Implementing an Automated SuperTuxKart Ice Hockey Player

Divya Thomas
University of Texas at Austin
divyasthomas@gmail.com

Abstract

This paper details the design and implementation of an AI agent to play ice hockey in pyTuxKart, which is a heavily modified version of the SuperTuxKart game designed for teaching purposes. This paper details the motivation, methods, evaluation metrics, and results obtained by my trained agent against benchmark agents known as TA Agents that were bundled along with the game environment. The final agent was an image-based agent that generated player actions to score goals against opponent teams based on the input image as seen by the agent and the agent state. The final image-based agent described in this article was able to achieve a score of 76% against the benchmark agent.

Keywords: ***Deep Learning, Computer Vision, CNN, Reinforcement Learning***

1. Introduction

SuperTuxKart is a 3D open-source arcade kart racing game with a variety of characters, tracks, and modes to play (Krähenbühl). In this paper, I discuss the design and implementation of a SuperTuxKart ice-hockey automated agent and benchmark its performance in SuperTuxKart Ice Hockey games against similar AI agents. Two kinds of agents could be built, namely, an image-based agent and a state-based agent. While both agents had the same objective of scoring as many goals and winning the match in a 2 vs. 2 tournament, they had differing inputs and evaluation metrics. This paper describes the implementation and design of an agent to compete with the TA agents, and the various design decisions and methods used to build the same.

This paper is relevant to Deep learning research as it leverages state-of-the-art techniques such as object detection using CNNs (O'shea & Nash, 2015) and U-Nets (Ronneberger et al.,2015), learning state-action space mapping via, DDPG (Lillicrap et al., 2015), and TD3 (Fujimoto et al., 2018) algorithms, and their specialization to solve an object locomotion task in the context of a game. It also demonstrates the performance of neural network-based agents in noisy and non-deterministic game engine scenarios, similar to research by peers in the field such as AlphaStar (Vinyals et al., 2019), SIMA (Raad et al., 2024), and other convolution or reinforcement learning-based game agents.

2. Motivation

2.1 Motivation to attempt state-based agents

Initially, I decided to create a state-based agent. This was motivated primarily because having the full knowledge of the player, puck, and opponent state in a zero-sum game suggested that the model only needed to learn the limited set of control inputs (7x1 vector) based on the state inputs (which were designed as a 21x1 vector) to win the game. Moreover, I hypothesized that the upper bound for improvement of such an agent was very high since I could improve the agents further by running more iterations of the learning agent against itself once it surpassed the TA agents.

This approach drew inspiration from the fact that algorithms like Alpha Go (Silver et al., 2016), Alpha Zero (Silver et al., 2017), and Alpha Star (Vinyals et al., 2019) have shown amazing evidence of learning novel strategies and achieving high performance by playing against themselves in such a manner.

2.2 Challenges encountered in state-based approach

In practice, however, the designed state agents weren't converging on a winning strategy even after 2000 2v2 games, and I realized that state agent training times often increase exponentially with the complexity of the action space. This was exacerbated by the physics-based dynamics of the Pytuxkart game engine, which meant that the state-action space was more complex than I initially envisioned. This coupled with the limitations of computing resources I had since I was competing in the project as an individual led to a late-stage pivot in strategy to use an image-based agent.

2.3 Motivation for the final image-based agent

Image-based agents, by the rules of the tournament, did not have complete knowledge of the state of all the players and the puck. Counter-intuitive to the fact that the information available to the agent was significantly reduced by this approach as compared to state-based agents, it offered an opportunity to leverage computer vision techniques such as Convolutional neural networks (CNNs) (O'shea & Nash, 2015) that have been shown to have remarkable reliability in their predictions. Furthermore, the rules of the game allowed us to write a handcrafted heuristic rule-based controller on top of a robust prediction method, which eventually led to superior performance overall within the time and resource constraints afforded in this project.

3. Methods

In this section, I will first describe the MLOps pipeline I used to design and iterate over multiple agents in parallel during the development of the gameplay agent.

I will then describe in detail the method that helped me to build a highly performant image-based agent. After that, I will also mention alternate methods and agents I iterated over in the quest toward building the final agent.

3.1 MLOps pipeline for development

The development of this project followed the standard MLOps iterative design and testing process outlined in Figure 1.

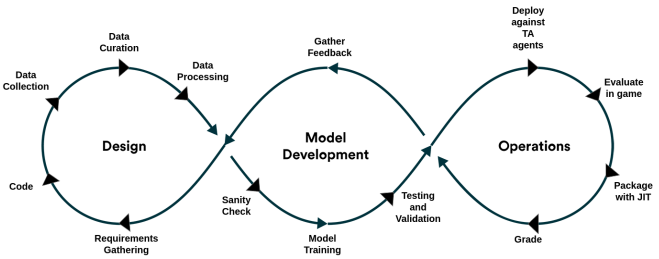

Figure 1: MLOps pipeline for this project

After modifying the tournament runner infrastructure to be conducive to an iterative design and validation process, data was collected and curated to generate a high-quality training and validation dataset. After that, the curated dataset was transformed from state space to image space to prepare a representative training set for the puck detection neural network. A few initial epochs of training were done to establish baseline behaviors such as decreasing loss exhibited by the candidate model. The model was allowed to train further only if these baseline sanity checks were passed to minimize computational cost excesses.

Once the models trained to maturity around 50 epochs, I evaluated the network performance on a representative game task such as puck detection (in case of image agent), or navigation to puck (in case of state agent) respectively. If the network exhibited robustness on the representative task, downstream systems such as the heuristic controller were developed based on the model, and the final combination of network + controller was deployed as an agent to test against the TA agents in the tournament setting. Automated and manual evaluations were done on the performance via monitoring and metrics which generated insights that went

into the iterative step of data collection, curation, and retraining. The above MLOps cycle was repeated a couple of times until the model was considered sufficiently robust to be packaged as a serialized and optimized JIT model and it was then released to be evaluated on the local and online grading systems.

3.2 Image-based agent Design

Implementing an image agent consists of multiple steps as outlined in the previous section. After curating a good set of representative data using a tournament runner repurposed as a data collection engine, I built an object detection deep convolutional network to detect the puck in a given image generated by the game engine. I coupled this network with a hand-tuned controller that used the predicted location of the puck on the screen and the state of the player's kart as inputs to navigate the game and score goals. The below subsections offer a detailed explanation of the design for various parts of the system.

3.2.1 Input parameters

An image-based agent was given access to the state of the players' karts and could also see an image for each player. It was not given access to the state of the puck or opponents during gameplay. The input consisted primarily of the following components:

1. A 3-channel 400x300 (width x height) matrix describing the image as seen by the camera following the kart.
2. Projection and view matrices for the camera with respect to the world coordinates of the game
3. Kinematic information for the kart such as kart position, rotation (determined via the kart front and kart center coordinates), and velocity with respect to the world coordinates of the game.

3.2.2 Data generation

I used a heavily modified version of the tournament runner to gather training data from the game. During this stage, the specifics of which agent played against which opponent were more relevant for the on-policy DQN-based (Mnih et al., 2015) state agents but less relevant for the off-policy state agents and the final image-based agents. For data collection for the imaging agent, to develop robustness against false positive detection of the puck when completely or partially occluded by different opponent agents, I randomized the karts used by the in-game AI and the TA agents in every game. Since the TA Agents and the in-game AI were all state agents, the physical appearance of the agents did not impact their performance, but a varied and representative collection of images was gathered. In the final iterations of the MLOps pipeline, the dataset was also seeded with a bias favoring the selection of the karts '*nolok*'

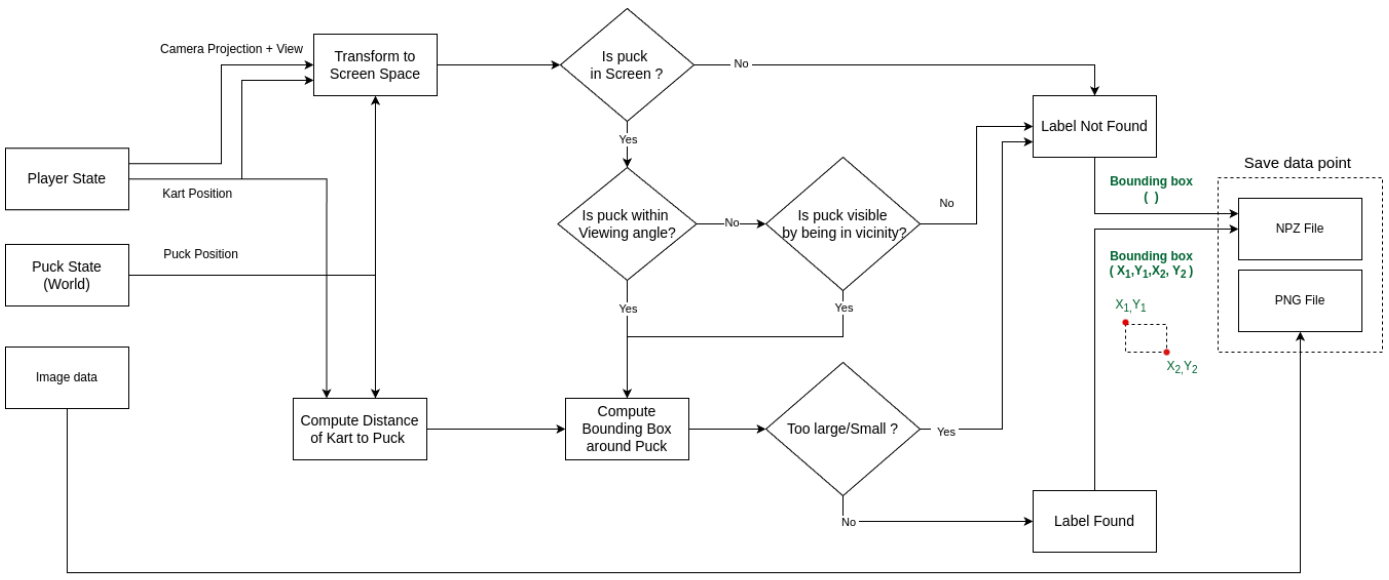

Figure 2: Dataset Processing FlowChart

and '*wilber*' as the player kart in the image as these were the karts used by the model's agent and equal probability of all karts in the opponent team. This was done to force the vision model to learn to ignore the player's kart which would be the main occupant of the lower-middle portion of the screen space in the final agent's image.

I gathered images and world-states of six games played by four randomly selected AI agents for 1200 frames per game, totaling 30,000 frames for training and 7000 frames for the validation dataset.

At every time step, I stored a numpy vector of world-state objects (Harris et al., 2020) from which I extracted ground truths for the vision task for curated images. The requirement for this stage was to accurately label the bounding box over the puck in case of an image where the puck was visible, and create an empty list for the bounding box vector in case the puck was not visible in the screen.

This post-processing was significantly optimized to reduce the need for manual sanitization of the generated data, due to project deadlines. The method of collecting and curating the data consists of the following checks as shown in Figure 2. The robustness of this automated dataset processing proved remarkable, with an error rate of less than 1 in 100 images being inaccurately labeled.

3.2.2.1 COMPUTING PUCK VISIBILITY

The location of the puck from the world coordinates to the screen coordinate was calculated by matrix multiplying the project and view matrices with the world coordinates and normalizing them. My filtering algorithm initially determines if the first coordinate in the normalized vector is between (-1,1) and the third coordinate of V_{view} is > 0 in which case the puck is considered visible. The normalized position of the puck was then scaled to screen coordinates and the origin was shifted from the coordinate system at the center of the image to the corner to match the screen information stored in the saved image's numpy serialization. Furthermore, The orientation of the kart was used to determine if the ball was behind the player.

The following equations are used to convert a point from world coordinates (W) to screen coordinates (S). The projection and view matrices for the camera P_{proj} and V_{view} are available in the game's state engine for each player kart.

$$S_x, S_y, S_z, S_w = P_{proj} \times V_{view} \times [W_x, W_y, W_z, 1] \quad (1)$$

$$S_{x_{Norm}}, S_{y_{Norm}} = [\frac{S_x}{S_w}, \frac{S_y}{S_w}] \quad (2)$$

$$X_S = \frac{width}{2} \times (1 + S_{x_{Norm}}) \quad (3)$$

$$Y_S = \frac{height}{2} \times (1 + S_{y_{Norm}}) \quad (4)$$

The sizing for the bounding box for the puck was calculated by a heuristic equation shown below:

$$p_{scale} = \frac{120}{(1 + Kart{-}to{-}Puck_{dist})^{1.21}} + 2 \quad (5)$$

$$Bx_{t-l}, By_{t-l} = [p_x - (p_W \times p_{scale}), p_y - (p_H \times p_{scale})] \quad (6)$$

$$Bx_{b-r}, By_{b-r} = [p_x + (p_W \times p_{scale}), p_y + (p_H \times p_{scale})] \quad (7)$$

Where,

B = Bounding box and p = Puck,

$x_{t\text{-}l}$ and $y_{t\text{-}l}$ are the x,y coordinates of the top-left corner of the bounding box and $x_{b\text{-}r}$ and $y_{b\text{-}r}$ are the x,y coordinates of the bottom-right corner of the box shown in Figure 2 in green.

p_W = 2 and p_H = 1 define the aspect ratio for the rectangle of the puck bounding box

Pucks that were plotted above the horizon, or in the air (calculated by drawing a line using the goal coordinates, and z of puck world state) and with a distance greater than 60 units were considered too small to be an accurate detection and empty bounding boxes were used in these cases.

Finally, the viewing angle was calculated as the angle between the kart and the puck and was limited to a 120-degree FOV, and any pucks on the edges of the screen within a 2-pixel border were filtered out.

You can see a sample of the results of the automated pipeline with the generated bounding boxes in Figure 3.

Figure 3: Sample detections from the data processing pipeline

A manual inspection of 100 randomly sampled ground-truth images showed that this method produced accurate puck bounding boxes and labels with 98% accuracy (2 false positives), which was sufficiently high-quality data for training a vision system for the image-based agent.

3.2.3 Model architecture

For my vision tasks, I used a U-Net (Ronneberger et al., 2015) backend with 4 convolutional blocks and 4 up convolution blocks. Each Convolutional block had 3 convolutional layers with Batch Normalization (Ioffe & Szegedy, 2015) on the output and ReLU non-linearity (Fukushima, 1969) between the layers. The blocks also have a skip connection between the input and the last Convolution feeding into the final ReLU non-linearity (Fukushima, 1969). This block structure was known to have a good feature detection power based on my previous experience and hence was chosen as the building block for the U-Net. My U-Net architecture consists of 4 blocks of these ConvBlocks, doubling channels in each block (16/32/64/128)

These are followed by 4 symmetric blocks of up-convolutions which upsample the activation maps (and halve the channels) to produce a single channel 300x400 heatmap with activations where the puck is present.

To improve convergence during training, I normalized the channels of the input image to the mean and standard deviation calculated over the entire training set. A diagram of the model architecture can be seen in Figure 4.

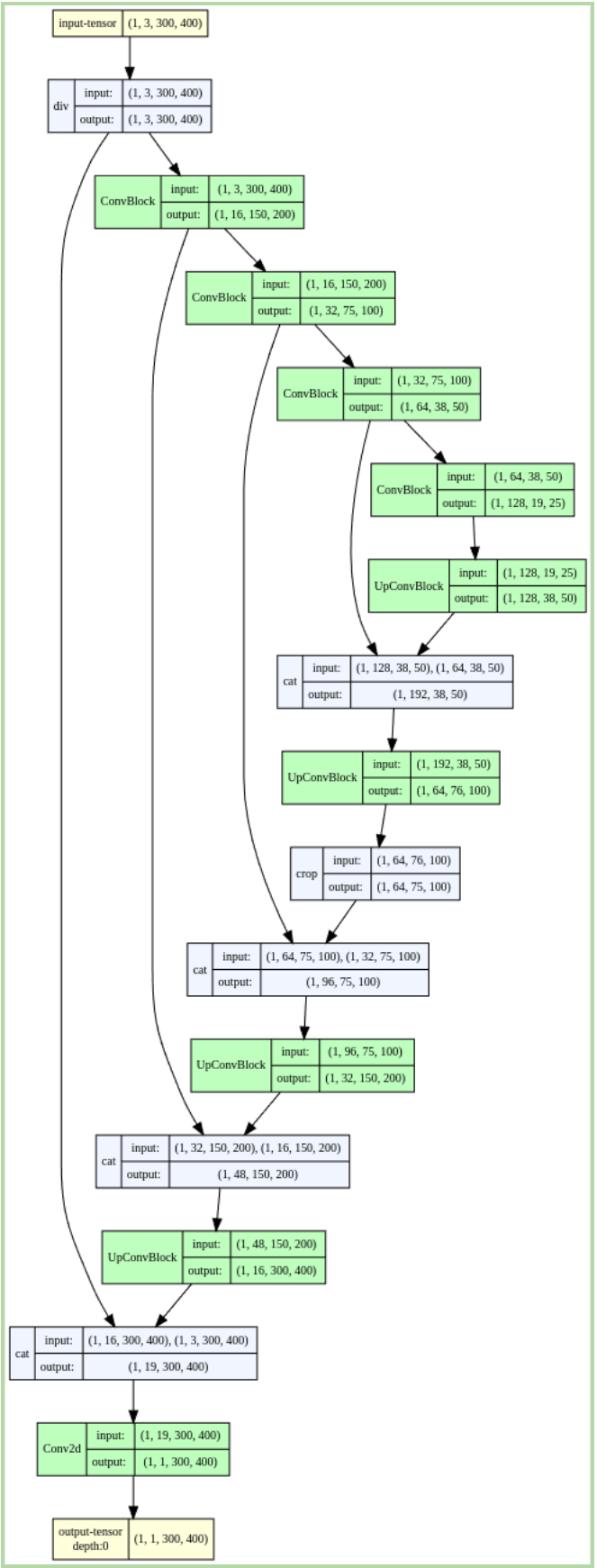

Figure 4: A diagram of the model architecture

3.2.4 Puck detection from heatmap

During the iterative development of the model it was also found that bounding boxes with slightly larger outlines exceeding the borders of the puck performed better for convergence as compared to tight borders around the puck as these would make the model mistake the hovercraft kart's skirt for a puck. This can be seen in Figure 5. The yellow circle indicates whatever the model detects as the puck. The hypothesis is that the ice arena around the puck has a particular pale blue and white palette which provides a nice contrast between the kart and puck which makes the detection better when training on the edges of the puck, whereas the hovercraft kart players will have the black skirt topped with either a blue or red hovercraft body.

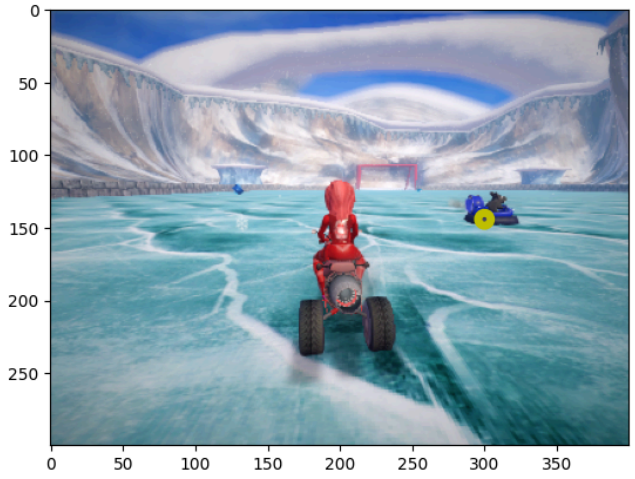

Figure 5: Example of incorrect detection by model

I performed a thresholding to detect the peaks in the heatmap. This task is well suited to a binary cross-entropy loss (I tested using Focal Loss, and RMSE loss but I found it to perform poorer as compared to the BCEWithLogits loss as the puck visibility is evenly distributed among the two classes in both train and test data sets). Another observation was that the model trained with color jitter (brightness, hue, saturation, and contrast) did not generalize well to the validation dataset. Again the hypothesis is that the color contrast between the puck and the surrounding ice rink becomes essential to identification even for the human eye and color jitter adds unnecessary noise that is not present in the image, so the final model was not trained using color Jitter.

3.2.5 Model training

The model was trained using a dataset of approximately 30,000 training images with a 60:40 split of images containing puck and those without, respectively. We found this ratio to yield the best results in terms of total convergence time and accuracy in detections. A total of 30 epochs were trained on the data with a batch size of 32. The best performance model used the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 0.005 and a weight decay of 0.000001 (1e-6). The network was trained using Pytorch with the BCEWithLogitsLoss loss function with the loss reduction option disabled. The average run of training for a single model took approximately 5 hours on a T4 GPU.

3.2.6 Controller

We handcrafted the controller based on a custom state machine comprising four major components.

1. A state variable calculator which provided the player with information about the state of the kart and the puck.
2. A set of replay buffers to store past kart states, actions, and past puck positions on the screen and in world coordinates respectively. This was inspired heavily by the state agent implementation which did not make it to the final image-based agent
3. A higher-level state checker with functions dedicated to inferring the state of the player kart such as Searching for Puck, Chasing puck, Attacking, etc.
4. Action functions to perform higher-level actions based on the state, such as chasing puck if the puck is visible on screen, escaping if inside a goal or stuck, etc.

3.2.6.1 STATE VARIABLE CALCULATOR

The state variable calculator performed low-level operations such as computing distance from kart to the puck, the angle between kart and puck, kart orientation to goal, kart velocity, and most importantly, computing the rough estimate of puck position in the world coordinates if it is visible in the screen

coordinate. The computation of the world coordinates (W) from the screen coordinate (S) for a point on the screen is detailed below. The projection and view matrices for the camera P_{proj} and V_{view} are available in the game's state engine for each player kart.

$$C_x, C_y, C_z, C_w = P_{proj}^{-1} \times [S_x, S_y, -1, 1] \quad (8)$$

$$C = C_{x_{Norm}}, C_{y_{Norm}}, C_{z_{Norm}} = [\frac{C_x}{C_w}, \frac{C_y}{C_w}, \frac{C_z}{C_w}] \quad (9)$$

Normalizing,

$$C_{unit} = \frac{C}{norm\,(C)} = [C_{x_{unit}}, C_{y_{unit}}, C_{z_{unit}}] \quad (10)$$

$$W'_x, W'_y, W'_z, W'_w = V_{view}^{-1} \times [C_{x_{unit}} \times d, C_{y_{unit}} \times d, C_{z_{unit}} \times d, 1] \quad (11)$$

$$W_x = \frac{W'_x}{W'_w}, W_y = \frac{W'_y}{W'_w} \text{ and } W_z = \frac{W'_z}{W'_w} \quad (12)$$

Where,

d = distance from the camera to a point

C = Camera

The calculation of screen-to-world coordinates is imprecise because a heuristic estimator is used for obtaining the depth field of the puck, and consequently the distance from the camera to the puck is also estimated. In my experiments, I found that the distance of the puck from the camera approximates a linear function of the distance of the puck center from the screen center (on the screen coordinates) as long as it is within a radius of 65 pixels from the screen center. This radius is considered a vicinity radius and is also used in the higher-level state checkers to determine if the kart is near the puck. With this method, I could approximate the position of the puck to within ±5 game units of the actual puck coordinates in the game as long as the puck was close to the screen center as mentioned above.

3.2.6.2 REPLAY BUFFER

Drawing inspiration from the state agent, a set of double-ended queue data structures was used to capture the past kart positions, player states, positions on the puck on screen, and past player actions respectively. This served as a memory to the controller that smoothed the outputs of the network and helped in eliminating outlier detections. For instance, the higher-level state checkers could use these buffers to understand when the position of the kart was reset, if the kart was stuck, or the optimal direction to turn while searching for the puck based on the last known puck location. The puck location buffer which stored the predicted last known location of the puck in world coordinates was shared by all instances of the player karts.

3.2.6.3 STATE CHECKER

On each step of the action function in the game, the state checker functions are invoked after the state variables are calculated. These functions suggest what is the next best action for the player based on the current state and this information is used to execute the player's subsequent action. The state checker and the action function (discussed in the next subsection) operate using a shared state-lock variable, which locks the action for a set number of turns to allow the agent time to complete an action without fluctuating too frequently between different suggested actions. Figure 6 shows the state machine of the kart controller with each state checker function (left side) and its suggested action (right side) shown as a tuple.

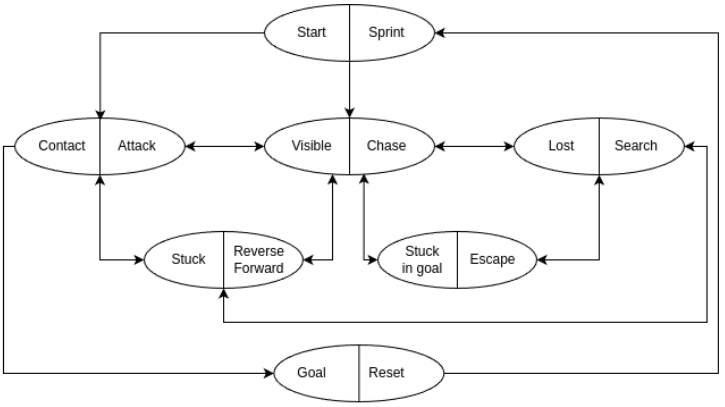

Figure 6: State machine of the kart controller

3.2.6.4 ACTION FUNCTIONS

The Action functions perform higher-level actions by providing the action output for a series of game steps. The hand-tuned controller has heuristics to determine the optimal acceleration and steering angle of the kart. The handling characteristics of the kart were defined as a DriveParams object that governed behaviors such as oversteer, understeer,

max acceleration, min and max velocity, drift angle, etc. The heuristic controller for the action functions relied heavily on motion planning approaches in robotics and used the third equation of motion in classical kinematics to compute the acceleration.

$$a = \frac{V_{target}^2 - V_{current}^2}{2 \times distance} \quad (13)$$

Where

a = acceleration

V = velocity

Steering angles were set by computing the angle between the target and current angle vectors and clipping it to max and minimum steering angles if it exceeded the steering cone of ±45 degrees. Due to the time constraints on the project, a fully accurate non-holonomic steering controller design based on the Ackerman steering (Agrawal et al., 2021) was attempted but subsequently skipped in favor of the above approximation.

At a high level, the following actions were implemented using this base controller. As discussed in the previous section every action acquired a state lock for `n` game steps based on the action characteristics.

1. Sprint to puck at the beginning and shoot the projectile at the last sprint step. The steering angles were hard-coded for each kart and acceleration was set to 1. This was locked to 25 steps for Player 0 and 45 steps for Player 1.
2. Chase puck, trying to maintain the puck center on screen. Currently, this is done only by using position control, however, a PID controller would be an enhancement in this action. This action was locked for 20 steps.
3. An attack action was defined to set the steering angle to θ as per Figure 7 if the puck was in the vicinity radius of the kart. This was locked for 5 steps of the game.
4. The search action was defined as taking the smallest radius circle with current velocity towards the last known puck position which was shared between the karts. State lock was for 2 steps before re-evaluation.
5. The Escape-from-Goal action was defined with pre-recorded steering and action inputs based on kart and goal position. This was locked for 20 steps as it was a hardcoded action.
6. Escape-if-Stuck action was defined as reverse if stuck facing forwards, accelerate forward if stuck driving reverse with a lock of 10 steps.

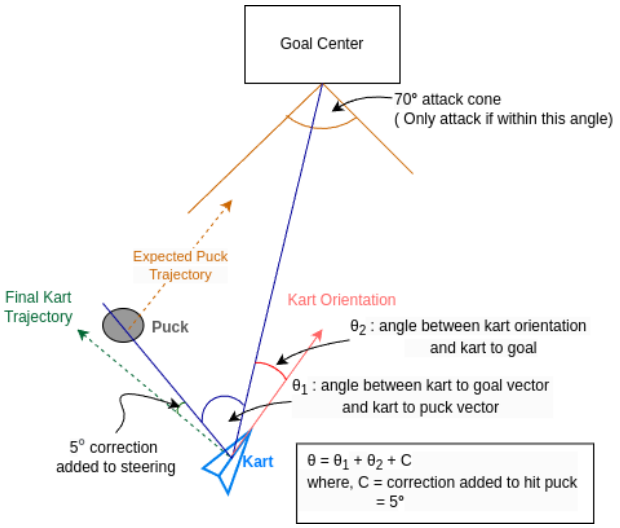

Figure 7: Calculation of steer angle for kart in Attack action.

3.3 State-based agent

As I mentioned in the introduction, I initially tried to implement a state-based agent. The steps I took for implementing a state-based agent consisted of first setting up a gym environment and then performing reinforcement learning using algorithms inspired by DDPG (Lillicrap et al., 2015), and TD3 (Fujimoto et al., 2018) algorithms multiple steps. In the sub-sections below, I go into some detail on each of these steps.

3.3.1 Input parameters for state agent

A state-based agent was given access to the state of the karts of both the players and opponents as well as the location of the puck. This meant that at any step of the gameplay, the network was able to use a tensor of the game state such as the position, orientation, and velocity of the player and opponent karts and the exact world coordinates of the puck. From these inputs, I was able to extract some basic features similar to the TA jurgen_agent using an identical feature

extraction function. Some of the higher level parameters extracted were: puck_center, kart_center, angle to puck from kart, angle to a goal from kart, the distance between the puck and goal line, etc.

3.3.2 Gym wrapper for PyTuxKart

To quickly iterate on various Reinforcement learning algorithms in the pyTuxKart game environment, I built a wrapper to convert the pytuxkart functions to the OpenAI gym specification. The key steps in this process were to create a PystkEnv class with the *reset, step, make,* and *sample* functions. Calling the *env.step()* function would then provide the *current_state*, *reward*, *next_state*, *done*, and *info* parameters for the reinforcement learning algorithm. This enabled me to quickly try various reinforcement learning algorithms in a standardized manner. Figure 8 showcases the gym wrapper's live in-game top-down view designed for ease of monitoring progress.

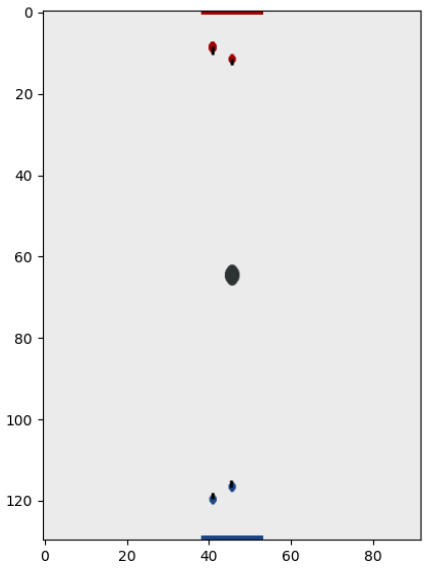

Figure 8: The live top-down view of the game arena.

3.3.3 Reinforcement Learning Models

Using the above framework, actor-critic approaches such as Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2015), and Twin Delayed DDPG (Dankwa & Zheng, 2019) were attempted. The actors for the networks comprised three fully connected linear layers with Layer Normalization (Ba et al., 2016) applied over the mini-batch inputs between layers.

These were passed through a ReLu non-linearity and followed by the $\boldsymbol{\mu}$ linear layer that was then converted the output to action space through a hyperbolic tangent function.

The critic networks had a similar architecture, with the Q linear layer producing the final state_action value. The hidden layer sizes were set to 400 and 300 respectively, with a batch size of 100, tau of 0.001, and gamma equal to 0.99. The learning rate for the actor was set to vary from 0.00025 to 0.025 in different experiments.

A warmup of 100 steps and a policy noise of 0.2 with a policy frequency of 8 was used for the Ornstein-Uhlenbeck Action Noise parameter ("Ornstein-Uhlenbeck process," n.d.). In the epsilon greedy approach, I used an exploration probability decay rate of 0.98.

4. Results

4.1 Model Evaluation Metrics

4.1.1 Model Evaluation Metrics for Image Agent

BCE with Logits Loss (Paszke et. al, 2019) was used to evaluate the performance of the classification models and tune hyperparameters. This loss combines the BCE Loss and a sigmoid layer into a single class. The unreduced loss we use in training the image-based agent can be defined as follows, where N = batch size, y is the ground truth, and $\widehat{y}$ is the prediction. This loss function takes advantage of the log-sim-exp trick for numerical stability

$$BCE = -\frac{1}{N}\sum_{i=0}^{N} y_i \cdot log(\hat{y}_i) + (1 - y_i) \cdot log(1 - \hat{y}_i) \tag{14}$$

The model with a lower value for loss was considered to be the better model. For the primary task of detecting the puck on the screen, we used the average precision (AP) metric, which sorts all the detections over the entire dataset by their confidence. It then measures precision and recall for puck detection. The average precision is computed as the mean of the precision at recall values ranging from 0 to 1 in 0.1-step increments. To determine a positive or negative detection, the distance between the object center and detection is checked to be ≤ 5 px, or if the detection of the puck lies

within the bounding box. Objects smaller than 15px area were ignored in this computation as puck detections at that distance were difficult to get fully right.

In addition to the computed metrics, manual inspection of the detections was done to estimate the true-ness of the detector over a series of games, and each model was given a score of 1-10 based on the quality of detections.

Finally, the top-scoring models were used as inputs to the controller to generate a game score for each model against all the TA agents in the iterative development cycle.

4.1.2 Model Evaluation Metrics for State Agent

Mean square error loss was used to evaluate the convergence of the state-agent along with a critic network. Additionally, the cumulative value of all the rewards acquired by the agent per game was used to validate the model for signs of progress. However, as mentioned before the state-based agent in my implementation was unable to converge even after 1000 games played 2vs2 against TA Agents.

4.2 Tournament Success Metrics

As mentioned in the introduction, the objective of the project was to build an agent to play and win a 2 vs. 2 SuperTuxKart Ice Hockey tournament against the TA agents. There were four TA agents - Geoffrey, Jurgen, Yann, and Yoshua and my agent played a total of 4 matches with each agent with a time limit of 1200 frames or 3 goals. Each game had two teams - 'Blue' and 'Red'. Each team got a point if they drove a puck into the goal post of the opposing team as well as if the opposing team drove a puck into their goal post. The TA agent played two games as Blue and two as Red. The average number of goals scored against the TA agents was computed linearly from 0 to 1 and if this value was 70% or above, the agent was considered successful.

4.3 Model Results

Our final agent was an image-based agent with the following metrics on our test dataset. We achieved a BCE loss of 0.000176 and an average precision of 0.893 which was considered sufficient for the task. Figure 9 shows our training and validation loss for this model over the 50 epochs. The majority of the errors in the detection came from false positives which we eliminate in the detection downstream by using the replay buffer in the controller discussed previously.

Figure 9: Training and Validation Loss for best model

Attempts to use multiple attention heads for predicting heatmaps, computing the bounding box size as well as depth field information were not successful and were hence not part of the final agent. Similarly, the state-based agent on which I spent most of the development time failed to converge to an increasing cumulative reward coinciding with an increase in the number of games.

4.4 Tournament Results

The goals scored by the TA agent as well as my agent are given in Table 1. My agent was able to achieve a score of 76% against the TA agents. Since my agent was able to score 76% against the TA agents, which exceeds the 70%

	Goals Scored		Goals Scored		Goals Scored		Goals Scored	
TA Agent Name	Geoffrey	My Agent	Jurgen	My Agent	Yann	My Agent	Yoshua	My Agent
Game 1	3	0	1	0	1	2	0	0
Game 2	2	1	1	1	1	2	0	1
Game 3	0	1	1	0	1	1	1	1
Game 4	1	1	1	2	1	2		0
Total	6	3	4	3	4	7	1	2

Table 1: Goals scored by TA agents and My Agent in each game

threshold requirement, I consider that my agent was successful in the tournament.

5. Conclusion

In this paper, I discussed the design and implementation of a SuperTuxKart ice-hockey automated agent and evaluated its performance in 2 vs. 2 SuperTuxKart Ice Hockey tournament against four course-provided state-based agents referred to as TA agents (Krähenbühl & Wu). To build a successful agent, the agent needed to be able to achieve an average score of 70% against the TA agents. My agent successfully achieved a score of 76% against the TA agents, beating the 70% threshold required. This indicates that the object detection model and the hand-tuned controller worked well during the gameplay.

If I had more time and computational resources, I wanted to work on a novel approach to use the two karts as one stereoscopic vision agent by driving them side-by-side and using the left and right eye images thus generated to produce a stereoscopic view of the playing field from which a neural network can generate a depth field similar to the method employed by Wang et al. (2021). I briefly experimented with this idea and found the `*Sara-the-racer*` kart is ideal for this task due to its narrower wheelbase and better maneuverability. As mentioned previously, other significant improvements would be building a better kart motion planner using vehicle dynamics by running tests on the game engine. This could be extended by developing a full-fledged non-holonomic vehicle controller which could then be abstracted into a class that the action stages would call.

The other branch of approaches I attempted was to develop a state-based agent. To this end, I ran three different flavors of algorithms DDPG (Lillicrap et al., 2015), TD3 using Ornstein-Uhlenbeck Action Noise, (Fujimoto et al., 2018) and TD3 using the epsilon-greedy approach. I also attempted various reward-shaping functions from simple linear ones such as the negative value of the puck to kart distance, to exponentially shape increasing rewards as the kart gets closer to the puck. I also normalized the state space and action space and reduced the action inputs to just 3 variables: acceleration, steering, and brake. However, I was unable to see an improvement after almost 2000 games and hit the point of diminishing returns. If I had more time and computational resources, I would work on debugging the implementation of the state agent and try to bring it to par with the image agent. Following this I would further develop this using Multi-Agent PPO (MAPPO) for on-policy learning (Yu et al., 2022) and Multi-Agent TD3 (Yu et. al, 2020) for off-policy learning approaches that leverage collaborative behaviors between the player karts to score goals similar to a real-world ice-hockey team. Finally, the TrackMania RL repository (pb4git, 2023) and related publications (Neinders, 2023) serve as an inspiration for developing the best possible pytuxKart controller for the game.

References

Agrawal, P., Sahai, S., Gautam, P., & Kelkar, S. S. (2021). Designing variable Ackerman steering geometry for formula student race car. *International Journal of Analytical, Experimental and Finite Element Analysis*, 8(1).

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv*:1607.06450.

Dankwa, S., & Zheng, W. (2019, August). Twin-delayed ddpg: A deep reinforcement learning technique to model a continuous movement of an intelligent robot agent. In *Proceedings of the 3rd international conference on vision, image and signal processing* (pp. 1-5).

Fujimoto, S., Hoof, H., & Meger, D. (2018, July). Addressing function approximation error in actor-critic methods. In *International conference on machine learning* (pp. 1587-1596). PMLR.

Fukushima, K. (1969). Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4), 322-333.

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. *Nature* 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

Ioffe, S., & Szegedy, C. (2015, June). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448-456). Pmlr.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:*1412.6980.

Krähenbühl, P. *Supertuxkart: A fork of Supertuxkart to hook an AI agent up to*. GitHub. https://github.com/philkr/supertuxkart?tab=readme-ov-file

Krähenbühl, P., & Wu, C.-Y. (n.d.). *Final Project - Supertuxkart Ice Hockey*. *Deep learning class template.* https://www.philkr.net/dl_class/homework/final/

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:*1509.02971.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, 518(7540), 529-533.

Neinders, L. J. (2023). *Improving Trackmania Reinforcement Learning Performance: A Comparison of Sophy and Trackmania AI* (Bachelor's thesis, University of Twente).

O'shea, Keiron, and Ryan Nash. "An introduction to convolutional neural networks." *arXiv preprint arXiv:*1511.08458 (2015).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems* 32 (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

pb4git. (2023). Pb4git/trackmania_rl_public: *Ai plays trackmania with reinforcement learning*. GitHub. https://github.com/pb4git/trackmania_rl_public

Raad, M., Ahuja, A., Barros, C., Besse, F., Bolt, A., Bolton, A.,… Young, N. (2024) *A generalist AI agent for 3D virtual environments.* https://deepmind.google/discover/blog/sima-generalist-ai-agent-for-3d-virtual-environments/

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference*, Munich, Germany, October 5-9, 2015, proceedings, part III 18 (pp. 234-241). Springer International Publishing.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587), 484-489.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:*1712.01815.

Velu, A., & Yu, C. (2020). Marlbenchmark/off-policy: *Pytorch implementations of popular off-policy multi-agent reinforcement learning algorithms, including qmix, VDN, MADDPG, and MATD3*. GitHub. https://github.com/marlbenchmark/off-policy

Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M.,Jaderberg, M., Czarnecki,W.,… Silver, D. (2019) Alphastar: Mastering the real-time strategy game starcraft II. *DeepMind blog*, *24*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii

Wang, H. M., Lin, H. Y., & Chang, C. C. (2021). Object detection and depth estimation approach based on deep convolutional neural networks. *Sensors*, 21(14), 4755.

Wikimedia Foundation. (2024, April 26). *Ornstein–Uhlenbeck process. Wikipedia.* https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process

Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., & Wu, Y. (2022). The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35, 24611-24624.

Yu, C., Velu, A., Vinitsky, E., Wang, Y., Bayen, A., & Wu, Y. (2020). Benchmarking multi-agent deep reinforcement learning algorithms.

Zhou, X., Wang, D., & Krähenbühl, P. (2019). Objects as points. *arXiv preprint arXiv*:1904.07850.