Generalizing Reinforcement Learning Agents on Ferrari Grand Prix Challenge Environments

Pratyush Menon, Nick Ni, Dhruv Patel, Vincent Racine

Colab Links:

Deep Q Network:
https://colab.research.google.com/drive/1OBT5tY2FUY1_c0vONeadSjGVTXkq8mGV

Vanilla Policy Gradient Model:
https://colab.research.google.com/drive/13wol1X8lIgHTIYjIRD0QxbayEvxNOv-X

A2C with PPO and Transfer Learning (unmentioned due to extremely slow training):
https://colab.research.google.com/drive/1P6IvNVTr_K9VsZ1zW8-8ocpxr1k3lWce

Word Count: 2466

Introduction:

In the past few years, it has become clear that deep reinforcement learning (RL) can solve highly complex video games[1] given the right reward function and unlimited time to interact with the environment [2]. However, as most popular RL benchmarks do not generalize well between tasks [3], RL research tends to “train on the test set” boasting results from the same environment it was trained on [2]. For the field of RL to advance, it is critical to mark a proper split between “train” and “test” dataset similar to supervised learning datasets to ensure we create agents that can generalize between tasks [2].

This project aims to create an agent that can cross-task generalize. To achieve this goal, an agent will be trained on the Ferrari Grand Prix Challenge (FGPC) “Base Track Map”, before investigating the performance of the agent on previously unseen tracks in FGPC such as Indianapolis, Belgium, and Japan.

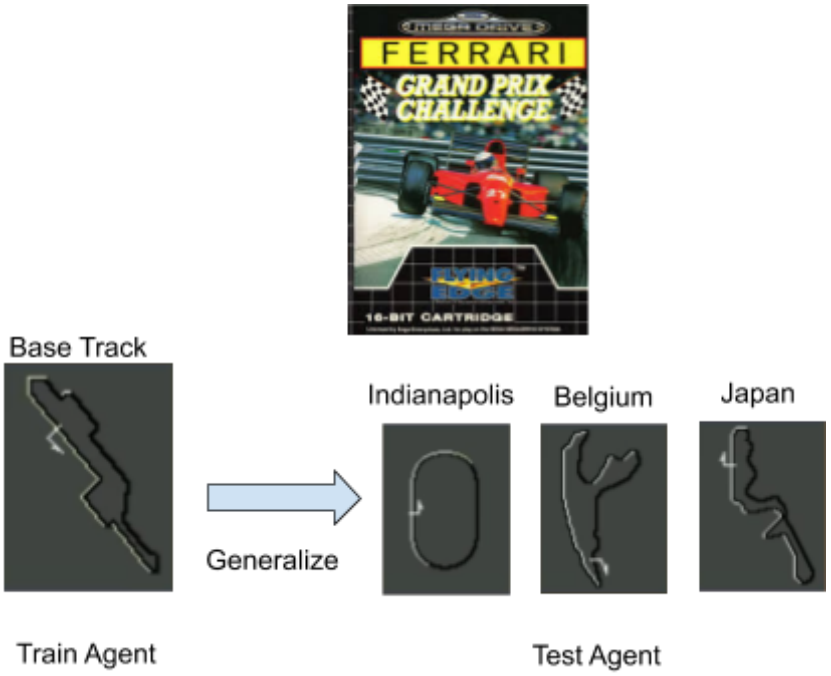

Figure 1: Project Illustration

Background & Related Work:

Two of the most relevant recent studies on transfer learning in video games setting are:
1. A study conducted by OpenAI, looked at using transfer learning as a method to create more generalized agents [2]. The aim of the study was to train a generalized agent to play various games of the *Sonic* franchise, by training the agent on certain levels and then testing it on previously unseen levels. Similarly to our project, the study compared the performance of various algorithms, such as policy gradients and deep Q networks.
2. A study by Pathak et al, found that by training an agent on one level of *Super Mario Bros* with curiosity as an intrinsic reward allowed the agent to enhance performance on two previously unseen *Super Mario Bros* levels [4].

Data Processing:

The three main data processing steps that we took were:

1. Restricting the action space. The SEGA Genesis controller used to play the game FGPC has a total 12 buttons combining to a total of 495 actions. However, most of the actions are either non-existent or only used on the menu screen. We, therefore, restricted the agent's action space to the four actions relevant to driving displayed in Figure 2.

Figure 2 : Ferrari Grand Prix Restricted action space

2. Cropping the raw game image. In order to reduce the number of required computations while not removing any relevant information the RGB game screen image was cropped before being fed into the neural network.

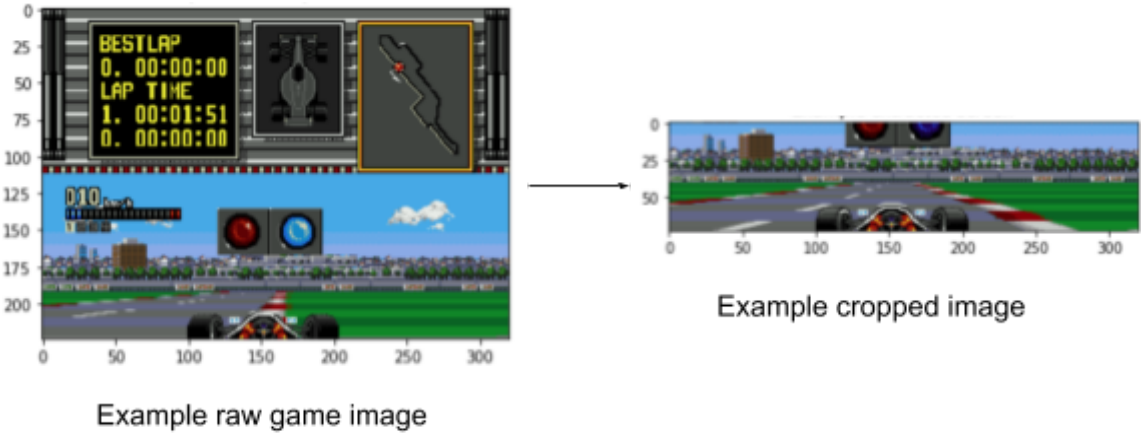

Figure 3: Image Cropping

3. Designed Convolutional layers to extract features from the game screen image. The CNN parameters are trainable. We originally used transfer learning to extract features in order to

reduce the number of computations, but neither Alexnet nor Resnet performed well. Hence we developed a three-layer CNN structured as displayed in Figure 4:

```
(Features): Sequential(
  (0): Conv2d(3, 10, kernel_size=(2, 2), stride=(2, 2))
  (1): BatchNorm2d(10, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU()
  (3): Conv2d(10, 17, kernel_size=(2, 2), stride=(2, 2))
  (4): BatchNorm2d(17, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU()
  (6): Conv2d(17, 25, kernel_size=(2, 2), stride=(2, 2))
  (7): BatchNorm2d(25, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): ReLU()
)
```

Figure 4: CNN structure

The convolutional layers overall maps an RGB image to 25 feature maps with reduced size.

Architecture:

Policy Gradient:

The policy gradient model tries to directly predict the action to choose based on the input state, via a 2 layer linear network (Figure 5), in addition to convolutional layers as described above. Every episode, it picks actions based on the current network parameters, and then it updates the policy based on the discounted rewards earned by each action. One of the main issues observed with the policy gradient model was that from a very early stage it converges to the local minimum of only pressing on the acceleration button. To encourage exploration, we forced the model to choose a completely random action for the entirety of each odd episode. This forced the agent to “see” different actions and improved the performance of the agent. However, it learned very slowly due to only being able to use information from one episode to update weights each iteration. As such, the decision was made to focus on the DQN model.

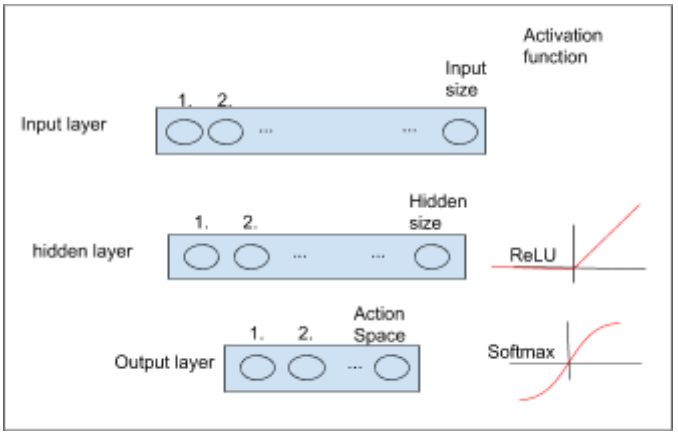

Figure 5: Policy Gradient Architecture

DQN:

The Deep Q-Network (DQN) model builds on the idea of Q-Learning. The main idea behind Q-Learning is that if we knew a function $Q^*(s,\ a)$ which tells us our expected return if we take an action, a in a given state, s then we can simply construct a policy which maximizes our rewards [5]:

$$\pi^*(s)\ =\ argmax_a\ Q^*(s,\ a)$$

where π represents the policy (a mapping between action and state) telling the agent what to do in that state. So, in a DQN, we try to approximate $Q^*(s,\ a)$ by using neural networks to learn the function. This approximate function is denoted as Q. We learn Q by allowing the agent to select an action given a state and observe the next state, and the reward that resulted from it. This transition (state, action, next state, and reward) tuple is stored in a buffer, referred to as the experience replay memory. By sampling random mini batches from it, we form an input dataset to train and update the neural network.

DQN uses the fact that every Q function for some policy π obeys the Bellman equation for the update [5]:

$$Q^\pi(s,\ a)\ =\ r\ +\ \gamma Q^\pi(s',\ \pi(s'))$$

where, $\gamma Q^\pi(s',\ \pi(s'))$ is the discounted future rewards and γ is a discount rate. The difference between $Q^\pi(s,a)$ and $r\ +\ \gamma Q^\pi(s',\ \pi(s'))$ is called the temporal difference (TD) error, δ [5].

$$\delta\ = Q^\pi(s,\ a)\ -\ (\ r\ +\ \gamma Q^\pi(s',\ \pi(s')))$$

In TD error calculation, the target function $Q^\pi(s',\pi(s'))$ is updated frequently, introducing instability, making training more difficult. So we create a target network that temporarily fixes the Q values and only synchronizes with the learned parameters of the latest network occasionally. This ensures that parameter changes don't impact target function immediately.

With this we can aim to minimize the TD error, δ, using Huber loss [5].

$$L\ =\ \tfrac{1}{|B|}\sum_B L(\delta)$$

$$L(\delta)\ =\ \{(\tfrac{1}{2}\delta^2)\ if\ |\delta|\ \leq 1, (|\delta|\ -\ \tfrac{1}{2})\ otherwise\}$$

When the error is small, it acts like mean squared error but when the error is large it acts like a mean absolute error, making it robust to outliers. The loss is computed over mini-batches of transitions (B).

The architecture we decided to use for our Q function and target function has 3 linear layers with ReLU activation in between, with 6000 inputs and 4 final outputs (Figure 6). The input comes from the CNN and output is the Q value for each action.

The model follows a ε- greedy policy, which means that with a probability of ε it will choose uniformly between the actions and the rest of time it will choose the action with highest Q value. Towards the beginning the agent does not have the knowledge of which actions are optimal given the state, so it must explore. As the agent is trained, the agent can pick optimal actions (exploitation). Thus, it was designed so ε is initially large and decreases exponentially as the agent is trained.

The agent always learns to maximize the rewards, thus it is important to design a function such that maximizing the rewards also accomplishes the goal. The initial reward function was the car's velocity, but the agent learned to only accelerate forward to maximize the reward without considering the grass or the collisions. After much experimentation, the final reward function had the form:

$$Reward = \{(S) \text{ if on the road}, (0.0016S^2 - 100) \text{ if on the grass or } S = 0\}$$

This function ensures that when the agent is on the road it maximizes the reward by accelerating and low speeds on the grass are penalized more compared to high speeds. This taught the agent to accelerate on the road and quickly leave the grass which was the intended behaviour.

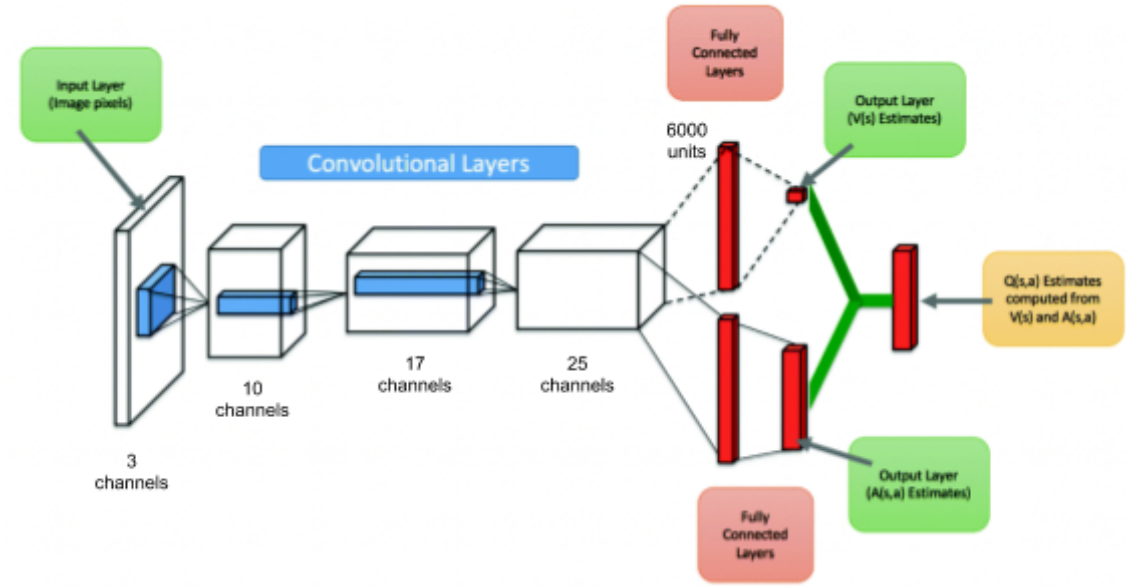

Figure 6: DQN architecture

Baseline Model:

The baseline model is a hand-coded heuristic agent that tries to avoid the grass by looking at pixel values near its position and turning away from any grass (green pixels) it detects (Figure 7) .

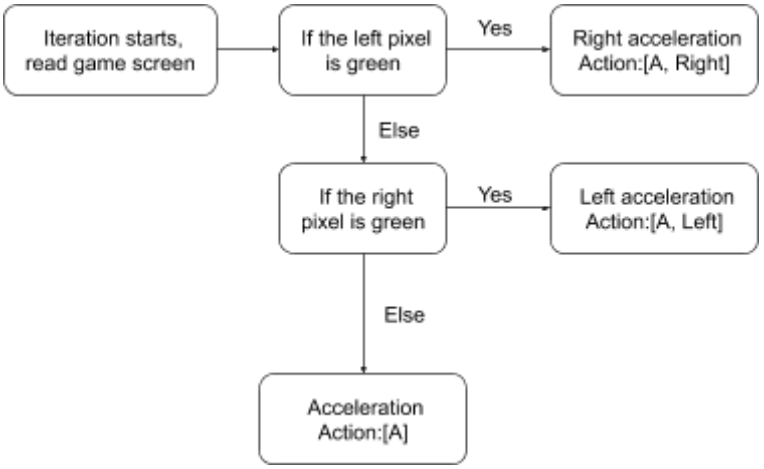

Figure 7: Baseline Algorithm

For example, in figure 8 the red dots in the graph indicate the pixels which will be tested. In this particular state, the right pixel is green and therefore the cart will accelerate to the left.

Figure 8: RGB Game Screen

Discussion of Results:

For quantitative results, the main DQN model as well as the best performing policy gradient model will be discussed.

Policy Gradient Model:

Figure 9 shows that the model initially learned quickly but got worse for quite a while before entering another phase of quick learning.

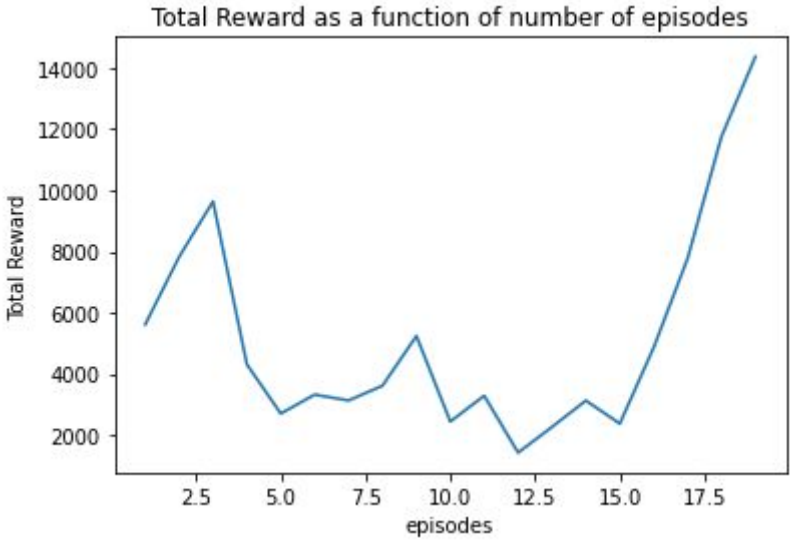

Figure 9: Rewards per non-random episodes of the policy gradient model

Unfortunately, later iterations once again got worse as shown in Figure 10.

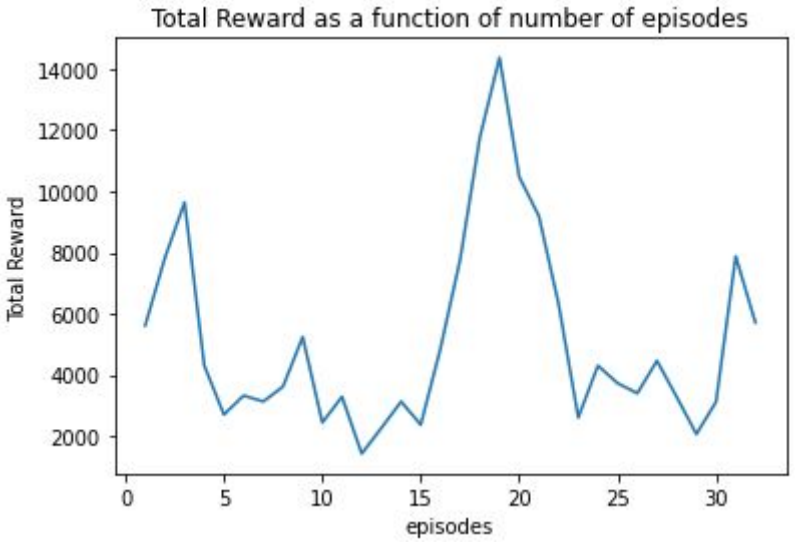

Figure 10: Extended rewards of the policy gradient model

Given more computing power and time, it is quite likely that the model would have reached another phase of quick learning and performed better, but the restrictions of Google Colab prevented further exploration of that theory.

Deep Q Network:

Figure 11: Average Reward per Lap for DQN Model

Figure 11 shows a clear improvement trajectory until it reaches the peak - the model actually used. This was likely due to the improved sample efficiency of DQN models, as will be discussed in more detail later. An interesting side effect of the final reward choice was that there was no longer a significant correlation between reward and average speed, as seen in Figure 12.

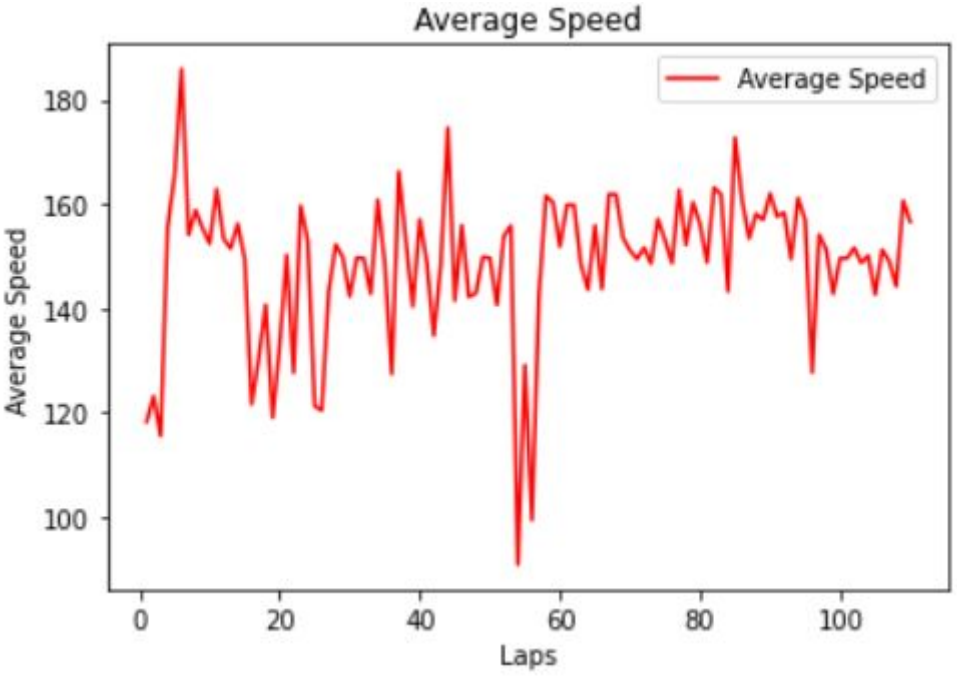

Figure 12: Average Speed per Lap for DQN Model

This is theorized to be because car handling becomes much harder as car speed increases in FGPC, so high average speeds correspond to the agent going off track more often.

Comparisons:
Since the main goal of any racing game is to complete laps as fast as possible, the measurement we used to compare the model was the lap time they each achieved on the base track.

Table 1: Model Success Comparisons

Model	**Lap Time**
Baseline Model	1:01.20
Policy Gradient Model	1:32.58
Deep Q Network	1:00.70

The policy gradient model was unable to beat the baseline model, so it was abandoned. On the other hand, the Deep Q Network performed better, but only slightly.

Qualitative Results:
Focusing on the DQN model, it was clear that it learned to perform more advanced maneuvers as it trained. Originally, it was essentially picking random actions and as such was very slow. It first learned to avoid collisions with billboards, but didn't know that grass was bad. With more training time, it learned to try and avoid grass as well as collisions. This could be seen as it was able to follow curves in the road, make U-turns, and even make sharp turns at high speeds. Examples such as these showed that the model was actually making inferences on what action would perform best based on the current state, proving that the algorithm was working.

Discussion of Performance on New Data:

To ensure the DQN was better than the heuristic model and that it had the ability to generalize to new tasks, comparisons on new environments were also key. Three alternate tracks of varying difficulty were chosen for the models to be tested on: Indianapolis, Belgium, and Japan. The results are shown in Table 2.

Table 2: Performance on New Tracks

Track Name	**DQN Lap Time**	**Baseline Model Lap Time**
Indianapolis	0:38.44	0:28.38
Belgium	1:08.45	1:11.33
Japan	1:19.37	1:31.98

The DQN performed worse than the baseline on the easiest track, but showed better results over the baseline as the difficulty of the tracks ramped up. This is likely since the easiest track was extremely different in shape to the base track that the agent was trained on, as shown in Figure 13.

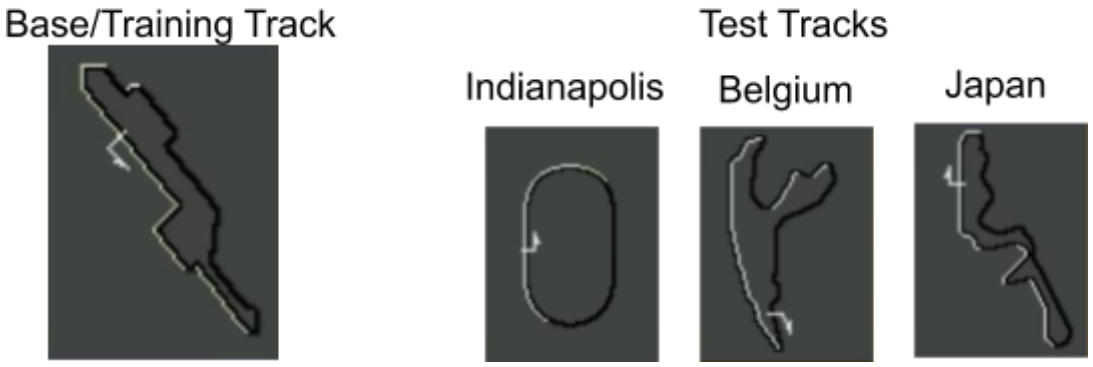

Figure 13: Comparison between base track and test tracks

Since it had not experienced such wide turns as on the Indianapolis track, the agent was unprepared to turn for so long and crashed. However, the agent clearly showed evidence of learning on the more difficult tracks as it knew how to navigate complex turns and zig zags, which the simple heuristic-based model was unable to do. This shows that the agent learned a lot from the base track and performed well on challenging environments; however, the base track alone was not enough to expose it to everything seen in other environments. A possible remedy could have been to choose multiple tracks to train the model on, and different tracks to test it on.

Discussion of Model Selection:

Based on our results, as well as theory, we inferred that DQN models perform better than policy gradient models on this task.

Policy Gradients handle large and continuous action space well, but has large variance updates which implies poor sample efficiencies, as it can only be updated after an episode of data gathering, and data from previous episodes (with different policies) is ignored [6]. DQN handles small and discrete action spaces better. It has small variance updates and better sample efficiencies, since it can be updated every few actions, and is trained on all the data collected from all previous episodes (experience replay memory) [6].

For the same number of episodes of training, DQN is updated much more frequently than PG, and has a lot more training data. With some exploration, the model learns quickly. To sum up, DQN is overall more efficient than PG given that we only have 4 discrete actions.

Ethical Consideration:

Transfer learning is an exciting technique for RL as it allows agents to achieve good performance in new environments with far less training by generalizing more quickly to varied domains [7]. However, since such agents are trained less in each specific environment, it means that they may not have been exposed to as many states in that environment as more specialized agents, which limits the agent's learning opportunities in less probable states. When agents find themselves in unexpected states, this leads to unexpected behaviour. Since our model will be trained on driving environments, a pertinent example would be if a self-driving car found itself in an unexpected state and this caused a car crash. To minimize this risk, it is important to ensure that the agent has been trained heavily on less probable states as much as it gets trained on the more common states before deployment in real life statements.

Future Work:

With more time, better results could have been achieved by further optimizing hyperparameters. There were many hyperparameters in the model, including: Batch Size, Discount Rate, Learning Rate, Exploration Probability, and Reward Function. Since the training process for each episode consists of thousands of steps, the effect of hyperparameter adjustment is seen extremely gradually - making it difficult to optimize such a vast selection of hyperparameters.

Another interesting next step would be to complete what we presented in our project proposal and test if our agent could generalize to other games with similar goals to FGPC, such as *MarioKart.* It would also be interesting to compare our results to other similar studies such as the one conducted by Stanford University where they showed that a network trained on *PuckWorld* was able to train faster and perform better on the game *Snake* than a randomly initialized network [8].

References:

[1] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A survey of deep reinforcement learning in video games,” IEEE, 2019.

[2] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, “Gotta learn fast: A new benchmark for generalization in rl,” Cornell University, 2018.

[3] 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, vol. 47, pp. 253–279, Jun. 2013.

[4] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction,” 2017. eprint: arXiv:1705.05363.

[5] Adam Paszke, “Reinforcement Learning (DQN) Tutorial” [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[6] J. Ba, “Lecture notes on dqn & pg csc413,” March 2020

[7] M. Taylor and P. Stone, “Transfer learning for reinforcement learning domains: A survey,”Journal of Machine Learning Research, 2009.

[8] C. Asawa, C. Elamri, and D. Pan, “Using transfer learning between games to improve deep reinforcement learning performance and stability,” Stanford University, 2017.