Group 60 Final Project Report

Deep Learning: Summer 2023

Abhishek Paul
ap62752

Ayush Kumar
ak45898

Mahir Morar
mm224567

***Abstract*—The project consisted of designing and training an agent to play SuperTuxKart to compete with the other trained agents in a 2v2 hockey game. Our strategy was to utilize reinforcement learning to minimize the player-puck and puck-goal distance. Our initial attempt at using Q Reinforcement Learning had trouble learning optimal strategies. Imitation Learning was then used to train the model on the winning strategies of over 2000 games. Our reinforcement learning model was then improved by using the imitation model as its initialization in order to further refine the agent's strategy. However, it was not able to perform that well against the other pre-trained models as it struggled in unseen scenarios. Future work can involve utilizing data augmentation during training in order to randomize initialization. A hand-crafted internal state controller could also be designed to see if it is superior to the AI models.**

I. Introduction

From its inception, Artificial Intelligence has been long used to solve complex tasks through the medium of games. More specifically, reinforcement learning has been a field of particular interest as it allows AI agents to learn to navigate complex virtual environments, optimize strategies and mirror human-like learning processes. The goal of this project was to evaluate one such use case by training a deep learning model to play a 2v2 hockey game of SuperTuxKart [1] against agents trained by the TAs and other students.

The overall structure of this project can be broken up into the following parts:

1. Make a decision on what kind of agent we want to play the ice hockey game
2. Design a model based on what agent we chose
3. Train the model to play against the given agents

Our performance would be dependent on how well our model does when playing against other pre-trained agents.

II. State-Based vs Image-Based Model

The first decision that needed to be made was to choose between using an image-based versus a state-based model.

Image-Based Agent - Focus on the vision component and have a hand-tuned controller. Required more computation to develop and test.

State-Based Agent - Will be given the state of the puck and both player and opponent agent. No hand-tuned controller allowed. More learning-based approach.

With that in mind, we started to look at the code as a team and found that our player.py for each model provided us with different information.

Within the image-based model we are given the following information about each player:

TABLE I: Table: Information for Image Based Agent

Object	Information
Camera	**- aspect:** Aspect ratio **- fov:** Field of view of the camera **- mode:** Most likely NORMAL (0) **- projection:** float 4x4 projection matrix **- view:** float 4x4 view matrix
Kart	**- front:** float3 vector pointing to the front of the kart **- location:** float3 location of the kart **- rotation:** float4 (quaternion) describing the orientation of kart (use front instead) **- size:** float3 dimensions of the kart **- velocity:** float3 velocity of the kart in 3D

Versus the following was provided to us in the state-based model:

TABLE II: Table: state agent information

Object	**Information**
Kart	**- front**: float3 vector pointing to the front of the kart **- location:** float3 location of the kart **- rotation:** float4 describing the orientation of kart (use front instead) **- size:** float3 dimensions of the kart **- velocity:** float3 velocity of the kart in 3D
Soccer State	**- location:** float3 world location of the puck

In our initial analysis, we were able to understand better how to tackle the state-based agent. We knew that we could take these inputs and make them into a tensor and create a basic model whose outputs are the actions that the player can take.

We hypothesized that the model we would have to create for the image-based agent would require more fine-tuning and was more prone to errors.

We created this chart to further analyze our options and to see if our hypothesis was correct:

TABLE III: Table: Information for state-based agent

Agent	Pros	Cons
Image	- Most control and understanding of the game	- Prone to error - Requires most compute
State	- More straightforward implementation - Requires less computation - Training can be quicker	- Lose granularity - Build RL model that learns over iterations (need to fine tune reward function)

Upon this objective review of our choices, we decided to build a state-based agent. A state-based agent operates by observing and making decisions based on a set of relevant variables that describe the current state of the game. These variables could include the player's position, acceleration, distances to objects, and other data points. Working with summarized data, state-based agents tend to have faster decision-making abilities. Thus, we believed we could create a strong model with the state-based agent.

III. Methods

A. Q Reinforcement Learning

Initially, we explored multiple options on how to train the data. Our team looked at the documentation found on OpenAI [2]. Our first objective was to create an adequate reward function to provide the model something to maximize.

Our first reward function was to reward the model for scoring with positive reinforcement while being scored on as a negative number. After exhausting hours of training, we found that this method did not train our model to produce any positive outcomes. We discovered that this strategy had limited effects and reinforcement learning is not as effective when there are sparse rewards.

Since we provided such a binary event for the model to try to optimize, it was struggling to understand which actions to take (i.e. policy) to maximize the reward function. Also since this model was playing with other trained agents, it was consistently being scored on and receiving a negative reward over time.

To fix this we went over and re-examined our reward function. We decided that, instead of having a discrete reward function (whether the agent scored), we need a continuous reward function. In order to encourage the players to go after the puck, which would help them score in the future, we altered the reward function to try to minimize the distance between the player, puck, and goal. Our thought with this new reward function was that it would encourage the model to move toward the puck which it was lacking in the prior attempt.

```
#reward is the distance between the ball and the goal line and the distance between
the ball and the players
ball_loc = np.array(soccer_state['ball']['location'])[[0,2]]
goal_center = np.array(soccer_state['goal_line'][self.team]).mean(0)[[0,2]]
player_loc = np.array([player['kart']['location'] for player in
player_state]).mean(0)[[0,2]]
player_to_puck = np.linalg.norm(player_loc - ball_loc)
ball_to_goal = np.linalg.norm(ball_loc - goal_center)
reward = (ball_to_goal-player_to_puck*0.001) / 100 + soccer_state['score'][self.team]
- soccer_state['score'][int(not self.team)]
```

Fig. 1: Code for the Reward Function

When training this function, we were also unable to encourage the players to actually score the puck. Observing at how the agents were playing, since there was an equal reward for the puck-goal and puck-player distance, the karts would not move when the opposing team brought the puck close to their goal.

B. Imitation Learning

Given our inability to fine-tune a reward function, we started to explore other ways to train the model. We believed that imitation learning, which entails teaching our agent based on the actions of a demonstrator, would prove to be more effective than the trial and error required of reinforcement learning [3]. In our research, we came across this blog post from Stanford [4]. This post trained a model to play Minecraft using videos of a human playing the game.

This post got our team to consider if creating an optimal reward function was the best use of our time. We already had access to pre-trained agents so we decided to use these agents as our 'experts' to train our model. Using these agents to demonstrate for our model seemed to be a more straightforward methodology.

The first part of this was to identify which agent had the best-trained model. After having all the agents play against each other, we found that image_jurgen_agent had the best performance among the others.

Our next step was to create the training data. To do this, we exported the pkl files from the different games between the agents. We created 2000 pkl files of training data for the model to use. Our goal was to imitate the winning strategy of each game.

With this data, we started to train our model. For our first attempt at the model, we chose to only focus on accelerating and steering. Using our training data, we iterated through each pkl file and grabbed the game state data.

We then looked at the acceleration and steering policy of the winner of each game and trained our model to mimic that strategy. We went through two iterations of this model.

The first iteration consisted of a network consisting of a set of fully connected layers:

```
class MyModel(nn.Module):
    def __init__(self,input_size,output_size,n_layers=1, layer_size=64):
        super().__init__()
        self.model = nn.Sequential(
        nn.Linear(input_size, layer_size),
        nn.ReLU(),
        *[nn.Sequential(nn.Linear(layer_size, layer_size), nn.ReLU())
        for _ in range(n_layers)],
        nn.Linear(layer_size, output_size),
    )
    def forward(self, x):
        return self.model(x)
```

Fig. 2: First iteration of imitation model

We found that this was not capturing all the nuances of the winning strategy. The second iteration utilized a four-layer convolution network:

```
class SecondMyModel(torch.nn.Module):
    def __init__(self,input_size,output_size,layer_size):
        super().__init__()
        self.network = torch.nn.Sequential(
        nn.Linear(input_size, layer_size),
        nn.ReLU(),
        torch.nn.Conv2d(layer_size, 32, 5, stride=2),
        torch.nn.ReLU(),
        torch.nn.Conv2d(32, 64, 5, stride=2),
        torch.nn.ReLU(),
        torch.nn.Conv2d(64, 96, 5, stride=2),
        torch.nn.ReLU(),
        torch.nn.Conv2d(96, 128, 5),
        torch.nn.ReLU(),
        torch.nn.Linear(128, output_size)
        )
    def forward(self, x):
        return self.network(x)
```

Fig. 3: Second iteration of imitation model

This performed better as our agent was able to score more goals against the opponents.

C. Reinforcement Learning Iteration 2

Once we had a working imitation learning model, we wanted to see if we could utilize it as an initialization for training reinforcement learning. This improved the training of the reinforcement learning model significantly as now instead of trying completely random strategies and seeing what works, it would utilize the strategies it learned from the imitation model as its starting point. This improved the rate of training as the model was making less "uninformed" choices.

IV. Results

A. Imitation Learning

We found that after tweaking the parameters of the model and training, the imitation learning model was able to play the game. However, it was far from perfect.

For example in one game when we played against image_jurgen_agent in the red, our agents were very close to scoring the goal but one of our players completely removed them self from the game. That player instead went to the upper right-hand corner and moved in circles. We found that this behavior would happen often especially as the player would move farther from the puck. The team was not able to figure out what caused this behavior.

The first iteration (fig 2) was giving us a score of 18/100. We decided to make the model more complex (fig 3) so it hopefully capture more of the nuance in the strategy.

The best result from this was about 25/100 with about 10 epochs.

B. Reinforcement Learning

As we mentioned in the methods we were relatively successful once we used the trained imitation model as the initialization of the reinforcement learning model. With this model, we found the agents were more likely to head toward the ball and take it toward the goal. One problem we found was that, while agents were able to perform well in the initial few seconds of the game, their performance degraded as the game went on.

Ultimately, we chose to keep the model as is since it would manage to score occasionally. We mapped out the frequency at which our model won against each of the other pretained model. These results can be found below:

TABLE IV: Table: model vs. agent win rates

Agent Name	**Win Rate**
image_jurgen_agent	0%
yann_agent	10%
yoshua_agent	8%
geoffrey_agent	5%
jurgen_agent	0%

The highest grade we could get on this was 32/100. A slight improvement, but obviously far from perfect. We had trained this with 30 epochs but noticed that the more we trained it, the lower the grade would become. We actually found that the 10th iteration of the model was able to produce the highest grade. After that, the grades gradually decreased. We believe this was because the model was over-fitted.

V. Future Improvements

The highest score we received on Canvas was a 32/100. While this was well under our goal of 70/100, we have learned a lot about deep learning and the effort needed to make a model successful. The first thing we realized was that training the model for more time and with more epochs does not always fix the problem. In fact, the longer the model trained, the worse it performed. As previously mentioned, we believe that this is because the model became over-trained. Given more time there are many things that we want to improve in order to improve model accuracy.

A. Data Augmentation

One of the main problems encountered in the project was that the performance of the agent decreased significantly after the first 5-10 seconds. We found the agents were performing well in the first few seconds, but after that the performance decreased. In our testing we found most of the goals scored by our model happened during the initial first few seconds of the game. These results were because we always started with the same initialization so, while the model had plenty of opportunity to explore the initial states of the game, as the game progressed the model would encounter more unknown states where it would fail. One way of fixing this issue would be to start the game with random initialization (i.e., the player and puck positions are randomized at the start of the game). This approach would significantly increase the number of different scenarios explored in training and hopefully increase the model's performance.

B. Explore Image based agent:

A disadvantage of reinforcement learning is that even if we observed an agent learning bad strategies during training we had little recourse to fix it other than modifying the reward function. Another approach we want to consider is utilizing a hand-crafted planner with an image-based agent. During testing, we found that even a simple heuristic of racing to the puck at the start of the game and trying to push it to the goal gave us a decent score. We could use an image-based agent with a deep network to determine the position of the puck in relation to the players and the goal. This would allow us to create a hand-crafted controller to instruct the agents on what to do in different scenarios (i.e., when the puck is by the wall, when the puck is ahead of the agent vs behind the agent, etc).

VI. Conclusion

This project was intended to capture our training process of a deep learning model to play SuperTuxKart. The first major decision was whether to use image-based or state-based agents. Ultimately, through careful analysis of our options, we chose to continue the project using state-based agents due to its more intuitive implementation and efficient decision-making. The first method we used in designing and training our model was Q reinforcement learning. The main challenge in this approach was coming up with a reward function that would better shape this model's behavior. After multiple iterations we decided on a continuous reward function that rewarded reducing both the agent-puck and the puck-goal distance. However, our trials were hampered by its slow training rate as the model's performance took many iterations to improve. Due to the challenges we faced in devising an effective reward structures and creating new strategies, we wanted to examine other methods of training the model.

Instead, we started investigating our options related to imitation learning. We had the TA's agents play 2000 games and used the winners as our 'experts'. Imitation learning provided greater positive results and allowed for further insight into this method's applicability and limitations. For example, expert bias came into play as or 'experts' were sub optimal, causing shortcomings in the model's behavior. This task was also relatively complex, so it was more difficult for the model to grasp the nuances of the demonstrator's behavior.

Once we had a working imitation learning model, we decided to revisit our reinforcement learning method and utilize our trained imitation learning model as its initialization. This combined approach significantly increased the training rate of the model as instead of trying out random strategies and seeing what worked, the model tried to use the strategies learned in the imitation model as a starting point. The final model used was the reinforcement learning model with our trained imitation model as its initialization.

Despite our inability to create a model that performed to our expectations, we gained a greater comprehension of deep learning through this project. Understanding the advantages and limitations of the different learning techniques will help us grow our technical skills and use deep learning in a more efficient and tactical manner in the future. Were we to complete this project again, we would also aim to further expand on our deep learning knowledge by exploring other approaches such as data augmentation and the image-based agent.

References

[1] SuperTuxKart, https://supertuxkart.net/Main_Page. Accessed 9 Aug. 2023.

[2] "Vanilla Policy Gradient — Spinning Up Documentation." Welcome to Spinning Up in Deep RL! — Spinning Up Documentation, https://spinningup.openai.com/en/latest/algorithms/vpg.html. Accessed 9 Aug. 2023.

[3] Daniel S. Brown, Wonjoon Goo & Scott Niekum "Better-than-Demonstrator Imitation Learning via Automatically-Ranked Demonstrations" Conference on Robot Learning (CoRL) 2019

[4] Garg¡/a¿, ¡a. "Learning to Imitate — SAIL Blog." SAIL Blog, 1 Nov. 2022, http://ai.stanford.edu/blog/learning-to-imitate/: :text=So%2C%20what%20is%20imitation %20learning,policy%20that%20mimics%20this%20behavior.