Reinforcement Learning with Car Racing

Hanqin Cai **Josh Davis** **Ethan Hargrove**

Sebastian Jackson **Zhi Loh**

Department of Computer Science
University of Bath
Bath, BA2 7AY
`hc2281, jwrd20, eh2169, sj2259, zl127`
Soure Code, Video Presentation, Agent Performance

1 Problem Definition

1.1 Description

Our objective is to employ deep reinforcement learning (RL) techniques to solve the Box-2D Car Racing Environment in the OpenAI gymnasium package. The task involves driving a race car (the agent) on a randomly generated racetrack in order to visit all the track tiles as quickly as possible. An episode terminates once all track tiles are visited, or the car drives off of the playfield.

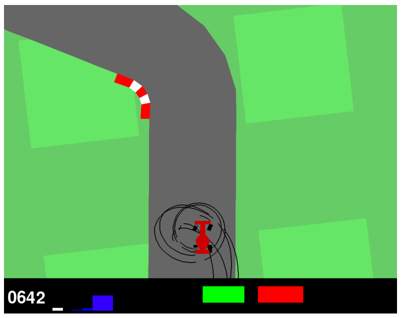


Figure 1: Snapshot of the "CarRacing" environment

1.2 State & Action Space

The environment "CarRacing-V2" is a benchmark for testing and comparing RL algorithms. The agent must learn to navigate a complex, continuous state space, where each state is represented by a set of $96 \times 96 \times 3$ RGB images. The total state space of our environment is $256^{(96 \times 96 \times 3)}$ (Brockman et al., 2016).

The environment allows both discrete and continuous representations of the action space. The continuous action space consists of three actions (s,g,b) where, s (in the range [-1,1]) stands for the steering angle, g (in the range [0,1]) for gas, and b (in the range [0,1]) for braking. The discrete action space instead has 5 actions: do nothing, steer left, steer right, gas, and brake. Both the action space and transition dynamics are deterministic.

1.3 Reward

The reward function is crucial to guide the agent's behaviour and in this environment, it was designed to encourage the agent to complete a loop of the racetrack as fast as possible. Completing one circuit indicates the end of an episode, however, if the car falls off the map it can preemptively terminate and give a reward of -100. Every track tile visited the agent receives a reward of +1000/N, where N is the number of tiles visited on the track, and is penalised by -0.1 for every frame. The documentation specifies the reward equation (Brockman et al., 2016):

$$R_t = 1000 - \frac{N}{10} \tag{1}$$

where N represents the number of tiles visited on the track. A time limit can be imposed on the car's off-track behaviour if it remains outside the racetrack for an extended period.

1.4 Suitability of environment

There are no limitations on the RL technique that the project can employ, as the environment allows for implementation in both discrete and continuous states. The rewards within the environment are frequent, with each track tile giving an immediate positive reward, while a time-based negative reward incentive the agent to complete the task faster, making it easier for the agent to discern which actions are beneficial. The race track is procedurally generated, making it challenging to model the environment, but the action space is deterministic. A deterministic action simplifies the problem and allows for the use of simpler RL algorithms.

2 Background

2.1 Value-Based Methods

The CarRacing-V2 environment is well suited for model-free RL methods because it has a stochastic environment and a discrete action space. One type of model-free method is the value-based method, which can scale well with large state and action spaces using function approximation techniques like neural networks. They are also known to converge quickly and are more sample efficient than policy-based methods (Rammohan et al., 2021).

However, there are some considerations:

- **Exploration-Exploitation trade-off:** Value-based methods choose the action with the highest expected reward based on the current estimate of the action-value. This leads to a trade-off between exploring unknown states in the environment and exploiting current knowledge.
- **Local Optima:** In large state-action spaces, the agent may get stuck in a local optima where it is difficult to discover better policies. In the CarRacing-V2 environment, the agent may learn to follow a suboptimal policy that completes a full circuit of the track without achieving the maximum possible returns. Value-based methods are prone to falling into local optima because they primarily focus on improving the current policy rather than exploring the state space for better policies.
- **Overfitting:** Value-based methods can 'overfit' to a specific policy, especially in large stochastic environments, and may not generalise to similar environments.

2.2 Policy Gradient Methods

Policy gradient methods are a subclass of model-free algorithms that optimise parametrised policies with respect to the expected long-term cumulative reward by performing gradient descent on its parameters. Unlike value-based methods, policy gradient methods can model both discrete and continuous actions for their respective spaces without requiring discretisation, making them more suitable for high-dimensional or continuous spaces.

However, policy gradient methods can suffer from higher variance in their gradient estimates due to their high degree of exploration in learning. This can result in slower convergence than value-based methods and make them less sample-efficient, especially in environments with dense rewards where the exploration aspect of stochastic policies may not be as necessary (Peters and Schaal, 2008). Therefore, in the CarRacing-V2 environment, careful tuning of exploration rates and learning rates is necessary to achieve better performance. Additionally, policy gradient methods can lead to sub-optimal policies with high-dimensional state spaces or long-horizon problems, as their optimisation can be more difficult due to the non-smoothness of the policy space. This is mitigated in the CarRacing-V2 environment.

In conclusion, policy-based methods directly learn a policy that maps states to actions without explicitly estimating the action-value. These methods are better suited to exploration in a large state space and can easily escape local optima since they encourage exploration. Early on in implementation we identified the proximal policy optimisation (PPO) algorithm as a suitable algorithm for the CarRacing-V2 environment.

2.3 Actor-Critic Methods

Actor-critic methods combine value-based and policy-based methods. These algorithms learn both a policy (actor) and a value function (critic) simultaneously. While value-based methods have a high sample learning efficiency and are effective in high-dimensional observation spaces they struggle in continuous action spaces (Rammohan et al., 2021). On the other hand, policy-based methods can learn stochastic policies in continuous, high-dimensional action space environments. Actor-critic methods can combine the advantages of value-based and policy-based methods to achieve sample efficient methods that can learn stochastic policies in continuous action spaces and high-dimensional observation spaces (Lillicrap et al., 2015). This can lead to more stable training and even better performance, especially if using the continuous action space of Car Racing.

2.4 Imitation Learning

Traditional reinforcement learning methods can face challenges due to low sample efficiency and the requirement for extensive exploration (Dong et al., 2020). This can make training computationally expensive, even for relatively simple tasks like 2D car-racing. In cases where an existing policy achieves satisfactory performance, reinforcement learning agents can leverage examples from that policy to learn more efficiently. This act of learning from expert demonstrations is called imitation learning.

Behavioural cloning is the bedrock that the more advanced imitation learning methods are built. It uses supervised learning with state-action pairs from expert demonstrations to train a neural network to mimic the behaviour of the given expert policy. This can be an effective way to achieve desired behaviour, however it can perform poorly in situations much different from the training set and it is unable to exceed the performance of the expert policy (Hussein et al., 2017). Applying behavioural cloning in the CarRacing-V2 environment can be a useful approach because we can easily capture the desired expert behaviour of staying on track. However, the agent may struggle to adapt to unique situations in the environment

3 Method

3.1 Pre-processing

Prior to applying our own training methods, it was necessary to simplify the environment before feeding it into our neural networks. Without this simplification step, the training algorithm would be overburdened by the input data, significantly hindering the agent's training speed. Our approach to streamlining information extraction from the image focused first on extracting the steering angle and speed from the screen then cropping out redundant pixels from the screen and then applying a contrast function that set the pixels to their closest high-contrast colour. Further steps involved the conversion of the RGB image to greyscale using the weighted method and the subsequent normalisation of pixel data, resulting in a massive reduction in the state space. The steering angle, speed, and the processed image were then fed into the neural network for training.

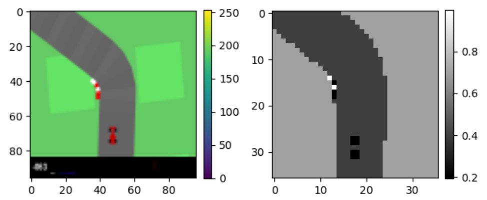


Figure 2: (Left) Un-processed image of the environment (Right) After pre-processing with decreased pixel size

3.2 DQN

Within the field of deep value-based RL methods, Deep Q-Networks (DQN) were the first deep RL method that used neural networks as function approximators to train agents in high-dimensional state spaces (Mnih et al., 2013). Although several more advanced techniques have surpassed DQN's performance since then, it remains a valuable benchmark for comparison between other RL and non-RL methods.

We adopted the vanilla DQN implementation from the paper "Playing Atari with Deep Reinforcement Learning" to the CarRacing-V2 environment. This consists of three components: a replay buffer, a target network, and a deep neural network.

Experience replay is a crucial component of DQN. It stores the agent's experience in a replay buffer and randomly samples in batches to train and update its Q-values. Without a replay buffer, the algorithm may calculate the loss only using one experience rather than a mini-batch of experiences. In the CarRacing-V2 environment this would result in a high variance in the update process, causing many steps where the car drives in the wrong direction. Although the average gradient over many steps would be correct, individual updates may be noisy. In addition, assuming the standard approach of collecting one-time step and updating the DQN neural network will use each sample of experience once before being discarded. The combination of these effects would likely make the learning process less sample efficient. Thus, experience replay reduces the correlation between consecutive updates and improves sample stability and efficiency Fedus et al. (2020).

This DQN implementation uses two deep neural networks: a main network and a target network. The main network updates each iteration using a backpropagation algorithm with a loss function that minimises the difference between the predicted Q-values and the target Q-values (Sutton and Barto, 2018).

In the CarRacing-V2 environment, the input to the neural network is a greyscale image with a depth of four frames. We pre-process each frame by stitching four frames together to get a meaningful representation of the car's movement. The neural network architecture consists of three hidden layers and an output layer Mnih et al. (2013). The output layer is a fully connected linear layer with a single output for each action in the game. So, the output of the network is a Q-value for each possible action that the agent can take, and the agent selects the action with the highest Q-value as its next move.

3.3 Pixel-Counting Control

In the continuous car-racing action space, there are three action parameters the agent must control: steering, gas, and brake. In order to develop our expert policy, we implemented a deterministic agent that determines its actions by counting the ratio of road pixels to non-road pixels along three lines of sight: left, right, and forward (see Figure 3).

The range of steering values is from -1 (full left turn) and +1 (full right turn). The pixel-counting agent determines its steering value by taking the ratio of road pixels to non-road pixels along the right line and subtracting the ratio of road pixels to non-road pixels along the left line. The gas value is set to half of the ratio of road pixels to non-road pixels along the forward line, times a multiplier that depends on the steering value to prevent understeering. The brake value is computed as the gas value multiplied by a handcrafted coefficient that depends on the gas and absolute steering value (see Appendix D). The coefficient is large when gas and steering have large magnitudes to cause the agent to slam on the brakes when taking a turn too fast. Conversely, the coefficient is small when gas and steering have large magnitudes to allow the agent to get back up to speed after the turn has been completed. The attributes serve as the foundation for our expert agent's behaviour.

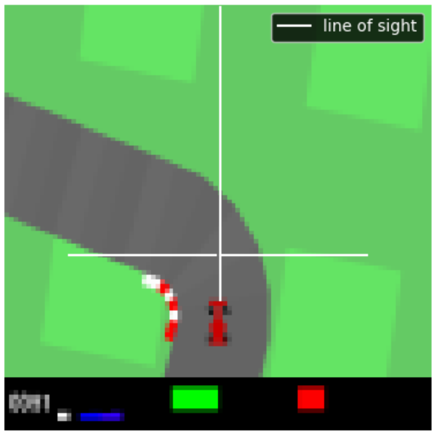


Figure 3: Lines of sight used by the pixel counting agent to determine actions.

3.4 Behaviour Cloning

The goal of behaviour cloning is to train a neural network that mimics the given expert policy. To do this we created a dataset of 25,024 state images and their corresponding expert-selected action and used the data to train a neural network. Since we are trying to learn a policy from only an image, we implemented a convolutional neural network (see Appendix B) to extract relevant features from the state image. After each convolutional and dense layer, we applied batch normalisation to achieve a more stable network with faster convergence. Following each batch normalisation, we used a dropout layer to randomly set inputs to zero with a probability of 50%, which helps to prevent overfitting. This network architecture was adapted from Yash Maniyar (2019), with our added batch normalisation and dropout layers. This network was trained on the expert dataset for 15 epochs with a batch size of 32 with an Adam optimiser with the TensorFlow default learning rate of 0.001 and a mean squared error loss function.

3.5 DDPG with Residual Policy Learning

Typically when reinforcement learning methods are applied to problems, the desired outcome is a policy that improves on existing performance. This can be done through residual policy learning, a reinforcement learning method that modifies an existing policy rather than learning a policy from zero knowledge Dong et al. (2020). This is typically done in actor-critic methods, such as Deep Deterministic Policy Gradient (DDPG), where there is a fixed pre-trained policy and a residual policy that is trained to output the difference between the output action for the fixed policy and the optimal action for a given state.

$$\pi^*(s) = \pi_{expert}(s) + \pi_{residual}(s)$$

This DDPG with residual policy learning preserves the expert policy ($\pi_{expert}(s)$) and adds small adjustments through a separate network ($\pi_{residual}(s)$) to improve performance. Using this approach, we can leverage the knowledge from the expert policy whilst allowing the agent to explore and learn from experience. During training we add a gaussian noise parameter to encourage exploration. Without noise, the agent may not find an optimal policy or fail to explore certain parts of the state space. This suggests in the CarRacing-V2 environment the noise parameter should have a small standard deviation to avoid the agent taking actions that are not useful for completing the task (see Appendix C).

4 Results

We started off with a basic DQN to train the agent, however, this was taking a long time to evaluate the model in the CarRacing-V2 environment. To confirm our suspicion, we adapted code that implemented the vanilla DQN model that was described in the DQN section. Figure 4shows the DQN agent trained for ten hours over 175 000 time steps that achieved an average return of 24. Given time and resource constraints we pivoted our approach towards a combination of reinforcement learning and supervised learning.

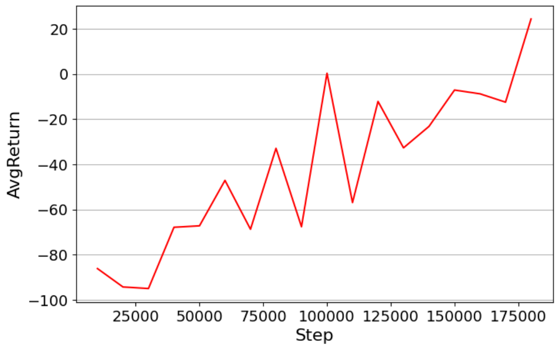


Figure 4: Average returns from the vanilla DQN agent.

We observed the reward achieved by each algorithm over 10 episodes, we then computed the mean, standard deviation, median, minimum, and maximum of those rewards (see table 1). Those results were then compared to actions selected uniformly at random and human performance, with each of our five members playing 10 episodes.

Method	Mean	Std.	Median	Min	Max
Random Action	-698.50	220.35	-863.81	-901.13	-394.88
Human	822.53	51.78	834.15	250.35	910.01
Pixel Counting	891.06	4.42	892.17	884.39	896.91
BC	760.23	371.48	883.15	-354.01	895.80
DDPG	882.94	10.32	882.36	863.10	900.06

Table 1: The mean, standard deviation, median, minimum and maximum values for each method.

5 Discussion

In one paper, the car-racing environment was said to be solved if after 100 consecutive trials, you obtain an average reward of 900 (Ha and Schmidhuber, 2018). The pixel-counting control algorithm is a robust agent that has been hand-tuned to always stays on the road, slow down to safely take sharp turns, and speed up on straightaways. To achieve this, it turns when the ratio of road pixels to non-road pixels is larger on one side of the car. As a consequence of this, the agent will wobble around corners and whenever it is not perfectly centered in the track, producing slightly sub-optimal behaviour. This could be improved by reducing the steering magnitude when the ratio road to non-road pixels is large on both sides of the car.

The DQN agent was trained based on the procedure in Mnih et al. (2013), which was expected to achieve a good performance. However, there were problems during the implementation. Although hyperparameters have been chosen for better rewards (see Appendix B) and the epsilon decays from 0.1 to 0.001 within 10000 steps, the rewards fluctuated between negative values and positive values. Since the neural network functions as a "black-box", it is difficult to find out the underlying reasons for the bad performance.

The behaviour cloning agent was trained off of expert examples given by the pixel-counting algorithm, so its behaviour is very similar when navigating through ideal scenarios. However, it experienced difficulties when facing scenarios not present in the training dataset. This can be seen when one of the car's wheels touches the grass, when this happens the agent halts to a stop and never continues driving. To improve the behaviour cloning performance, a diverse dataset including expert actions in difficult and sub-optimal positions would be needed.

The DDPG with residual learning agent aims to learn the difference between the behaviour cloning agent and the optimal policy. Since its output actions are the sum of the cloning network's outputs and the residual network's outputs, in most scenarios it behaves quite similarly to the behaviour cloning agent. However, there are a few noticeable ways in which it differs. The agent performs very safely to avoid scenarios where the "expert" policy completely stops, such as when a wheel leaves the track. To avoid this, it slows down and turns early when approaching the edge of the track. If the agent learned from a more robust expert policy it would be able to behave more dangerously, potentially allowing the car to drive faster and take sharper turns by allowing part of the car to leave the track. Behaviour similar to that of actual racing drivers was also observed. That is following the racing line - the optimum route to minimise time spent in the corner and maintain maximum speed. It does this by swinging out wide before the corner and then swinging sharply across the apex of the corner. Due to these improved behaviours, DDPG with residual learning outperforms its given expert policy.

6 Future Work

The main option for future work is to explore different RL algorithms beyond the ones we have implemented. Whilst the methods we have already implemented have performed well, there is room for improvement using more complicated or better-suited algorithms. For instance, we could implement the Proximal Policy Optimization (PPO), which is better suited to continuous action spaces and more sample-efficient in complex environments such as car racing (Schulman et al., 2017).

Another possibility is to improve our current implementations. For example, we could fine-tune the hyperparameters such as the exploration rate or the size of the replay buffer to optimize the model's performance. However, research suggests that increasing the replay buffer capacity may not result in significant improvements in DQN's performance (Fedus et al., 2020).

In addition to exploring different algorithms and improving our current implementations, we could also investigate two frontiers in RL. The first of these is intrinsic motivation, which allows agents to explore due to an intrinsic reward/motivation rather than external factors. However, this approach may not be the best option for our CarRacing-V2 environment since it is not sparse in terms of rewards. Another option is hierarchical reinforcement learning, which focuses on breaking down tasks into sub-goals that the agent can learn. In a car racing setting, these could be high-level options such as "accelerate for 3 seconds" or "turn left 15 degrees". By focusing on learning skills more effectively and knowing when to apply them, hierarchical reinforcement learning can lead to faster learning and better overall performance. However, the challenge would be to choose appropriate options to maximize the reward when racing around the track. In conclusion, exploring different RL algorithms, improving current implementations, and investigating frontiers in RL such as intrinsic motivation and hierarchical reinforcement learning could be promising avenues for future work in our project.

7 Personal Experience

During the project, we encountered various challenges that required us to refine and adapt our algorithms. Initially, we decided to apply PPO to the CarRacing-V2 environment based on its reputation as a leading approach (Wang, He and Tan, 2020). However, we experienced difficulties during agent training, with the agent failing to learn effectively. To implement PPO in the CarRacing-V2 environment would require a solid understanding of the environment combined with the required computational resources. Consequently, we switched to DDPG with residual policy learning.

Interestingly, we found that behavioural cloning yielded impressive results, despite some issues with the agent occasionally driving into the grass. We speculate that with more training time, both DQN and DDPG could potentially achieve similar levels of performance, as we observed a positive correlation between the rewards achieved by these algorithms and the number of training steps taken.

Overall, our experience with the project was both challenging and rewarding. We learned a great deal about the strengths and limitations of different reinforcement learning algorithms, as well as the importance of carefully managing resources and iterating on our approach to achieve the best results.

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W., 2016. Openai gym. *Corr* [Online], abs/1606.01540. `1606.01540`, Available from: `http://arxiv.org/abs/1606.01540`.

Dong, H., Dong, H., Ding, Z., Zhang, S. and Chang, 2020. *Deep reinforcement learning*. Springer.

Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M. and Dabney, W., 2020. Revisiting fundamentals of experience replay. *International conference on machine learning*. PMLR, pp.3061–3071.

Ha, D. and Schmidhuber, J., 2018. Recurrent world models facilitate policy evolution. `arXiv:1809.01999`.

Hussein, A., Gaber, M.M., Elyan, E. and Jayne, C., 2017. Imitation learning: A survey of learning methods. *Acm computing surveys (csur)*, 50(2), pp.1–35.

Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. *arxiv preprint arxiv:1509.02971*.

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

OpenAI, G., 2023. Gymnasium documentation. Available from: `https://gymnasium.farama.org/environments/box2d/car_racing/`.

Peters, J. and Schaal, S., 2008. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4), pp.682–697.

Rammohan, S., Yu, S., He, B., Hsiung, E., Rosen, E., Tellex, S. and Konidaris, G., 2021. Value-based reinforcement learning for continuous control robotic manipulation in multi-task sparse reward settings. *arxiv preprint arxiv:2107.13356*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. *arxiv preprint arxiv:1707.06347*.

Sutton, R.S. and Barto, A.G., 2018. *Reinforcement learning: An introduction*. MIT press.

Wang, Y., He, H. and Tan, X., 2020. Truly proximal policy optimization. In: R.P. Adams and V. Gogate, eds. *Proceedings of the 35th uncertainty in artificial intelligence conference* [Online]. PMLR, *Proceedings of machine learning research*, vol. 115, pp.113–122. Available from: `https://proceedings.mlr.press/v115/wang20b.html`.

Yash Maniyar, N.M., 2019. Racetrack navigation on openaigym with deep reinforcement learning [Online]. Available from: `https://web.stanford.edu/class/aa228/reports/2019/final9.pdf`.

Appendix

Appendix A: Detailed Problem Domain Description

Most of these details have been obtained from (OpenAI, 2023)

States

The states are a pre-processed version of the observation which is a top-down 96x96 RGB image of the car and race track. The pre-processing reduces the pixels and grey scales the image to improve efficiency when learning. In terms of the DQN agent using PyTorch, the image size is reduced to 42x42 with 4 stack frames.

Actions

If continuous there are 3 actions : 0: steering, -1 is full left, +1 is full right 1: gas 2: breaking If discrete there are 5 actions: 0: do nothing 1: steer left 2: steer right 3: gas 4: brake

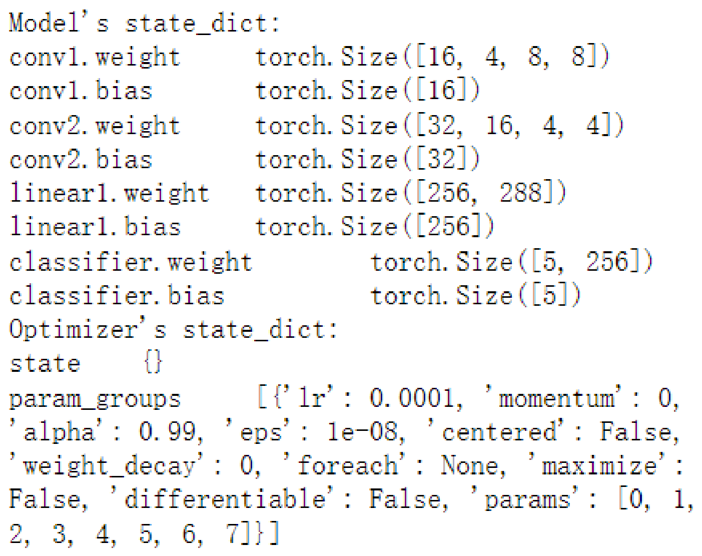
Rewards

The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points.

Transition Dynamics

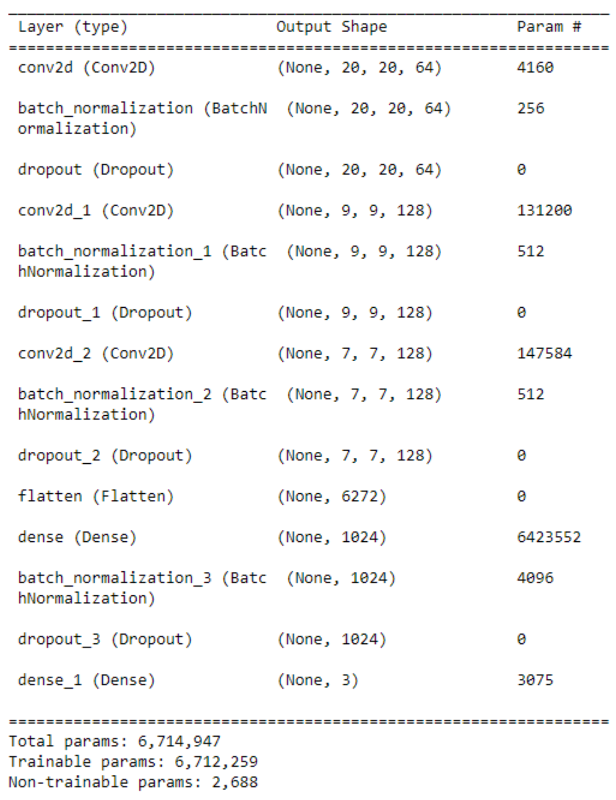
The transition dynamics are determined by the simulated physics of Box2D. The car's position, velocity, and orientation are updated based on the agent's actions and the environment's physics. The environment is deterministic, meaning that given the same state and action, the environment will always produce the same next state. However, there is stochasticity in the environment due to small variations in the physics simulation and the actions of other cars.

Appendix B: Convolutional Neural Network Architecture



```
Model's state_dict:
conv1.weight     torch.Size([16, 4, 8, 8])
conv1.bias       torch.Size([16])
conv2.weight     torch.Size([32, 16, 4, 4])
conv2.bias       torch.Size([32])
linear1.weight   torch.Size([256, 288])
linear1.bias     torch.Size([256])
classifier.weight        torch.Size([5, 256])
classifier.bias          torch.Size([5])
Optimizer's state_dict:
state    {}
param_groups     [{'lr': 0.0001, 'momentum': 0,
'alpha': 0.99, 'eps': 1e-08, 'centered': False,
'weight_decay': 0, 'foreach': None, 'maximize':
False, 'differentiable': False, 'params': [0, 1,
2, 3, 4, 5, 6, 7]}]
```

Figure 5: A view of state dictionary of the convolutional neural network model and the optimizer used in DQN.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 20, 20, 64)        4160

 batch_normalization (BatchN  (None, 20, 20, 64)       256
 ormalization)

 dropout (Dropout)           (None, 20, 20, 64)        0

 conv2d_1 (Conv2D)           (None, 9, 9, 128)         131200

 batch_normalization_1 (Batc  (None, 9, 9, 128)        512
 hNormalization)

 dropout_1 (Dropout)         (None, 9, 9, 128)         0

 conv2d_2 (Conv2D)           (None, 7, 7, 128)         147584

 batch_normalization_2 (Batc  (None, 7, 7, 128)        512
 hNormalization)

 dropout_2 (Dropout)         (None, 7, 7, 128)         0

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 1024)              6423552

 batch_normalization_3 (Batc  (None, 1024)             4096
 hNormalization)

 dropout_3 (Dropout)         (None, 1024)              0

 dense_1 (Dense)             (None, 3)                 3075

=================================================================
Total params: 6,714,947
Trainable params: 6,712,259
Non-trainable params: 2,688
```

Figure 6: A view of parameters and output shapes in the convolutional neural network used in behaviour cloning and DDPG with residual policy learning.

Appendix C: DDPG with Residual Learning Hyperparameters

Critic LR	0.0006
Actor LR	0.0005
Gaussian Noise Mean	0
Gaussian Noise Std.	0.005
Episodes	200
Discount Factor	0.975
Buffer Size	20000
Mini-batch Size	32
Steps Between Updates	5

Table 2: Hyperparameters used for traing the DDPG with Residual Learning Agent.

Appendix D: Pixel-Counting Control Algorithm

```python
# To get steering, count road pixels on the left and right of the car.
left_line = processed_img[56:57,:34][0]
right_line = processed_img[56:57,38:][0]
left_count, right_count = 0, 0

for left_pixel, right_pixel in zip(left_line, right_line):
    if left_pixel[1] == 101:
        left_count += 1
    if right_pixel[1] == 101:
        right_count += 1

steering = right_count / len(right_line) - left_count / len(left_line)
```

Figure 7: Using left and right road to non-road pixel ratios to get steering value.

```python
# To get gas, count road pixels in front of the car.
middle_line = processed_img.shape[1] / 2
front_line = processed_img[:67,middle_line:middle_line+1]
front_count = 0
for pixel in front_line:
    if pixel[0, 1] == 101:
        front_count += 1
front_count_norm = front_count / len(front_line)
if steering > 0.75:
    gas = front_count_norm * 0.05
elif steering > 0.5:
    gas = front_count_norm * 0.1
elif steering > 0.5:
    gas = front_count_norm * 0.25
else:
    gas = front_count / len(front_line)
```

Figure 8: Using the steering value and the forward road to non-road pixel ratio to get the gas value.

```python
# Use gas and steering to get brake
if (gas > 0.75) and (abs(steering) > 0.75):
    brake = gas * 0.975
elif (gas > 0.75) and (abs(steering) > 0.5):
    brake = gas * 0.85
elif (gas > 0.75) and (abs(steering) > 0.25):
    brake = gas * 0.55
elif (gas > 0.5) and (abs(steering) > 0.75):
    brake = gas * 0.975
elif (gas > 0.5) and (abs(steering) > 0.5):
    brake = gas * 0.75
elif (gas > 0.5) and (abs(steering) > 0.25):
    brake = gas * 0.45
elif (gas > 0.25) and (abs(steering) > 0.75):
    brake = gas * 0.975
elif (gas > 0.25) and (abs(steering) > 0.5):
    brake = gas * 0.65
elif (gas > 0.25) and (abs(steering) > 0.25):
    brake = gas * 0.35
else:
    brake = 0.05

return [steering, 0.5 * gas, brake]
```

Figure 9: Obtaining the brake value using steering and gas values. Gas value is then halved and the action is returned.