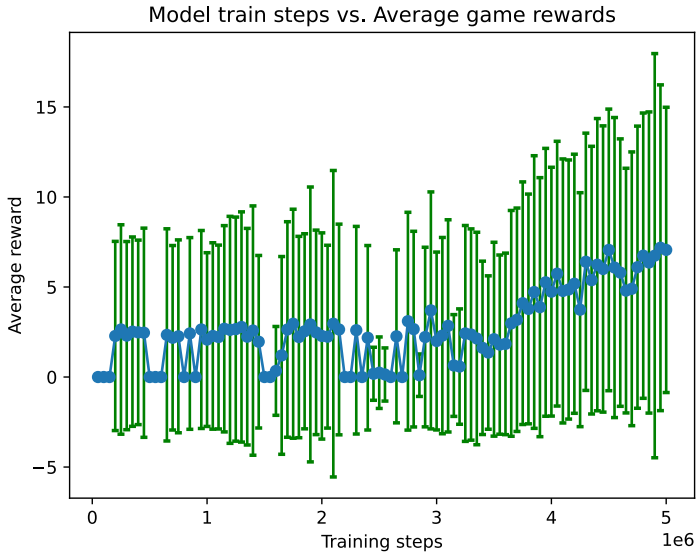


Figure 3.6: Performance of the agent learning to hit a random target starting from the cradled position.

The second sub-module of the modular scheme is this agent, who learns how to shoot the ball toward the target from the cradled position. Rewards were assigned as in Table 3.2. The agent was able to maintain a higher average from the start of the training as the ball starting stationary from the cradled position is intuitively more controllable.

3.6.1.3 Learning to shoot directly

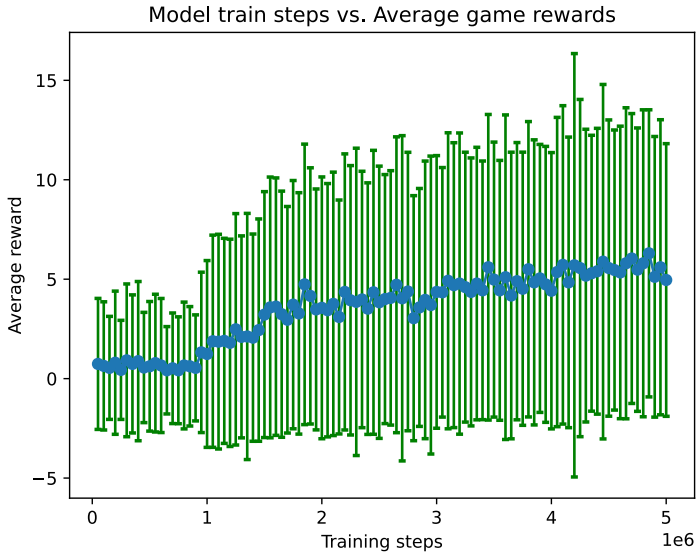


Figure 3.7: Performance of the agent learning to hit the random target as a single task

An agent is trained to hit the randomly positioned target in the playfield with the ball dropped from a random location in the playfield. Rewards were assigned as in Table 3.2. Figure 3.7 shows the performance of the agent in 5 second games averaged over 250 games. We can see that the agent has slowly improved the cumulative reward which requires hitting the target without overusing the flippers.

3.6.2 Direct Learning vs. Modular Learning

A modular learning scheme is obtained by utilizing the agent who learned to cradle the ball (Section 3.6.1.1) and the agent who learned to shoot from cradled position (Section 3.6.1.2). At the game start, the first agent is utilized to capture the ball. Once the ball is cradled, the second agent takes control and shoots the ball up. Either hitting the desired target or the ball falling down the playfield triggers a switch-back to the first agent.

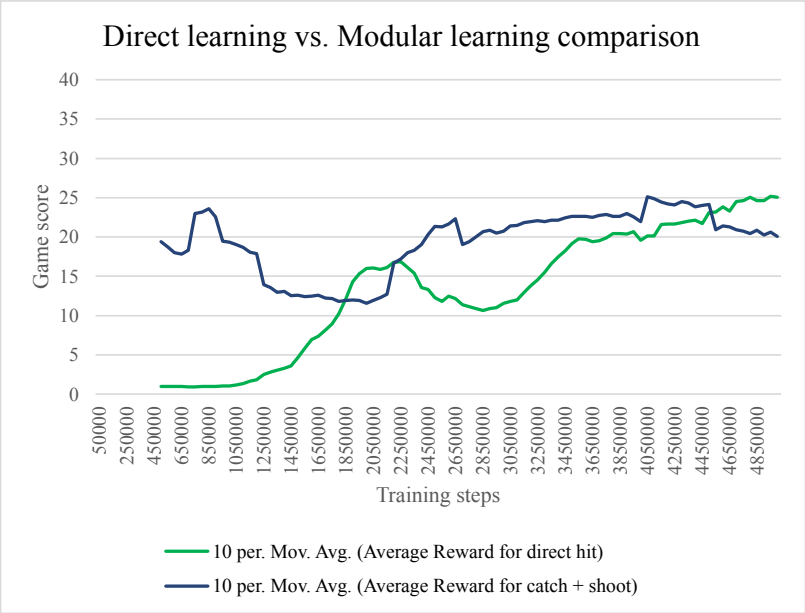


Figure 3.8: Comparison between direct learning and modular learning

Figure 3.8 shows how the performance of a single agent learning to aim a shot directly compares with the combination of two modular agents learning to catch the ball and shoot. An evaluation was done using 1000 second games with just 10 points for each successful hit to the target. The graph shows the moving average of 10 data points. Results indicate that the modular learning method had performed better from the start whereas the direct learning agent could catch up as the number of training samples grew. We can also see that the modular scheme is less stable than the other. We believe this instability can be improved with use of an intelligent and adaptive higher level controller instead of the simple rule based switching method we used.

Possible future steps would be training hierarchical learning agents who utilize the modular knowledge in comprehending complex tasks. DQN based higher level agents[33], HIRO [34], and HAC [35] are among the many possible avenues in hierarchical learning we can pursue [36, 37].

In our learning experiments, goals involved optimizing a combination of sub-goals where some take precedence over others. For example, while aiming a shot towards a selected target, reducing the overuse of the flipper and preventing the ball from sinking

were also present as sub-goals. These sub-goals were introduced to the agents via the reward scheme. Agents had to converge to a policy with an optimal balance between all sub-goals on their own, which can increase the number of training samples required and the possibility of getting trapped in a local extremum. It might be possible to reduce the chance of encountering a local extremum with a gradual training plan where we place a dynamic reward scheme that guides the agent on which goal to focus based on current performance. Same can be applied to learning via simple scenarios initially and gradually pushing the agent to learn complex ones. This is essentially curriculum learning[38] which is used to inject domain knowledge into the training making the process more data efficient. As a future experiment, we can evaluate the improvement of performance and training efficiency with a curated curriculum with our expertise in the task at hand.

Real-world framework

Figure 4.1: Commercial pinball machines in the laboratory

The real-world framework is built for training and testing control agents on real-life pinball machines. The framework is generalized to a certain level, allowing it to be used on different machines with little modifications. Figure 4.1 shows the set of commercial pinball machines we have in our laboratory. The real-world framework consists of a camera to capture the pinball playfield, a hardware actuator that controls the pinball game switches, and a PC containing the agent software. Figure 4.2 shows the components of the framework. The overhead camera streams video to the main computer, which has an image processing

module and a control module. The main computer sends control signals to the hardware actuator which consists of a microcontroller and an opto-isolated control circuit. Details on each of these framework components are discussed in the next sections.

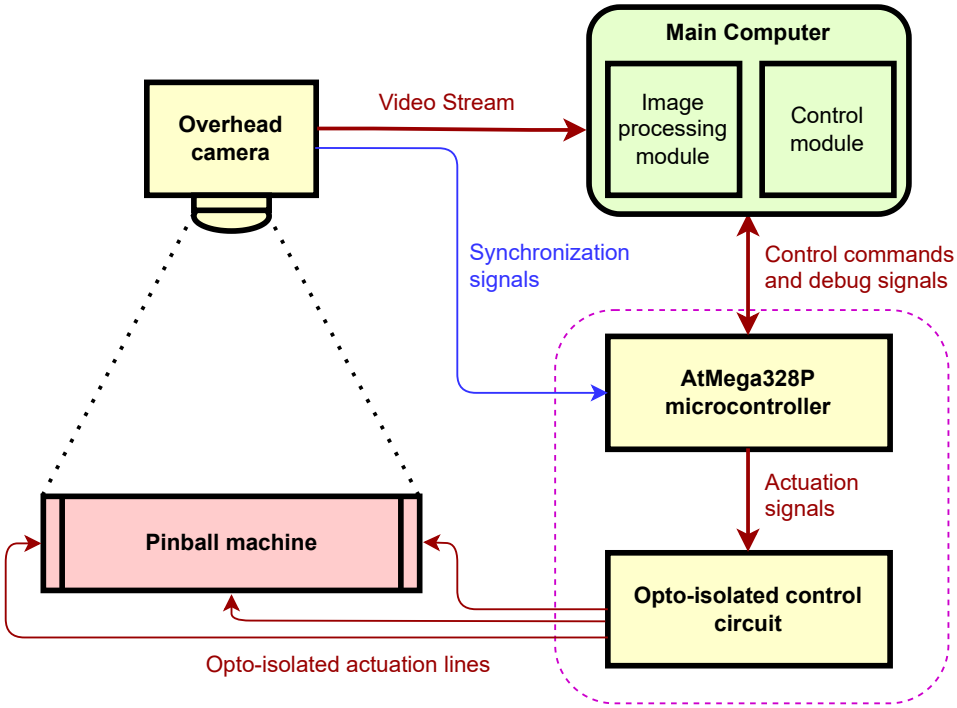


Figure 4.2: Components of the real-world framework

4.1 Overhead Camera

(a) Camera and the lens (b) Camera placement

Figure 4.3: Overhead camera of the framework.

We place an overhead camera that captures the area of the pinball machine's playfield. Figure 4.3a shows a closer look at the camera and Figure 4.3b shows how the camera is positioned over the 'Stranger Things' commercial pinball machine. The camera we used is 'FLIR Blackfly S BFS-UE-32S4C', which is extensively customizable and capable of capturing RGB images at 118 frames per second in 2048x1536 resolution. It also contains a high-precision real-time clock (accuracy up to 1ms) and a configurable GPIO port, which we use for the framework functions. The camera settings, including exposure time, resolution, frame-rate, color-mode, trigger mode were configurable via API calls. The camera uses USB 3.0 to stream data and has the capability of transmitting an image of 2048x1536

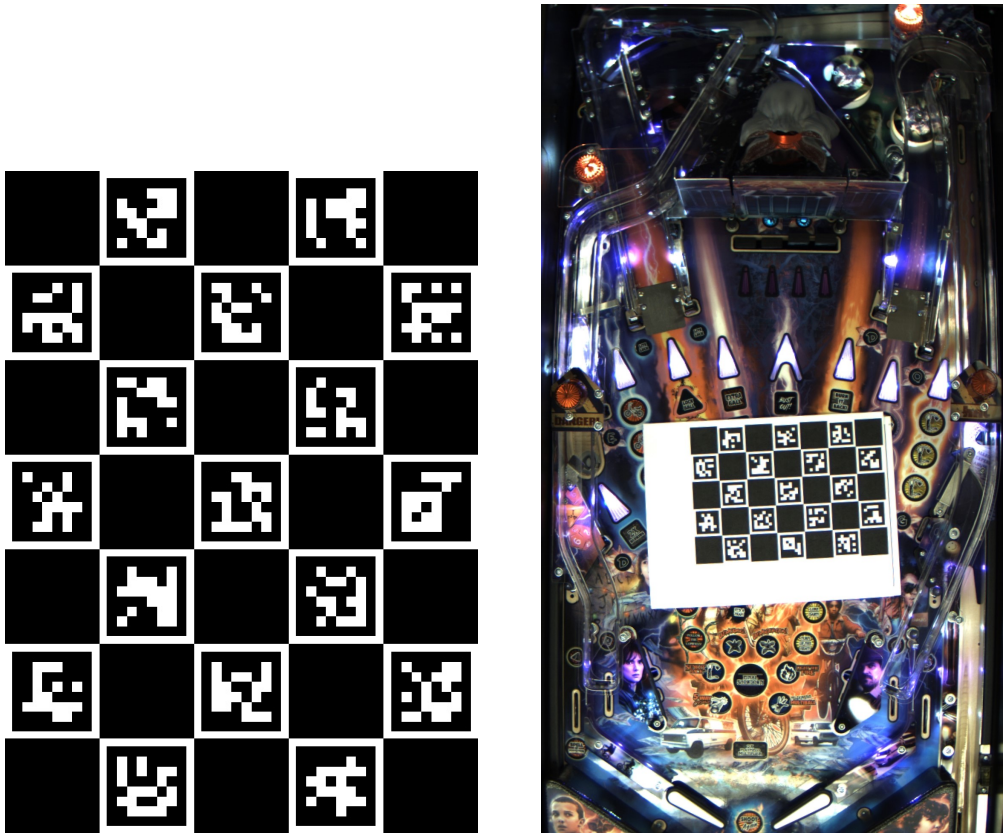
resolution under 1ms of latency according to our measurements. The frame rate of the image stream could be increased by reducing the image resolution. For example, 800x600 images can be streamed at 200 frames per second.

For our application, the camera was configured to capture images in the resolution of 2048x1140, which has a close aspect ratio to that of the pinball playfield. The GPIO port of the camera is configured to send an opto-isolated digital signal containing pulses for each capture trigger of the camera. This pulse train is used in the hardware actuation module to synchronize the frame capture times with the pinball actuation times, which we will provide details under the 'hardware actuator' topic in Section 4.4. Each image captured from the camera is sent with an embedded timestamp from the internal real-time clock. We also configured the camera to allow reading the internal real-time clock data through API requests, which is used by our software.

An 'Edmonds 8.5 mm C-mount' wide-angle lens is mounted to the camera in which the focus distance and the aperture were adjustable. After positioning the camera, the lens is adjusted to focus on the playfield. The iris is kept in the maximum open position, which was f/1.3 aperture, to make best use of the environment light. The camera exposure time was adjusted empirically to a level which is high enough to prevent an under-exposure effect and low enough to prevent a motion blur effect, which can occur due to fast movements of the ball. Having good environmental lighting conditions help this process.

Images captured through a wide-angle lens suffer from the barrel-distortion effect. Camera calibration is the process of evaluating camera geometric model parameters [39]. There are many methods used to correct image distortion through camera calibration.([40, 41, 42, 43, 44] to cite a few) We used OpenCV implementation of Zhang's method to calculate the camera model parameters [44]. Zhang's algorithm requires a set of camera images containing a fixed pattern, captured with different poses and locations of 3D space visible to the view of the camera. From many different patterns that are being used for this purpose such as checker board, Square matrix, Circular dot, ARTag, etc.[45], we went with

a custom generated 'ChArUco' pattern, which is a combination of checker board pattern consisting sharp edges and 'ArUco' markers, which are unique cell patterns, making it less prone to detection errors [46, 47].



(a) The generated 'ChArUco' pattern (b) One of the collected image samples

Figure 4.4: 'ChArUco' pattern for camera calibration

Figure 4.4 shows the 'ChArUco' pattern used in our camera calibration process and how it was used in sample collection. Zhang's algorithm provides us with an estimation of the camera matrix, distortion coefficients and rotation and translation vectors. OpenCV provides us a method to calculate a mapping function to undistort according to the calibration parameters. In our application, we used the generated mapping information in two ways: undistorting the image as a whole and mapping individual pixels to the undistorted space. Figure 4.5 shows, (a) a distorted image , (b) same image corrected and cropped. We will discuss further on it's usage in the next section.

(a) Distorted image (b) Corrected (and cropped) image

Figure 4.5: Example of Distortion correction sample

4.2 Image processing module: ball tracking and feature detection

We use a PC running on Windows 10 as the main computer. We developed our software using Python 3. It contains two components of the framework: *Image processing module* and *control module*. *Image processing module* takes in the images streamed from the camera and extract features of importance, in real-time. For image processing, we use OpenCV built with cuda support. Images streamed over USB 3.0 from the camera are captured using the python libraries provided in the Spinnaker SDK, which is the official SDK of the 'FLIR blackfly' cameras [48].

As discussed earlier, the images we receive from the camera are distorted and need

to be corrected to get accurate feature coordinates. Initially, we went with correcting the whole image as the first step of the image processing pipeline, but the time required for the calculations involved in the process were exceeding the time constrains of real-time operation. Hence, we went with processing the images without correction, and applied the pre-calculated mapping function to calculate the correct positions for feature points at the end. This method allowed us to perform calculations on individual pixels as needed, which was much faster.

In Section 3, we found that motion feature based learning is more effective in sample efficiency. To derive the same set of features, we have to detect the ball position and state of each flipper from the received images. Ball velocity and higher order derivatives can be derived from temporal ball position data. We will first discuss how ball tracking and flipper state tracking were done in a simple custom-built pinball playfield and then in the commercial pinball machine, 'Stranger Things' by Stern Inc.

4.2.1 Ball tracking in a custom built pinball playfield

Figure 4.6: Custom-built simple test-bed

For experimentation, we use a simple custom-built pinball playfield with minimum necessary features as shown in Figure 4.6. It contains two flippers, a ball plunger, and a plain playfield. The flippers were powered with solenoids, which were driven with MOSFETs on a 48V DC supply. Power to the solenoids was controlled according to the state of the flipper. Zero power was applied to keep the flipper in the lowered position, high power was applied to activate the flipper, and when shooting the flipper up and a small amount of power was applied to hold the flipper in up position. This was achieved by

driving the MOSFETs with Pulse Width Modulation (PWM) signals generated by a microcontroller, according to actuation commands and the current state of the flipper. Actuation commands were sent from the PC and the state of the flipper was detected with the limit switches attached to each solenoid. The rest of the experiment setup follows the framework description we presented earlier.

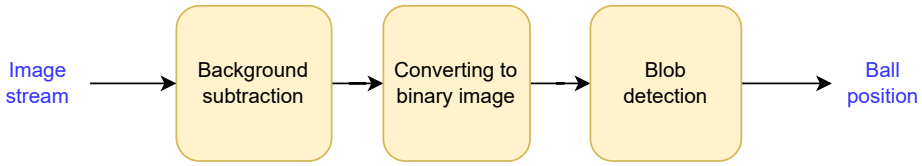


Figure 4.7: Ball tracking methodology for the simple test-bed

Since the playfield is plain and static, we used an image of the playfield without the ball as the background. Figure 4.7 shows the flow of operations we follow to extract the ball position. First, the background was subtracted from each image in grayscale color space. The result is then converted into a binary image by clustering the pixels into two groups, black and white by a threshold value. This threshold value was adjusted by experimentation to reduce noise. Finally, the binary image is processed to detect contours using OpenCV implementation of Suzuki's method [49]. After, we filter the detected blobs according to a set range of values on enclosing pixel count and detect the ball. We take the centroid of the blob as the center point of the ball. Flipper state information was available from the limit switches attached to each flipper.

Some preliminary experiments were done in this playfield in collaboration with my colleague, Michael Ikuru. They included profiling the resultant trajectory of the ball according to the shot delay from the cradled position and testing the applicability of Deep-Q-Learning based on states defined with a virtual grid. We will omit the details of those experiments as they were collaborative work. However, conclusions from those experiments helped in shaping the direction of this work.

4.2.2 Ball tracking on commercial pinball machines

Figure 4.8: Stranger Things game playfield

Commercial pinball machines increase the complexity of ball tracking. Figure 4.8 shows the playfield of the 'Stranger Things' pinball machine which we used for our experiment. Generally, the playfield of a commercial pinball machine is a dynamic environment with frequent lighting and color changes due to blinking lights, spot indicators, area lights, etc. Depending on the game design, there are interactive components which have different motions during game-play. For example, there are bouncers which have elastic bands

showing significant shape deflection during a ball bounce, there are stand targets which hide or move as ball touches them. These movements also induce changes in the shadows and regions with glare in the playfield. The pinball ball is coated in silver color with a mirror-like finish, which causes it to reflect different colors and lights of the pinball playfield. And the shadow of the ball changes in size and shape according to the lighting of the region, which the ball is rolling through. Another challenging aspect is the random vibrations in the playfield relative to the fixed camera, caused by the impact of shooting and bouncing of various elements during gameplay.

We experimented with deep learning methods and classical image processing techniques to handle this challenge. Deep learning methods required thousands of labeled ball samples from the playfield to give reasonable tracking accuracy and sample collection process had to be repeated for each playfield we used our framework on. Since our system is built with generality in mind, we went for a solution based on classical image processing techniques which expands upon the method described in section 4.2.1.

The issues discussed above, are also considered as some of the key challenges in the background separation problem in general[50]. Various methods were proposed to solve this problem including histogram analysis [51], filtering based methods [52, 53, 54], clustering methods [55, 56] and statistical methods [57, 58]. For our application, we also had to pick a method that is computationally light so the time required to calculate meets the real-time time constraints. To sample the background in our work, we first capture N images of the playfield each time we initiate a game. Our controller waits until the background sample collection is completed to insert the ball into the playfield. Image samples take the form of a three dimensional matrix,

$$I_i = \{I_i(x_j, y_k, c_n) : 0 < j \geq w, 0 < k \geq h, n \in \{R, G, B\}\}, \tag{4.1}$$

where (x, y) are pixel coordinates, w is the width of the image, h is height of the image, and c resembles intensity values in the color channels. Initially, we used an averaging method, where we calculate a static background image, I_b by taking the average of pixels in each color channel.

$$I_b = \frac{1}{N} \sum_{i=1}^{N} I_i(x_j, y_k, c_n). \tag{4.2}$$

Then we extracted the foreground pixels by subtracting X_b from the images and thresholding to remove noise. This method delivered decent results, but had issues in regions of the playfield where dynamic color changes occurred.

We experimented with the methods in the literature to overcome the problem with the averaging method. The determined solution is a simplified version of Gaussian Mixture Models (GMM) based background subtraction introduced by Stauffer et al. [58]. Stauffer models intensity values of each pixel of the image as a mixture of Gaussian distributions. For a given pixel X_t, the probability of the pixel belonging to the GMM is

$$P(X_t) = \sum_{k=1}^{K} \omega_k * \eta(X_t, \mu_k, \Sigma_k), \tag{4.3}$$

where K is the number of Gaussian distributions, ω_k, μ_k, Σ_k are the weight, mean and the covariance matrix of the k^{th} Gaussian respectively, and η is the probability distribution function. For simplicity, we assume the color channels are independent from each other and $K = 2$. We used the SciPy implementation of the Expectation Maximization (EM) algorithm to estimate the parameters of the Gaussian distributions for each pixel using the N background images. Stauffer picks a set number of Gaussian distributions with highest ω/ρ ratio where ω and ρ are the weight and standard-deviation of a given distribution. From the two Gaussian distributions trained, we pick the one with higher ω/ρ ratio as

the background distribution, since that would capture the background information and the wider, shorter distribution would correspond to the noise.

We obtain the Gaussian models corresponding to the background pixels during the initialization phase of our tracking system. We set $N = 100$, which is the number of background image samples. Then the obtained Gaussian models are used to extract foreground from the images coming from the stream. If a pixel intensity values are 2.5 times away from standard deviation of the Gaussian, it is taken as a foreground pixel. We did not update the distributions in middle of the games so that the regular processing tasks were not interrupted. As a future development of the framework, an online background modeling scheme can be adapted.

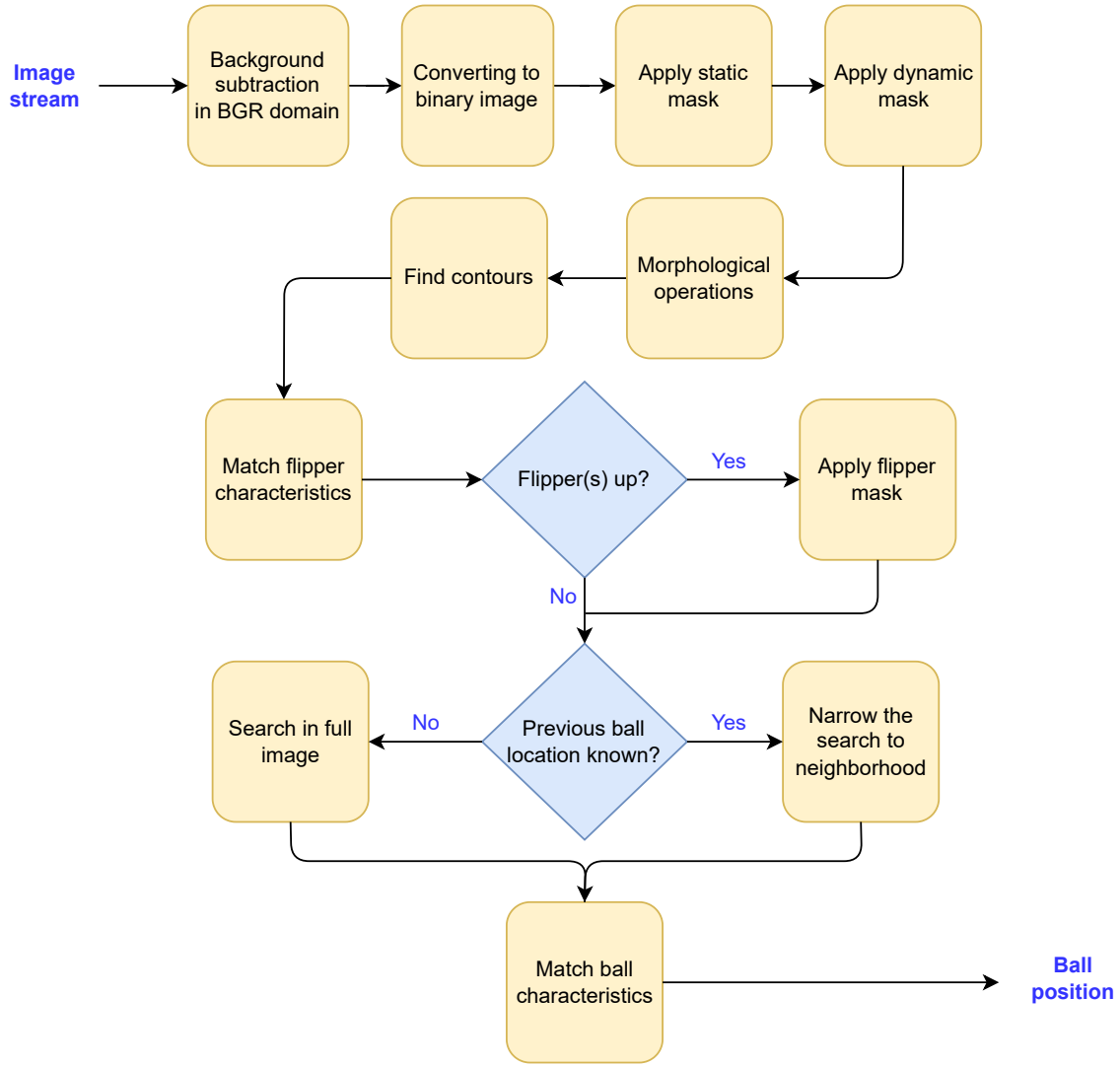


Figure 4.9: Image processing pipeline

Figure 4.9 shows the steps followed in image processing. We will describe each step in brief.

1. Image received from the image stream undergoes the background subtraction in the method described above. Foreground pixel data is sent to the next step.

2. Converts the image to black and white, where black corresponds to the background pixels and white corresponds to the foreground pixels. Threshold value separating black and white is adjusted to reduce noisy pixels.

3. To avoid any noise from the non-playfield regions in the image, a fixed mask is applied to filter them out.

4. The dynamic mask is applied. Dynamic mask is created online in which masked regions are generated according to the following rules.

   (a) If there are additional foreground objects detected in an image after identifying the ball (and flippers if applicable), those regions are added to the mask.

   (b) If the ball is not found in the image and there are matching foreground objects masked out by dynamic mask, that region is removed from the mask.

   This dynamic mask is updated at the end of processing pipeline of each frame.

5. In this step, we apply two morphological operations on the image: dilation followed by erosion with a same sized kernel. This step, further removes any noisy pixels and combine clusters of close-by blobs into a single blob. The effect of this operation is controlled by adjusting the kernel size.

6. We find contours in the binary image by making use of OpenCV implementation of Suzuki's method [49].

7. Next, the found contours are compared with the predefined set of properties (includes a range of area, bounding rectangle dimension range) to find either left flipper or right flipper (or both) are up.

8. If either of the flippers are up, a corresponding mask is applied on the image to cover up the flippers. This is done in order to identify the ball in scenarios where the ball is either touching the flipper (eg: when ball is starting from the cradled position) or moving near the shadows of flippers.

9. Then we change the search region for the ball according to following criteria.

(a) If the ball location is known from the previous frame, narrow down the search region to a square, where the center of the square is the position of the ball in the previous frame.

(b) Else search on the entire image.

10. Final step is to compare the contours in the image with the predefined properties of the ball to identify the blob corresponding to the ball. The centroid of the blob is considered as the center of the ball.

Detected ball position as well as flipper states are passed on to the next stages of the framework. In Graphical User Interface (GUI) mode, the detected ball position and the flipper states are displayed graphically. In real-time operation in higher frame rates, GUI mode is disabled to meet the processing time restrictions. This image-processing pipeline completely operates on the Central Processing Unit (CPU). As an alternative approach we implemented a pipeline with all image processing tasks run on Graphics Processing Unit (GPU) to leverage the power of cuda processors[59]. Though the image processing steps took less time on the GPU than the CPU, the overhead time of transferring image matrices to the GPU, and vice versa, increased the overall processing time. Hence we chose CPU implementation to meet the time constraints of our application.

4.2.3 Kalman filter to filter positional data

The ball position data sent from the image processing pipeline is noisy. Some of the factors that contributes to the noise are,

- Tracking method is sensitive to the variation of ball shadow. It can not differentiate the shadow from the reflective ball.
- Vibrations of the playfield due to large impact forces generated by it's components.
- Occasional errors in the image processing module.

The effect of noise is greatly visible in the derivatives of the ball position data as differentiation amplifies noise. A adequate filtering technique should be able to effectively reduce the effect of noise to improve the accuracy of the data. To implement a filter, we approximated the ball motion with a constant acceleration model in both the X and Y directions. In this approximation, we neglect the effect of friction, air resistance, and consider only the gravitational acceleration component acting on the ball due to the angle of the playfield. This model is not valid for instances where the ball hits another object (i.e.: flipper, bouncer, etc) Hence, we use this filter on known portions of the game-play which does not violate these assumptions. These portions include the ball falling or moving up freely in the playfield, rolling down from a straight ramp or a ball guide.

Under those approximations, the ball position (x, y), at time t, can be modeled as

$$x\left(t; x_0, \dot{x}_0, a_x\right) = \frac{a_x}{2}t^2 + \dot{x}_0 t + x_0 \tag{4.4}$$

$$y\left(t; y_0, \dot{y}_0, a_y\right) = \frac{a_y}{2}t^2 + \dot{y}_0 t + y_0 \tag{4.5}$$

$$\dot{x}\left(t; \dot{x}_0, a_x\right) = a_x t + \dot{x}_0 \tag{4.6}$$

$$\dot{y}\left(t; \dot{y}_0, a_y\right) = a_y t + \dot{y}_0. \tag{4.7}$$

where, (a_x, a_y) are the acceleration components due to net forces on the ball in x and y directions. The acceleration components can be assumed to be constant as we selectively apply this model only on regions where the net force is constant. That include the scenarios where the ball is rolling freely down a ramp, moving up after being shot, etc. Considering only first and second order derivatives of the position, we define the ball motion parameters as $\mathbf{m} := [x, \dot{x}, \ddot{x}, y, \dot{y}, \ddot{y}]^\top$ and the ball position measurements as $\mathbf{b} := [x, y]^\top$. We base the Kalman filter on the following dynamic model.

$$\hat{\mathbf{m}}_t = \mathbf{F}\hat{\mathbf{m}}_{t-1} + \mathbf{G}\mathbf{u}_t + \mathbf{w}_{t-1} \tag{4.8}$$

$$\mathbf{b}_t = \mathbf{H}\mathbf{m}_t + \mathbf{v}_t, \tag{4.9}$$

where $\mathbf{F}$ is the state transition matrix, $\mathbf{G}$ is the control matrix, $\mathbf{w}$ is the process noise, $\mathbf{H}$ is the observation model and $\mathbf{v}$ is the measurement noise.

For regions where the constant acceleration model is applicable, we do not have a control input. Hence, $\mathbf{u} = 0$. We assume $\mathbf{w} \sim \mathcal{N}(0, \mathbf{Q})$, a zero mean Gaussian noise with time invariant covariance $\mathbf{Q}$. We also assume $\mathbf{v} \sim \mathcal{N}(0, \mathbf{R})$, a zero mean Gaussian noise with time invariant covariance $\mathbf{R}$.

Using (4.4)-(4.7), we can derive the following state transition relationship and state transition matrix $\mathbf{F}$:

$$\underbrace{\begin{bmatrix} \hat{x}_t \\ \hat{\dot{x}}_t \\ \hat{\ddot{x}}_t \\ \hat{y}_t \\ \hat{\dot{y}}_t \\ \hat{\ddot{y}}_t \end{bmatrix}}_{\hat{\mathbf{m}}_t} = \underbrace{\begin{bmatrix} 1 & \Delta t & 0.5\Delta t^2 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & 0.5\Delta t^2 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{F}} \underbrace{\begin{bmatrix} \hat{x}_{t-1} \\ \hat{\dot{x}}_{t-1} \\ \hat{\ddot{x}}_{t-1} \\ \hat{y}_{t-1} \\ \hat{\dot{y}}_{t-1} \\ \hat{\ddot{y}}_{t-1} \end{bmatrix}}_{\hat{\mathbf{m}}_{t-1}} \tag{4.10}$$

Using the definitions of $\mathbf{m}$ and $\mathbf{b}$, $\mathbf{H}$ can be derived:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4.11}$$

To account for the inaccuracies of the model imposed due to our assumption of constant acceleration, we presume there is a random variation in the acceleration components with a constant variance σ_a^2. We derive process noise covariance, $\mathbf{Q}$:

$$\mathbf{Q} = \mathbf{F}\mathbf{Q}_a\mathbf{F}^{\mathbf{T}}$$

$$\text{where } \mathbf{Q}_a = \sigma_a^2 \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{thus, } \mathbf{Q} = \sigma_a^2 \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} & 0 & 0 & 0 \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t & 0 & 0 & 0 \\ \frac{\Delta t^2}{2} & \Delta t & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ 0 & 0 & 0 & \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ 0 & 0 & 0 & \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \tag{4.12}$$

For simplicity, we assume the measurements x and y are uncorrelated and they both have a time invariant random uncertainty variance of $\sigma_{\mathbf{b}}^2$. Hence $\mathbf{R}$ is,

$$\mathbf{R} = \begin{bmatrix} \sigma_{\mathbf{b}}^2 & 0 \\ 0 & \sigma_{\mathbf{b}}^2 \end{bmatrix}. \tag{4.13}$$

Using these derivations for $\mathbf{F}$,$\mathbf{H}$, $\mathbf{Q}$ and $\mathbf{R}$, we implemented the filter based on standard prediction equations and update equations of Kalman filter [60, 61, 62]. The process noise variance, σ_a^2, and measurement uncertainty variance, $\sigma_{\mathbf{b}}^2$, were tuned empirically.

4.3 Control module

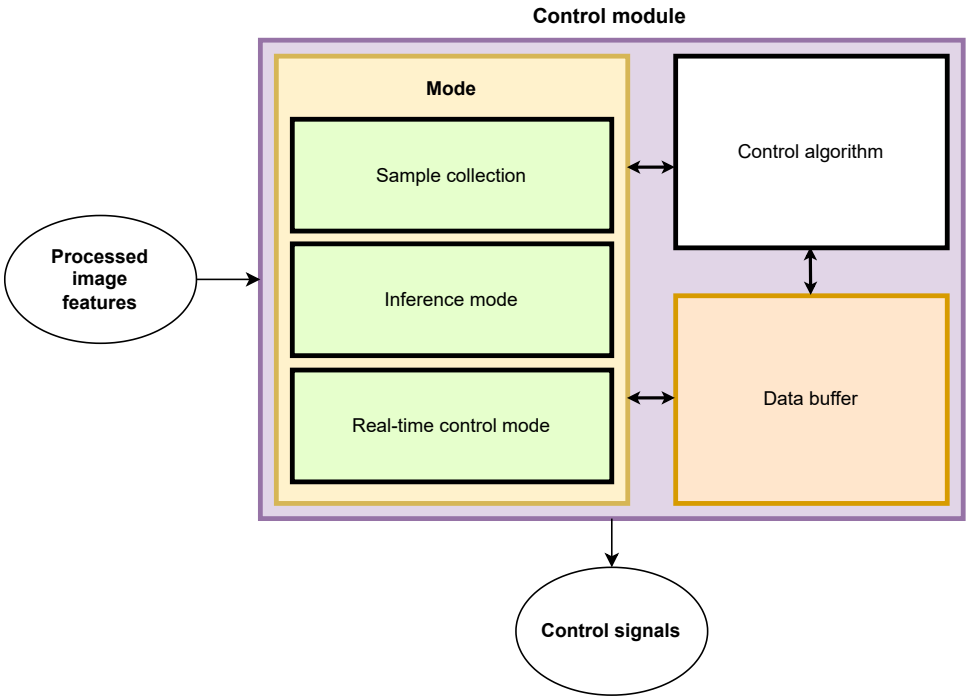


Figure 4.10: Components of control module

The control module sits in the PC next to the image processing module (Section 4.2) in the diagram and handles data processing and decision making. As shown in Figure 4.10, the control module receives feature information from the image processing module and generates control signals as outputs. The main three sub modules inside the control module

are: mode selector, control algorithm and data buffer.

- **Mode selector:** consists of configurations for three different modes: sample collection mode, inference mode and real-time control mode.
  - Sample collection mode: This generates control signals to make random shots or make shots according to pre-configured patterns or conditions, and collects the shot related data (timestamp, extracted features and corresponding images). Collected data is sent to data buffer to save accordingly.
  - Inference mode: This is used to do a dry run of the control algorithms on either real-time data stream or previously saved shot data. This mode is used to analyze the output of the controller without exhausting the real machine where applicable. This mode does not require adhering to real-time constraints.
  - Real-time control mode: In this configuration, data stream from image processing module is used to generate decisions from the control algorithm module and control signals are sent to actuate them. Similar to sample collection mode, shot data is sent to the data buffer to be used as training data for online-training algorithms or for evaluating the accuracy of the controller later.
- **Control algorithm**: different control algorithms can be implemented inside this module. It has essential handles for training and inferencing. It is also possible to plug in externally trained models to this module. The control algorithm module has access to real-time data or saved data for training. Decisions made in inference mode are sent out as control signals for actuation.
- **Data buffer**: acts as an intermediate data storage until data is written to hard drive or used by other modules. When another module sends data to be saved to disk, the data buffer keeps them in memory until time-critical operations are completed and initiates the time consuming data write operation. The data buffer also retains historical

data for the use of online-learning agents, control algorithms and GUI operations for features like displaying shot replays. Use of data-buffer is a critical step in real-time operation of the system.

4.4 Hardware actuator

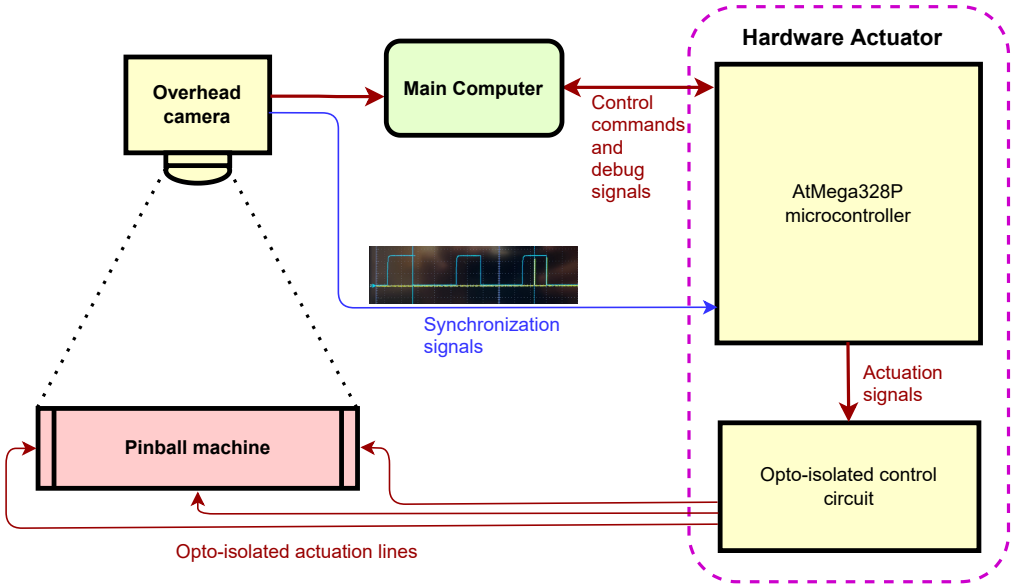


Figure 4.11: Hardware Actuator Module in the the system diagram

The final component of the realworld framework is the hardware actuator. It has the responsibility of controlling the pinball buttons according to control signals received from the computer while being precise on timing. The two main components in this module is AtMega328P microcontroller and the opto-isolated control circuit.

- **AtMega328P microcontroller**: This is connected with the main computer with a serial link built using USART interface. Microcontroller receives synchronization pulse train from the overhead camera in which each pulse starts at frame capture start event. A hardware interrupt is configured to capture the start of each pulse, which is used to synchronize the micro-controller time with the frame capture events. A

hardware timer is configured to count microseconds from the event of interrupt. This allows the controller to send out actuation signals, either in-sync with the next image capture time or with a customizable delay from that moment accurate to microseconds in a non-blocking manner. The actuation signals for left flipper, right flipper and the ball shoot/game start is sent out to the opto-isolated control circuit.

- **Opto-isolated control circuit**: This is used to electrically isolate our system from the pinball machine. Incoming signals are relayed to the output via individual opto-isolating transistors in which outputs are connected to each pinball machine button terminals via removable clips.

Real-world experiments: Model based learning

The aim of the experiments done in this section is to explore the methods of training a sample-efficient agent to play a real-world pinball machine. We utilize the real-world framework we developed in Section 4 for the experiments. Based on results we got from simulation based experiments in Section 3, we know extracting features can reduce the number of training-samples in comparison to image pixel based learning method. Hence, we use extracted features as the input. Even with motion feature based inputs, we see that model-free methods required thousands of samples to train in the simulator experiments. That number is still infeasible with a real-world machine.

We experimented on transferring neural network models trained on the simulation framework to real-world machines via transfer learning techniques, but that was not able to produce significant improvement on the results. We believe the number of training samples feasible on the real-world machine might not be sufficient for transfer learning agents to adapt. The differences in perception noise, sampling rate and actuation delays can be contributing factors. Data efficient transfer learning is a highly relevant and interesting future research direction for this project.

We moved on to using a model-based approach. We trained our agents to act as Model Predictive Controller (MPC) in the following experiments. We will describe the pinball model, then the ML techniques used to train agents, and finally, the evaluation results.

5.1 Modeling the game of pinball

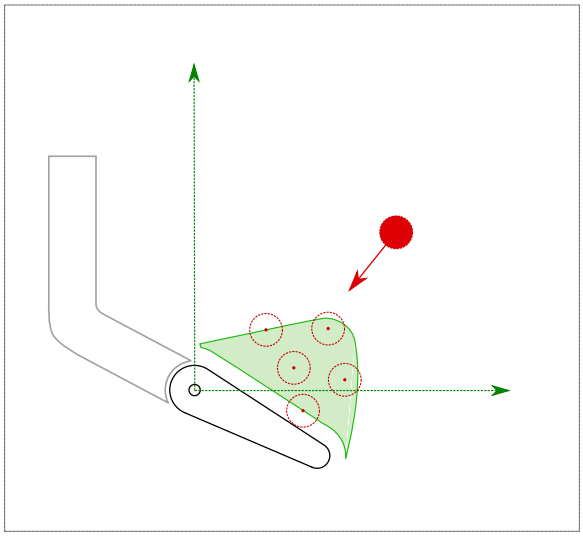


Figure 5.1: *Control Region*: Area in which the flipper can change the ball dynamics

Figure 5.1 shows the *Control Region* in green. It is the area in which the player is able to hit the ball and impact the game-flow. When the ball is outside this region, the game is completely governed by the momentum of the ball, reaction of the ball to other game components, governing forces over the elements, and noise. The challenge for the player is to act when the ball is within the *control region* and re-direct the ball to desired paths.

We use a Cartesian coordinate system, (x, y) with the origin located in the pivot point of the flipper as shown in the Figure 5.1. The ball position data provided by the image processing module is transformed to this Cartesian coordinate system. We define the state of the system as a vector concatenating position and velocity vectors, $\mathbf{z} := (x, y, \dot{x}, \dot{y})$. The trained agent has to precisely time shots by anticipating the motion of the ball. To reduce the complexity of the problem, we consider a common scenario of the game where the ball rolls down from a ramp and guided towards the flipper from the side, namely 'Rolling Shot Scenario' (RSS), which we previously introduced in [22]. In the RSS, the player is able to plan well and make a shot to almost all accessible areas of the playfield from the chosen flipper. Hence, pinball players often uses other flipper techniques to move the ball into the RSS and make a calculated shot from there.

5.1.1 Rolling Shot Scenario

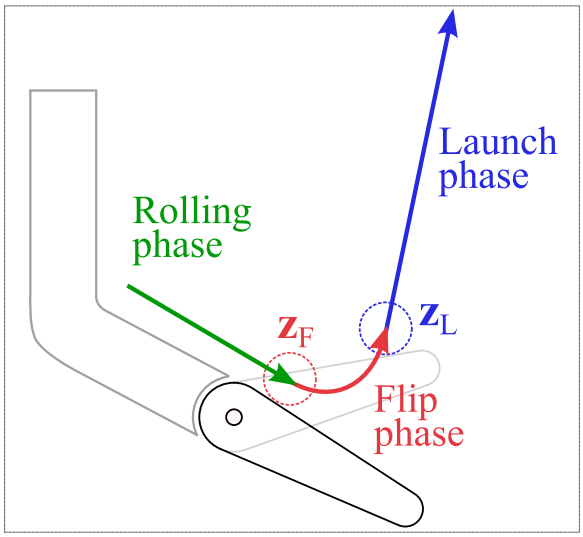


Figure 5.2: Rolling Shot Scenario (RSS).

Figure 5.2 shows the three phases of RSS: rolling phase, flip phase and launch phase. $\mathbf{z}_F$ is the starting state of flip phase and $\mathbf{z}_L$ is the starting state of the launch phase. In the RSS, the ball rolls down from the side ball guide of the flipper and at a moment when the ball is on top of the flipper, the flipper is activated to shoot the ball up. The system moves to the flip phase at the moment the flipper is activated. The flip phase exists until the ball is being pushed up by the flipper and maintains surface contact. At the moment the ball looses surface contact with the flipper, the system moves into the launch phase. The launch phase lasts until the ball hits another object in the playfield.

We define t_F as the moment in time the flipper activates and the corresponding state as $\mathbf{z}_F := \mathbf{z}(t_F) = (x_F, y_F, \dot{x}_F, \dot{x}_F)$. Similarly, we define t_L as the moment in time the system enters the launch phase and corresponding state as $\mathbf{z}_L := \mathbf{z}(t_L) = (x_L, y_L, \dot{x}_L, \dot{y}_L)$. The player changes the phase of the system from the rolling phase to the flip phase by activating the flipper by pressing the corresponding flipper activation button, which is located in the left and right sides of the pinball machine by default. Next, we will discuss the motion dynamics of each phase of the RSS.

- **Rolling phase dynamics**

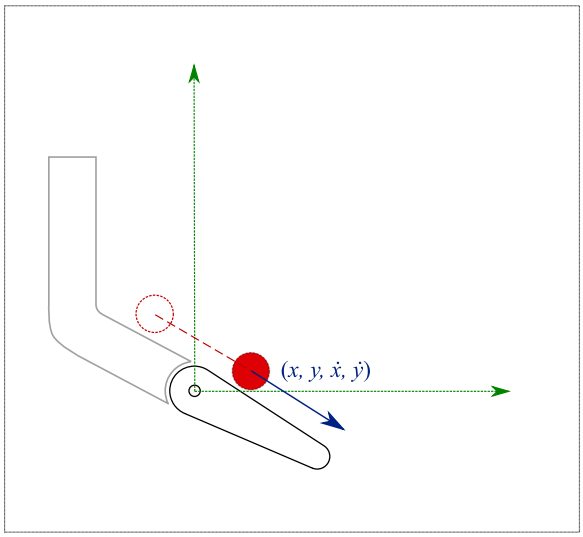


Figure 5.3: Ball motion in rolling phase.

Figure 5.3 shows an arbitrary system state in the rolling phase. Net forces acting on the ball consists of a frictional component and gravitational component. They are governed by the pinball playfield inclination, ball guide angle, and the resting angle of the flipper. The gravitational component is significantly larger than the frictional component and assuming friction to be negligible,we can formulate the following dynamic model for any given time in the rolling phase.

$$x\left(t; x_0, \dot{x}_0, a_x\right) = \frac{a_x}{2}t^2 + \dot{x}_0 t + x_0 \tag{5.1}$$

$$y\left(t; y_0, \dot{y}_0, a_y\right) = \frac{a_y}{2}t^2 + \dot{y}_0 t + y_0 \tag{5.2}$$

$$\dot{x}\left(t; \dot{x}_0, a_x\right) = a_x t + \dot{x}_0 \tag{5.3}$$

$$\dot{y}\left(t; \dot{y}_0, a_y\right) = a_y t + \dot{y}_0. \tag{5.4}$$

Here, (x_0, y_0) is an arbitrary initial starting position of the ball at $t = t_0$ and a_x, a_y are the acceleration components acting on the ball. Since we only consider gravitational components on the ball, we can presume (a_x, a_y) to be constant.

- **Launch phase dynamics**

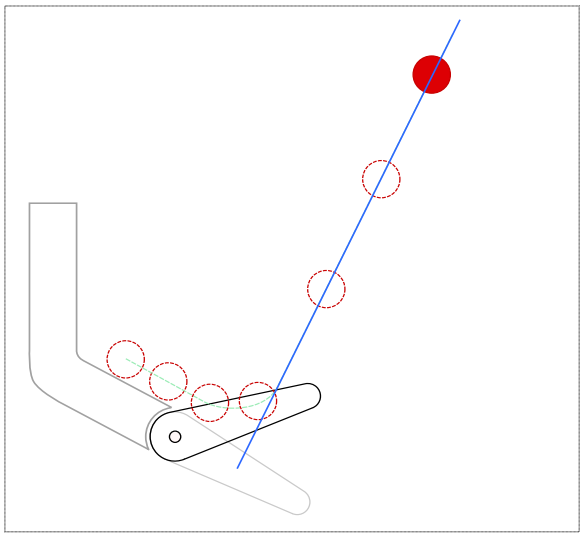


Figure 5.4: The launch trajectory

In the launch phase, the ball has the most momentum when entering this phase, which is state $\mathbf{z}_L$. By analyzing some shot samples empirically, we can observe that the magnitude of the ball velocity in the y direction $\dot{y}_L$ renders the effect of gravitational component in y direction insignificant. Also, as the playfield is leveled from left to right in general, and neglecting frictional components, we can assume the ball is moving with a constant velocity $\dot{x}_L$ in the x direction. Hence, the launch phase can be approximated with a straight line, $g(x, y)$ with a constant velocity model as shown in Figure 5.4.

For the the initial ball position in the launch phase $\mathbf{z}_L := (x_L, y_L, \dot{x}_L, \dot{y}_L)$ the launch trajectory can be parameterized as

$$\begin{aligned} g(x,y) &= x - \frac{\dot{x}_L}{\dot{y}_L} y + \dot{x}_L \frac{y_L}{\dot{y}_L} - x_L = 0 \\ &= x - \beta_1 y - \beta_0, \end{aligned} \tag{5.5}$$

where $\beta_1 := \frac{\dot{x}_L}{\dot{y}_L}$ and $\beta_0 := -\dot{x}_L \frac{y_L}{\dot{y}_L} + x_L$ are the launch trajectory parameters.

- **Flip phase dynamics**

  Modeling the flip phase dynamics are not essential at this point. However, for a given $\mathbf{z}_F$, the flipping dynamics have to be deterministic given the external conditions do not change. We can use that to build a deterministic relation between $\mathbf{z}_F$ and $\mathbf{z}_L$. Thus the launch trajectory parameters β_1 and β_0, which only depend upon $\mathbf{z}_L$, relates to $\mathbf{z}_F$ deterministically. We can say there are functions,

$$\beta_1 = \hat{f}_{\beta_1}(\mathbf{z}_F) \tag{5.6}$$

$$\beta_0 = \hat{f}_{\beta_0}(\mathbf{z}_F) \tag{5.7}$$

  where $\hat{f}_{\beta_1}$ and $\hat{f}_{\beta_0}$ are mapping functions. A human player develops intuitive knowledge on those mapping functions whereas artificial agents have to learn them via machine learning. Ideally, if we have accurate information on pinball machine dynamics, we should be able to mathematically derive those functions [63]. However, that might not be practical as we would have to account for slightest changes on the system to be precise. For example, machine wear and tear will change the system dynamics. Using a learning based method will allow the system to adapt to the system with less effort.

In the MPC approach, we train the agents with machine learning techniques to learn $\hat{f}_{\beta_1}$ and $\hat{f}_{\beta_0}$ functions. Next we will discuss how the training data was collected.

5.1.2 Data collection

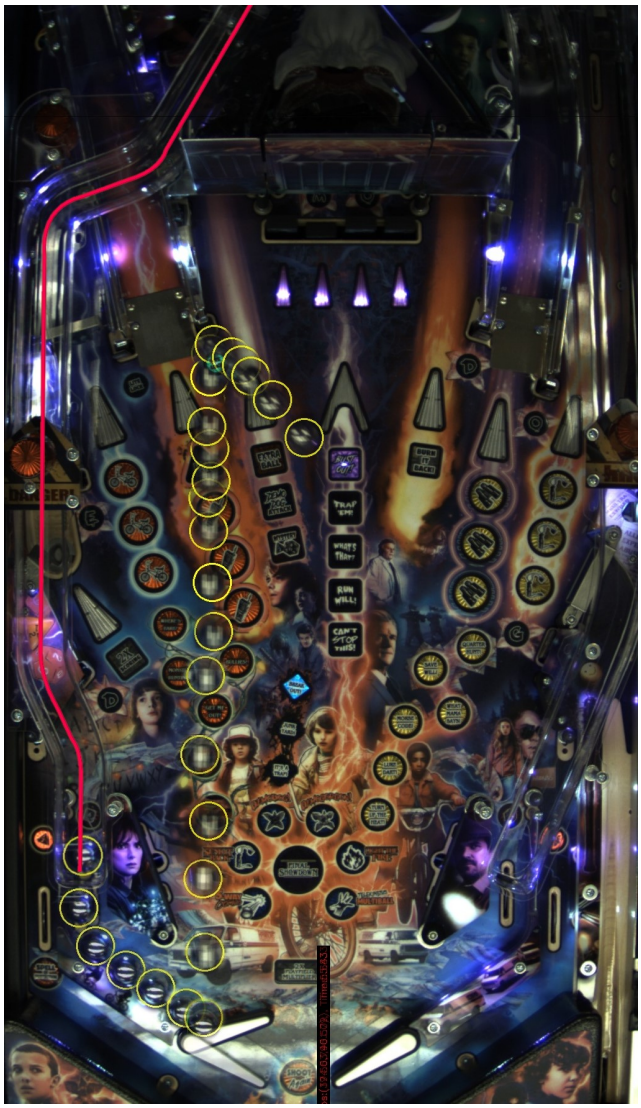


Figure 5.5: Tracked ball positions in a sample RSS shot

As mentioned before, we use a 'Stranger Things' commercial pinball machine made by Stern Pinball Inc. for the experiment. Our framework was configured to detect when a ball was rolling down the left ramp representing a start to the RSS. Figure 5.5 shows the tracked ball positions in a sample shot in yellow circles. The shape of the left side ramp is shown in red. Once the ball has rolled down the left ramp, the sample collection module of the framework starts recording the ball positions. When the ball is rolling over the flipper, a shot is made by the framework at a desired state to move the ball through flip phase into the

launch phase. The launch trajectory is recorded until the ball changes it's moving direction after hitting another object.

In a single data collection run, where we drop the ball from a roughly fixed location in the ramp, the data collection module takes in the number of trials we want to run and then systematically collects shot samples by sweeping the region of possible shots with the goal of creating an even distribution of the samples. This method was repeated for 3 different ball drop locations in the ramp (upper, middle and lower sections) and with the ball cradled in the left flipper (i.e. flipper is in the up position at the start and the ball is placed on the head of the flipper). For each shot sample, the recorded launch phase ball positions are fitted into a line using Deming regression and the launch trajectory parameters, β_1 and β_0 are calculated[64]. 120 samples were collected as training data and 80 samples were collected as validation data.

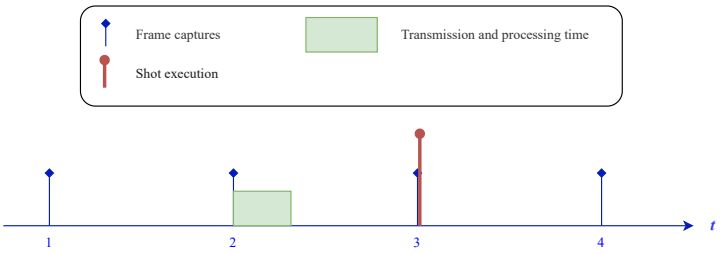


Figure 5.6: A sample timeline of processing a frame and actuating a shot

Since the game of pinball depends on precise timing, both sample collection as well as shot actuation, need sufficient timing accuracy. The processing time for a frame can slightly vary from frame to frame due to operating system conditions. To avoid ambiguity in the timing of the shot samples, we synchronize every shot actuation with the next immediate frame capture event as shown in Figure 5.6. In this way, we fix the time from frame capture to shot actuation, equal to the image capture period, which is 20ms. The same method is used when making shots with the trained models to make timing consistent between training and evaluation.

5.2 Model Predictive Controller

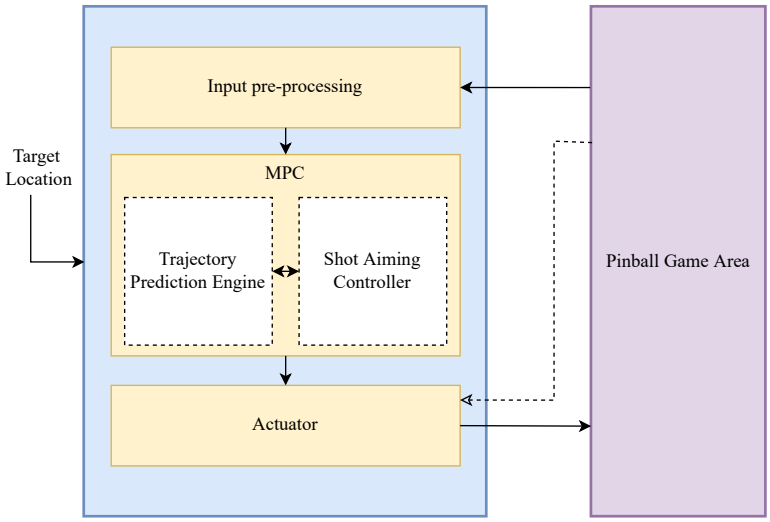


Figure 5.7: Model Predictive Controller Architecture

With use of the mathematical model we derived in section 5.1, we create a Model Predictive Controller (MPC) to aim shots in the game of pinball. Figure 5.7 shows the integrated components of the complete MPC system. Feedback from the pinball game is taken using the camera in the framework and actuation of the game is done using the hardware actuator of the framework. The dashed arrow indicates the synchronization signal coming from the camera. Input pre-processing is done with the image processing module. The components of the MPC: Trajectory Prediction Engine (TPE) and Shot Aiming Controller (SAC) are implemented in the control algorithm module of the framework.

The MPC receives a set target location. TPE has to predict the launch trajectories for any given system state. Utilizing the predicted launch trajectories, the SAC will strategically plan and execute a shot to reach the given target.

5.2.1 Trajectory Prediction Engine

(5.6) and (5.7) show that there exist mapping functions from a given system state, $\mathbf{z}_t$ to β_1 and β_0 as $\hat{f}_{\beta_1}(\mathbf{z}_t)$ and $\hat{f}_{\beta_0}(\mathbf{z}_t)$. We model these mapping functions using machine learning techniques which we describe in detail in the next section. If the models are able to predict the launch trajectories for system state with enough accuracy, those models can be used to aim a shot utilizing the SAC.

5.2.2 Shot Aiming Controller

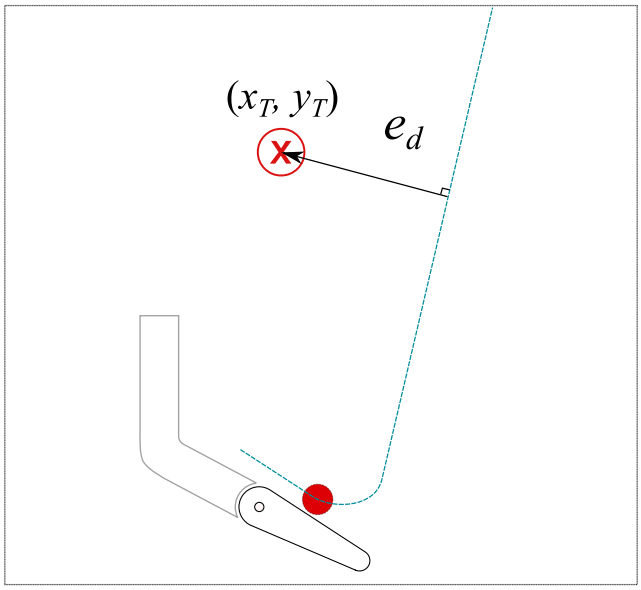


Figure 5.8: Error, e_d for a given ball launch trajectory

Based on the launch trajectory parameter predictions, $\hat{\beta}_1$ and $\hat{\beta}_0$, provided by the TPE functions, $\hat{f}_{\beta_1}(\mathbf{z}_t)$, and $\hat{f}_{\beta_0}(\mathbf{z}_t)$, the launch trajectory can be predicted using (5.5) as

$$\hat{g}(x, y) = x - \hat{\beta}_1 y - \hat{\beta}_0. \tag{5.8}$$

For a given target location coordinates (x_T, y_T), we define an error term, e_d which is the perpendicular signed distance between $\hat{g}(x, y)$ and (x_T, y_T) as shown in Figure 5.8. Using (5.8), e_d can be written as

$$e_d = E_d(x_T, y_T, \hat{\beta}_1, \hat{\beta}_0) = \frac{x_T - \hat{\beta}_1 y_T - \hat{\beta}_0}{\sqrt{\hat{\beta}_1^2 + 1}}. \tag{5.9}$$

When $e_d \approx 0$, the launch trajectory should go through the target location, which should result in a successful target hit. In order to find a future state that will correspond to $e_d \approx 0$, we first have to find the search space containing feasible future states, given the current state of the system.

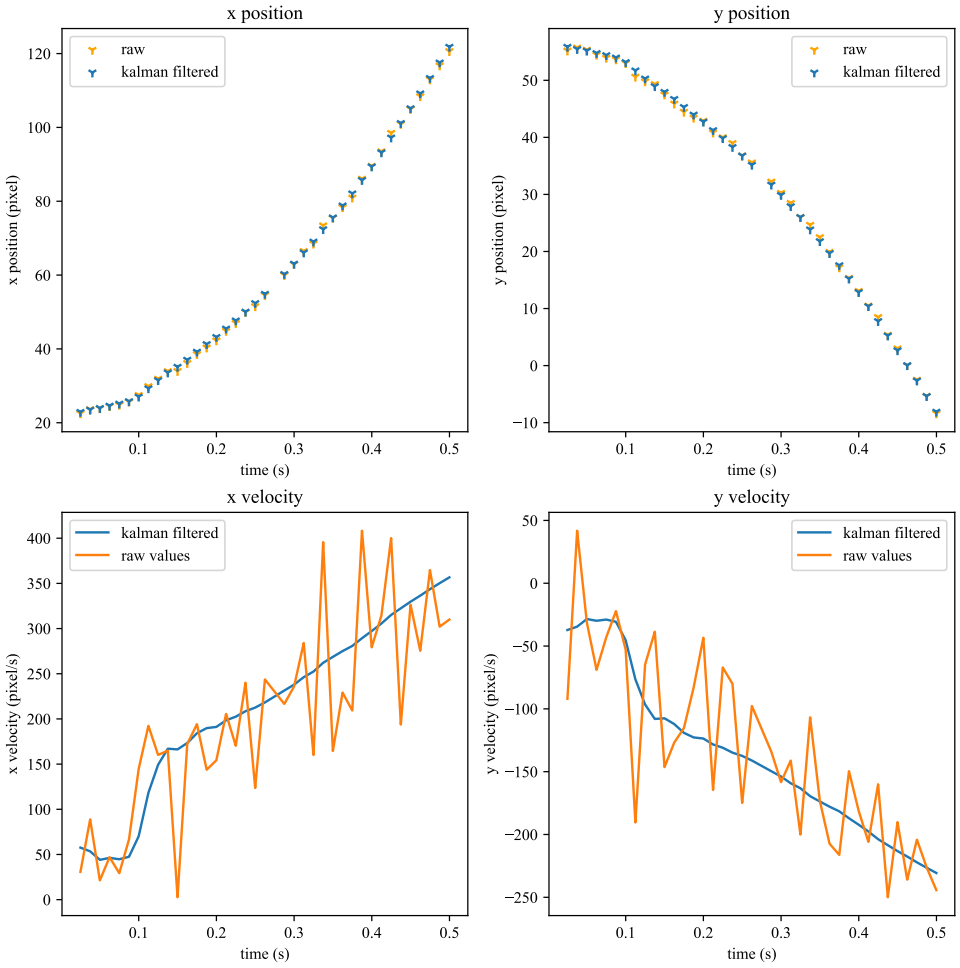


Figure 5.9: Kalman filtering in a rolling phase example

The motion model described with (4.4)-(4.7) under Kalman filtering in Section 4.2.3

is applicable to the rolling phase. Figure 5.9 shows the Kalman filtered values in a rolling phase of an arbitrary RSS sample. Graph shows that the Kalman filter was able to regulate the variations of the velocity with it's estimates. Using Kalman filtered motion parameters at time t, $\hat{\mathbf{m}} = [\hat{x}, \hat{\dot{x}}, \hat{\ddot{x}}, \hat{y}, \hat{\dot{y}}, \hat{\ddot{y}}]^\top$ we can derive the following equations for an arbitrary small time step into the future,

$$\hat{x}_{t+\delta t} = \frac{\hat{\ddot{x}}_t}{2}\delta t^2 + \hat{\dot{x}}_t \delta t + \hat{x}_t \tag{5.10}$$

$$\hat{y}_{t+\delta t} = \frac{\hat{\ddot{y}}_t}{2}\delta t^2 + \hat{\dot{y}}_t \delta t + \hat{y}_t \tag{5.11}$$

$$\hat{\dot{x}}_{t+\delta t} = \hat{\ddot{x}}_t \delta t + \hat{\dot{x}}_t \tag{5.12}$$

$$\hat{\dot{y}}_{t+\delta t} = \hat{\ddot{y}}_t \delta t + \hat{\dot{y}}_t. \tag{5.13}$$

where the predicted system state at time $t + \delta t$ is $\mathbf{z}_{t+\delta t} := [\hat{x}_{t+\delta t}, \hat{y}_{t+\delta t}, \hat{\dot{x}}_{t+\delta t}, \hat{\dot{y}}_{t+\delta t}]^\top$. States satisfying the above equations are the feasible future states for the system, given the current state.

By empirical analysis we can conclude that for a ball rolling down the flipper, the error function $e_d(t)$ is a strictly increasing monotonic function for a given target location. By (5.6) and (5.7) we can derive the following for a given target location (x_T, y_T),

$$\begin{aligned} e_d(t) &= E_d(x_T, y_T, \beta_1, \beta_0) \\ &= E_d(x_T, y_T, \hat{f}_{\beta_1}(\mathbf{z}_t), \hat{f}_{\beta_0}(\mathbf{z}_t)). \end{aligned} \tag{5.14}$$

Hence the predicted error at time $t + \delta t$, $\hat{e}_d(t + \delta t)$ is

$$\hat{e}_d(t + \delta t) = E_d(x_T, y_T, \hat{f}_{\beta_1}(\hat{\mathbf{z}}_{t+\delta t}), \hat{f}_{\beta_0}(\hat{\mathbf{z}}_{t+\delta t})). \tag{5.15}$$

Since it is not always feasible to find an inverse of the machine learning models which we use as $\hat{f}_{\beta 1}$ and $\hat{f}_{\beta 0}$, we suggest two alternative methods to find the delay δt^*, where $e_d(t + \delta t^*) \approx 0$:

1. **Iterative solver:** We can solve for the root of the equation (5.15) using numerical methods. Specifically, we use Secant method to solve for the root. This numerical solver has to run in a serial manner.

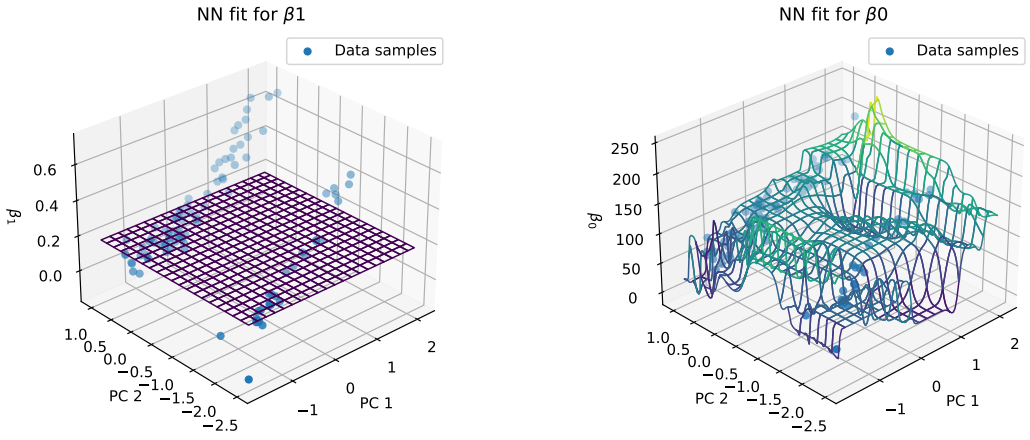
2. **Sampling solver:** This is an approximation method where the next frame period is subdivided into finite amount of samples and corresponding state and error are calculated to find δt^*, which is closest to the root. These calculations can be done in parallel for all time samples, which can save the effective execution time.

After finding the delay time δt^*, an accurately timed shot has to be made in order to shoot the ball to the target.

5.3 Neural Network based MPC

In this experiment we explore the use of a multi layered neural network to predict launch phase parameters from the state of the system as approximations to $\hat{f}_{\beta_1}(\mathbf{z}_t)$ and $\hat{f}_{\beta_0}(\mathbf{z}_t)$. We train two neural networks as the prediction functions, $\hat{f}_{\beta_1}(\mathbf{z}_t)$ and $\hat{f}_{\beta_0}(\mathbf{z}_t)$.

5.3.1 Hyperparameter tuning

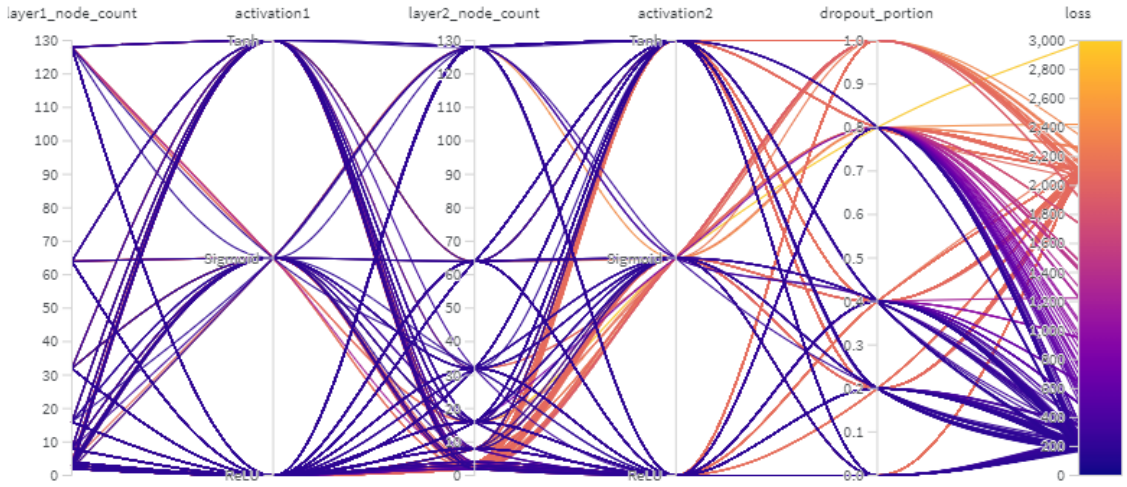


(a) Possible under-fit

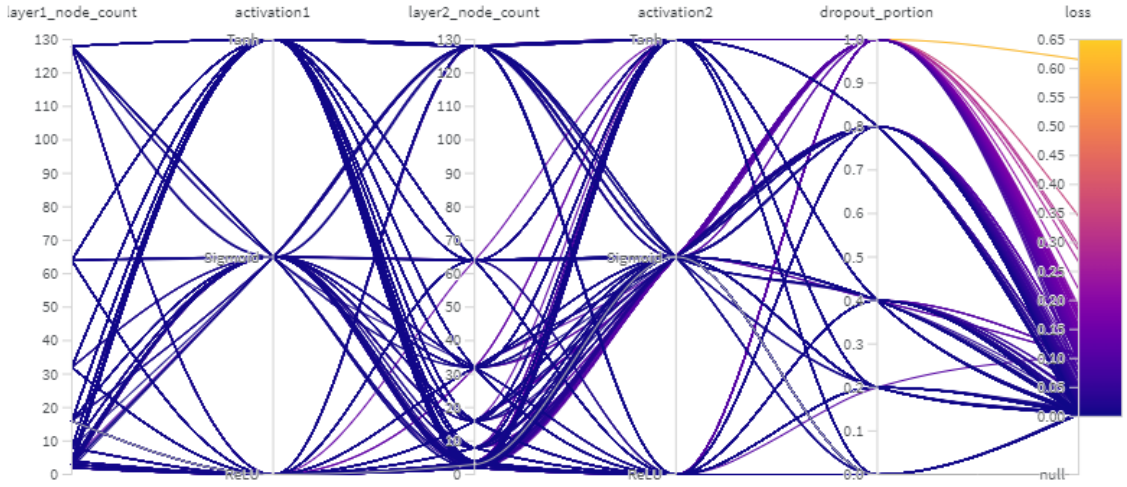
(b) Possible over-fit

Figure 5.10: Examples for possible under-fit and over-fit models

To come up with a good neural network structure, number of hidden layers, number of nodes per layer, activation functions, dropout rate, etc. had to be optimized. First we decided the number of layers empirically by visualizing the generated models. A set of models were trained with differing number of hidden layers with 20 models per each layer count. Since our state space has four dimensions which is not convenient to visualize, we checked the possibility of dimension reduction. With Principal Component Analysis on the sample data we found two principal components which were able to capture more than 90% of the variance of data which we name as 'PC1' and 'PC2'. We use these two principal components to visualize our data and the models. We visualized the validation data and the trained models to identify under-fit and over-fit cases as in the examples shown in Figure 5.10 by observation. By eliminating the model groups with most observable under-fit and over-fit cases, we decided to go with neural network models with 2 hidden layers.



(a) $\hat{f}_{\beta_0}$ training

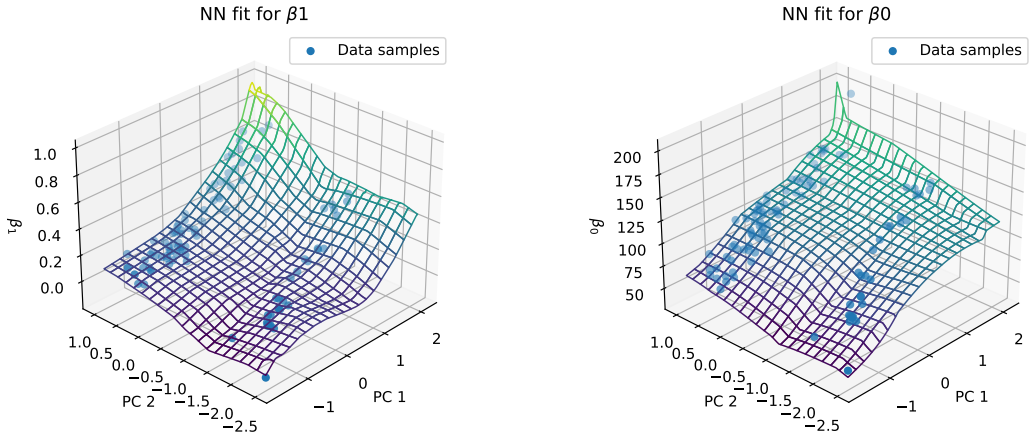


(b) $\hat{f}_{\beta_1}$ training

Figure 5.11: Model training loss

To tune the hyperparameters: the hidden layer 1 node count, the hidden layer 1 activation function, the hidden layer 2 node count, the hidden layer 2 activation function and the dropout percentage, we used a Baysian Optimization (BO) based method [65, 66]. A finite search space is built by assigning a set of discrete values to each hyperparameter. BO

was applied over parallel model training sessions. Figure 5.11 show a summary view of the model training results. In addition to this, we visually observed the distribution of models in the final stages of BO using the principal components as we did earlier to rule out over-fit models.



(a) β_1 (b) β_0

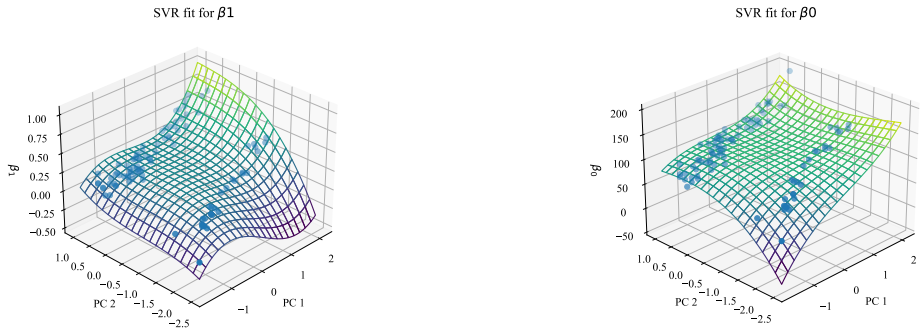
Figure 5.12: Visualization of the Neural Network model picked for predicting the launch trajectory

Figure 5.12 shows the distribution of the selected models at the end of the hyper-parameter tuning process. Final models consisted of 16 neuron hidden layer with tanh activation and another 128 neuron hidden layer with ReLu activation function. Optimal dropout percentage was 0% and 20% for $\hat{f}_{\beta_0}$ and $\hat{f}_{\beta_1}$ respectively, from the training results.

5.4 Support Vector Regression based MPC

This is another approach we took on the MPC. As the use of Support Vector Regression (SVR) also provided promising results, we will include the details here. SVR is a generalized version of Support Vector Machines; hence the model provides a continues-valued output [67]. The choice of SVR as a modeling method was also driven by the fact that it is able complete training in a few seconds. A short training time is a useful property for an online learning agent on the system. For the scope of this work, we will see how

SVR trains an agent with the collected sample data in an offline manner.



(a) β_1 (b) β_0

Figure 5.13: Visualization of SVR models

Using the same training dataset, SVR models were trained as $\hat{f}_{\beta_1}(\mathbf{z}_t)$ and $\hat{f}_{\beta_0}(\mathbf{z}_t)$ prediction functions. Trained models were visualized using principal components of datasamples. By testing with a set of different kernels: linear, polynomial, radial basis function and sigmoid, we found the polynomial kernel fits the data better. Visualization of selected SVR models are shown in Figure 5.13.

5.5 Results

5.5.1 Evaluation metric

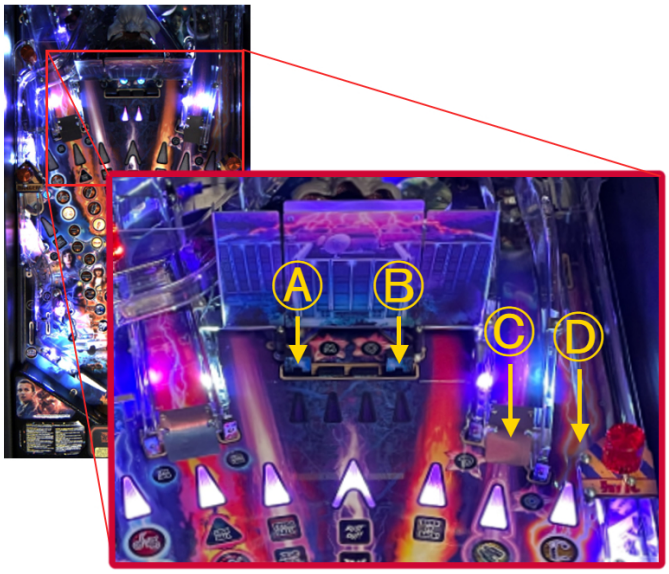


Figure 5.14: The four target locations.

To evaluate the performance of the developed ML models, we selected four frequently used targets in the 'Stranger Things' pinball game. The four targets are shown in Figure 5.14. Target A and B, which are stand targets, are 0.8 inches wide and target C and D, which are entry points to ramps, are 1.8 inches wide. We use each target's center point as the target position. According to our setup, one pixel in the camera corresponds to 0.019 inches in the playfield. The evaluation is done by making shots to the selected four targets and analyzing the deviation of launch trajectories from the target points. To compare the performance, we also implement the polynomial fit method presented in [22] which works on a two variable system state description, and we asses it over the same evaluation conditions.

5.5.2 Asynchronous Shot Aiming Controller

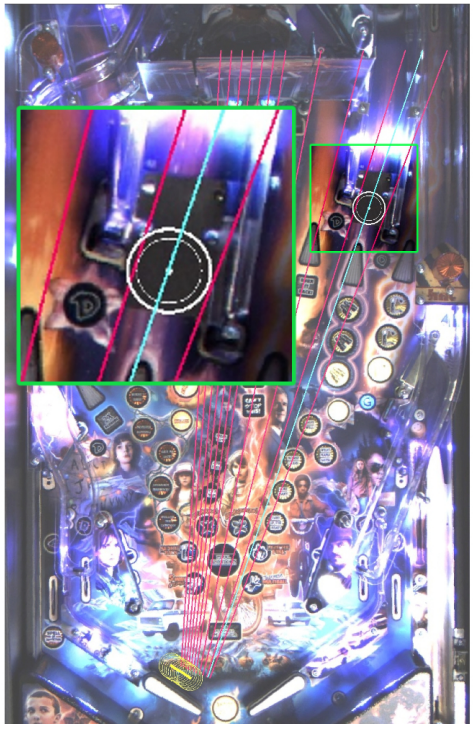
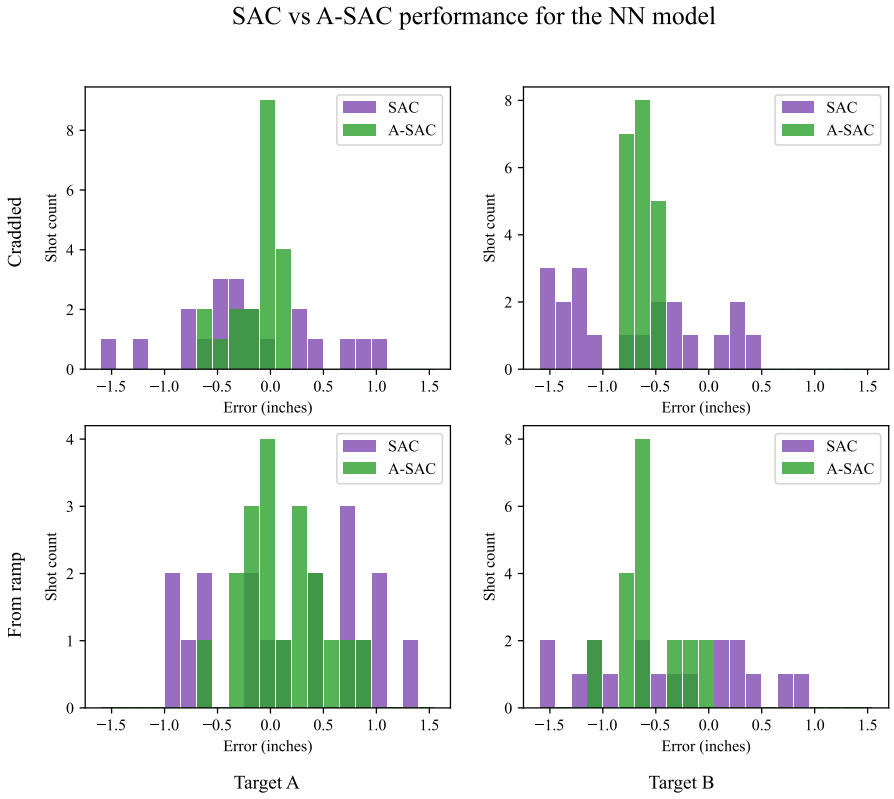


Figure 5.15: Difference between synchronous and asynchronous shot aiming methods.

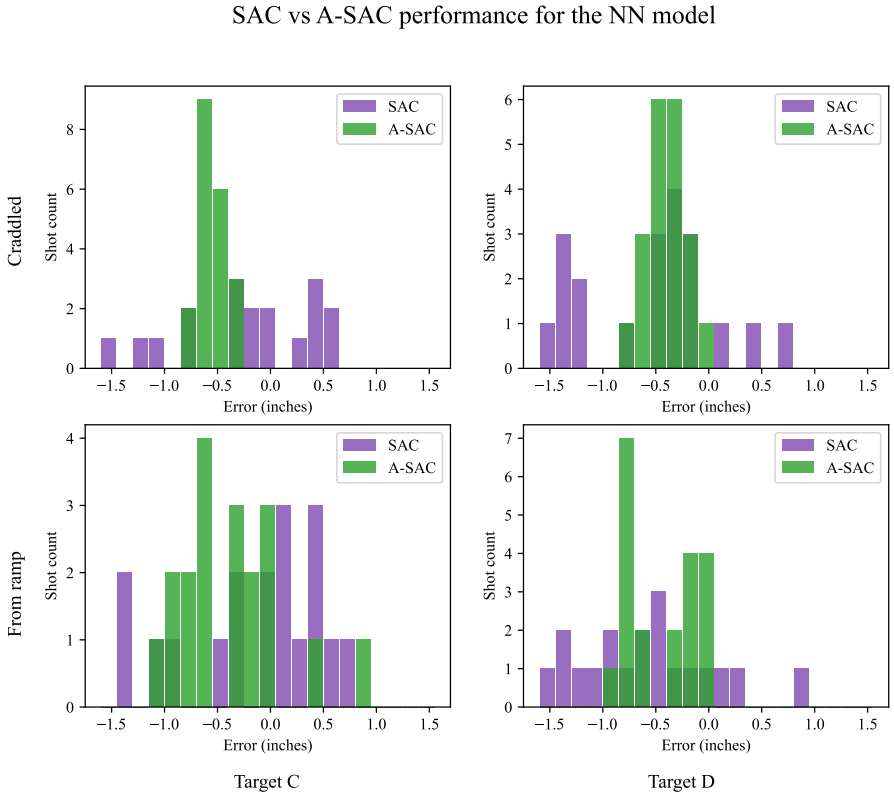
We use the term 'Asynchronous' in our SAC to explain the difference between the SAC we previously introduced in [22] and this implementation. Figure 5.15 shows a trail of the ball rolling down the flipper in the rolling phase as yellow circles on top of the left flippers. The red lines correspond to predicted launch trajectories from a given model at each frame capture moments. The center of the white circle is the selected target. In [22], the SAC is restricted to making shots synchronously with image capture events. Hence, the possible shots it can take are only the ones shown in red. The asynchronous shot aiming controller (A-SAC) we introduce, is able to make a shot anytime, asynchronously with image capture events. In an ideal scenario, A-SAC is able to aim and execute a shot precisely to go through a target position at any point in the playfield as shown by the cyan line in the Figure.

From the two methods presented for implementing the A-SAC in Section 5.2.2, the iterative solver is capable of reaching the exact answer, whereas the sampling solver ap-

proximates the answer, in-which the precision depends on the number of samples allowed. The SVR method and polynomial method is able to calculate a prediction in less than $1ms$ whereas the neural network model inference takes $\approx 11ms$ for single input in the PC we used (Intel(R) Xeon(R) CPU E5-1650 3.60GHz, 32GB RAM and Nvidia GTX960 GPU). Since we were running our system at $50Hz$, we are time bound to complete the full pipeline from capturing the image, processing the data, calculating the optimal shot to actuating, within the frame period which is $20ms$. With this time constraint, secant solver was only applicable on the polynomial fit method and SVR. For the neural network based method we went with sampling solver leveraging the batch inference capability of 'PyTorch' framework which was able to process a batch of 20 state samples, $\approx 14ms$ in average.



(a) Target A and B



(b) Targe C and D

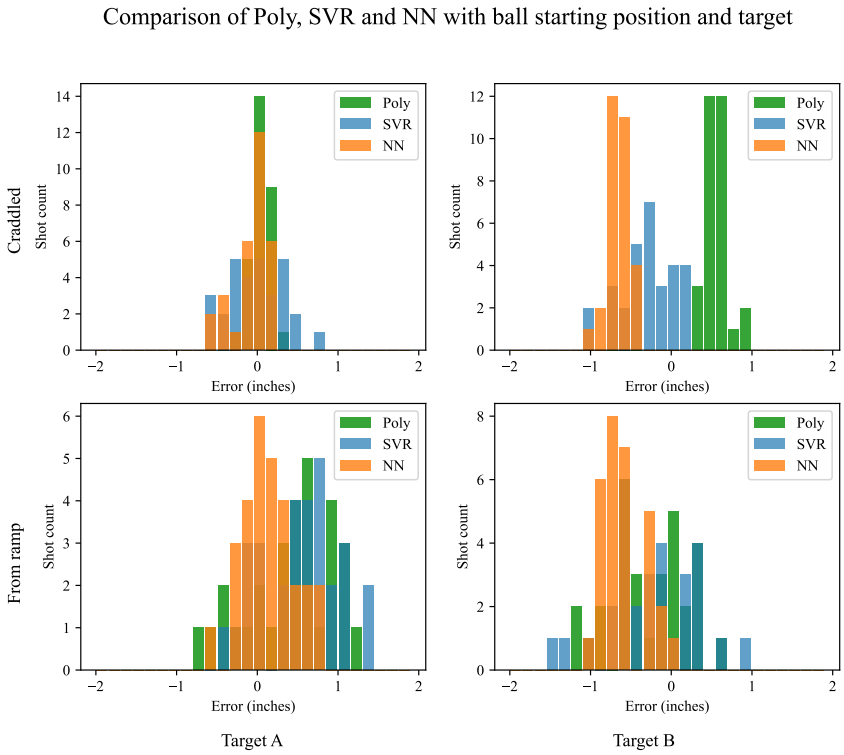
Figure 5.16: Error distribution of shots using SAC and A-SAC for different targets and starting locations

To evaluate the performance of the shot aiming controller, we conducted two trials. One with SAC and another with A-SAC. Both trials consisted of 20 shot samples for each target, from the two ball start locations: from middle of the ramp and from the cradled position. In both trials, we used the same Neural Network model. Figure 5.16 shows the result. It is evident that A-SAC was able to confine the error to a smaller region than the SAC.

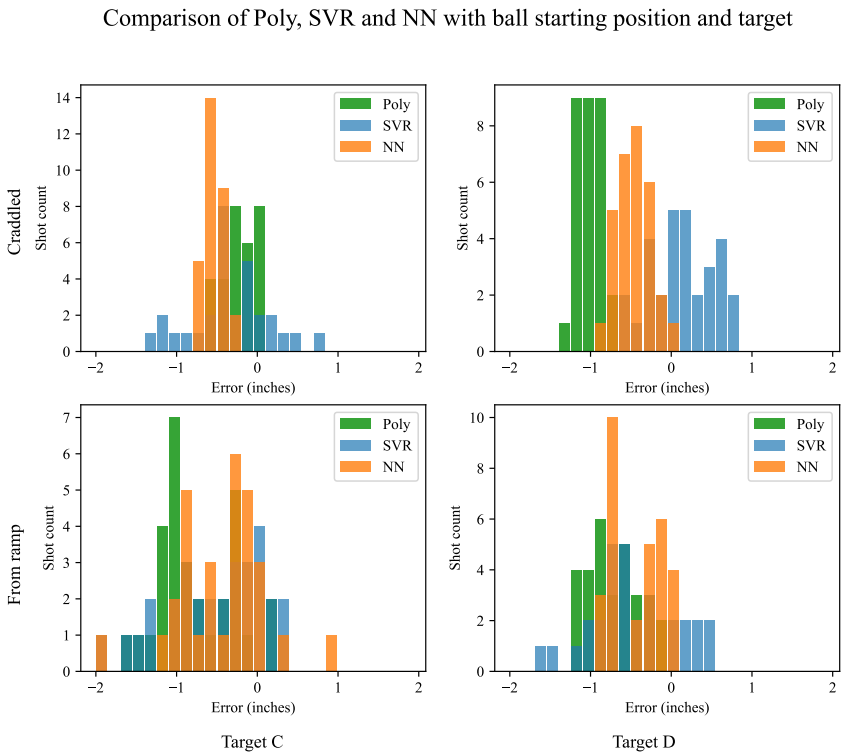
5.5.3 Model Predictive Controller

To evaluate the performance of the MPCs we trained, we conducted three trials. One with the polynomial model, one with the SVR model and one with the NN model. All models used A-SAC for shot aiming. Each trial consisted of 30 shot samples per each target and per each of the two starting locations. The results are shown in Figure 5.17, with separate plots for each target and each ball starting location. From the 720 total trial shots, 9 data samples were retaken as the errors reported were higher than ± 7, which we suspect to be caused by errors in the tracking system or the computer.

The results show that three models have shown different accuracy levels depending on the ball start location and the selected target. Generally for a human player, the ball starting from the cradled position makes shooting easier than the ball rolling down the ramp, as the velocity of the ball is significantly lower when ball starts from the cradled position. We can decide if the outcome is a hit or miss by comparing the error with the required accuracy values for each target. The estimated accuracy for hitting target A and B are ± 0.95 inches and for target C and D, which are ramp entry points, are ± 1.1 inches.



(a) Target A and B



(b) Target C and D

Figure 5.17: Error distributions for Polynomial model, SVR model and Neural network model, all using A-SAC

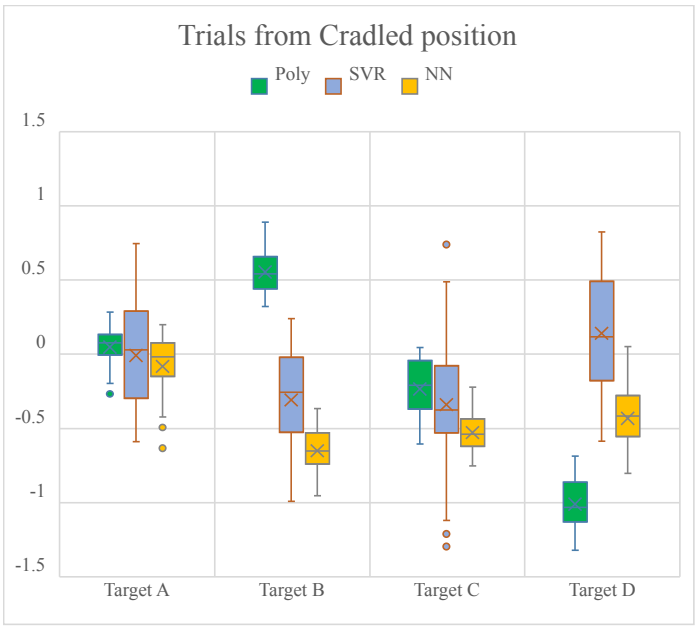


Figure 5.18: Error distribution for trials with the ball started from the cradled position.

Figure 5.18 shows the distribution of error for the ball starting from the cradled position. As expected, Target A was the easiest for all 3 models as the ball moves the slowest in the range of target A. From the 3 models, polynomial model and NN model show a lower variation in error comparing to the the SVR model for all 4 targets. For different targets, polynomial model seems to have a bias to either of plus or minus error sides. The SVR model result variance is higher; still the deviation of the means from zero error is less than that of the polynomial model.

Comparing with the required accuracy levels for each target, the Polynomial model has shown above a 92% hit rate, SVR model has shown above a 96% hit rate and NN model has shown above a 98% hit rate for all targets overall.

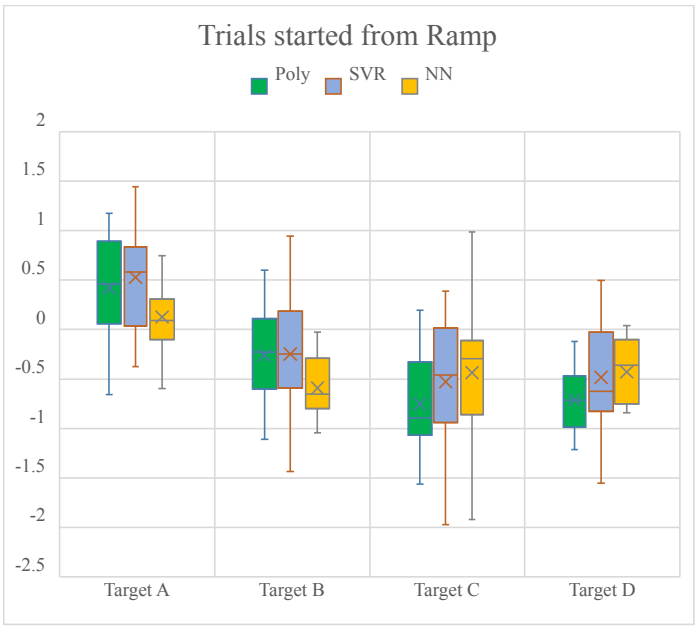


Figure 5.19: Error distribution for trials with the ball started from middle of the ramp.

Figure 5.19 shows the error distribution for the trials where the ball started from the middle of the ramp. Usually, the difficulty of shooting the ball from this location is higher for a human player as the ball has already gained momentum through the drop from the height. Both the NN and Polynomial models seem to have a wider variance in error distribution for the same target when compared with the ball starting from the cradled position. Comparing the required accuracy levels for each target, both the Polynomial model and the SVR model have shown hit rates above 85%, and the NN model has shown a hit rate above 96% on all targets.

Overall, all three models have learned to shoot the selected targets with at least an accuracy of 85%. The SVR model which has a comparatively larger variance in most of the cases, can be considered to be more unstable than the Polynomial model and NN model. Of the three methods, the NN method has reported the best results of at least a 96% hit rate on all scenarios. Also we have to factor in for the data errors of the limited number of data points we supplied the models with, errors in the pinball framework and the effects of the hyperparameters we used for the models, which can have an effect on the overall results of the models.

Discussion and Conclusion

The goal of this work was to create a real-time Cyber Physical System sandbox using pinball machines and to test the applicability of ML techniques for control. We have developed a simulator framework to test ML techniques on pinball in a virtual setting. Using that, we have experimented on model-free control methods that fall into reinforcement learning category. We have developed a real-world framework to be used with physical pinball machines which we had used to experiment on model-based techniques, incorporating the learnings from the simulator based experiments. We have used three different ML techniques on MPC to see what they can accomplish and how they compare with each other. Also, we discussed the methods and challenges in developing a controller for a real-time Cyber Physical System from both controller design perspective and hardware and software development perspective.

From the results, we saw that the introduction of a hardware based synchronization and asynchronous shots had a positive effect on the results. All three ML methods we tested were able to learn to aim and shoot with at least 85% accuracy with just 120 data-points to train. We observed that using a model-based method reduced the data requirement from hundreds of thousands, which we saw in simulator based experiments, to around a hundred, which we saw in real-world experiments. Hence, we can conclude that ML based MPC was able to learn a good shot aiming controller for the RSS in a data-efficient manner. And from the three ML methods we tested for the RSS, the Neural Network based method was able to deliver a hit rate of 96%.

Selection of the state space on this work was done in a generalized manner, considering possible future expansion of this project. The same state space is applicable on the entire *control region*, which essentially covers the entire range of possible scenarios in the game of pinball. The learned controllers in this work can be used as control modules in a hierarchical or modular learning scheme as discussed in the simulator based experiments section.

The development of the real-world framework is done in an generalized manner, so it can be ported to any other physical pinball machine with little to no modifications. Hence, the same framework can be used to experiment with transfer learning, where we can test to what extent we can reuse the knowledge of an agent trained from one machine on another. The SVR method was chosen due to the low computational power and time required to train an agent with new data. We can extend our work to create an online learning agent who improves itself with all new pinball shots they make and able to utilize them immediately.

As final remarks, we can state that Machine Learning based control on real-time CPS is an under-explored domain. Specifically, training, testing and evaluation of control methods on physical hardware exposes the real challenges in the process. Hence, we believe the sandbox environments we created and the knowledge we gained through the experiments will be helpful in future research on real-time CPS controllers.

Bibliography

[1] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: the next computing revolution," in *Design automation conference*, pp. 731–736, IEEE, 2010.

[2] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[4] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, "Zero-shot text-to-image generation," in *International Conference on Machine Learning*, pp. 8821–8831, PMLR, 2021.

[5] J. Zander, "Model-based testing for execution algorithms in the simulation of cyber-physical systems," in *2013 IEEE AUTOTESTCON*, pp. 1–7, IEEE, 2013.

[6] I. Buzhinsky, C. Pang, and V. Vyatkin, "Formal modeling of testing software for cyber-physical automation systems," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 3, pp. 301–306, IEEE, 2015.

[7] Z. Jiang, M. Pajic, and R. Mangharam, "Cyber–physical modeling of implantable cardiac medical devices," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 122–137, 2011.

[8] J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Distributed real-time software for cyber–physical systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 45–59, 2012.

[9] H. Neema, P. Volgyesi, B. Potteiger, W. Emfinger, X. Koutsoukos, G. Karsai, Y. Vorobeychik, and J. Sztipanovits, "Sure: An experimentation and evaluation testbed for cps security and resilience: Demo abstract," in *Proceedings of the 7th International Conference on Cyber-Physical Systems*, pp. 1–1, 2016.

[10] V. Matena, T. Bures, I. Gerostathopoulos, and P. Hnetynka, "Model problem and testbed for experiments with adaptation in smart cyber-physical systems," in *2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 82–88, IEEE, 2016.

[11] H. Zhang, D. Ge, J. Liu, and Y. Zhang, "Multifunctional cyber-physical system testbed based on a source-grid combined scheduling control simulation system," *IET Generation, Transmission & Distribution*, vol. 11, no. 12, pp. 3144–3151, 2017.

[12] H. Gao, Y. Peng, K. Jia, Z. Wen, and H. Li, "Cyber-physical systems testbed based on cloud computing and software defined network," in *2015 International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, pp. 337–340, IEEE, 2015.

[13] T. L. Crenshaw and S. Beyer, "Upbot: a testbed for cyber-physical systems," in *Proceedings of the 3rd international conference on Cyber security experimentation and test*, pp. 1–8, 2010.

[14] S. Poudel, Z. Ni, and N. Malla, "Real-time cyber physical system testbed for power system security and control," *International Journal of Electrical Power & Energy Systems*, vol. 90, pp. 124–133, 2017.

[15] M. Khan, S. Alam, A. Mohamed, and K. A. Harras, "Simulating drone-be-gone: Agile low-cost cyber-physical uav testbed," in *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pp. 1491–1492, 2016.

[16] M. H. Cintuglu and O. A. Mohammed, "Cloud communication for remote access smart grid testbeds," in *2016 IEEE Power and Energy Society General Meeting (PESGM)*, pp. 1–5, IEEE, 2016.

[17] Y. Zhou, J. Wang, H. Du, H. Li, P. Hu, G. Wang, and R. Shao, "The multi-agent based evaluation of connected vehicle systems," in *2014 IEEE Vehicular Networking Conference (VNC)*, pp. 131–132, IEEE, 2014.

[18] N. Winstead, "Some explorations in reinforcement learning techniques applied to the problem of learning to play pinball," in *Proceedings of the AAAI-03 Workshop on Entertainment and AI/A-Life*, pp. 1–5, 1996.

[19] K. Fragkiadaki, P. Agrawal, S. Levine, and J. Malik, "Learning visual predictive models of physics for playing billiards," *arXiv preprint arXiv:1511.07404*, 2015.

[20] J. Renz, X. Ge, M. Stephenson, and P. Zhang, "Ai meets angry birds," *Nature Machine Intelligence*, vol. 1, no. 7, pp. 328–328, 2019.

[21] A. Metcalf, "Pinball: High-speed real-time tracking and playing," 2011.

[22] Z. E. Fuchs, P. Saranguhewa, and M. Ikuru, *Real-Time Model Predictive Control for Shot Aiming in a Physical Pinball Machine*, p. 1–8. IEEE, Aug 2021.

[23] "Future pinball." `https://futurepinball.com/`. [Online], Date last accessed 04-August-2022.

[24] "Visual pinball." `https://github.com/vpinball/vpinball`. [Online], Date last accessed 04-August-2022.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

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

[27] F. S. Melo, "Convergence of q-learning: A simple proof," *Institute Of Systems and Robotics, Tech. Rep*, pp. 1–4, 2001.

[28] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," *arXiv preprint arXiv:1912.10944*, 2019.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[30] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," 2019.

[31] S. Bhat, C. L. Isbell, and M. Mateas, "On the difficulty of modular reinforcement learning for real-world partial programming," in *AAAI*, pp. 318–323, 2006.

[32] C. Simpkins and C. Isbell, "Composable modular reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 4975–4982, 2019.

[33] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," *Advances in neural information processing systems*, vol. 29, 2016.

[34] O. Nachum, S. S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," *Advances in neural information processing systems*, vol. 31, 2018.

[35] A. Levy, G. Konidaris, R. Platt, and K. Saenko, "Learning multi-level hierarchies with hindsight," *arXiv preprint arXiv:1712.00948*, 2017.

[36] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, "Hierarchical reinforcement learning: A comprehensive survey," *ACM Comput. Surv.*, vol. 54, jun 2021.

[37] M. M. Botvinick, Y. Niv, and A. G. Barto, "Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective," *Cognition*, vol. 113, no. 3, pp. 262–280, 2009.

[38] X. Wang, Y. Chen, and W. Zhu, "A survey on curriculum learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[39] L. G. Shapiro, G. C. Stockman, *et al.*, *Computer vision*, vol. 3. Prentice Hall New Jersey, 2001.

[40] R. Tsai, "A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses," *IEEE Journal on Robotics and Automation*, vol. 3, no. 4, pp. 323–344, 1987.

[41] J. Heikkila and O. Silvén, "A four-step camera calibration procedure with implicit image correction," in *Proceedings of IEEE computer society conference on computer vision and pattern recognition*, pp. 1106–1112, IEEE, 1997.

[42] P. Sturm and S. Maybank, "On plane-based camera calibration: A general algorithm, singularities, applications," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 1, pp. 432–437 Vol. 1, 1999.

[43] C. X. Guo, F. M. Mirzaei, and S. I. Roumeliotis, "An analytical least-squares solution to the odometer-camera extrinsic calibration problem," in *2012 IEEE International Conference on Robotics and Automation*, pp. 3962–3968, 2012.

[44] Z. Zhang, "A flexible new technique for camera calibration," *IEEE transactions on pattern analysis and machine intelligence*, vol. 22, no. 11, p. 1330–1334, 2000.

[45] Z. Gao, M. Zhu, and J. Yu, "A novel camera calibration pattern robust to incomplete pattern projection," *IEEE Sensors Journal*, vol. 21, no. 8, pp. 10051–10060, 2021.

[46] O. Kedilioglu, T. M. Bocco, M. Landesberger, A. Rizzo, and J. Franke, "Arucoe: Enhanced aruco marker," in *2021 21st International Conference on Control, Automation and Systems (ICCAS)*, pp. 878–881, 2021.

[47] G. H. An, S. Lee, M.-W. Seo, K. Yun, W.-S. Cheong, and S.-J. Kang, "Charuco board-based omnidirectional camera calibration method," *Electronics*, vol. 7, no. 12, 2018.

[48] "Spinnaker sdk." `https://www.flir.com/products/spinnaker-sdk`. [Online], Date last accessed 28-August-2022.

[49] S. Suzuki *et al.*, "Topological structural analysis of digitized binary images by border following," *Computer vision, graphics, and image processing*, vol. 30, no. 1, pp. 32–46, 1985.

[50] T. Bouwmans, "Traditional and recent approaches in background modeling for foreground detection: An overview," *Computer science review*, vol. 11, pp. 31–66, 2014.

[51] J. Zheng, Y. Wang, N. L. Nihan, and M. E. Hallenbeck, "Extracting roadway background image: Mode-based approach," *Transportation research record*, vol. 1944, no. 1, pp. 82–88, 2006.

[52] K. Toyama, J. Krumm, B. Brumitt, and B. Meyers, "Wallflower: Principles and practice of background maintenance," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 1, pp. 255–261, IEEE, 1999.

[53] S. Messelodi, C. M. Modena, N. Segata, and M. Zanin, "A kalman filter based background updating algorithm robust to sharp illumination changes," in *International Conference on Image Analysis and Processing*, pp. 163–170, Springer, 2005.

[54] E. Zhang, F. Chen, and W. Zhang, "A novel particle filter based background subtraction method," in *2006 International Conference on Computational Intelligence and Security*, vol. 2, pp. 1837–1840, IEEE, 2006.

[55] M. Shah, J. D. Deng, and B. J. Woodford, "Video background modeling: recent approaches, issues and our proposed techniques," *Machine vision and applications*, vol. 25, no. 5, pp. 1105–1119, 2014.

[56] A. Ilyas, M. Scuturici, and S. Miguet, "Real time foreground-background segmentation using a modified codebook model," in *2009 Sixth IEEE International Conference on Advanced Video and Signal Based Surveillance*, pp. 454–459, IEEE, 2009.

[57] T. Bouwmans, "Subspace learning for background modeling: A survey," *Recent Patents on Computer Science*, vol. 2, no. 3, pp. 223–234, 2009.

[58] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings. 1999 IEEE computer society conference on computer vision and pattern recognition (Cat. No PR00149)*, vol. 2, pp. 246–252, IEEE, 1999.

[59] R. Di Salvo and C. Pino, "Image and video processing on gpu: Implementation scheme, applications and future directions," in *Advances in Mechanical and Electronic Engineering* (D. Jin and S. Lin, eds.), (Berlin, Heidelberg), pp. 375–382, Springer Berlin Heidelberg, 2013.

[60] Q. Li, R. Li, K. Ji, and W. Dai, "Kalman filter and its application," in *2015 8th International Conference on Intelligent Networks and Intelligent Systems (ICINIS)*, pp. 74–77, 2015.

[61] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.

[62] G. Bishop, G. Welch, *et al.*, "An introduction to the kalman filter," *Proc of SIGGRAPH, Course*, vol. 8, no. 27599-23175, p. 41, 2001.

[63] D. Limon, J. Calliess, and J. M. Maciejowski, "Learning-based nonlinear model predictive control," *IFAC-PapersOnLine*, vol. 50, p. 7769–7776, Jul 2017.

[64] W. E. Deming, "Statistical adjustment of data.," 1943.

[65] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[66] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[67] M. Awad and R. Khanna, *Support Vector Regression*, pp. 67–80. Berkeley, CA: Apress, 2015.