CS 342: Final Project

Building a better kart driver

Group Members:
Joseph Bourque
John Madden
Shea Rozmiarek

Neural networks is similar to any field of computer science in the sense that it is easy to get distracted from what you're trying to accomplish for the sake of finding an interesting or creative solution. However, in the end what really matters is finding the best solution for the job at hand. For our final project, our group split up to do independent research. We shared ideas about what did and didn't work, before finally finding a working solution and converging back together as a group to polish said solution. In the end though, it is just as important to discuss the solutions which failed as they had a huge impact in shaping our final solution.

The first approach we tried was the same cross entropy solution used to complete homework 10. After discussing this approach, it was decided to use this solution as a jumping off point as each of us saw decent results with this approach in homework 10 and we initially felt a gradient free solution would be an excellent choice for Tux due to the complexity of the problem at hand and our hope was that it would allow tux to learn which actions to take based on information provided by the state. Upon putting this solution into practice however, it quickly became apparent that the convergence was far too slow to run to completion given the 1.5-hour time limit to run a program on the lab computers. Therefore, to try and avoid saving and reloading .tfg files multiple times, we split up in order to research possible ways to speed up convergence time and improve our working policy.

The first idea we talked about was possibly adding genetic aspects to the cross-entropy solution for homework 10 in an effort to get the network to converge faster. We modified the cross-entropy function to take in a numpy array and then restricted the survivors for each group to about 2 or 3. Between runs, the survivors were averaged

together, and the mean was passed back into the cross entropy function as a parameter. Once the set of random weights was generated, they were averaged together with the passed mean to create a compound mean. This idea was based on concepts in biology. Primarily that when two individuals in nature mate, their offspring are in a way, an average of the genes of the two parents (minus crossing over). Additionally, certain actions such as going the wrong way or progress along the track were used to bias the ranking score to ensure that the surviving parameter sets weren't a result of Tux cheating to maximize his reward. The hope that was by creating a compound average that it would result in balancing out any negative or bad parameters generated and result in faster convergence. This approach showed a significant amount of progress, however it still did not converge fast enough and still required many samples per epoch to see noteworthy progress. Additionally, both the dataset and the .tfg graph had to be saved between runs in order to pass all data needed to continue making progress. In the end, time constraints and the emergence of a more effective solution resulted in the abandonment of this method. However, given more time I believe this could still have been an effective solution albeit not as effective as the solution used.

Another policy which emerged as a result of trying to improve the cross-entropy method was one which involved imitation learning. Through trying to bias the cross-entropy method for faster convergence, we ended up with a hard-coded solution which was able to complete the lap with a reasonably high success rate. In addition, we also discovered how to export the data from the competing AI carts to obtain a second set of actions which completed all three laps every time. Using this data, the idea was that we would be able to use the data obtained as labels to train a CNN to simply perform some

arbitrary action action a at time t. This would be akin to blindfolding someone and then telling them which actions to take to get through the course. This seemed like a very promising idea at first, as it would reduce our problem to a simple classification problem vs. a more complex solution such as action prediction or segmentation. Unfortunately, while this was a fine solution in theory, it fell short in practice as Tux was able to find a policy which cheated to give a higher score without finishing the course. Our policy ended up being tux going straight 90% of the time as it was more beneficial to go straight vs turn as turning had risks associated with the center of the track and the angle. Eliminating penalties for angle and the center of the track resulted in Tux moving backwards from the starting position as it forced the position to reset to a higher value giving him a higher score than going forward. While we did attempt to add biases to correct this behavior, we found that doing so simply resulted in another hand-coded policy. This combined with the emergence of a more effective solution lead to the abandonment of this policy as well.

While trying many ideas for helping tux to drive, one huge breakthrough we had was that training on the state itself offered more information to tux than training on the current image representing the frame. Initially we simply passed in the state to a fully connected network which saw reasonable results as well as an extremely fast training time. Adding a CNN which analyzed the image frame and then concatenated the state information with the processed image before throwing it into a final fully connected layer turned out to be even more effective. This solution saw not only an incredibly fast training time, but also saw Tux complete all three laps on many instances. We believe that the reason the state is more effective than the image is that the state gives much more information to tux as to what is going on in the game. It tells tux where he is on the track,

what direction he's pointing, and how far off the track he is. Therefore, training on this information and smashing it together with the image can help tux to determine areas of the map which are more beneficial than others. While there is still a risk of Tux performing erroneous behavior, and finding a policy which is sub-optimal, but which results in a higher score is still possible; in most cases Tux still finds that going forward results in a higher score than going in reverse.

In conclusion, we learned that the environment in which you train your agent can be just as important as the architecture itself. Numerous approaches were abandoned simply because the restraints on the machines in which we were running made the approach too obtuse to pursue further. In the end, the Real challenge was not only teaching Tux to drive; and rather was teach tux to drive within the boundaries of the runtime environment provided. In the end, our chosen policy needed to reflect this, and we found that a much more straightforward policy was much more successful than our other in-depth policies due to hardware limitations. Despite the limitations, we did find a reasonable policy and gained a much greater understanding of the concepts covered in class as well as a greater understanding about how to design effective architectures in a limited runtime environment.