John L. Peterson (jlp5729)
Philipp Krähenbühl
CS394D – Deep Learning
12 December 2021

Final Project: Ice Hockey in PySuperTuxKart (PySTK)

For this assignment, I decided to use an image-based model. Issues I encountered include generating and reading data in a PyTorch-accessible format, training the model with adequate labeled data, and most of all, developing strategies for and fine-tuning of the controller/player module. Although the controller did not perform as well as hoped in the end, it was not due to lack of effort or ideas/strategies.

Initially I researched the state-based route. While I was interested, I did not feel that I had enough experience programming this topic in the homework assignments to confidently start in a new direction. Starting on the image-based model, I initially tried several routes to generate, download, and process the data before learning more about the included modules/methods. My initial attempts to develop CSV files to export then re-import were abandoned in favor of generating the state recorder pickle file with slight modifications to structure and processing that data. The high-RAM Google Colab instance ended up being necessary due to the size of the data file. Initially I chose to label the training images with just a puck screen coordinate and a Boolean "puck onscreen" value. Utilizing the bit shift instance data in PySTK for puck onscreen and the projection / view matrices for the on-screen coordinates made this easy once I understood how to access the data. Having generated the data, created a model, and conducted some initial training, I then realized that it may be difficult for a controller module to determine required direction and speed without some estimate of distance. While I initially explored estimating height and width of the puck, I found it much simpler to calculate a distance value to the data, modify the network, and retrain the model.

The model used in this submission is a convolutional network with a block-based structure. I opted not to use drop-outs or skip connections, as the goal was to keep the model light and efficient. There are four main block layers, each with a Batch Normalization layer, a Convolutional layer, and a Rectified Linear Unit. In order to predict the "puck onscreen" and the "puck distance" values, I use a Linear classifier with two output channels. For the "puck coordinates" prediction I use a convolutional classification layer and a spatial argmax function to pick the highest coordinate value from a full-resolution matrix, minimizing resolution loss. For training I used a L1 Loss for both puck distance and Boolean puck value, and a Binary Cross Entropy Loss with a Sigmoid layer for the coordinate values. The BCE with Logits loss in PyTorch was efficient and generated values in the range 0 to 1 necessary for on-screen coordinate prediction.

Several issues were encountered during training the convolutional model. First, as referenced earlier, was the need to rewrite and retrain the model to include a distance estimate. This newly-added distance value became an issue in training, as the model would not train effectively once the distance value was added to the loss functions. After hours of graduate student descent, I realized that normalizing the data would be required, as the distance values predicted were much larger than either the coordinates or Boolean values. Still having issues with the distance value and coordinate losses not consolidating, I returned to the training data and applied a negative distance value and negative coordinate values to any image that had a "False" value for "puck onscreen." This helped to correlate the data further and reduce loss upon both training and validation. Negative distance values also became another helpful threshold upon which the controller could act.

Having generated data, composed the model, and trained the model, the next step was the controller. This task ended up being a much larger portion of the assignment than initially expected. My initial plan for the controller consisted of three parts: generating basic logic to drive toward the puck, developing logic to determine which side the player was on and avoid own goals, and then to tweak and improve methods from there. The first step was determine threshold values for puck onscreen and distance; this was done using trial and error. Code to steer toward the puck when it was onscreen was relatively simple once the thresholds were determined, but the size and scope of the problems became clear while creating methods to "find" the puck when it was offscreen and avoid getting stuck.

My first attempt to avoid getting stuck involved checking location and direction and backing up when above a certain threshold of each. This did not work reliably, as the racer would immediately start moving forward again. Another attempt to avoid getting stuck was to drive in a circle in a pseudo-random way. However, this still did not solve the problem entirely, and still resulted in problems like getting caught in the goal. Another attempt was using rescue, but as it did not change the direction or location of the kart significantly, it would easily lead to a "rescue" loop where the kart was useless for large portions of the game. I attempted to utilize "memory" and reference the previous states as to whether the kart was recently rescued to avoid this, but could not find an optimal point where rescue was not over or under-utilized, and abandoned this option. After several more attempts, the most success I found came from developing a "seek location" function to steer toward a specified point, either forward or backward. The logic compares the current

kart direction vector to the intended direction vector, and issues commands to drive either forward or backward. This value can be specified, but if not, it determines which direction to drive based on the way the kart is currently facing. In the final version, when the kart does not see the puck, it drives backward to the "home base" location, which is the corner for Player 1, and in front of the goal for Player 2. I did incorporate memory to check if the kart is stuck by comparing the previous few values of speed and location. If stuck, the controller attempts to either drive forward or backward out of the spot using the seek location function.

With logic in place to avoid getting stuck, another challenge was maneuvering to the puck in a manner that would avoid scoring in the player's own goal on the opponents' behalf. In random trials, simple puck-seeking logic would sometimes score, but given randomness of the game state and the opponent, could not be expected to perform reliably, and would just as often end up scoring own goals. My first idea was to check the direction and location of the kart, and if heading toward the opponent's goal line, steer slightly to the side of the puck. This required trial and error to determine the threshold at which to enable this function, the steer amount needed to hit the puck to the side and not miss entirely, and further fine-tuning. Additional adjusts the steering amount by comparing the direction of the kart to the direction of the goal. This allows the driver to hold onto the puck longer and attempt to steer toward the corner/wall instead of only hitting it away; that logic is implemented for both players 1 and 2.

While avoiding getting stuck and playing defense were both important, the most important metric was left: goals scored. Initial attempts to steer the puck toward the target goal consistently resulted in under or over-steering. The puck often either slammed into the wall or was lost to the opponents. The next attempt involved comparing the x-coordinate of the kart direction to the x-coordinate of the goal and steering gradually in direction of the goal. However, this did not compensate enough. Further revisions included adding a separate step to adjust steering when the puck was close but not in possession; once the puck was "locked on," further adjustments were made. Even so, these adjustments did not consistently score goals. After more research and consulting Piazza, I decided to utilize an angle-based approach to determine the steering angle to the goal. The controller attempts to steer the kart wide, based on the angle to the goal, the distance from kart to puck, and distance from puck to goal. This adjustment will be lessened both when the puck is in possession of the kart and when the puck is close to the goal. One issue early on with this was kart oscillations close to goal, causing what looked like a surefire goal to shoot sideways instead. I encountered several edge cases as well, including backing up to approach a puck in the bottom corner of the screen, hitting the puck on a larger angle when against the wall, and attempting to hit the side of the puck when driving perpendicular to the goal to "nudge" it in. By and large, successful goal attempts come from a clear path, but the parameters and cases of my controller can turn mediocre situations into goals on a semi-regular basis.

Several strategies were attempted and abandoned. The first and most promising was a quadrant-based approach. This would keep Player 1 and Player 2 in separate but overlapping areas, and take different actions based on which further subdivision of those sections they were in. This ended up being too complicated; it was enough of a challenge just to get the agents moving and scoring. The second strategy was to have one player as attack, and the other as defense. This strategy lasted for the majority of the time working on the controller; this allowed me to develop more accurate attacking logic. One of the issues came in getting the defensive player back to the specified goalie box when it had driven out to seek the puck. This was improved with the "seek location" function mentioned in the above paragraphs. The defensive strategy was heavily modified; Player 2 now tracks back to the defensive goalie box when it doesn't see the puck, but is not restrained to this area at all times, in order to attempt to score more goals. Another strategies that was attempted was to induce randomness into steering. This was incorporated into the "seek location" function and into logic to avoid getting stuck, but did not become a primary part of the strategy otherwise. Finally, the negative distance and coordinate values that the network predicted ended up providing a more robust controller; if the model predicted an incorrect "onscreen" value, this could be checked against the coordinates and distance values. One option I wanted to try was coordinating actions between player 1 and 2, but I did not have time or expertise to traverse this route.

Overall, this assignment ended up largely consisting of work to develop a successful controller, and about 70% of the time working on the assignment (more than 40 hours total) were devoted to it. I did not choose to work in a group, as personal commitments prevented me from starting the project earlier than 1.5 weeks before due date. I did not feel that timeline would be fair to my prospective team members. I also would have liked to explore the state-based model in order to develop experience in developing training imitation, reinforcement or other models. Unfortunately, due to my single member group, lack of extra time, and unfamiliarity with the actual programming for those subjects (not to mention very rusty programming skills that made just generating data and formatting for a PyTorch Model much more involved than it should have been), I ultimately chose the image-based route. While many problems documented above proved frustrating and challenging, this project still had several memorable and enjoyable aspects.