An Image-Based Agent with a Hand-Tuned Controller for Ice Hockey Strategy in SuperTuxKart

CS 394D: Deep Learning

Benjamin Hunter, Michael Pontikes, Eric Ravet, Luis Smith

30 April, 2023

Abstract

This paper proposes an image-based agent with a hand-tuned controller for ice hockey gameplay in the SuperTuxKart environment[1]. Our primary objectives are to maximize the number of goals scored and win the match. We use an image-based agent to infer the location of the puck, which is then used by our controller to generate gameplay actions. We generate training data from simulated matches, train a fully convolutional network for object detection and distance estimation, and develop two hand-tuned controller strategies. Our approach demonstrates the effectiveness of image-based agents in complex game environments and presents a detailed analysis of the model's performance and the hand-tuned controller strategies.

Section 1: Problem Setup

1.1. Ice Hockey Tournament

The task is to create the gameplay actions for a team consisting of two SuperTuxKart ice-hockey players. The team has the following objectives: to score as many goals as possible and to win the match. These objectives skew towards an offensive/attacking strategy since we are specifically evaluated on the average number of goals we score per match. To evaluate performance, the team will play 2 vs 2 tournaments against agents that were created by the TAs and the professor. The tournament environment consists of an ice hockey rink with two goals and standard SuperTuxKart physics.

1.2. Image Agent

We chose to create an image agent. The agent gets an image from the kart it controls to infer where the puck is located. The image resolution is 300 x 400 pixels (height x width) with three RGB color channels. The image-based agent has access to some game state including player kart locations, orientation, and velocity. However, the agent does not get access to the state of the puck or opponents during the gameplay. An additional requirement is a time limit of 50 milliseconds per call to the Team.act function on a GPU, which limits the complexity of our model and ensures real-time performance.

[1] https://github.com/philkr/pystk

Section 2: Generation Training and Evaluation Data

2.1. Data Logged and Used in Training Process

Data logged and used in the training process includes Team 1 and Team 2 state information.

State information includes:

- Camera information (e.g. view and projection matrices)
- Kart location information (e.g. location of the back and front of the kart)
- Kart steering information (e.g. max steer angle and velocity)

We also have access to the Team 1 images and Team 2 images, which shows the image from the vantage point of each kart for each team. Additionally, we have access to the ice hockey game state, which contains the location and size of the puck and the location of the goals.

All location information is logged in world coordinates. World coordinates can be described by 3 different dimensions (x, y, z) where:

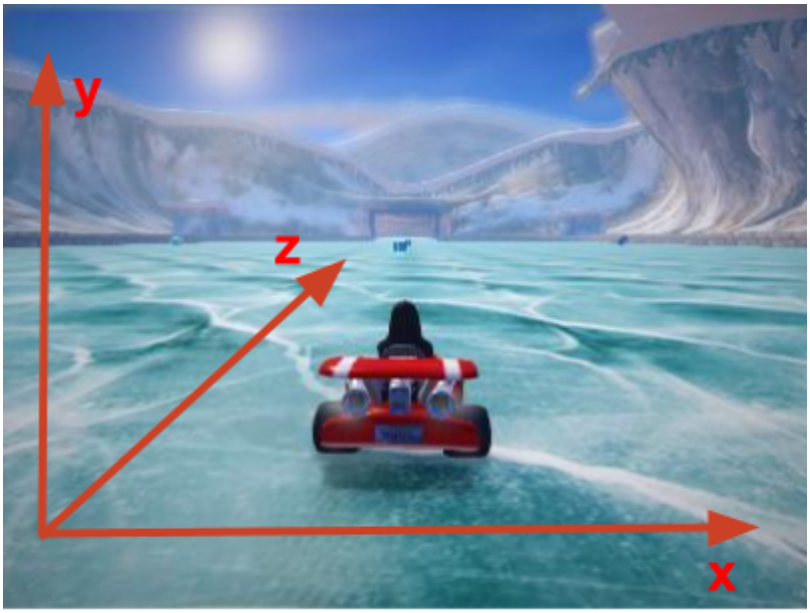

Figure 1: World coordinates of the ice hockey track

- x = coordinate in the direction of the width of the field (i.e. distance between sidelines)
- y = coordinate for height

- z = coordinate in the direction of the length of the field
  (i.e. distance between goals)

2.2. Simulated Matches

We simulate 8 matches as training data. In order to increase the robustness of the training data, we use different starting criteria for each match, such as different starting positions for each team. For example, half the games a team will be the red team, and half the games a team will be the blue team. We also use different starting puck positions, and randomized starting puck velocity. We simulate a 9th match that is unseen in the training data and strictly used for evaluation purposes.

Match	Team 1	Team 2	Puck Position	Puck Velocity (x,y)	Train or Evaluation?	Num Frames
#1	Red	Blue	[0, 1]	rand(-1,1), rand(-1,1)	Train	3,000
#2	Red	Blue	[0, -1]	rand(-1,1), rand(-1,1)	Train	3,000
#3	Red	Blue	[1, 0]	rand(-1,1), rand(-1,1)	Train	3,000
#4	Red	Blue	[-1, 0]	rand(-1,1), rand(-1,1)	Train	3,000
#5	Blue	Red	[0, 1]	rand(-1,1), rand(-1,1)	Train	3,000
#6	Blue	Red	[0, -1]	rand(-1,1), rand(-1,1)	Train	3,000
#7	Blue	Red	[1, 0]	rand(-1,1), rand(-1,1)	Train	3,000
#8	Blue	Red	[-1, 0]	rand(-1,1), rand(-1,1)	Train	3,000
#9	Red	Blue	[0, 1]	rand(-1,1), rand(-1,1)	Evaluation	3,000

Table 1: Training and Evaluation Simulated Match Parameters

For training, we used 8 total matches of 3,000 frames and 2 player karts per frame. This resulted in a total of 48,000 training examples. For evaluation, we used 1 match and had 6,000 evaluation examples.

To generate varied sets of training data, we also experimented with different permutations of teams, including:

- AI
- Dummy Team with no action
- Image agent (using perfect information from the soccer game state)

- Image agent (using model prediction)
- TA state agents

After experimenting with these combinations, we chose to use the AI teams for data generation because this allowed us to reliably generate quality data and conduct hyperparameter tuning on our model while our controller was under construction. We assessed data quality visually by reviewing video output from the teams. We looked for a data set that included diverse scenarios and positions of karts, puck, and goals throughout the ice-hockey map. We specifically wanted data that included karts successfully finding the puck and scoring.

Once the image agent was able to score consistently with perfect information from the soccer game state, we tested data generation using our image agent. Data generation using our own model predictions to control the kart proved impractical due to performance of the data generation. A data set of 48,000 training frames would have taken almost an hour on local hardware. Without an obvious advantage to the new dataset, we continued training against the data generated AI teams.

Section 3: Model Training

3.1. Pre-processing Data

For every match, we use data from one player from Team 1 and one player from Team 2. This gives us an increased amount of training data as opposed to only considering one player from one team. We treat a single frame from the vantage point of a specific kart as a training example. As a preprocessing step, we normalize the pixel values to a range between 0 and 1.

We create two sets of labels for our training data:

1. Puck center points in terms of pixel coordinates of the 2-D image the kart sees
2. Euclidean distance of the kart's location in world coordinates and the puck's location in world coordinates

The purpose of the puck centerpoint was to help us choose a target aim point. The purpose of the euclidean distance of the puck was to help us decide target velocity and acceleration.

3.1.1. Generating Center-Point Labels

Object detections are in terms of pixel coordinates of the 2-D image the kart sees. To calculate this, we use the soccer state's 3-D world coordinates of the object and convert them to 2-D screen coordinates. First, we translate the 3-D cartesian coordinates (x, y, z) to 4-D homogeneous coordinates (x', y', z', w) by appending a 1 to the end. Using coordinates in this homogeneous form allows us to represent points at infinity (w = 0) and apply the translation transformation to the camera which is impossible with a 3-D matrix. To convert back to cartesian coordinates, we simply divide the first three components by the fourth component (x = x'/w, y = y'/w, z = z'/w).

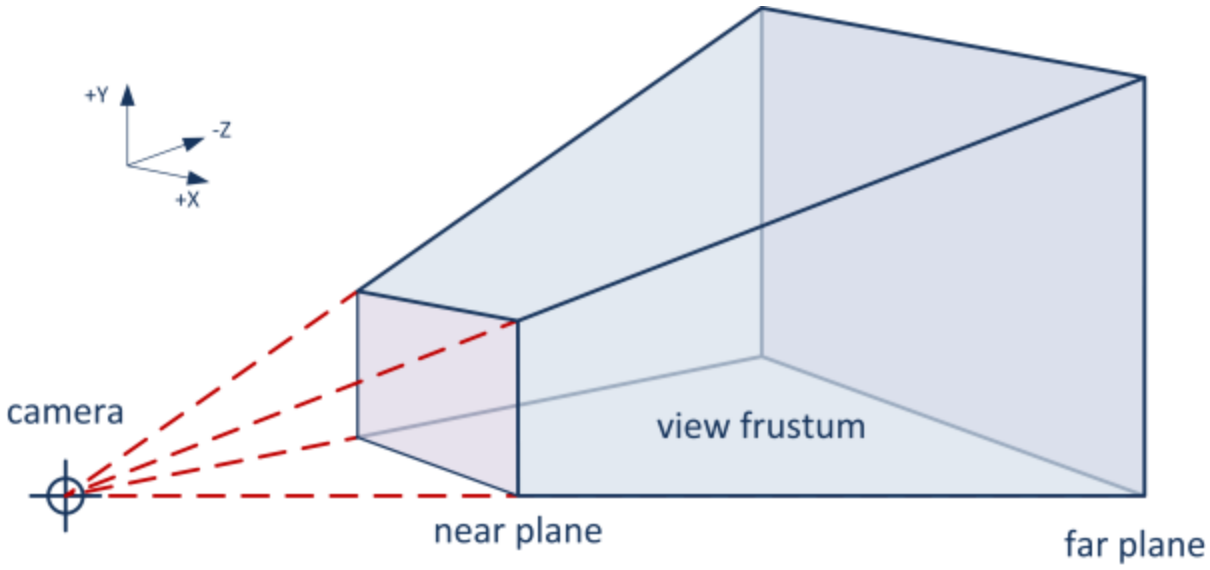

Figure 2**:** Visualization of a view frustum [2]

Applying the projection and view matrices to a 4-D point in world space takes the points within the viewing frustum and squashes it into the view box which ranges from -1 to +1 in the x, y, and z dimensions, where x and y represent the width and height of the screen and the depth z represent the near and far plane of the frustum. The z dimension is discarded because our screen is only 2 dimensions.

Visible points lie within the range of [-1, +1] in all x, y, and z dimensions[3]. To account for objects that may be partially visible on the screen, we include a buffer equal to the object size. The object is approximated as a sphere to simplify the calculation by using the 2-norm. The resulting on-screen position in x, y dimensions is converted to pixel coordinates to match our model's heatmap, and subsequently appended to the training set as a label.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \text{Proj} \times \text{View} \times \begin{bmatrix} \text{x} \\ \text{y} \\ \text{z} \\ 1 \end{bmatrix}$$

[2] https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling
[3] https://www8.cs.umu.se/kurser/5DV051/HT12/lab/plane_extraction.pdf

Equation 1: Projection of visible points

3.2. Model Evaluation Criteria

To evaluate our model we used average precision. Average precision computes the average precision value for recall value over 0 to 1[4]. Average precision requires criteria to determine whether an object detection is successful or unsuccessful. In our case, we used two different criteria: average precision for object detection within 5 pixels and average precision for distance estimation within 10% distance. These metrics assess the model's ability to accurately locate objects in the image and estimate their distance from the kart. A model that performs well on these two metrics would provide the controller with the necessary information to choose aimpoints and velocities to better score goals.

We also wanted to be able to determine a minimum score threshold to consider a puck detection as valid or not. To do this, we log the scores for each valid detection and each invalid detection. We then analyze the distribution of the scores for the valid detections and invalid detections. We manually choose a score threshold for our trained model to use during tournament gameplay. For example, in our analysis, a score threshold of -1 for our final model would result in 97% valid and 3% invalid detections.

3.3. Model Architecture

For building an object detection model, we sought an architecture that would allow us to determine the center point of the puck. Thus, we wanted to be able to score each pixel in terms of the likelihood that it is a puck. An architectural framework that accomplished this goal was a Fully Convolutional Network.

3.3.1 Fully Convolutional Network (FCN)

A Fully Convolutional Network (FCN) has the ability to quickly and accurately get the segmentation heatmap of an image[5] for object detection tasks, as it contains a combination of downsampling and upsampling layers to capture hierarchical features and spatial information. This is done through a series of convolutions that first downsample the image into a series of feature maps that then are scaled back up to the image size using upsampling or up-convolutions. This process, in addition to adding skip convolutions to help bring in data from earlier layers, creates a heatmap that provides a score for each pixel on the original input image. Additionally, FCNs are known for having high performance and low-latency. At the time of publication, simultaneous detection and segmentation (SDS)[6] models were the previous

[4] https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173
[5] https://arxiv.org/pdf/1411.4038.pdf
[6] https://arxiv.org/pdf/1411.4038.pdf

state-of-the-art system. However, FCNs demonstrated a 20% improvement in mean Intersection over Union (IU), and ran orders of magnitude faster than SDS models.

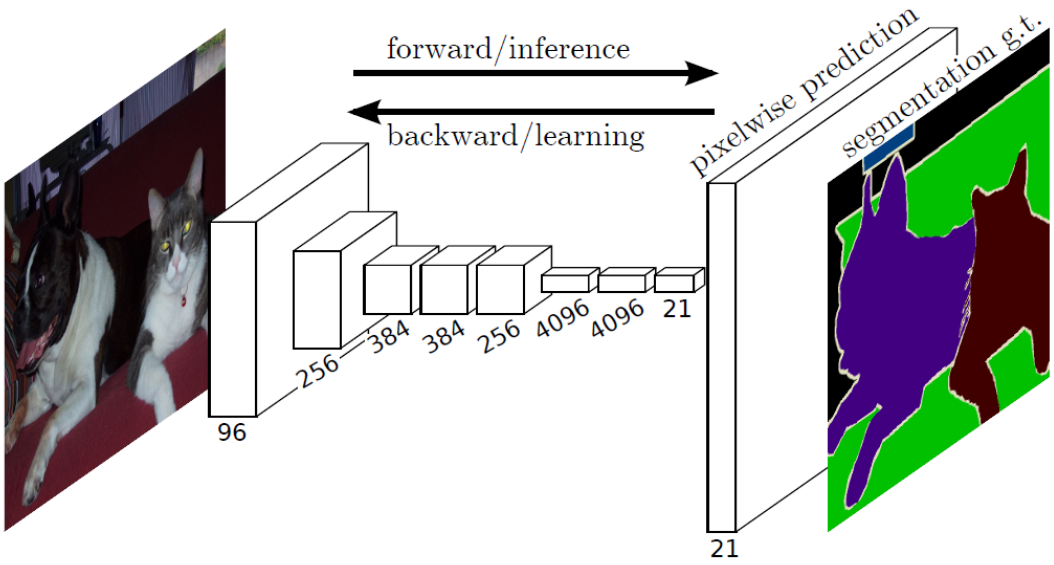

Figure 3: Visualization of an example of a fully convolutional model[7]

3.3.2 FCN for Puck Detection

As discussed in the previous section, a FCN model allows us to quickly and accurately get the segmentation heatmap of an image with a score for each pixel for each object we have chosen to detect. From the heatmap, we are able to extract the locations of the peak scores, which corresponds to the predicted center point of the object[8]. We originally experimented with detecting the puck, the teammate kart, opposing team karts, the opposing team goal, our own goal and items. However, ultimately our controller logic relied only on the puck, so we chose to only detect the puck.

Since we only need to detect a single puck for each kart, this means that whatever pixel has the highest score in the heatmap is going to be the point that is the most likely to be the puck on the screen. There is also a minimum score threshold we provide to minimize false detections. We pass the location of the puck detection for each kart to the controller so that it can make an informed decision on the next action.

[7] https://arxiv.org/pdf/1411.4038.pdf
[8] https://arxiv.org/pdf/1411.4038.pdf

Figure 4: Ground truth label, frame from Python SuperTuxKart, model prediction (pictured from left to right). The puck is in red and the goal is in green.

3.3.3 Model Architecture

The architecture of the model is organized as a series of down-block and up-block layers, with skip connections between corresponding layers. The input shape is a 300x400 image with three color channels for RGB, and the final output consists of two tensors representing class probabilities and object distances. The model contains a total of 794,988 parameters resulting in just over 3MB of weights.

The forward pass of the model processes the input image through a series of convolutional, batch normalization, and ReLU activation layers, utilizing skip connections to concatenate the outputs from corresponding layers. The model's detect function extracts object detection results from the network's outputs by calling extract_peak and returns a list of detections for each class, limited to a configurable maximum number of detections per image per class. In practice we only need one detection for the puck.

The architecture consists of the following layers:

- A series of 4 Block layers, each with a combination of Conv2d and BatchNorm2d layers. The number of output channels increases progressively from 16 to 128. The Block class includes a skip connection, which helps preserve spatial information as the input passes through the network.
- A series of 4 UpBlock layers with ConvTranspose2d layers, which upsample the feature maps, decreasing the number of output channels from 128 to 16, while increasing spatial dimensions.
- Two final Conv2d layers which output the class probabilities and object distances, respectively.
- See Appendix A: Torchinfo Report of Model Architecture for the complete structure.

3.4. Model Hyperparameters Attempted and Evaluation Results

To find the optimal hyperparameters for our model, we experimented with batch size, training data size, learning rate, loss weights, and training epochs. Our best weights were discovered with a batch size of 40 (the maximum size our GPU allowed) and an initial learning rate of 0.001 after 60 epochs. Our full data set took 200 seconds per epoch for a total of 3 hours and 20 minutes of training time. All our training runs used the Adam optimizer and scheduler ReduceLROnPlateau[9] with patience of 5 epochs.

During training, we store model weights in a unique file every time the validation loss of the model improves. This method improves on the early stopping technique and allows us to identify the best weights outside the training loop. To find the best weights we compute the average precision for object detection within 5 pixels and the average precision for distance estimation within 10% distance as discussed in 3.2 Model Evaluation Criteria.

Table 2 summarizes the results of different model hyperparameters. Key findings from the table are:

- Initial learning rate of 0.001 was significantly more effective than other learning rates, even with the ReduceLROnPlateau scheduler. In general, lower initial learning rates (0.001 and 0.01) provided better precision results than higher initial learning rates (0.1 and 1).
- Training beyond 60 epochs did not significantly improve model performance.
- To fairly evaluate each set of weights, we use a single validation dataset to generate the precision performance in Table 2.
- Distance performance closely correlated to precision, so we omit it from the table.

[9] https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html

Run ID	Batch Size	Initial Learning Rate	Loss Weights	Epochs	Best Epoch	Validation Loss	Precision (5 px threshold)
1	16	0.001	0.1	60	55	0.136	0.9029
2	32	0.001	0.1	60	57	0.139	0.8178
3	40	0.001	0.1	60	60	0.148	0.9039
4	40	0.001	0.1	60	45	0.194	0.9017
5	40	0.001	0.1	120	62	0.196	0.9035
6	40	0.001	1	120	60	1.85	0.7969
7	40	0.01	0.1	40	40	0.175	0.814
8	40	0.1	0.1	40	17	3.784	0.0002
9	40	1	0.1	40	30	22.097	0.0006
10	40	0.0001	0.1	40	40	0.235	0.6094
11	30	0.01	0.1	40	40	0.195	0.8123
12	30	0.1	0.1	40	15	3.77	0
13	30	1	0.1	40	40	20.456	0.0005
14	30	0.0001	0.1	40	39	0.222	0.6848
16	40	0.1	0.1	100	99	0.234	0.616
100	20	0.01	0.1	80	80	0.36	0.061

Table 2: Model Hyperparameter Results

Section 4: Two Controller Strategies

We implemented a simple and advanced controller strategy that used our object detection models. Both strategies required the ability to determine whether the puck was in the view of a particular kart and then choose an action. To determine whether the puck is in a kart's view, we feed the kart's image to our machine-learning model. We ask the model to return 1 puck detection, and we use a minimum score threshold to determine whether the model returns a valid detection or not.

4.1. A Simple Controller

The simple controller was the first attempt at creating a controller that would work with our game data at a very basic level. The primary goal of the first model was to learn about the different aspects we can control and the data our karts receive as part of the game. This controller was first tested on the game state data, which does not use our model to get the puck's location and distance.

4.1.1 Controller Strategy

Our first controller's strategy can be categorized as using a passive approach to finding the puck, with an aggressive puck chasing strategy. In its first revision, it alternates between the following main strategies:

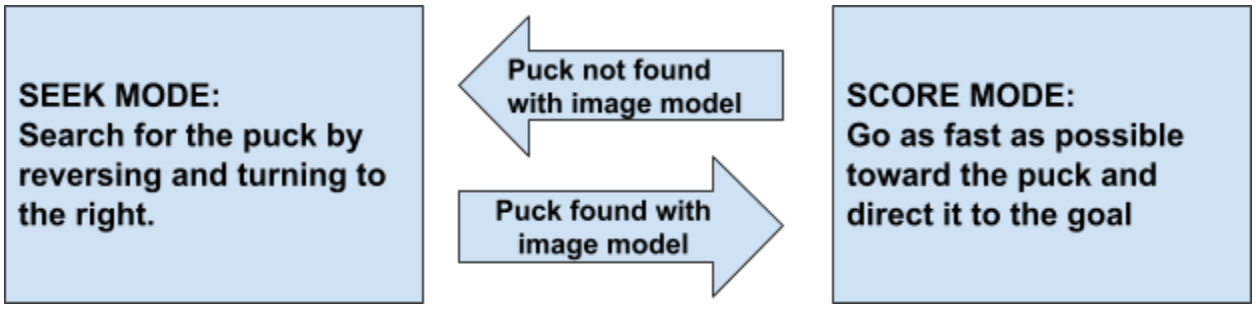

Figure 5: State graph of simple controller

If we successfully detected the puck, the controller converts the models' detection from pixel coordinates to (x,y) screen coordinates that are normalized to [-1, 1]. We then use the x-coordinate of the screen coordinate to determine an aim point. The aim point targets an angle between the puck's location and the opposing team's goal to score.

Below we describe the "Seek" mode and the "Score" mode:

- Seek mode simply reverses the kart to try and find the puck if the puck can't be found. The kart will set the brake to true and turn the steering variable to its max value in one direction (1.0)
- Score mode will attempt to speed toward the puck at max speed, hitting the puck on the left or right based on which side of the field the puck is currently on. The kart will use its own location to determine if it should hit the puck on the left or right. For example, if the kart is on the left half of the field, it will hit the puck on the left side. The amount of distance from the center of the puck is a static 7 pixels and does not change based on puck location. The kart will also go at max speed to get to the puck as quickly as possible (acceleration is set to 1.0). The puck location is found using the data from the model.

4.1.2 Evaluation

We first ran this controller and model combination on the local grader to determine the compatibility with the online grader. When run against the online grader, the first strategy scored an average of 0.44 goals per match.

Opponent	**Game 1**	**Game 2**	**Game 3**	**Game 4**	**Total**
Geoffrey	1:2	1:1	1:1	0:0	3:4
Jurgen	1:2	0:0	0:3	0:2	1:7
Yann	0:0	0:3	1:2	0:1	1:6
Yoshua	0:0	1:0	1:0	0:0	2:0
Average Goals		0.44 : 1.06	**Total Goals**		7 : 17

Table 3: Simple Controller Scores. Scores are in the format of: "Our Score:Opponent Score"

4.1.3 Controller Limitations

This controller had a few flaws with its logic. First, if you are far away from the puck, the model may have difficulty finding the puck and can keep spinning in circles for a long time, sometimes getting stuck in goal. Also, if the puck is on the other side of the field and the kart can't see the puck, it's possible to get stuck in the corner of the goal. Next, the max acceleration caused issues when trying to hit the puck into the goal at an angle since the kart was going so fast that it was unable to fully correct its trajectory as it approached the puck. Last, if the goal was not in view, the kart will just hit the puck straight so it was possible for our karts to hit the puck into our own goal unintentionally.

There was also imperfect information. The controller has no reference of the 3-D location of the puck, just a 2-D location and an estimated distance of the puck. This version of the controller was not using these elements to infer the 3-D location of the puck as described in later sections.

In further exploration work for the simple controller, there was an effort made to address some of the issues we saw. To address the reversing endlessly issue, we added some logic that alternates between going toward the center of the field and then after a set number of frames start reversing. A second improvement was to make the karts go forward for a couple of seconds when the game starts regardless of what the kart sees so that the kart can have a greater chance of seeing the puck. Despite these changes, it did not seem to improve the end results of the game very much from what was seen in the first time we tested the controller.

4.2. Converting from Screen Coordinates to World Coordinates

To improve the controller, we found it would be useful if we were able to convert the object detection from (x,y) screen coordinates to estimated world coordinates. To do so, we first used a heuristic distance-based approach. As this was unsuccessful, we then used a technique to “un-project” screen coordinates to global coordinates.

4.2.1. A Heuristic Distance-based Approach

Given that our model predicted euclidean distance, we first attempted to use that euclidean distance to estimate global coordinates. Since we had the kart’s global coordinates, the kart’s orientation, the predicted euclidean distance of the puck, and the predicted screen coordinates of the puck, we deliberated on methods to estimate the puck’s global coordinates. However, figuring out an accurate method for this proved challenging, as it was difficult to figure out how much of the distance of the puck was along the x-axis, and how much of the distance of the puck was along the z-axis. Next, we explored a geometric method for estimating global coordinates.

4.2.2. Un-projecting Screen Coordinates to Global Coordinates

Instead of going from a world point to a screen which is what the projection and view matrices do, our approach was to do the opposite of this and take the screen point and calculate a world point.

To do this, we do the following steps[10]:

1. Invert the y coordinate. This is necessary since the screen y-axis goes in the opposite direction as the 3-D y-axis.

[10] https://dondi.lmu.build/share/cg/unproject-explained.pdf

2. Convert the 2-D screen coordinate to a 4-D homogenous coordinate on the near (*z*=0) and far (*z*=1) plane of the view box. Then multiply by the inverted product of the projection and view matrix to convert them from the view space to the world space.

$$\begin{bmatrix} x_{near} \\ y_{near} \\ z_{near} \\ w_{near} \end{bmatrix} = (Proj \times View)^{-1} \times \begin{bmatrix} x_{screen} \\ -y_{screen} \\ 0 \\ 1 \end{bmatrix}$$

Equation 2: Screen coordinates to near plane

$$\begin{bmatrix} x_{far} \\ y_{far} \\ z_{far} \\ w_{far} \end{bmatrix} = (Proj \times View)^{-1} \times \begin{bmatrix} x_{screen} \\ -y_{screen} \\ 1 \\ 1 \end{bmatrix}$$

Equation 3: Screen coordinates to far plane

3. Divide the points on the near and far plane by *w* to convert them to 3-D cartesian coordinates.

$$\begin{bmatrix} x'_{near} \\ y'_{near} \\ z'_{near} \end{bmatrix} = \begin{bmatrix} x_{near} \\ y_{near} \\ z_{near} \end{bmatrix} / w_{near}$$

Equation 4: Near plane conversion from homogeneous to cartesian coordinates

$$\begin{bmatrix} x'_{far} \\ y'_{far} \\ z'_{far} \end{bmatrix} = \begin{bmatrix} x_{far} \\ y_{far} \\ z_{far} \end{bmatrix} / w_{far}$$

Equation 5: Far plane conversion from homogenous to cartesian coordinates

4. With these two points we have a ray intersecting our object while passing through the near and far plane. Since our world is flat, we know the depth at which it intersects is when the y-component is equal to the object height.

$$y_{object} = y'_{near} + u * (y'_{far} - y'_{near})$$

Equation 6: Calculation of depth in view box

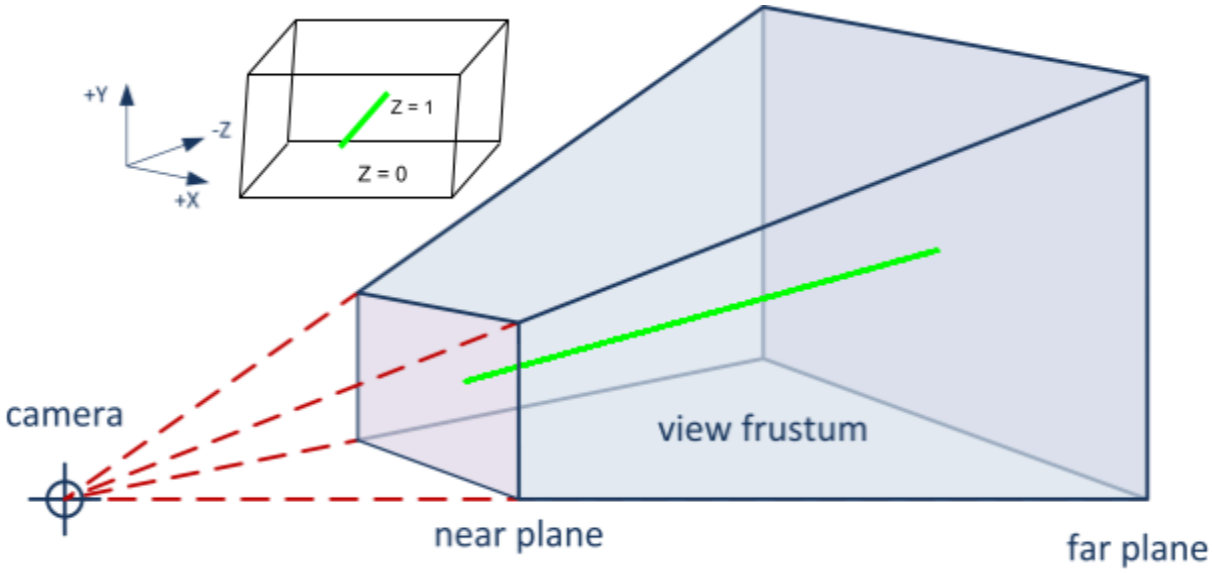

Figure 6: Ray passing through the view frustum[1] and view box

5. We rearrange this equation to find the value of *u* that corresponds to the point where the ray intersects the plane to get the *x* and *y* coordinates at that point

$$\begin{bmatrix} x_{puck} \\ y_{puck} \\ z_{puck} \end{bmatrix} = \begin{bmatrix} x'_{near} \\ y'_{near} \\ z'_{near} \end{bmatrix} + u * \left(\begin{bmatrix} x'_{far} \\ y'_{far} \\ z'_{far} \end{bmatrix} - \begin{bmatrix} x'_{near} \\ y'_{near} \\ z'_{near} \end{bmatrix} \right)$$

Equation 7: Combination of near/far points to find object position

4.3. An Advanced Controller Strategy

In our second approach, we attempted to remedy the issues in our first approach by estimating the puck's 3-D position in the world based on the 2-D image passed through the player state. With a better estimate of the kart's position relative to the puck and goal, we could design a new strategy to outperform the previous controller. To do so, we used the 3-D coordinates of the kart in the player state and the estimated 3-D coordinates of the puck to direct the kart to the optimal place to hit the puck.

4.3.1. Image Processing

Every time a team's act method is called, several variables are computed and cached to aid the controller in making decisions. Primarily, the puck detections are converted to world coordinates. If one player is unable to perceive the puck, the other player can provide the puck coordinates. These coordinates are then stored for velocity computations in the next frame. The velocity values are averaged over the last 5 frames, with more weight assigned to the latest observations. Additionally, we retain information on whether we are attacking the goal at the negative or positive end of the rink.

The locations of the puck, goal, and kart are crucial in determining our next move. We use this information to calculate a couple of vital angles. First, we measure the angle between the puck, goal, and kart, as illustrated in Figure 7(a). This helps us determine our kart's position relative to the puck and goal. An angle of 180 degrees indicates that we are directly lined up behind the puck, facing the goal, while 0 degrees means we are behind and directly in line between the goal and puck. The second angle we calculate is between the direction in which our kart is facing and the puck, shown in Figure 7(b). A value of 0 degrees means the kart is facing the puck directly, while 180 degrees means it's directly behind the kart.

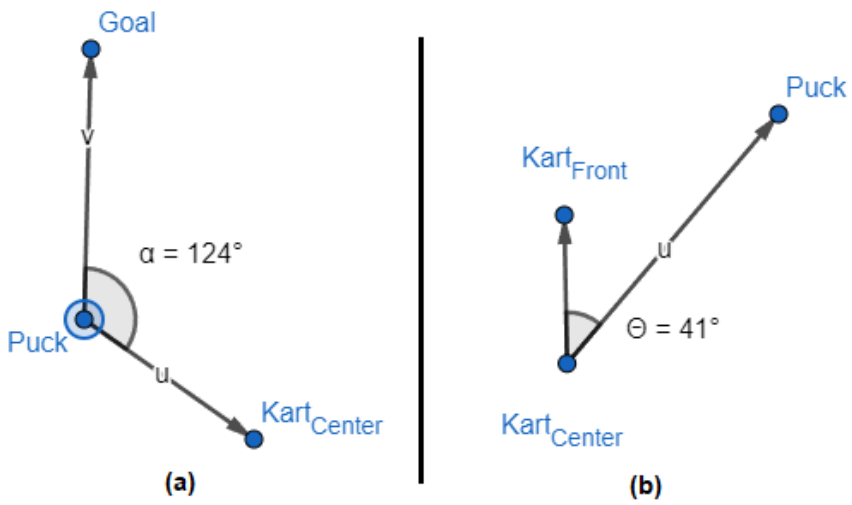

Figure 7: Kart Puck Positioning Angles

4.3.2. Kart Control

In each frame, the kart heads towards a 3-D point using the move_to_3D_point function. To steer, the puck-kart angle is used with a heavy-weight factor applied, resulting in values of -/+ 1 unless the kart is facing its objective directly.

The kart's acceleration is based on the difference between its speed and set_speed, which is the ideal speed for turning. The target_speed variable is passed to the function to determine the maximum speed for the current mode and state. If the angle between the kart and the objective point is over 0 degrees, a damping factor is applied to reduce the target speed. A minimum speed is then set to ensure the kart can always turn and avoid getting stuck. The complete formula for set_speed is given by:

$$set\_speed = (1 - \theta/90)^2 * target\_speed + min\_speed$$

Equation 8: Formula for setting speed of move_to_3D_point

If the kart is traveling 150% of the set_speed, it also applies the brakes.

To prevent karts from interfering with each other, a kart will set acceleration to zero if they get within 5 units of each other.

To ensure we push the puck towards the goal, we aim at the point on the puck opposite of our opponents goal. This is calculated by finding the vector from the puck to the goal then normalizing and inverting it. The vector is then scaled by radius of the puck so the

point is on the outside of the puck as shown in Figure 8 below. This prevents the kart from pushing through the puck to get to the point.

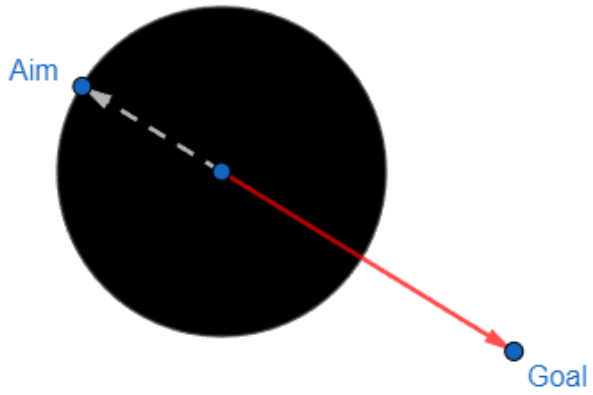

Figure 8: Optimal aim point on puck

4.3.3. Mode Selection

The kart's course of action is determined by analyzing the player state and image values. The decision-making process explained below varies based on the values computed at each step. However, if the 'hold_mode' variable is activated, the selection process is overridden and the kart remains in its current mode. A flow chart of the mode selection logic is shown in Section 4.3.2.5.

4.3.2.1. Start Mode

Figure 9: Kart in start mode accelerates forward.

The karts default to “Start Mode” at the beginning of each match. It’s also activated when the kart’s direction vector changes with a zero velocity, which indicates a goal has been scored and it’s about to reset. In this mode, the kart travels straight with max acceleration for 10 frames. This allows us to get close to the puck before relying on the model for detection.

4.3.2.2. Find Ball Mode

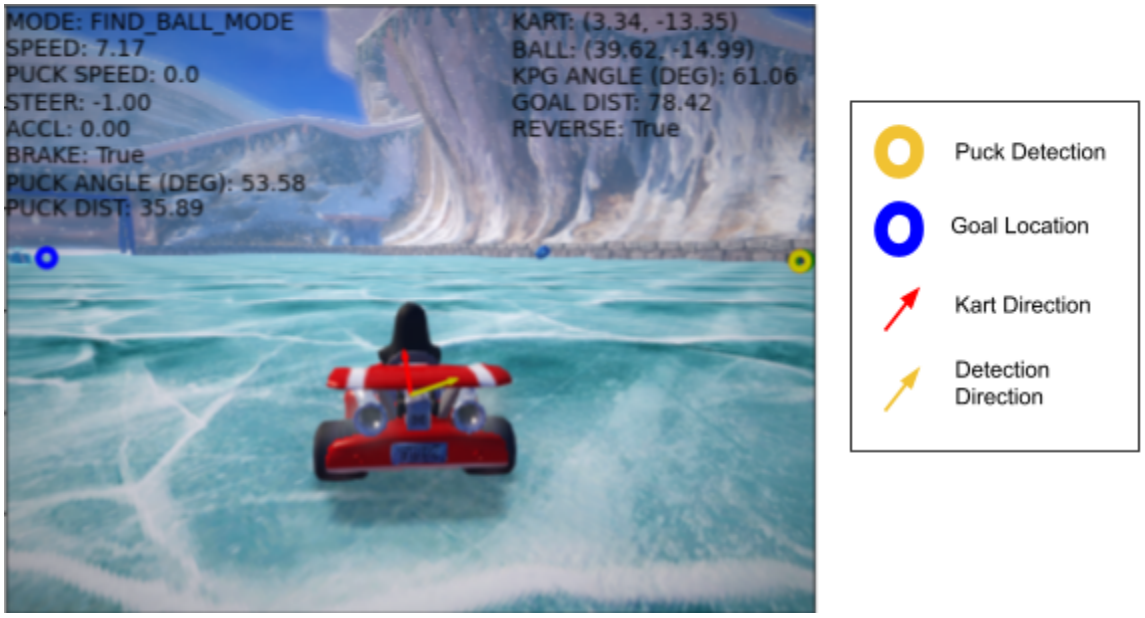

Figure 10: Find Ball mode. Reverses until puck is directly in front.

When neither player can detect the puck and its location from the last frame is unknown, the kart enters the “Find Ball” mode.The kart sets acceleration to zero and applies the brake to begin reversing. The kart will continue turning in the same direction it was before it lost the puck. If the puck’s location gets updated while in this mode, including detections from the other kart, it’ll use that direction instead. Once the kart is stopped and begins moving in reverse (determined by the dot product of the kart’s velocity and direction), the steering direction swaps. We stay in this mode until the kart is facing the puck (puck-kart angle < 30 degrees).

4.3.2.3. Intercept Mode

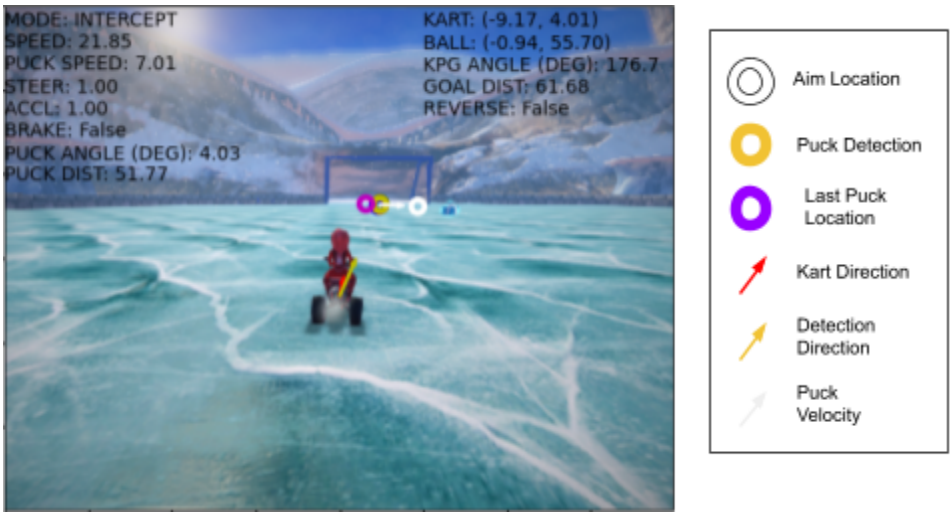

Figure 11: Kart heading in intercept direction

In “Intercept” mode, the kart aims to intercept the puck when it isn't directly behind it, defined as the kart-puck-goal angle less than 150 degrees and less than 5 units away. To do this, it uses the puck's position and velocity, along with its own position and speed, to predict the time of intercept. The formula[11] for the intercept time is show below.

$$0 = (||\overrightarrow{v_{puck}}||_2^2 - v_{kart})t^2 + 2(\overrightarrow{v_{puck}} \cdot (x_{puck} - x_{kart}))t + ||x_{puck} - x_{kart}||_2^2$$

Equation 9: Intercept time for kart and moving puck

Once the time is known, the kart calculates the position of intercept and its vector, based on the puck's initial position and velocity.

$$x_{intercept} = x_{puck} + t * v_{puck}$$

Equation 10: Intercept location

Figure 12 provides a diagram of this process.

[11] https://stackoverflow.com/questions/29919156/calculating-intercepting-vector

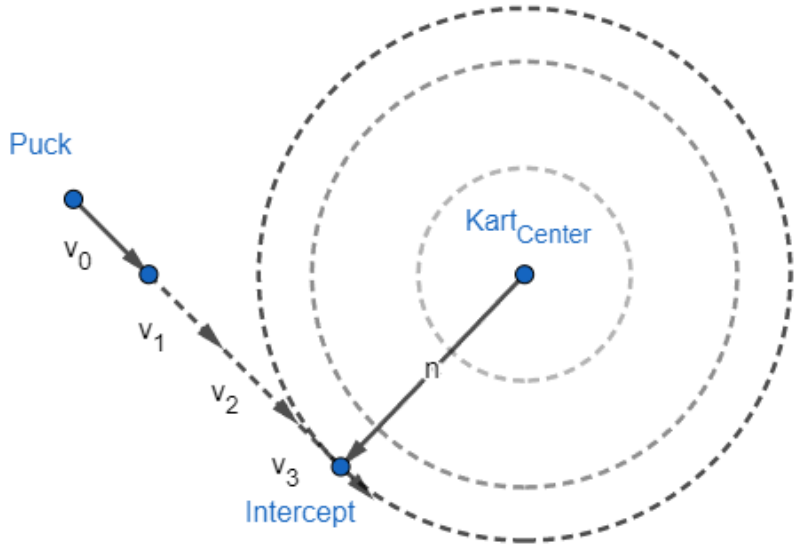

Figure 12: Intercept time, location, and vector diagram

In this mode, the puck also tries to get around the puck if the kart-puck-goal angle is less than 90 degrees. To do so it adjusts the aim point in the direction of the vector perpendicular between the kart and puck as shown in Figure 13. The closer the kart-puck-angle is to 0 degrees, the more it pushes the aim point in that direction.

If the angle is greater than 90 degrees, it adjusts the aim point further past the goal-puck vector. As the kart-puck-goal angle approaches 180, the magnitude of this adjustment goes to zero as seen in Figure 14.

The target speed in this mode decreases as a function of its distance to the puck. It's defined as the minimum between the max speed of the kart and the distance to the puck.

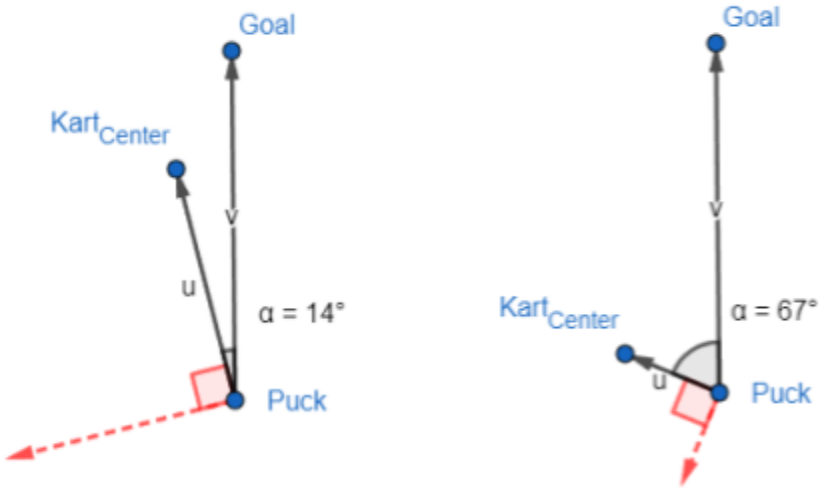

Figure 13: Aim point adjustments when between goal and puck.

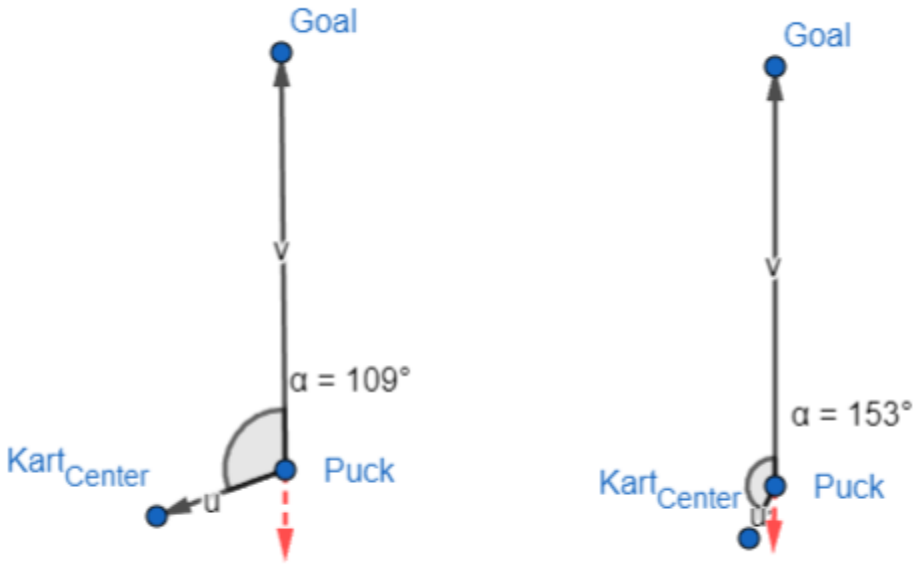

Figure 14: Aim point adjustments when behind puck

4.3.2.4. Dribble Mode

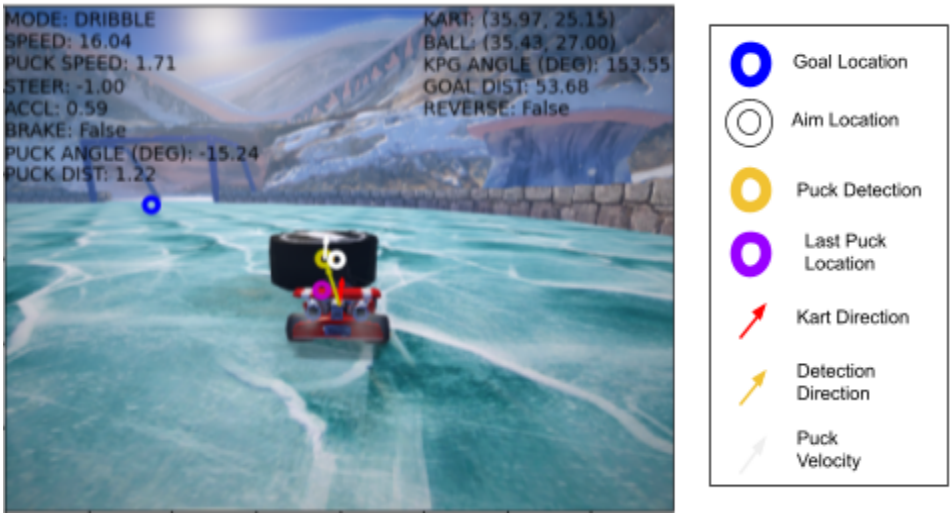

Figure 15: Dribble mode activate when behind puck

When the kart-puck angle is below 30 degrees and the puck is less than 5 units away, the kart enters dribble mode, signaling control of the puck. The kart steers directly toward the point behind the puck to push it toward the opponent's goal. The target speed in this mode depends on the distance to the goal and angle behind the puck, and slows down as it approaches or loses alignment with the goal. One downside is that slowing near the goal puts the puck in a dangerous position in the event we're unable to score.

4.3.2.5. Mode Diagram

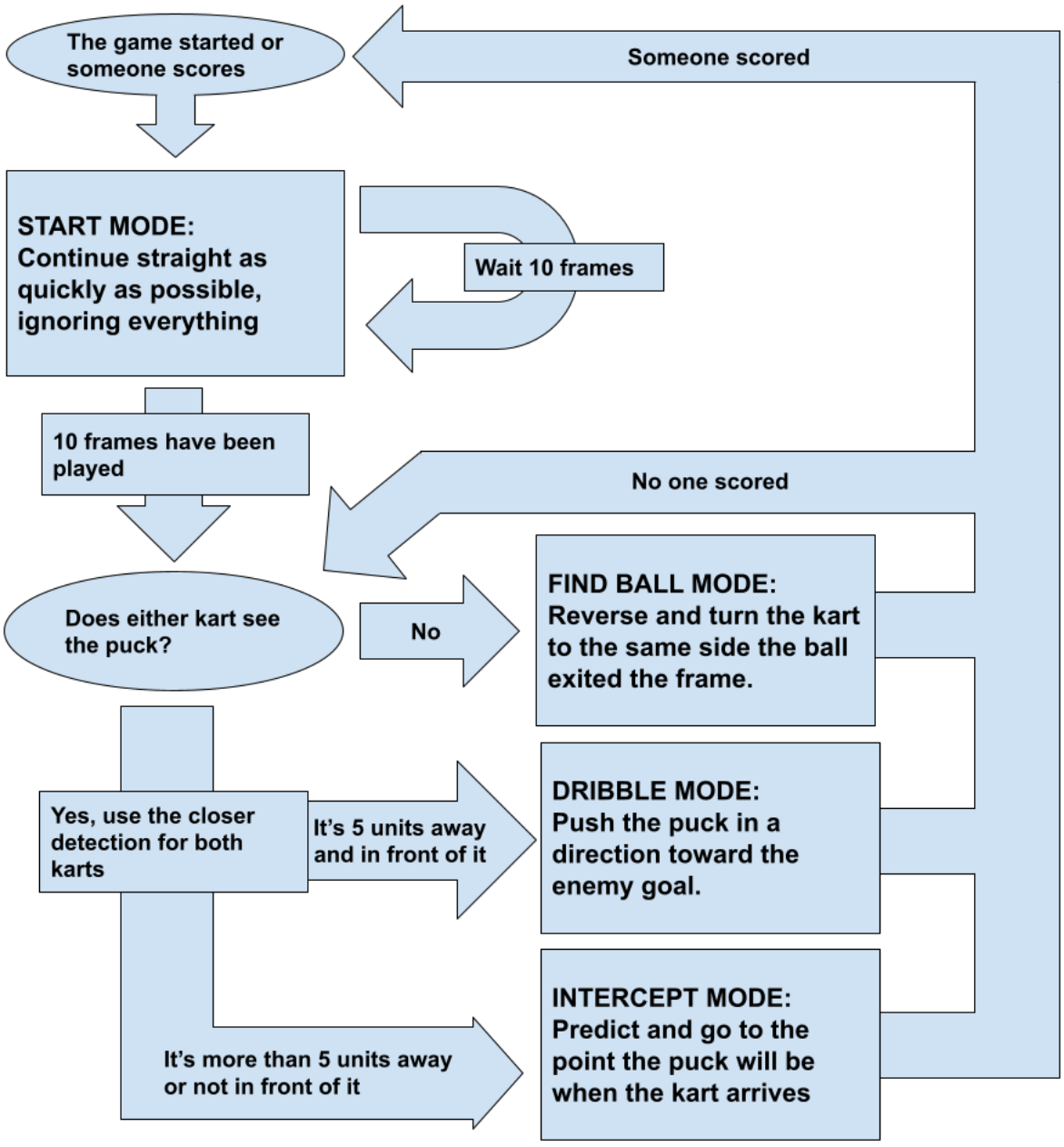

Figure 16: Mode selection flowchart

4.3.2.6. Evaluation

We first ran this controller and model combination on the local grader to determine the compatibility with the online grader. When run against the online grader, the first strategy scored an average of 1.13 goals per match.

Opponent	**Game 1**	**Game 2**	**Game 3**	**Game 4**	**Total**
Geoffrey	0:0	2:0	2:0	1:0	5:0
Jurgen	2:0	1:2	2:1	0:1	5:4
Yann	1:1	1:1	0:1	0:2	2:5
Yoshua	3:0	1:0	0:0	2:1	6:1
Average Goals		1.13:0.63	**Total Goals**		18:10

Table 4: Advanced Controller Scores. Scores are in the format of: “Our Score:Opponent Score”

This was a significant improvement as compared to the 0.44 goals scored by our first controller, and demonstrated the effectiveness of the additional strategies implemented as part of our second controller.

4.3.2.7. Controller Limitations

While trying to gain control of the puck, this kart loses speed and gives up control to the opponent. This was obvious when analyzing videos against speedy agents like Jurgen. Our controller would often lose control of the puck and not be able to regain it because it slows down as it nears the puck.

Additionally, when the kart approaches the goal from the side, it encounters difficulty scoring. Despite traversing the mouth of the goal, it fails to propel the puck into the net. Our belief is that this flaw stems from aiming for the center point of the goal rather than utilizing both posts.

Section 5: Conclusions and Further Work

Based on our ML model evaluation criteria, our model did a satisfactory job detecting the puck with a high degree of precision. The challenges mostly lied in hand-tuning the controller to be able to accurately use those predictions to score the puck. However, improvement is still possible.

To enhance performance, we could modify the model architecture and player roles. Currently, our model predicts only the puck's location and distance, but we could expand it to cover items and the opponents' location too. Nitro gives an edge when the match restarts and the AI agent scores many goals using this tactic. Detecting opponents' positions would steer the puck away from them and toward the goal.

Making temporal predictions is another improvement. To do this, we must increase the model input to accept multiple frames and include object velocity in the labels. This would enable predictions beyond a single frame, like the puck rolling around the corner and in front of the goal.

Besides the model architecture, players need different strategies. Often they got in each other's way and added little value. One solution is to assign a player to defend and stay near the rink's back half. Another is to have only one player attack while the other provides support in case the attacker loses control.

Appendix A: Torchinfo Report of Model Architecture

Layer (type:depth-idx)	Output Shape	Param #
Detector	[1, 2, 300, 400]	--
├─Block: 1-1	[1, 16, 150, 200]	--
│ └─Conv2d: 2-1	[1, 16, 150, 200]	448
│ └─BatchNorm2d: 2-2	[1, 16, 150, 200]	32
│ └─Conv2d: 2-3	[1, 16, 150, 200]	2,320
│ └─BatchNorm2d: 2-4	[1, 16, 150, 200]	32
│ └─Conv2d: 2-5	[1, 16, 150, 200]	2,320
│ └─BatchNorm2d: 2-6	[1, 16, 150, 200]	32
│ └─Conv2d: 2-7	[1, 16, 150, 200]	64
├─Block: 1-2	[1, 32, 75, 100]	--
│ └─Conv2d: 2-8	[1, 32, 75, 100]	4,640
│ └─BatchNorm2d: 2-9	[1, 32, 75, 100]	64
│ └─Conv2d: 2-10	[1, 32, 75, 100]	9,248
│ └─BatchNorm2d: 2-11	[1, 32, 75, 100]	64
│ └─Conv2d: 2-12	[1, 32, 75, 100]	9,248
│ └─BatchNorm2d: 2-13	[1, 32, 75, 100]	64
│ └─Conv2d: 2-14	[1, 32, 75, 100]	544
├─Block: 1-3	[1, 64, 38, 50]	--
│ └─Conv2d: 2-15	[1, 64, 38, 50]	18,496
│ └─BatchNorm2d: 2-16	[1, 64, 38, 50]	128
│ └─Conv2d: 2-17	[1, 64, 38, 50]	36,928
│ └─BatchNorm2d: 2-18	[1, 64, 38, 50]	128
│ └─Conv2d: 2-19	[1, 64, 38, 50]	36,928
│ └─BatchNorm2d: 2-20	[1, 64, 38, 50]	128
│ └─Conv2d: 2-21	[1, 64, 38, 50]	2,112
├─Block: 1-4	[1, 128, 19, 25]	--
│ └─Conv2d: 2-22	[1, 128, 19, 25]	73,856
│ └─BatchNorm2d: 2-23	[1, 128, 19, 25]	256
│ └─Conv2d: 2-24	[1, 128, 19, 25]	147,584
│ └─BatchNorm2d: 2-25	[1, 128, 19, 25]	256
│ └─Conv2d: 2-26	[1, 128, 19, 25]	147,584
│ └─BatchNorm2d: 2-27	[1, 128, 19, 25]	256
│ └─Conv2d: 2-28	[1, 128, 19, 25]	8,320
├─UpBlock: 1-5	[1, 128, 38, 50]	--
│ └─ConvTranspose2d: 2-29	[1, 128, 38, 50]	147,584
├─UpBlock: 1-6	[1, 64, 76, 100]	--
│ └─ConvTranspose2d: 2-30	[1, 64, 76, 100]	110,656
├─UpBlock: 1-7	[1, 32, 150, 200]	--
│ └─ConvTranspose2d: 2-31	[1, 32, 150, 200]	27,680
├─UpBlock: 1-8	[1, 16, 300, 400]	--
│ └─ConvTranspose2d: 2-32	[1, 16, 300, 400]	6,928

```
├─Conv2d: 1-9                            [1, 2, 300, 400]          40
├─Conv2d: 1-10                           [1, 1, 300, 400]          20

Total params: 794,988
Trainable params: 794,988
Non-trainable params: 0
Total mult-adds (G): 3.48

Input size (MB): 1.44
Forward/backward pass size (MB): 82.29
Params size (MB): 3.18
Estimated Total Size (MB): 86.91
```