NeuralKart: A Real-Time Mario Kart 64 AI

https://github.com/rameshvarun/NeuralKart

Harrison Ho
Stanford University
h2o@stanford.edu

Varun Ramesh
stanford University
vramesh2@stanford.edu

Eduardo Torres Montaño
Stanford University
torresm9@stanford.edu

Abstract

We developed a real-time Mario Kart 64 autopilot which trains and plays without human intervention. Our model has two main components. First, an omniscient search AI with complete control of the emulator simulates different possible actions and generates a training set associating screenshots with a steering angle. Second, a convolutional neural network (CNN) trains on the resulting dataset. Finally, to increase our ability to recover from errors, we randomly sample states from the CNN during real-time play and run the search AI from those states to augment the dataset. The resulting autopilot bot is independently able to recognize road features and correct over- and under-steering while playing Mario Kart 64. Videos of the autopilot playing in real-time are available at `https://www.youtube.com/playlist?list=PLSHD7WB3aI6Ks04Z7kS_UskyG_uY02EzY`.

1. Introduction

The popular kart racing game Mario Kart 64 presents an opportunity to develop real-time autopilot controllers in a simplified scenario that resembles real autonomous driving. As an Nintendo 64 video game, tools exist to examine and manipulate machine state, which allows us to generate large datasets rapidly. At the same time, Mario Kart introduces unique challenges – racers must navigate hazards and jumps, as well as use items effectively. We wanted to develop an real-time autopilot that could complete races and avoid hazards, while only looking at an image of the screen.

To simplify the problem, we constrain the autopilot to constantly hold accelerate, as Mario Kart races can be easily completed without braking. We also ignore the ability to drift as well as use items. With these constraints, the autopilot simply needs to return a steering value for a given screenshot of the game.

Because we only take the screen as input, we need to extract features that can tell us about the terrain and hazards in front of the kart. However, there is a wide variety of terrain textures across the different tracks in Mario Kart, and trying to hard-code feature extractors is infeasible. By using deep learning, specifically CNNs, we can automatically learn feature extraction while training our model end-to-end.

Our problem lies at the intersection between three fields of research: real-time deep learning controllers, autonomous driving, and game-playing. Thus we combine the research in each of these fields to develop an approach that yields competitive performance on Mario Kart tracks.

2. Background / Related Work

2.1. Imitation Learning

Real-time deep learning controllers are often trained using imitation learning. In imitation learning, an expert is recorded performing the task, and observations and resulting actions are recorded at each time-step. A neural network is then trained using these recordings as a dataset, thus learning to "imitate" the expert. The potential for imitation learning and neural networks in applications such as robotics has been noted since the late 1990s [17].

However, imitation learning controllers suffer from a fundamental distribution mismatch problem. In practical terms, experts are often too good, and rarely find themselves in error states from which they must recover. Thus, the controller never learns to correct itself or recover, and small errors in prediction accumulate over time. Ross, et. al introduce the DAGGER (dataset aggregation) algorithm which resolves this issue [16]. After initially training a weak model from human input, they run the controller and sample observations from the resulting trajectories. Then, a human manually labels the sampled observations. The dataset is then augmented with the new human-labeled data and the model is retrained. The resulting model performs well on games such as Super Mario Bros. and Super Tux Kart, a 3D racing game similar to Mario Kart.

2.2. Reinforcement Learning

Deep learning controllers can also be trained using reinforcement learning. Unlike imitation learning, reinforcement learning requires no human input at all. Instead, the AI repeatedly tries to execute runs, some of which will be more successful than others. The AI then modifies the network to make successful runs more likely.

Much of the deep reinforcement learning literature has been evaluated in the Arcade Learning Environment (ALE), which provides emulated versions of many Atari games [6]. The ALE also provides reward functions for the games, which is a requirement for deep learning.

Several deep reinforcement learning algorithms have been introduced. Deep Q Networks (DQN) were developed to play in the ALE [14], resulting in human-level performance. DQNs are regressions which learn to map an observation to expected rewards for each available action. The controller runs the DQN and selects the actions with the highest expected value. Newer techniques include deep deterministic policy gradients, which learn to map an observation directly to an action, and can thus operate over continuous action spaces [13].

By contrast to imitation learning, which is a form of supervised learning, reinforcement learning methods are more complicated and take longer to converge. However, they sidestep the distribution mismatch problem, as the AI only ever trains on data that was generated by the AI itself.

2.3. Game Playing

As mentioned above, Mnih et. al first applied deep learning to play Atari games in real-time [14]. Guo et. al significantly improve upon the DQN results, developing the best real-time Atari game player to date [11]. They first create a planning-based agent that can read memory and simulate many possibilities. This agent receives far higher scores than the DQN agent, although it cannot run in real-time. They then train a CNN model to imitate the actions proposed by the search-based agent. The new CNN model performs worse than the planning agent, but better than the DQN agent. Our approach is fundamentally based off of the strategy used in this paper, where we train a CNN to imitate our own offline search agent. Guo et. al also use a form of DAGGER to resolve issues with imitation learning. Finally, CNNs and imitation learning have been applied to other interactive video games such as Super Smash Bros [8].

2.4. Autonomous Vehicles

The earliest application of neural networks to autonomous vehicles, from 1989, is ALVINN, where a three-layer fully connected network was trained to map road images to recorded steering data [15]. Since then, companies and universities, such as Google and Stanford, have pursued autonomous driving systems [4] [12]. These systems often aggregate over a multitude of features to predict optimal trajectories.

In 2016, Researchers at NVIDIA designed an modern end-to-end system for training self-driving cars using CNNs [7]. For the training set, they collected 72 hours of driving data in different weather conditions, associating images collected from a front-facing camera with the steering angle. In addition, they augment the dataset with shifts and rotations to inform the network how to recover from poor positions or orientations, and train a CNN on the resulting dataset. Their model performs well and is able to drive autonomously approximately 98% of the time on a normal drive. The CNN architecture used in our autopilot is a slightly modified version of the architecture first introduced in this paper.

2.5. Mario Kart 64

The NEAT algorithm (Neural Evolution of Augmenting Topologies) has previously been applied to Mario Kart 64 [2], and the resulting model is able to use advanced techniques such as drifting and using items at opportune times. However, it "cheats" by reading game registers during real-time play, information not directly accessible by human players. Our model differs by only relying on the game screen during real-time play, but can read game registers during training. In general, NEAT has only been applied to shallow networks, and thus is suitable for tasks where features have already been extracted [18]. It remains to be seen if NEAT can be used to evolve CNN architectures, though some work does exist in that area [9].

Previous work has been done to apply CNNs to Mario Kart 64. TensorKart learns to map screenshots to human inputs, and is also able to generalize training data over different track scenarios [3]. It uses the model developed by the NVIDIA autopilot paper. However, as a pure imitation-learning system, it cannot recover well from error conditions. In addition, it requires unnatural human play; turns must be performed gradually for TensorKart's CNN to learn properly, and turns which fluctuate in steering angles confuse the AI. Despite these shortcomings, we used the TensorKart model and training code as a starting point, ultimately eliminating the need for human game-play entirely.

3. Method

3.1. Bizhawk

In order to play Mario Kart races in an automated way, we take advantage of the Bizhawk emulator. Bizhawk provides an interface to run Lua scripts while playing Mario Kart, which allows us to save/load states, play for any number of frames, access in-game memory locations, and save screenshots. In addition, we can programmatically determine which buttons are pressed at any given time.

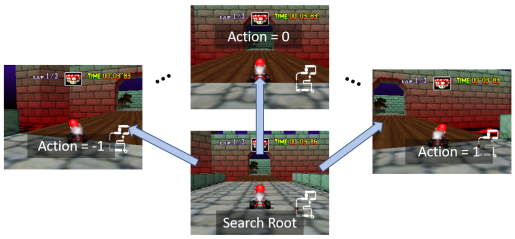

Figure 1. A demonstration of the search process. The search AI simulates the outcomes of 11 different angles, chooses the angle yielding the greatest progress, and stores the search root image and steering angle as a single datapoint.

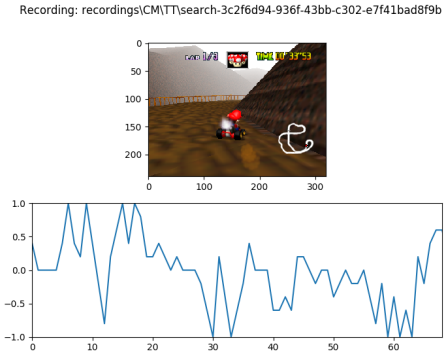

Figure 2. The bottom graph displays the steering values that the search AI has chosen up to the current point in time.

3.2. Search AI

The first component of our approach is a search based AI, which can determine the best steering action to take from a given game state. The search AI runs offline, using the Bizhawk emulator to simulate different actions. During a search, the AI saves its current position as the root state. It then tries 11 different steering values from this root state and simulates the results of the gameplay for 30 frames. The search AI chooses the angle associated with the greatest reward, which simply consists of a weighted sum of the current progress (a float between 0 and 3 indicating how much of a race has been completed) and the current kart speed. Both of these values can be read using in-game memory addresses. Finally, the search AI proceeds, using the selected angle, for 30 frames, and repeats the process using the new state as the next root state. Figure 1 demonstrates the search process visually.

To collect data, we save the root state game screen and the corresponding angle chosen by the search AI. This creates recordings, as shown in Figure 2. In total, we have collected 18658 training examples across four tracks, with a 10% randomly chosen validation split.

3.3. Real-time CNN

The second component of our model is a convolutional neural network. Our network incorporates 5 batch normalization-2D convolution-ReLU layers, followed by 5 dense layers that end in a regression. It uses an input shape of 200x66, which means that the input images are resized before being processed.

Our model is based off of TensorKart's CNN architecture, which is itself a Keras implementation of NVIDIA's autopilot model [7]. Our model modifies the prior CNNs by including several batch normalization layers, which we found helped with reducing over-fitting and smoothing turns taken by the CNN. We train the CNN on the dataset generated by the search AI, using a euclidean loss. Training is performed in Keras, using the Tensorflow backend [10] [5]. The network is trained with an Adam optimizer. At each epoch, we only save our weights if the validation loss has decreased.

Each track's recordings are treated as a separate dataset. We train the model separately on each track, saving a separate weights file. See Section 5.3 for results when we train on all tracks together.

3.4. DAGGER Algorithm

Because the search AI can see future consequences for any action, it rarely enters error states, such as when a driver is slowly drifting off the road and needs to correct course. As a result, training the CNN on the search AI alone can yield poor performance; errors in the outputs of the CNN will compound, and the CNN simply doesn't know how to recover.

To resolve this, we use the DAGGER algorithm. We first run the search AI by itself on the track; the resulting data is used to initialize the weights of the CNN. Next, we allow the CNN to play using its predicted steering angles. We then randomly pause the CNN and run the search AI from the current point. We run the search AI for 120 frames and save image-steering angle pairs; the resulting pairs are used to augment the dataset with which we retrain the CNN. Every time we train, we use the previous weights as an initialization. The interaction between the CNN and search AIs is demonstrated in figure 3.

The constants we chose for alternating between running the search AI and the CNN could potentially be tuned further. In practice, we found that the search AI was able to recover from error states within 120 frames (2 seconds), which gave good examples for escaping such conditions.

3.5. Playing in Real-time

In order to play a game with the CNN AI, we start a TCP Python server that loads the Keras model. The server has a simple line-oriented protocol where clients can send requests for predictions and receive floating points in return.

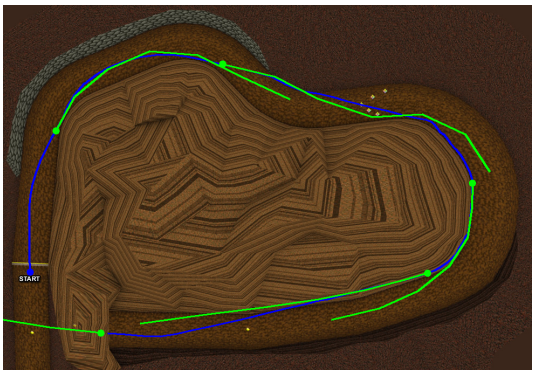

Figure 3. The paths that the kart takes using the DAGGER approach. The blue line shows the path of the CNN AI playing in real-time. The green lines show the trajectories chosen by the search AI when started at states randomly sampled from the CNN AI's play-through.

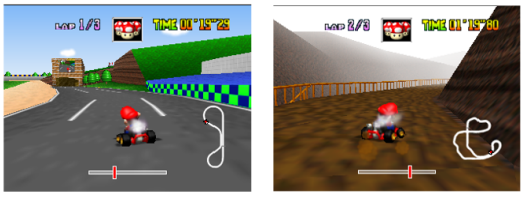

Figure 4. Examples of real-time play, with a slider depicting the output of the network.

Next, a Lua script running in Bizhawk connects to the server using the LuaSocket library. As fast as it can, the Lua script takes a screenshot, sends a request to the server, receives the prediction, and sets the joystick value. For debugging purposes, we draw a slider on the game screen which represents the chosen steering as output by the network.

All of the networking is done asynchronously, meaning that the game doesn't halt while we wait for predictions. Although the CNN is deterministic, the random variations in network timing mean that the trajectory taken by the autopilot is different every single time.

3.6. Input Remapping

N64 joysticks return signed bytes, which range from -128 to 127. When originally developing the search AI, we linearly interpolated our potential angles from this range. Unfortunately, most of this space is a dead-zone, so many of the steering choices resulted in identical trajectories. Furthermore, the horizontal displacement of a turn is not linear w.r.t. the joystick value. This resulted in some trajectories that were too similar and others that had noticeable gaps in between. This is undesirable, as the search AI should have a set of distinct trajectories that uniformly cover the track in front of the kart. To solve these issues, we came up with a mapping function $J(s)$ that maps a "steer" input domain $s \in [-1, 1]$ to joystick values, such that gaps between trajectories are evenly spaced and there are no repeated trajectories. Figure 5 shows that our input remapping scheme has a nearly linear relationship with horizontal displacement.

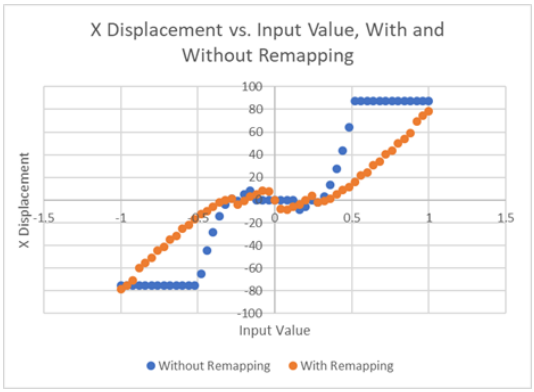

Figure 5. The horizontal displacement of the player with respect to the input trajectory is non-linear, and most of the values are taken up by dead zones. Our input remapping removes the dead-zones and makes the displacement linear with respect to the input value.

$$\alpha(s) = (\text{sgn}(s) \times \sqrt{0.24 \times |s| + 0.01} + 1)/2$$
$$J(s) = \lfloor -128(1 - \alpha(s)) + 127\alpha(s) \rfloor$$

During search, we apply $J(s)$ before taking any action in the emulator; the value that we save in our recording is the value of s. We train on values of s, and predict values of s, which changes how our loss function responds to steering error. While playing the game, we calculate $J(s)$ before we send any predictions to the emulator.

4. Results

4.1. Quantitative Evaluation

We evaluated our autopilot based on its achieved time in the single-player time trial mode. For each track, we run 10 races in real-time and calculate the mean race time as reported by the in-game timer.

We ran our model on four different courses in Mario Kart 64: Luigi's Raceway, Moo Moo Farm, Choco Mountain, and Rainbow Road. We chose these courses for two reasons. First, these courses have walls throughout the track; on some courses without well-defined walls, the model would drive off road or fall off, slowing training progress. Second, some courses do not have well-defined progress waypoints. The search AI relies on game-defined waypoints to determine progress through the courses. Some waypoints either do not reflect a track accurately or are placed around the border of certain tracks, causing the search AI to drive on a suboptimal trajectory.

For comparison, we had a human test-drive each track twice: the first time to familiarize with the track, and the

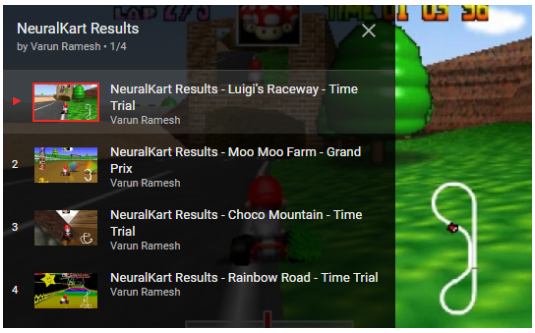

Figure 6. Recordings of our AI racing on various tracks are available at `https://www.youtube.com/playlist?list=PLSHD7WB3aI6Ks04Z7kS_UskyG_uY02EzY`.

second time to record their time. The human follows the same limitations as the AI, and cannot brake, drift, or use items. The resulting times are displayed in table 1.

Track	Autopilot Time (s)	Human Time (s)
Moo Moo Farm	97.46	94.07
Luigi's Raceway	129.09, 1 DNF*	125.30
Choco Mountain	138.37, 2 DNF*	129.50
Rainbow Road	389.18	365.60

Table 1. Achieved track times for the autopilot bot and the human; the autopilot times have been averaged over 10 runs. *DNF signifies that the autopilot got stuck and was unable to finish some number of races.

As seen in the times, the autopilot performs slightly worse than human players, but still yields competitive performance. The autopilot performs best on Moo Moo Farm and Luigi's Raceway, both of which have gentle turns. In contrast, Choco Mountain and Rainbow Road have sharper turns, a thinner raceway, and closer walls; navigating these tracks without accidentally bumping into walls and losing speed is a challenge for the autopilot.

4.2. Qualitative Evaluation

After inspecting our performance, we found that our AI on Luigi's Raceway and Moo Moo Farm was actually stable to perturbations by an external force. This is shown in Figure 7, and demonstrates how effectual the DAGGER iteration process is.

We also found that the autopilot is capable of making short, quick adjustments, as opposed to choosing a stable steering angle for an entire turn. This occurs even though the CNN uses a regression over the different steering angles, instead of classification. The steering behavior resembles how a human would play Mario Kart; instead of choosing a single continuous angle for an entire turn, human players often use short quick adjustments. The prior work, TensorKart, did not observe this behavior.

Figure 7. On Luigi's Raceway, our AI is stable to perturbations. Here, an actual joystick is overriding our AI, pushing it to the right. However, the AI correctly sees that the proper response is to turn to the left. We don't observe the same level of stability on every track.

Figure 8. Our AI is trained in Time-Trial mode, but can still race in Grand Prix mode. Grand Prix introduces new UI elements, item boxes, opponents, and hazards like bananas.

Although all of our training was done in Time-Trial mode, we found that our AI could race quite well on Luigi's Raceway and Moo Moo Farm in Grand Prix mode (shown in Figure 8). This means that the AI is able to ignore the information added by new elements that appear only in Grand Prix mode, despite never having seen those elements before.

We examined situations where the autopilot would slow down. In many cases, the autopilot would slide against walls or drive on the edge of the road next to sand or grass, both of which slow down the kart. An example of the latter is demonstrated in figure 9. We believe this is a result of the search AI (which is deterministic) not understanding risky situations that are likely to lead into error states during real-time play. The CNN then inherits this risky behavior, but is unable to execute it exactly, thus sliding off the road or into a wall.

On Choco Mountain, we observe that the AI bumps into the wall quite frequently while turning, which is the pri-

Figure 9. The autopilot drives halfway on the road and halfway on the grass. The autopilot often behaves sub-optimally in risky situations.

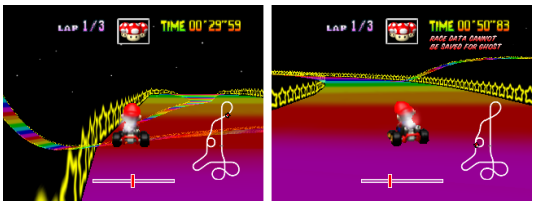

Figure 10. On Rainbow Road, the autopilot takes the sharper turns better than the wide turns.

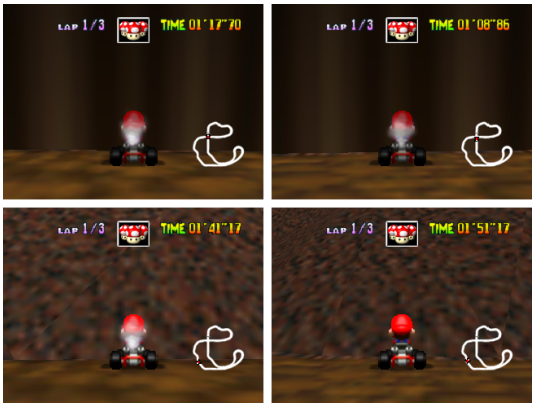

Figure 11. If the AI finds itself staring at a wall, its not obvious whether it should turn right or left. The images in the first column should correspond to a left turn, and the images on the right column should correspond to a right turn.

mary cause of slowdown on that track. This may be due to sharp turns on the track, which the model is not equipped to handle. On Rainbow Road, we observe similar behavior at several turns. However, we found that the AI actually takes the sharpest turns on Rainbow Road quite well. In fact, it tends to bump into the wall only for the wider turns, as seen in Figure 10. This may be due the sharper turns simply being more evident in the down-sampled image fed into the network.

The primary cause of unfinished runs is due to situations

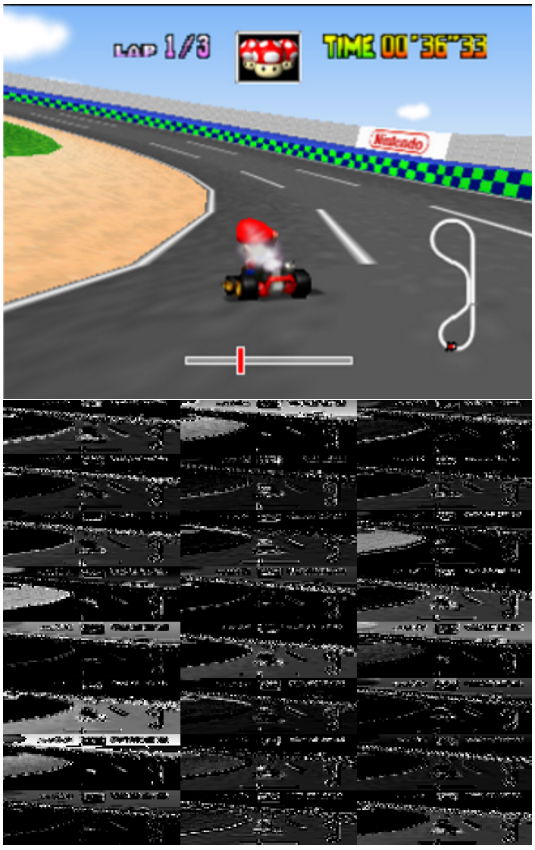

Figure 12. A visualization of the first layer activation functions for a single input image. Certain features are distinctly emphasized, such as the sand and the road.

such as those in Figure 11, where the AI finds itself staring head-on into a wall. Because walls on the left and right-hand sides of the track often have the same texture, the AI does not understand its orientation. It often simply outputs 0, thus getting stuck, or picks the wrong direction and starts to go in reverse (which is detected as an unfinished run).

Our AI is unable to handle situations where it may have to turn around or drive backwards to recover. When the AI is turned around, it typically begins to drive the course in reverse, oblivious that it is making negative progress.

4.3. Network Visualizations

To see what kinds of image features the network was looking for, we generated activation maps for a selection of filters from the first convolutional layer, shown in Figure 12. The activation maps suggest that the network is able to correctly isolate pixels corresponding to the road, walls, and sand.

We also generated saliency maps and class activation maps using the Python package `keras-vis` [1]. Our usage is somewhat unusual, as these visualizations are designed to debug classification models, but our model is a

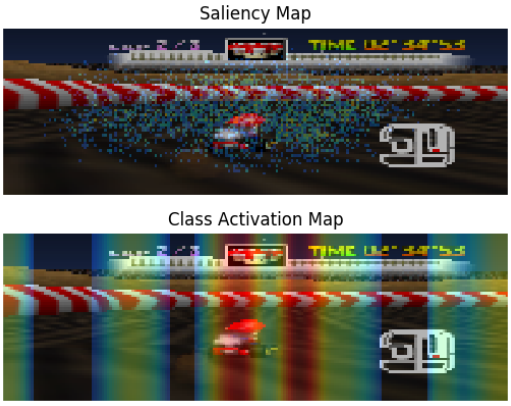

Figure 13. Our saliency map suggests that we successfully ignore the UI and minimap, concentrating attention instead on pixels in the center. The class activation map is harder to interpret.

regression. Thus, we only visualize images corresponding to a right turn (positive steer). The saliency and class activation maps will then reveal the portions of the image that correspond with an increased output value, thus contributing to a positive steer. Unfortunately, both visualizations, shown in Figure 13 are hard to interpret.

The saliency map suggests that a set of pixels clustered near the center of the image are responsible for the decision to turn right. This at least implies that the CNN ignores the UI elements as well as the minimap. The class activation map is even harder to interpret. The map is split into vertical bands due to the last convolutional layer of our network outputting 1×18 filter activations. Thus, vertical information is lost, and our fully-connected layer only operates across horizontal activations.

5. Experiments

5.1. Image Reflection

In order to generate more training samples and improve generalization, we attempted to reflect our training images horizontally, along with the associated steering direction. We concatenated the normal dataset and the reflected dataset together and re-trained. We found that our model stopped taking all turns and continuously outputted steering values near 0. This suggests that our model may not be looking at the curvature of the road in order to determine steering, and may actually be looking at the textures of various walls and terrains for decision making.

5.2. Classification-based Model

We tested a discrete model where, instead of outputting a steering value through regression, the CNN would classify an image into 11 different categories. Each category represents the 11 possible steering angles that the search AI can take. We used the same base model, with the only change being an output of 11 softmax probabilities and a cross entropy loss function.

The resulting model performed worse than the regression model, and was unable to consistently clear Luigi's Raceway. The model was only able to clear 1 out of 10 runs, with a run time of 310.50 seconds - considerably worse than the regression model's average run time of 129.09 seconds. We observed that the AI would make odd, jerky turns due to the discretized steering angle, and was unable to recover from error states. This may be because misclassifications are treated equally in the cross-entropy loss, when some misclasssifications are objectively worse than others for steering. For example, given that the ground truth steering angle for a state is -0.4, a prediction of -0.6 is better than a prediction of 1.0. The regression model better captures this property.

5.3. Training on All Tracks Together

For the results presented in Table 1, each track has a separate set of training data and generates a separate weights file. We concatenated all of our training data into one dataset and generated a unified weights file for all of the tracks. The race times of the unified weights are shown in Table 2.

Track	Individual Data (s)	All Data (s)
Moo Moo Farm	97.46	97.63
Luigi's Raceway	129.09, 1 DNF	129.03
Choco Mountain	138.37, 2 DNF	131.93, 3 DNF
Rainbow Road	389.18	396.74, 1 DNF

Table 2. The performance of our model when trained on all of the data at once, versus keeping a separate dataset and weights file for each track. DNF signifies that some runs did not finish.

The autopilot performs approximately the same on Moo Moo Farm and Luigi's Raceway, performs slightly better on Choco Mountain, and performs slightly worse on Rainbow Road. This suggests that we may be overfitting to Choco Mountain, and that data from other tracks is helping us generalize. It also suggests that Rainbow Road is not benefiting from data from other tracks, potentially due to the unusual setting and textures present on the track.

5.4. Beam Search

Our search AI uses a single depth level, which for several tracks is enough to play with human-like performance. Unfortunately, the search AI cannot complete some tracks due to difficult turns or misplaced waypoints, which eliminates our ability to train the CNN on that track. To resolve this, we implemented a beam search, which stores the top k results for some positive integer k at each time step. This enabled the search AI to explore multiple time steps in the future without the full cost of an exhaustive search.

Figure 14. From saliency maps, we know that the minimap is largely ignored by the CNN. However, it contains a rough estimate of position and orientation, and could be used to improve performance.

By using beam search with DAGGER iteration, we were able to train our CNN on Mario Raceway, a more difficult track with sharp turns and few walls. Preliminary evaluation of the model gave a mean finishing time of 205.38 seconds, with 6 out of 10 runs finishing. In comparison, a human player achieved a finishing time of 103.76 seconds. As seen here, the model required almost twice as much time to finish the race; more training iterations and tuning of the beam search is needed to achieve better results.

6. Conclusion

Our results demonstrate that end to end neural systems can yield good performance as real-time controllers in games like Mario Kart 64. Imitation learning, which is easier to implement and converges faster than reinforcement learning, can be adapted to be completely autonomous through the use of an offline planning agent. DAGGER iteration can be done automatically in order to develop controller stability.

7. Future Ideas

As shown in Figure 11, in situations where the AI gets stuck on walls, the AI often doesn't know which direction to turn to get back on course. To fix this, we could use traditional computer vision techniques to extract the position and direction of the player's icon on the minimap, shown in Figure 14. These values could then be added as inputs to the dense layers of our network. With the minimap position, the network may be able to tell the difference between two areas of a track that are otherwise indistinguishable.

Our CNN only takes a single screenshot at each evaluation step. We could improve this by adding prior frames to each input, giving our CNN the ability to track features over time. For example, the model may wish to swerve more harshly if obstacles are rapidly approaching, or turn more gently otherwise. This is especially vital in Grand Prix mode, where items can cause the player to slow down or speed up in unpredictable ways.

We may also explore button inputs other than simply steering. In particular, the jump / drift button can yield large differences in track times, and is vital to high-level Mario Kart play. However, this would effectively double our search space.

Reinforcement learning, such as with Deep Q Learning or Policy Gradients, can reward good performance and punish error conditions for our model. This may resolve the behavior described in section 4.2, where the autopilot will drive near sand, walls, or other hazards. These risky situations are likely to lead to error conditions; reinforcement learning can push us towards safer states.

References

[1] Keras visualization toolkit. `https://raghakot.github.io/keras-vis/`. Accessed: 2017-06-12.

[2] Mario kart 64 with neural evolution of augmenting topologies (neat). `https://www.youtube.com/watch?v=tmltm0ZHkHw`. Accessed: 2017-05-15.

[3] Tensorkart: self-driving mariokart with tensorflow. `http://kevinhughes.ca/blog/tensor-kart`. Accessed: 2017-05-01.

[4] What we're driving at. `https://googleblog.blogspot.com/2010/10/what-were-driving-at.html`. Accessed: 2017-06-12.

[5] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 06 2013.

[7] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[8] Z. Chen and D. Yi. The game imitation: Deep supervised convolutional networks for quick video game AI. *CoRR*, abs/1702.05663, 2017.

[9] B. Cheung and C. Sable. Hybrid evolution of convolutional networks. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, volume 1, pages 293–297, Dec 2011.

[10] F. Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[11] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In Z. Ghahramani, M. Welling,

C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3338–3346. Curran Associates, Inc., 2014.

[12] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichman, M. Werling, and S. Thrun. Towards fully autonomous driving: systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, 2011.

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] D. A. Pomerleau. Alvinn, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University, Computer Science Department, 1989.

[16] S. Ross, G. J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011.

[17] S. Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[18] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.