CS 342 Final Project Writeup

Andrew Nolte, William Wang, Grant He, Russell Coleman

December 2020

1 Introduction

The problem that our team tackled was to create a SuperTuxKart AI that would score as many goals as possible, in games of ice hockey. We tried to motivate our ideas by first playing rounds of ice hockey ourselves, watching expert humans do it on YouTube, and having every team member individually create a controller strategy before we merged our work together.

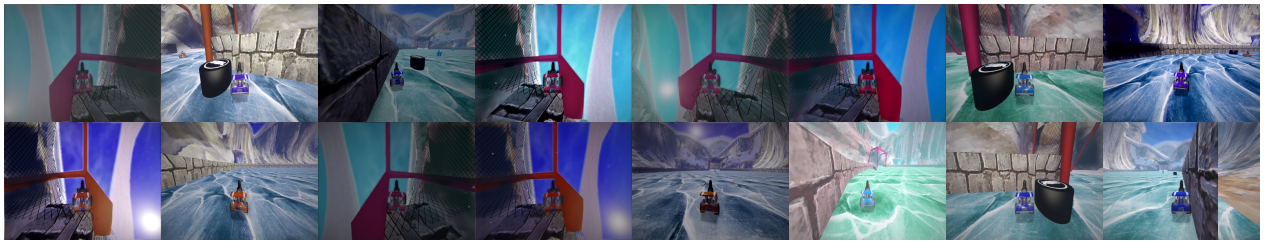
2 Model Architecture

We went through several iterations of our models. We knew our architecture would have to be similar to our code from homework 4, so we had different group members try training their individual homework 4 solutions, as well as adapting the master solution. We also tried generic dense prediction networks as well as networks that dense predict with heatmaps and `extract_peak()`. In the end, the adapted master solution was chosen with heatmaps, as it gave the subjective best performance and was easier to tweak to immediate changes.

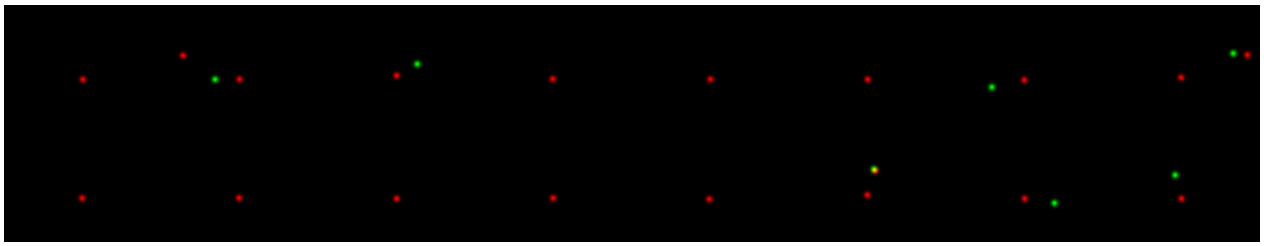
With this adapted master solution, we had three variants: the small network, the medium network, and the full-size network. The full size network takes in the full $(300, 400)$ image, the medium and small take half and quarter size respectively, with the sky cropped out. While the medium and small were faster, we realized we ended up needing to use the full size because being just a few pixels off can very much change the predicted world coordinates of the puck.

The full predictions of the network are its predictions of other karts, where the puck is, where nitros are, where pickups are, a depth map, and a single scalar outputting whether or not the puck is in the frame. To do this, we layered additional convolution layers on top of the last upconvolution layer of the master solution network, with an appropriate filter size to give what we need, and used an adaptive max pool to bring the `is_puck` value down to a single scalar.

Included below are examples from the output of the model (by random chance, there are a lot of images in which the kart is stuck in the goal in this selection of 16 photos). This is full resolution, without sky cropped out, for better visualization.

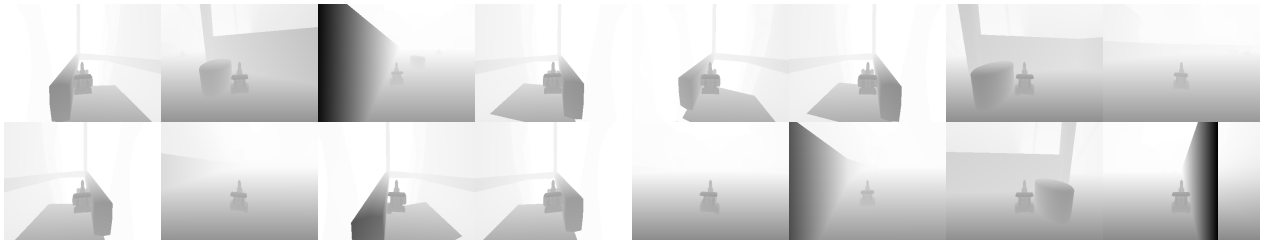


Ground truth detections (green is puck, red is kart):

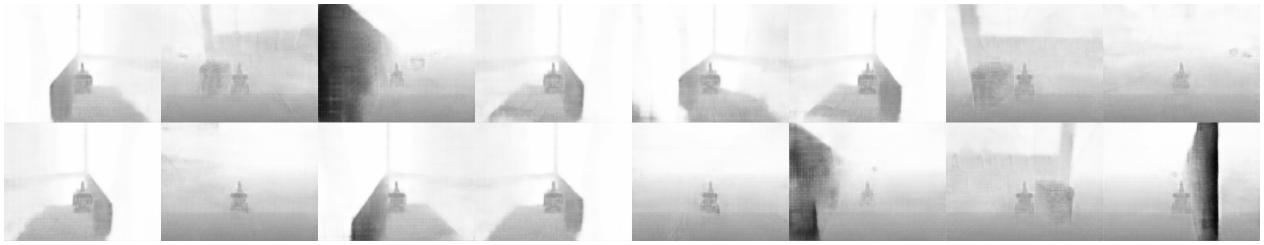


Model detections:

Ground truth depth:



Model depth:



There is also an `is_puck` loss not included here, as it is a single scalar and thus not conducive to image visualization.

Here are some links to videos of the predictions of the full size model vs the small model, and notice how the small model is often more than a few pixels off, something detrimental to the projection matrix with predictions on the horizon, motivating our choice of the large model (note: this is AI vs AI, only model predictions are shown, the controller is not playing here).

3 Training

We created a series of different datasets. The first ones created were offline datasets, in which we recorded depth map information, as well as box location information for each of the items. Our plan was to start training on the offline dataset, as it is easier to make, and then move over to training on the online dataset as time went on. The goal to prevent overfitting was to have an absolutely massive dataset, and so we left it generating over the span of about 24 hours to create a dataset of about 504GB. When training our later models on the huge dataset, they were not even able to make it past one epoch of training in 10 hrs, yet had great performance, as they had not had a chance to overfit.

While we did successfully achieve creating the online dataset, porting over some of the intricacies (e.g. generating the right heatmaps) at runtime proved to be non-trivial, and we decided that with such a large offline dataset, our time would be better spent tweaking the controller, and so we did not pursue this path further.

However, this ended up being a mistake, as part of the benefit of the online dataset is that it plays in realistic circumstances and with the right kart (as we changed karts after the dataset was generated). As a result, our model's performance was better on the Tux kart than the Wilber kart, which we had written the controller for, and so we had to do a last-minute rewrite of the controller back away to the Tux kart.

4 Controller

We designed our controller to be as simple as possible, requiring only an estimate of the global puck location on the 2D plane (in addition to `player_info`) to decide what to do. To avoid our teammates fighting over a puck, we decided to make one agent the "scorer" and the other the "goalie". The scorer would always try to get behind the puck and push it towards the enemy goal, while the goalie would sit in front of the goal until the puck got within a certain distance.

Here is a link to a video of an early version of the full stack agent, where you can see 2d predictions drawn to a map of the side (blue being kart location, red predicted puck, green ground-truth puck)

4.1 Scorer

The default control state of our scorer agent was to drive towards the puck. However, we realized early on that simply charging at the puck would only rarely lead to an actual goal. Rather, we needed a way to steer towards the line between the puck and the goal, moving the aimpoint towards the puck/goal when we approached the line. We accomplished this by first calculating the line between the goal and the puck at the start of every frame. Then, we found the distance d between the kart and puck and calculated a target distance αd, for some α between 0 and 1 which we tuned. The final aimpoint was then determined as the point on the goal-puck line which lay "behind" the puck at a distance αd. By calculating this aimpoint each frame, we could drive a smooth curve towards the puck which would cause it to be driven directly toward the goal at the moment of impact.

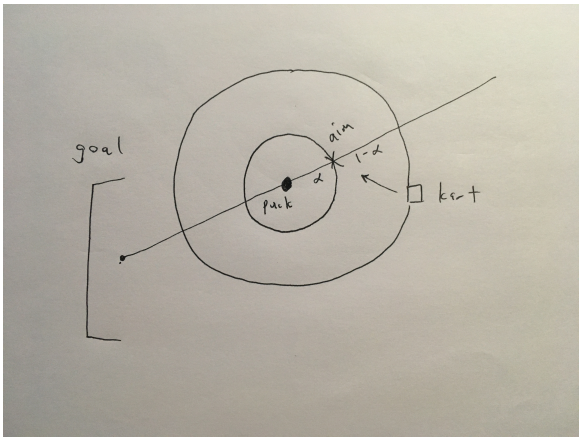


Figure 1: Scorer steering logic.

4.2 Goalie

Our goalie was much simpler, using a 3-state machine to alternate between sitting at the goal, charging at the puck when it got too close, and returning to the goal.

5 Inverse Camera Projection

In order to connect the camera coordinate output of the model with the world coordinates that the controller expects, we can use the given projection and view matrix, along with the (mostly) constant height of the puck to almost perfectly projection onto world coordinates. Intuitively this makes sense that this is possible, as our view of the puck lies on a line coming out of the camera, and the puck lies on a flat plane. We can find the intersection of the plane and the line.

The camera projection function is $\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$

...where x, y, z are camera coordinates. z is not used to draw on the camera, but it is an indicator of depth. So $z < 0 \implies$ the point is behind us, and z provides a depth ordering. x, y also range between -1 and 1, with the outside of that box being outside the field of view of the camera.

... and X, Y, Z are world/3d camera coordinates.

Normally, the view matrix transforms from world to camera 3d coordinates (where z axis is coming out of camera), and then the projection matrix transforms that into 2d camera coordinates as shown above. Since the view matrix transformation is similar, these can be combined into one "projection" matrix.

Note that the last value in each coordinate is 1. This essentially means that whenever we work with the matrix, we divide the end vector by the last value to get 1 in that spot. This takes care of math regarding depth, as a high depth means a final projection that is closer to the center of the image.

In attempting to get the inverse operation, we can get this formula: $\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \mathbf{P}^{-1} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$

z is the only unknown we need. Because we know the height of the puck (Y), we can derive it. Note, p_{ab} in these calculations are indexing into the inverse of P.

$Y = \frac{p_{10}x+p_{11}y+p_{12}z+p_{13}}{p_{30}x+p_{31}y+p_{32}z+p_{33}}$

$(p_{30}x + p_{31}y + p_{32}z + p_{33})Y = p_{10}x + p_{11}y + p_{12}z + p_{13}$

$p_{30}xY + p_{31}yY + p_{32}zY + p_{33}Y = p_{10}x + p_{11}y + p_{12}z + p_{13}$

$p_{32}zY - p_{12}z = p_{10}x + p_{11}y + p_{13} - (p_{30}xY + p_{31}yY + p_{33}Y)$

$z = \frac{p_{10}x+p_{11}y+p_{13}-(p_{30}xY+p_{31}yY+p_{33}Y)}{p_{32}Y-p_{12}}$

Once z is found, we simply multiply the 2d homogeneous coordinates by P^{-1} and divide by the last element to get world coordinates.

6 Failed attempts

We attempted an imitation learning approach based on creating an oracle agent using hacked state information, and then doing raw imitation learning from states to actions. However, we did not do a good job creating a controller for this, rather having it predict the raw steering angle with an MSE loss and such, and it was not able to achieve high enough performance so we scrapped the idea.

We also included lots of things in our network, such as depth predictions, nitro/pickup predictions, etc. that we planned to use, but were not able to. In playthroughs of the game as a human, we found nitros and pickups to be essential in winning, but we found that without a more confident network our agent did not benefit from the use of these, and that a simple controller was better, and so it was scrapped. Still, we hope that some of the extra prediction information that was no longer used (e.g. depth map) provided a good auxiliary loss for the network in order to be able to get higher quality predictions.