# Racetrack Navigation on OpenAIGym with Deep Reinforcement Learning

**Yash Maniyar**
Department of Computer Science
Stanford University
ymaniyar@stanford.edu

**Nidhi Manoj**
Department of Computer Science
Stanford University
nmanoj@stanford.edu

## Abstract

Automated cars and vehicles pose a pressing and challenging technical problem. We tackle car navigation in randomly generated racetrack using deep reinforcement learning techniques such as Double Q-learning (DDQN) and the OpenAI Gym environment. We model a reward system and experimented with a lot of hyperparameter tuning, model architectures, and input image processing. We were able to achieve human-level learning on car navigation with a predicted q-value of 1000 after 18,000 iterations of the environment. By employing more computation resources, dropout, and other DeepRL efforts we can hope to further improve performance in teaching an agent to navigate a racetrack.
Github repo: https://github.com/nidhimanoj4/CS238CarNavigation

## 1 Introduction

Car navigation is integral to many current automated systems. We are interested in the problem of teaching an agent – an “intelligent" car – how to navigate independently around a simulated race track. The car has to take actions (turn, accelerate, and brake certain magnitudes) to follow the grey-marked path and complete the randomly generated track, all while doing so as quickly as possible. Car navigation has a multitude of practical applications and is a challenging, current, and pressing problem that many autonomous vehicle companies are trying to tackle!

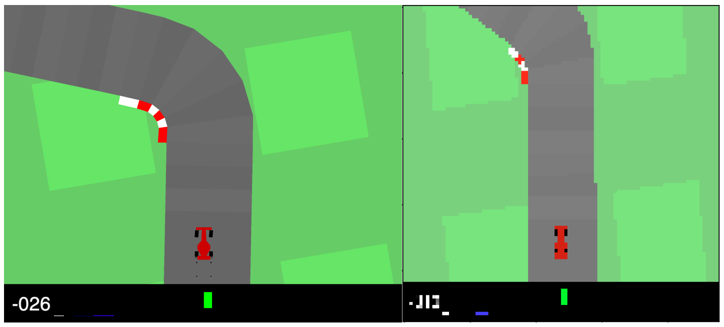
## 2 Problem

### 2.1 Environment

We use OpenAI Gym’s CarRacing-v0 environment (https://gym.openai.com/envs/CarRacing-v0/) to model our car and the world racetrack. A random track is generated in every run of the simulation (hereby referred to as an "episode"). Each state that the agent can observe is a $96 \times 96$ RGB pixel frame, representing a top-down view of an area including and surrounding the car’s current location. Between each frame, the car makes an action, which has three components: steering direction, gas, and brake. Finally, we reward two behaviors in this environment: (1) staying on the track tiles (the rectangular gray sections that make up the racetrack), and (2) completing the course as quickly as possible.

### 2.2 Sources of Uncertainty

The racetrack has a huge state space of $10 * 256^{3*96^2}$ possible states given each state consists of 10 frames of a $96 \times 96$ grid of RGB pixels. The racetrack is randomly generated in each episode and because of the way the environment is set up, when the car takes an action, we don’t know in which



Actual game screen $96 \times 96$ pixels: input image

Figure 1: A red car navigates a racetrack and tries to stay on the grey-marked path, avoiding the green areas, off-course regions, and red-white turning blockers. The left image is an example of what a human would see while playing the game, while the right image is what the environment supplies to our model.

next state the car will end up. Additionally, the $96 \times 96$ input images are highly pixelated versions of the actual game screen, which may be helpful because it reduces the dimensionality of the input images, but is also a source of uncertainty because it blurs the actual game state, as shown in Figure 1. Using image input at all instead of position and velocity values as input is a source of uncertainty as well, because the model has to learn for itself where in the image the car is located, and which parts of the image represent the road, etc.

## 3 Related Work

As discussed in Decision making under uncertainty[1], Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. We decided to use a Q-learning approach as well to tackle the car navigation task.

$$Q(s, a) = r(s, a) + \gamma \max_{a} Q(s', a)$$

Figure 2: Q-learning

Because our problem takes images as input data, papers about Deep Reinforcement Learning techniques are very relevant. Specifically, "Deep Reinforcement Learning with Double Q-learning,“ a paper by a team of researchers and engineers in Google DeepMind, outlines a technique called Double Q-learning (DDQN), which builds on standard Deep Q-learning and improves on convergence time and accuracy by using one Q-Network to choose each action and another target Q-Network to evaluate these actions.

This approach has already been used to solve the Space Invaders game[2] (another OpenAI Gym environment), and the tutorial proved to be very useful as we applied DDQN to the CarRacing-v0 environment. The DDQN technique is further described in the Approach section.

We also consulted Deep Reinforcement Learning Approaches for Multi-Agent Dense Traffic Navigation[4] which employs a model-free, off-policy reinforcement learning process and does hyperparameter tuning on policies for the DeepTraffic challenge for Self Driving Cars. This paper provides learnings about prior and successful methods of reward discounting, exploration decay (being wary of greedy exploitation), and other methods to optimize stability and performance. After a lot of research into related work we had decided that the DDQN algorithm was the ideal approach for our task, and this this paper helped provide insights about hyperparameter tuning, especially how to handle exploration and exploitation. This was helpful in guiding model performance analysis and in deciding the different models that we experimented with.

## 4 Approach

We want to learn a policy that dictates the best action for the car to make at each state to successfully navigate through the two-dimensional track. In other words, after each state (a stack of $N$ frames), the car must choose in what direction and to what magnitude it wants to turn, speed up, and slow down for the next $N$ frames.

### 4.1 New state definition

The OpenAI Gym environment defines a state as a single frame in the game, but as a part of our approach, we stack $N$ consecutive frames together to represent a single state. Under this new definition of a state, taking an action $a = (s, g, b)$ simply means choosing the same steer, gas, and brake values for $N$ frames. Defining a state this way captures movement of the car through the world – it is difficult to see the effects of steer, gas, and brake values from one frame to the next, so we need to view states as series of several frames. Initially, we started with $N = 10$ frames in a state.

### 4.2 Action space discretization

Although the OpenAI Gym environment accepts any value in a continuous range for steer, gas, and brake, we discretized the action space so that we have a finite set of actions, which helped improve learning and simplify the learning process. Initially, we had the steer value choices of -0.25 (slight left), 0 (straight), or 0.25 (slight right); gas value choices of 0.1 (slight acceleration), 0.4 (medium acceleration), or 0.8 (high acceleration); and brake choices of 0 (no brake), 0.1 (very slight brake), or 0.5 (slight brake). This restricts the actions that the agent could make in the game, but also biases the car to move in a certain way; all three brake choices are low, and steering was limited to slight turns, which means the car generally moves quickly and does not make sharp turns.

### 4.3 Deep Reinforcement Learning with Double Q-learning

Because our states are series of 96 $\times$ 96 RGB images, the state space is simply too large to be able to learn the entire MDP, or to even use simple generalization techniques to estimate Q-values at unseen states. So we chose to explore Deep Reinforcement Learning techniques: specifically, we used Double Q-learning.

As described in the Google DeepMind paper[3], Double Q-learning is an augmented version of Deep Q-learning, which simply combines standard Q-learning with a deep neural network architecture. Double Q-learning works by training two models concurrently, and using one model to evaluate and update the other (the Q-value prediction is calculated as shown in Figure 3). This is done instead of choosing maximum Q-value actions in a single model as in Q-learning and using them to update weights, which usually leads to overestimation.

$$Q_{\max} = \begin{Bmatrix} r_{j+1} & \textit{if episode terminates at } j+1 \\ r_{j+1} + \gamma \max\limits_{a_{j+1}} Q_{target}\left(s_{j+1}, a_{j+1}; \theta^{-}\right) & \textit{otherwise} \end{Bmatrix}$$

Figure 3: DDQN Q-value calculation

Double Q-learning has been used to solve other OpenAI Gym gaming environments, but nothing quite as dynamic as CarRacing-v0, so we gave it a try. Our main and target models consist of 3 convolution and activation layer pairs, followed by flatten and dense layers. The output vector has as many values as there are actions, so a softmax gives us the index of the best action to take.

During training, a decaying $\epsilon$ value gives us our exploration probability. Initially, this $\epsilon$ is set to 0.1, and it eventually plateaus at 0.05. On any iteration, there is a $\epsilon$ probability that the agent will choose some random action to take, and a $1 - \epsilon$ that the agent will go with the model's predicted best action. This ensures that the agent gets plenty of opportunity to try new actions and explore new states that it would have never experienced if it only followed the model's predictions. This is useful for uncovering new, better strategies to drive the course.

Also, during training, every single $(s, a, r, s', d)$ experience tuple is added to a ReplayBuffer, a class that stores up to some maximum number of tuples and can later be sampled for learning. After we reach a minimum number of observed tuples, we begin to improve our model on every iteration by sampling a batch of past experience tuples from the buffer and using it to train model weights.

## 5 Evaluation Metric

The agent's goal is to have the car complete the full track (visit all the tiles in the track). The reward system is as follows. We penalize -0.1 points for every frame that the car took in its path to ensure that the car does not dally and tries to finish the track as soon as possible. A reward of 1000/N points is awarded for every track tile that the car visits. Lastly, a penalty of -100 points is assigned if the car goes too far off-course (outside of the area defined as the PLAYFIELD). This score (reward) is updated upon each frame. The game finishes after the car has visited all the track tiles and the game may stop prematurely if the car navigates too far off-course, in which case there is an immediate elimination and the episode is over.

Since the reward system depends heavily on the racetrack, which is randomly generated in each episode, and since a single action can throw the car off the track, reward is not the best evaluation metric for success. Thus, we model the problem by trying to maximize the Q-value, which is the expected discounted reward if we perform action a from state s then follow the optimal policy from then onwards[1].

## 6 Experiments/Results

We ran 4 trials with different hyperparameters and architectural variants. We trained each model for around 10,000 to 20,000 iterations, for a total training period of around 60 hours. We used Keras for our implementation and used an Adam optimizer with learning rate on plateau starting at 0.00001.

### 6.1 Small CNN: `basic`

Since the state space is so large, we experimented with different methods of reducing the state space. We tried to crop the 96x96 pixel grid to a 50x50 pixel grid around the car and also tried converting the RGB grid to a black and white grid. We found that none of these led to any benefit and thus decided to instead employ a Convolutional Neural Network (CNN) to process the input states (grid frames). The output of the CNN is then passed into the DDQN algorithm. The CNN architecture is explained in Table 1. The input into Convolution 1 is of shape 84x(84x3)x10 where the 3 accounts for the RGB color aspect of the state and the 10 handles the number of frames in each state. This model reaches a predicted Q-value of 400 in 21,000 iterations.

### 6.2 Large CNN: `new_cnn`

This model uses a different model architecture. Since our input data is on the magnitude of 84x(84x3)x10, we realized that the small CNN architecture was reducing our data significantly so we increased the size of the convolutions (using more filters) in this larger CNN architecture which is seen in Table 2. We still continue to use 10 frames for our state definition. This model reached a predicted Q-value of 400 in just 9,000 iterations and thus is learning much faster than the smaller CNN architecture which took 21,000 iterations to reach the same predicted Q-value.

### 6.3 Large CNN with five frames per state: `five_frames`

This model architecture is similar to `new_cnn`, except that instead of stacking 10 frames per state, we stack 5. The intuition here is that we wanted the agent to be making decisions more frequently. This model reached predicted Q-values of 400 only after a full 20,000 iterations.

Table 1: Small CNN Architecture

| Layer | Details |
|---|---|
| Convolution 1 | 32 filters of size 8x8 and stride of 4 |
| ReLU | – |
| Convolution 2 | 64 filters of size 4x4 and stride of 2 |
| ReLU | – |
| Convolution 3 | 64 filters of size 3x3 and stride of 1 |
| ReLU | – |
| Flatten | – |
| Dense | Size 512 |
| ReLU | – |
| Dense | Size corresponds to number of possible actions |

Table 2: Large CNN Architecture

| Layer | Details |
|---|---|
| Convolution 1 | 64 filters of size 8x8 and stride of 4 |
| ReLU | – |
| Convolution 2 | 128 filters of size 4x4 and stride of 2 |
| ReLU | – |
| Convolution 3 | 128 filters of size 3x3 and stride of 1 |
| ReLU | – |
| Flatten | – |
| Dense | Size 1024 |
| ReLU | – |
| Dense | Size corresponds to number of possible actions |

### 6.4 Large CNN with larger action space: `more_actions`

This model is the same as `new_cnn`, except with a larger action space. Instead of choosing three possible values for each of steer, gas, and brake, we choose from 5 evenly spread out values for each action parameter. This increases the action space from 27 to 125. This model reached predicted Q-values of 1000 after just a 18,000 iterations. This model thus seems to perform well in comparison to `five_frames` which only reached a predicted Q-value of 400 in around the same amount of iterations.

### 6.5 Results

The performance and predicted Q-values achieved by all the models is reflected in Table 3. The respective plots of Q-value vs time is seen in Figure 4.

### 6.6 Qualitative Review

As a qualitative review of the models' performances, we ran each model through 3 episodes of the car racing game, noting how well the the agent was able to make turns and stay on the track (which proved to be the two most difficult tasks). The images are seen in Figure 5. `new_cnn` was able to stay on track and make turns the best out of the four models.

## 7 Conclusion

The two models with the best predicted Q-values were `new_cnn` and `more_actions`, both reaching Q-value $\approx 400$ by iteration 9000. When we let `more_actions` run for longer, we saw that it achieved a predicted Q-value of 1000 after 18,000 iterations. These two were followed by `basic`. The `five_frames` model probably did not perform quite as well because each of the states (using only 5 instead of 10 frames) were too small; this game is very dynamic, so the model needs as much

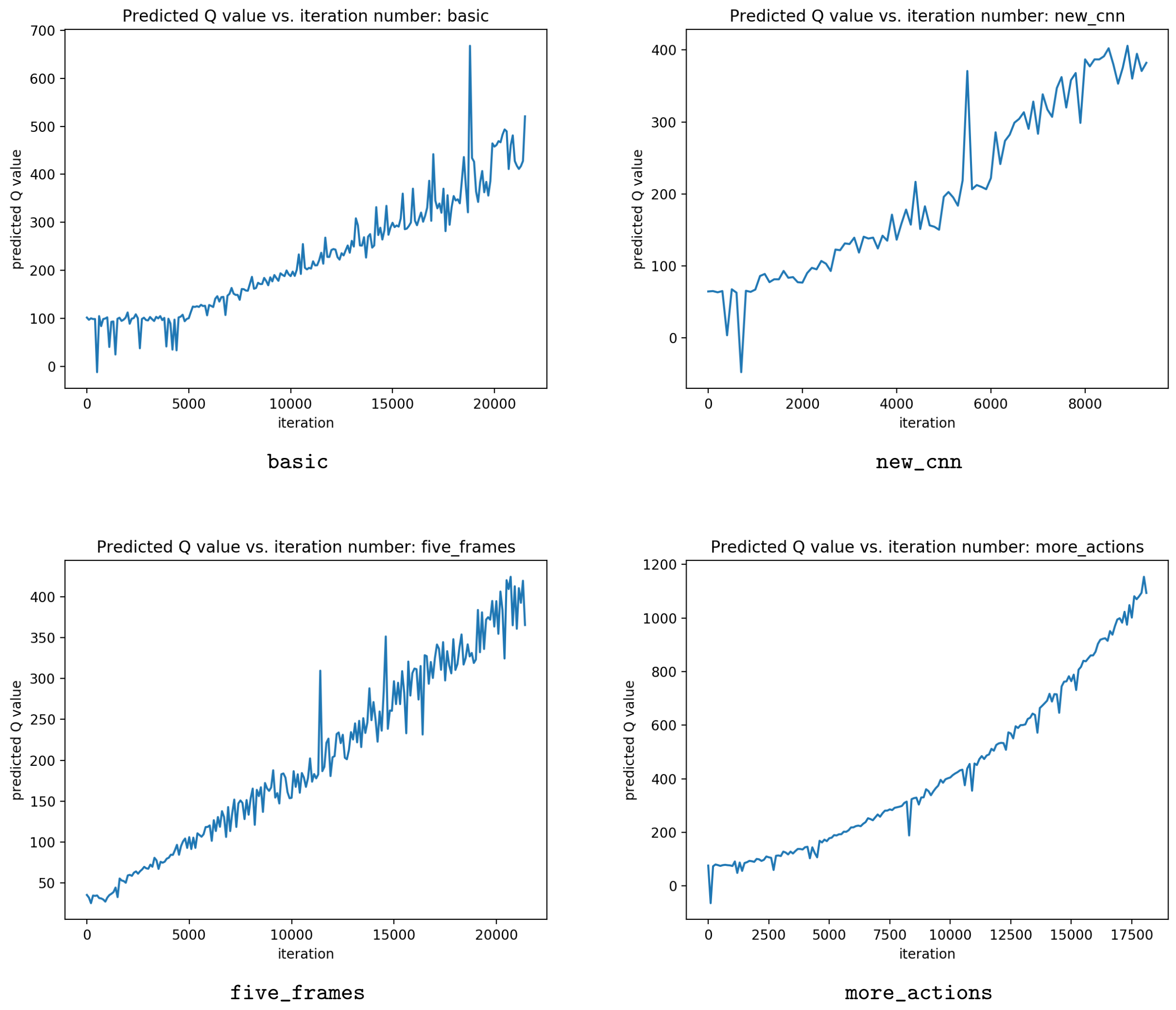


Figure 4: Training four different models

Table 3: Comparison of final predicted Q-values

| Model | Number of iterations | Approximate predicted Q-value |
|---|---|---|
| `basic` | 21000 | 400 |
| `new_cnn` | 9000 | 400 |
| `five_frames` | 21000 | 400 |
| `more_actions` | 18000 | 1000 |

context as possible to make sense of a state and the effect of an action. `new_cnn` was probably the best model from a qualitative standpoint, as the agent was able to make turns and stay on track for the most part. This is probably because the `new_cnn` model had larger intermediate tensors when compared to `basic`, so the model was not losing as much information from the huge input tensor between convolutions. Also, having a smaller action space helped the agent, because of the bias we put in our choices for actions. `new_cnn` had the car moving faster. On the other hand, `more_actions` generated a very slow, hesitant car which maintained rewards by never venturing out into the green, but did not traverse nearly as much of the track as did `new_cnn`. Although we were not able to train a model that could complete the entire racetrack, we were able to achieve the capability of a human who is still learning the game.

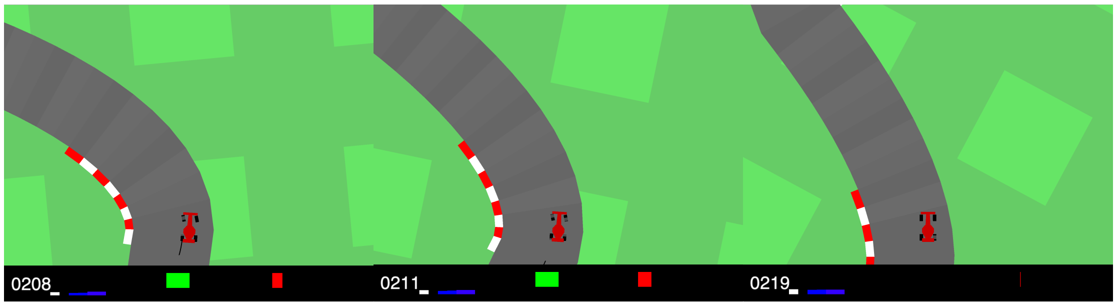


Figure 5: The car makes a turn (with model = `new_cnn`)

In the future, we may want to improve performance of our models by extending training; from the plots, we see that the predicted Q-values had not converged by the end of training even though we ran for a long time, due to time constraints. Thus we know that we can improve model performance with more computation resources. Additionally we can introduce more uncertainty by adding noise to actions in the environment, which would hopefully lead to a more robust model.

We may also look to other DeepRL methods to solve the same problem. Preferably, we would want something that trains faster and performs well in such dynamic environments as this. One possible approach is Evolution Strategies, which train a model function without using backpropagation, and could be very useful in quickly training an agent to race around the track.

### Acknowledgments

Thanks to Mykel Kochenderfer, Tomer Arnon, and the rest of the CS238 course staff for their support and guidance with this project.

### Contributions

Yash and Nidhi co-worked on implementing code as well as writing, tuning, and training the models. All paper-writing was done jointly. We spent 20 hours writing models and over 60 hours training models. This time does not include researching papers, writing other pieces of running code, and writing papers.

## References

[1] Kochenderfer, M. Decision making under uncertainty. MIT Lincoln Laboratory Series, 2015.

[2] Sagar, A. (2019) Deep Reinforcement Learning Tutorial with Open AI Gym. https://towardsdatascience.com/deep-reinforcement-learning-tutorial-with-open-ai-gym-c0de4471f368

[3] van Hasselt, H. & Guez, A. & Silver, D. (2015) Deep Reinforcement Learning with Double Q-learning. *Google DeepMind*.

[4] Fridman, L. DeepTraffic: Crowdsourced Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation. 32nd Conference on Neural Information Processing Systems (NIPS 2018).