MarI/O Kart: Autonomous Driving in Mario-Kart

Drake Young
Dennis Mills

Project GitHub: https://github.com/MarIQKart/Mario-Kart-Autonomous-Driving-

Abstract:

Autonomous driving is a revolutionary new field of artificial intelligence (AI), wherein the embedded computers of vehicles are used to determine the driving behavior of the automobiles. Even within the past year, we have seen great leaps forward with regards to the technology and methodology used in autonomous driving environments. One of great interest to us was the use of the video game Grand Theft Auto V as the training environment for the autonomous driving agent, which achieved comparable real-world performance to other agents trained by observing real drivers. This research raised an interesting question for us: does the visual complexity of the training environment have a significant impact on the performance of an autonomous driving agent? We attempted to answer this question quantitatively, by training identical reinforcement learning agents within two Mario Kart racing video games. We trained the agent once on the less graphically advanced Mario Kart N64, and once on the more graphically advanced Mario Kart Double Dash. For the sake of consistency, the same map was used on both games as a baseline for comparison. Though we were more concerned with the comparable performance of the two models over the overall performance of either, we still approached this project with the goal of creating a generic agent which was capable of performing on both game environments at comparable levels. Because of this approach, our model was found to struggle with high speeds, but would perform well when moving slowly (we hypothesize that this is due to the greater reaction-window that our agent has at slower speeds).

INTRODUCTION

I. Problem Statement

The goal is to create a reinforcement agent that is capable of learning how to drive autonomously across different versions of Mario-Kart games. The desired agent will play the game in the Time Trials mode in order to simplify the problem. In Time Trials mode there is only one player on the track, and the objective is to get lap times down to as little as possible. The goal for our agent was to not only complete laps, but to be able to learn generically enough to learn regardless of the Mario Kart game environment and translate any knowledge across the various environments. The agent will be trained on a Nintendo-64 version of the game, as well as a GameCube version of the game; after which it will be determined if either of the games were better for training the model for playing Mario-Kart. The agent was trained using console emulation on the PC, meaning that no physical Nintendo-64 or GameCube consoles are required in order to train the model or run the program.

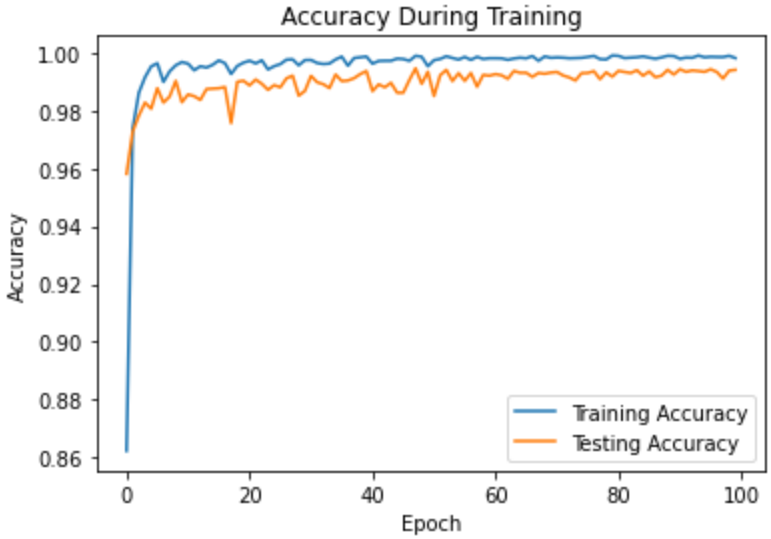
II. Model

State Space: Our final design of the model used a Convolutional Neural Network (CNN) that was trained using supervised learning to classify the current state that the agent is in at any given time -- Value Function Approximation via state-aggregation. It was trained on video data partitioned into frame images when driving in the corresponding classes for each game we were planning to use the agent on. We used the classifier to interpret game frames in real-time and classify one of 9 positions (classes) where mario may be located within the environment for a given frame. The following classifications from our CNN compose the resulting state space for our model: Center, Near Left, Near Right, Off Left, Off Right, Wall Left, Wall Right, Tunnel Left, Tunnel Right. The figures below show the state space.

Below is the Keras-generated summary of the CNN model architecture used for classifying the game frames into the corresponding state space.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_layer_1 (Conv2D)        (None, 62, 78, 32)        320
_________________________________________________________________
maxpool_1 (MaxPooling2D)     (None, 31, 39, 32)        0
_________________________________________________________________
conv_layer_2 (Conv2D)        (None, 29, 37, 64)        18496
_________________________________________________________________
maxpool_2 (MaxPooling2D)     (None, 14, 18, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 16128)             0
_________________________________________________________________
dense (Dense)                (None, 9)                 145161
=================================================================
Total params: 163,977
Trainable params: 163,977
Non-trainable params: 0
_________________________________________________________________
```

Finally, below is the training and validation accuracy for the classifier, as generated during training and plotted using python's matplotlib package. As shown, the final validation metrics for the model, after training for 100 epochs with a batch size of 128, resulted in an accuracy of 0.9948 and a loss of 0.1605



The dataset used for training our classifier consisted of 59,710 total frame images (44,782 in the training partition and 14,928 in the test/validation partition of the dataset).

Action Space: From any of those states, the agent has the option of taking any of the following actions: Left, Right, Throttle. Of course, there are many more actions that are possible in the Mario-Kart franchise (using power ups, jumping, camera controls, etc…), but since the agent is designed to play across a variety of different Mario-Kart games where those mechanics differ

greatly they are all excluded from the action space in order to ensure that the agent will have a consistent response to the actions taken across multiple games from different eras of Mario-Kart.

Action Space

Left	Throttle	Right

Training Method: The agent is trained by using Q-Learning with the described action space and state space above. The state space was condensed from the continuous one that the game provides (wherein, each frame captured is its own state) into our discretized version in order to make training the Q-Table much faster (value function approximation with the use of a CNN to performs state-aggregation into one of 9 possible states), and also to ensure that the model still retains a sense of generality that would allow the model trained in one game to still drive in another game. The agent would be allowed to start on a map and the Q-Table would begin to be built. Over time this Q-Table will converge to the optimal value function for our given policy. We utilize a TD(0) methodology for training purposes, each time a frame is captured, we reward the previous frame's state-action pair based on the resulting state.

Q-Table

State/Action	Left	Throttle	Right
Off Left	(-100, N)	(100, N)	(100, N)
Near Left	(-100, N)	(100, N)	(100, N)
Center	(0, N)	(100, N)	(0, N)
Near Right	(100, N)	(100, N)	(-100, N)
Off Right	(100, N)	(100, N)	(-100, N)
Wall Left	(-100, N)	(0, N)	(100, N)
Wall Right	(100, N)	(0, N)	(-100, N)
Tunnel Left	(100, N)	(0, N)	(-100, N)
Tunnel Right	(-100, N)	(0, N)	(100, N)

The above is an expected Q-Table after training with our implemented TD(0) approach. Where N varies between training sessions. The N value for each entry is the total number of times that specific state-action pair is encountered during training (we maintain this count in order to

compute a “progressive average” in temporal difference environments with a history length greater than a single entry.

Reward Process: The agent was rewarded based on a lookup table. Since the action space and state space were small, it was easy enough to make a dictionary of all the possible scenarios that the agent would encounter and make rewards for them. A tabular representation of the reward lookup dictionary is shown below. Notice that because we trained with a TD(0) approach, the Q-Table converges to exactly match the reward table. If we modified the time-distribution to back-propagate rewards over multiple states in recent history, these values may change. Such an approach is supported, with a gamma of 1 currently enabled, but for final submission, we utilize a history-length of exactly one past state-action pair.

Reward Table

State/Action	Left	Throttle	Right
Off Left	-100	100	100
Near Left	-100	100	100
Center	0	100	0
Near Right	100	100	-100
Off Right	100	100	-100
Wall Left	-100	0	100
Wall Right	100	0	-100
Tunnel Left	100	0	-100
Tunnel Right	-100	0	100

Emulation: The chosen emulators for the project were Mupen64 for the Nintendo 64 emulation of the game and the Dolphin emulator for the GameCube emulation of the game. In order to get the pixel data from the emulator to the program was by using the MSS Python library (Multiple-Screen-Shots), which allows for constant screen recording in real time with incredibly low overhead. In order to handle keyboard interaction between the agent and the emulator the Keyboard Python library was used. One important note is that the Keyboard library is only available on Linux and Windows, but it isn’t supported on Mac. This library was chosen because it was one of the few libraries that simulated true Operating-System level events that the

emulators would recognize as being legitimate. However, some emulators (like Mupen64) are known to "swallow" event callbacks for arrow keys. This means that the program could send over an arrow key, say the left one for example, the emulator wouldn't recognize it. However, most emulators have no problems with events that come from the alphabetical keys on the keyboard, so if one were to reproduce the results shown they would need to adjust their mappings in the emulator settings in order to reflect only the alphabetical keys. The ones used in the current project are:

Emulator/Key	Left	Throttle	Right
Mupen64	L	X	R
Dolphin	L	X	R

In order to ensure that the events are truly processed by the emulator, a delay of 0.15 seconds is used for each key to be "held down" by the agent. If the key press is much faster than that, it will not be registered by all emulators. This cuts the granularity of control down by quite a lot, marking only 6 complete presses that can be made per second by the agent. However, this is a library limitation, and there is little that can be done about it.

III. Results

Nintendo 64 Version: The agent seems to understand the environment well, and it also makes good choices when it is near a wall or side of the road. The problems come in when the agent moves too quickly or is near a stretch of road where the walls are very far to the sides. Because the agent doesn't have a concept of the curvature of the track, the agent will recognize the upcoming curvature as still being in one of those regions that our state space is divided into. It's only once the curvature becomes drastic, or the agent moves too quickly, that the agent will change from one state to another rapidly. For example, if the agent is moving to the right and the agent is in the center state, but the road has a strong curvature to the left, the agent will quickly go from being in the center state to being on the right wall. This can be especially dangerous for the agent when it is near a corner. On most of the attempts to complete a lap, the agent would very rarely be able to get past the tunnel on Luigi's Raceway for this reason. The geometry of the map near the tunnel has a mountain that forms an intersection with the road, creating a roadblock everywhere but in the center state. The agent made it through this level once or twice, but it took numerous attempts, and one of those attempts was only after the agent was manually corrected to get past the tunnel. The best time the agent got on a single lap was 8 minutes and 22 seconds. In other words, the agent is not reliably capable of completing a single lap in the Time Trials mode

on the Nintendo 64 version of the game, and even when it does complete a lap it doesn't do so in any reasonable manner.

GameCube Version: As could be expected, the GameCube version didn't do any better. The more graphically intense game greatly confuses the model, even despite the extra training data that was provided for the Double Dash game. In addition to the more graphically intense environment, there are more animated objects that move in the background which make it harder to train our model. The model is currently not capable of getting around corners in this mode either. The hope was that the smoother geometry would create less corners that the agent could get stuck on, but regardless our agent found a way to still get stuck quite frequently. The agent was tested in a few different maps in this mode to see if it had any kind of ability to play the game, but no map that was tested showed any promise. The track it did the best on in this mode was Luigi Circuit, but that is because it is the same map as Luigi's Raceway. Since the track that exists in both games, it got trained on the data for that map twice. This was enough to make it slightly biased to that track. In the first edition of the model there was no training data provided from the Double Dash game, but this made no difference considering the agent could still not complete a single lap in this mode. This shows that the training bias on Luigi's Raceway/ Luigi's Circuit was irrelevant in this case. Though this overcame our initial hypothesis that the state classifier was to blame for the poor performance initially, expanding the classifier's training dataset was not sufficient to overcome some of the underlying issues -- our inability to access the game's variable data (lap time, hitboxes, etc.) and the limitations of our classifier to not consider curvature or motion.

IV. Discussion

Problems

1. Since the model is so generalized, it prevented it from being decent at any game it was tasked with playing. In the process of trying to find a reliable way for the agent to play any Mario-Kart game it was given, it ended up being bad at all of them in just about equal ways. While the agent did a little bit better on the Nintendo 64 version of the game, it still is generally considered to not be successful in this regard. In other terms, we found the conclusion that a special agent will always outperform a generic agent at the task it has been specialized for.

2. The model lacks the kind of environmental information that other approaches have used that made them successful. Other models online, such as TensorKart by Kevin Hughes uses a supervised learning approach. He trained his model on data that he recorded himself playing as a target value function, and then through value function approximation methods he was able to train his model to play like he did. Another approach that was

taken into consideration was Rameshvarun's NeuralKart approach where they built Lua scripts as an embedded emulator plugin so that they could extract information from ROM files and maintain direct communication to the emulator from their program. This was too costly in terms of time, and it is a very difficult task to write an emulator plugin from scratch. Those approaches gave the developers much more control over their model, as they had access to much more refined and accurate information about their environment. These approaches would not work for us since the mechanics differ greatly from each version of the game, and writing an emulator plugin would be ineffective for our approach since our goal was to target games across multiple emulators and consoles. Because of this, we only had access to the same visual information that a human player would be able to access -- the game image itself. We ran into an unexpected problem that was raised from this issue -- we are unable to reward based on lap-time/score/hitbox location/distance traveled in correct direction/etc. -- we couldn't parse that data out of the video frames algorithmically. The only information we had to access for rewarding state-action pairs was the previous and the current state-action pairs, which results in our agent failing to learn any complex behaviors or strategies, since it could only act based on one of 9 states to reach the most optimal of the 9 for rewards.

3. Our approach changed many times throughout the course of development. Initially, we tried to gather state space information by algorithmically detecting the roads and obstacles, which relied heavily on an algorithm called "Template Matching", which relies on sliding a "template" image across the image frame, and calculating the mean-squared-error for each position of the template and selecting the position with the lowest error. Of course, template matching is very slow and couldn't keep up with the game's 30fps framerate. After modifying the program to skip frames and not render the gameplay at the same time it was slightly better, but still not even remotely fast enough to be able to be used in the project. After abandoning the Template Matching algorithm, we rewrote our state-aggregation algorithm to utilize a new approach: Canny Edge Detection, which computes the gradient between pixels in the image and marks an "edge" at pixels whose gradient is above a specified threshold, was also explored in order to see if reducing the information in the model would be a better approach (we took a representative 6 "stripes" horizontally across the frame to compute the canny edges for, and calculated whether the average edge was left or right of the center of the frame for each row), but this failed to produce anything worthwhile as this approach would consider driving into a wall or perpendicular to the track as being "perfectly centered". The final approach came very late in development -- the use of a supervised CNN classifier to interpret the input frame as one of nine classes (states), which left us with little time to tweak the model and do the kind of parameter tuning that might have allowed it to perform in a more reasonable manner.

4. When we initially trained and developed our models, we were unprepared for how dramatically different the Luigi's Raceway map was between the two games. This provided a new set of issues relating to driving in unfamiliar environments between the two tracks, especially since the map on the GameCube version has multiple routes which can be traversed (there are forks in the road), and the GameCube version does not contain the tunnel that the classifier was trained for in the N64 version.

Future Works/Improvements

1. Had time allowed, we would like to consider hyper-parameter tuning for our CNN classifier in order to reduce the loss in classification, or use a differently structured. Likewise, we would like to see how other network structures perform (such as google's Inception model instead of our current network architecture).

2. One possible improvement would be to try to use a continuous state space by using a Deep Neural Network to examine the pixel data of the screen and use that as our state. This would mean that we'd be using a value function approximation technique like Gradient Descent in order to solve for our weights for the network. The reason this was not explored initially is because it takes a long time to train the network, and that training time becomes exponentially larger when the agent needs to perform on many different environments like ours was intended to do. In addition, there was not even a guarantee that this approach would work, so it was left as a last resort.We feel that we may have seen greater success utilizing a Deep Q-Learning approach over our current state-aggregation classification.

3. It might be best to try to train the model to perform well at one game first with a slow method like template matching and train a Q-Table as a kind of pseudo-value function. From there, move to another game and train a new Q-Table. After a few different Q-Tables have been trained, it might be better to aggregate them into a new Q-Table (maybe through weighted averaging) to try to get a model that performs well on more than 1 game.

4. We would like to utilize emulator plugins in order to access game data to further improve our reward function. We would like to use lap completion time, or time spent on the track as rewards rather than a state aggregation. If we could reward without the use of frame-image interpretation, we could use a better VFA function to learn a weights vector to predict reward for each action instead of a CNN classifier to aggregate the states. This is because we know that state aggregation will rarely outperform other methods of VFA.

5. We believe it would be interesting to see how our current, and potential future, implementations would perform if trained and implemented on other tracks (perhaps tracks which share more similarities between the two versions of the game). We think it would be useful information to see how constrained our model was by the specific tracks it was used, and whether the tracks are interchangeable, or if the implementation could only find success on the "known" tracks.

6. We were inspired by the attempt to play games by other teams where they use multiple (4+ frames) to capture the "sense of motion" with respect to the game's state. Had time allowed, we would like to have trained our classifier to take a 4-frame history as input to compute a richer state space that takes motion into consideration, rather than a frame-by-frame history of the continuous state space.

V. References (Will format later)

[1] Article on the Use of Grand Theft Auto V for Training Autonomous Driving: https://www.thesun.co.uk/motors/3355779/researchers-using-grand-theft-auto-video-game-to-train-driverless-car-technology/

[2] Sample Reference to a Neural Network Implementation with the use of Emulator Plugins: https://github.com/rameshvarun/NeuralKart

[3] Reference Implementation with the use of Tensorflow for Neural Network Training: https://github.com/kevinhughes27/TensorKart

[4] Article on the Use and Implementation of Template Matching in OpenCV: https://towardsdatascience.com/object-detection-on-python-using-template-matching-ab4243a0ca62

[5] N64 Implementation of Neural Network to play Mario Kart: https://www.youtube.com/watch?v=Eo07BAsyQ24

[6] Further Reference Implementations on the N64 Version of Mario Kart: https://news.developer.nvidia.com/watch-an-ai-play-mario-kart/

[7] Q Learning Implementation on the SNES version of Mario Kart (name inspiration for project): https://www.youtube.com/watch?v=Tnu4O_xEmVk

[8] More In-Depth Examination of the MarIQ Implementation: https://www.youtube.com/watch?v=BQifVQPnRQk