Deep Q-Learning for Autonomous Racing: Implementation and Analysis in CarRacing-v3

Xian Ma

December 9, 2024

Abstract

This report presents the implemention and analysis of a reinforcement learning agent using Deep Q Learning for the CarRacing-v3 environment in OpenAI's Gymnasium. It explores the use of CNNs (convolutional neural networks) in processing visual input for autonomous racing. CNNs will be used to process visual input for autonomous racing. The agent will incorporate multiple DQN optimization techniques.

The environment's state space is represented by a 96x96 RGB image. In order for the CNN to properly process the image, it was simplified by taking the mean of the third axis of RGB values. This method was imperfect, it was functional but not optimal. The training approach utilized a epsilon-greedy exploration strategy with a high initial epsilon value and a low minimum epsilon value. The learning rate adaptively changed during training as progress stalled. This approach allowed for the trained model to significantly improve over only 2000 episodes. It was able to obtain a reward of 900+ despite its -54 initial reward.

1 Introduction

1.1 Motivation

Unlike traditional programming approaches to problem solving, reinforcement learning allows for an agent to learn through environmental interaction and feedback. Reinforcement learning agents can develop sophisticated policies through trial and error, making them particularly well-suited for complex tasks where optimal solutions are difficult to program directly.

Autonomous driving is one particular domain that exemplifies the aforementioned challenges. It requires real-time visual processing, decision making, and precise control. While real-world autonomous driving development demands extensive capital investment, simulation environments offer an accessible alternative.

This project leverages CarRacing-v3, a top-down racing environment as a means to explore reinforcement learning in the contex of autonomous driving. The environment provides a simplified representation of the challenges of autonomous driving through incorporation of elements such as visual processing, input control, and the requirement of learning purely through feedback. The constrained environment allows for a focus on the core mechanics of reinforcement learning without additional complexities that the real-world would introduce.

1.2 Objectives

The objective of CarRacing-v3's environment is to have the race car navigate around the randomly generated track and visit all the tiles on the track. In the case where the agent navigates far outside of the track, the episode ends and the agent is penalized with a reward of -100 and dies.

While the environment's success criteria is defined above, the project establishes additional objectives to evaluate the performance of the agent. These include some of those outlined in the project proposal as well as additional objectives to better evaluate the performance of the agent.

- Achieve a reward of 700 over 100 consecutive episodes.
- Achieve an average reward of 800+ over 100 episodes.
- Show improvement in reward gain overtime during training.
- Demonstrate at least a 50% increase in reward compared to the baseline agent.

1.3 Dataset

Unlike traditional machine learning projects that rely on datasets, this project utilized CarRacing-v3, a simulation environment from OpenAI's

Gymnasium. The environment procedurally generates a new racing track for each episode. The ”dataset” is the environment itself.

The environment provides an observation space of top-down 96x96x3 RGB image of the car and race track. The action space is configurable between a continuous or discrete action space.

While a continuous and discrete action space are possible, a 5 action discrete action space was chosen. The initial experiments that utilized a continuous action space yielded poor results. These results were prevalent despite optimization attempts. Therefore, this project focused on a simplified 5 action discrete action space. The actions are as follows:

- 0: do nothing
- 1: steer left
- 2: steer right
- 3: gas
- 4: brake

The environment provides a reward function that penalizes the agent -0.1 for every frame and rewards +1000/N for completing the track where N is the number of tiles on the track. There are implicit penalties for driving far off the track. The agent will be penalized -100 and die.

2 Methodology

2.1 Network Architecture

The agent employs a Convolutional Neural Network (CNN) to process the 96x96 visual input from the environment. The CNN architecture consists of three layers followed by two fully connected layers. The CNN architecture is shown below:

Layer	**Filters**	**Kernel Size**	**Stride**
Conv1	32	8×8	4
Conv2	64	4×4	2
Conv3	64	3×3	1

Table 1: Configuration of convolutional layers

The flattened output from the convolutional layers feeds into two fully connected layers:

- **Hidden Layer:** 512 neurons (Processes everything)

- **Output Layer:** 5 neurons (One for each possible action)

I used a Rectified Linear Unit (ReLU) activation function after each layer to introduce nonlinearity. This aids in the network's ability to learn non-linear patterns.

The convolutional layers are used to capture the visual information of the environment. Examples of this include where the edges of the road are and where the agent currently is on the track. The fully connected layers are used to process the information captured and output an appropriate action to take.

In order to simplify the process, I reduced the input dimensionality by taking the mean of the third axis of the RGB values. This may not have been the optimal solution, but it was a functional one that still allowed the agent to retain essential visual information.

2.1.1 Justification For Layer Configurations

In the first layer, I used a 32 filter with a kernel size of 8x8 and a stride of 4. By utilizing a kernel of 8x8 and a 32 filter, the model will be able to capture large features of the environment like the track shape. The stride of 4 allowed the model to reduce the size of the image.

The second layer used a 64 filter with a kernel size of 4x4 and a stride of 2. This layer was used to capture more detailed features of the environment. By doubling the filter size and halving the stride, the model should be able to capture smaller features.

The third layer used a 64 filter with a kernel size of 3x3 and a stride of 1. This layer was used to capture the finest details of the environment. By keeping the filter size the same and reducing the stride, the model should be able to capture the finest details of the environment.

The hidden layer was set to 512 neurons. This was chosen because it was most likely large enough to process all the features from the previous layers. It is also a common choice for hidden layers where the task is moderately complex.

The output layer was set to 5 neurons. This is because there are 5 possible actions the agent can take in the discrete action space. It is a direct mapping between the neurons and actions.

As layers progress, each subsequent layer captures more detailed features of the environment. It "zooms in" on the environment progressively. Moreover, it adds a ReLU to aid it in capturing more complex patterns.

2.2 Training Framework

2.2.1 DQN Implementation

To address the problem of CarRacing-v3, a Deep-Q Network (DQN) algorithm was implemented. At its core, it leverages two neural networks: a policy network that chooses actions and a target network that helps stabilize the training process. The dual network structure was critical to the agent's success.

Experience Replay Buffer The experience replay buffer was a critical component. The model employed a buffer of 100000 experiences. Each experience is represented as a tuple (state, action, reward, next state, done). Batches of 32 experiences are then sampled during training. This helps reduce correlation between training samples dueo to the fact that it avoids consecutive samples that are similar.

Training Process The training process of the agent contained several key components. The gamma value was set to 0.99. This value was chosen because it is a common value for gamma in reinforcement learning. It allowed the agent to consider future rewards when making decisions. The loss function used was L1 loss. This loss function was chosen since it prevents exploding gradients that occur from large Q-value differences. Gradient clipping was also used to prevent the exploding gradients problem. The target network was updated every 10 episodes to prevent the Q-values from spiraling out of control.

Adaptive Moment Estimation (Adam) Optimizer The agent employed an Adaptive Moment Estimation (Adam) optimizer to dynamically adjust the learning rate based on its own training performance. In the scenario that the agent began to plateau, the learning rate was adjusted. The learning began at 0.0001 and decreased to 6.25e-06 based on performance plateaus. The range of the learning rate was 1e-04 to 1e-06. 1e-06 was never reached during the 2000 episodes.

Hardware Configuration The training leveraged GPU acceleration with ROCm (AMD alternative to CUDA). This significantly sped up the training process compared to a CPU only implementation. This optimization was critical in allowing for 2000 episodes to be completed in under 24 hours. The GPU used was a 7900XTX. The total training time was 16.6 hours.

Epsilon-Greedy Exploration The agent used an epsilon-greedy exploration strategy. The epsilon value began at 1.0 with a decay factor of 0.995. The epsilon value decayed over time with a minimum value of 0.01.

2.2.2 Action Space Mapping

Initial experimentation with a continuous action space yielded extremely poor results despite optimization attempts. This led to the adoption of a discrete action space. Each discrete action is mapped to a 3-dimensional continuous action vector in `dqn_agent.py`. The mapping is as follows:

- Do nothing: [0.0, 0.0, 0.0]
- Steer left: [-1.0, 0.0, 0.0]
- Steer right: [1.0, 0.0, 0.0]
- Gas: [0.0, 1.0, 0.0]
- Brake: [0.0, 0.0, 0.8]

While a discrete action space sacrifices the precision of continuous action spaces, it was chosen because it simplifies the learning process for the agent. The results demonstrated that a five action discrete action space was sufficient for the agent to learn to navigate the track.

2.2.3 Hyperparameter Scheduling

The agent's performance was monitored and metrics were tracked in order to schedule the hyperparameters. The best reward achieved was tracked and used as a reference point. The best reward was initially set to $-\infty$. The scheduling gave a 200 episode patience period during which the hyperparameters were not adjusted. If a better reward was achieved, the hyperparameters were updated and the patience period was reset. If a better reward was not achieved within the 200 episode patience period, the learning rate was halved.

3 Experiments

3.1 Training Setup

The training loop for the agent in CarRacing-v3 consisted of 2000 episodes. The training employed two environments. One environment was used for training and set with the seed 42. The other one acted as the validation set and set with the seed 420. Despite each episode utilizing a randomly generated track, utilizing two environments ensured that the agent was not overfitting patterns that stemmed from a seed's generated tracks.

The validation environment was used to evaluate the performance of the agent every 100 episodes to monitor the agent's progress. The validation process consisted of running 5 episodes and averaging the reward. If the averaged reward was greater than previous averaged value, then the model was dumped and saved.

Hardware Configuration The training leveraged GPU acceleration with ROCm on WSL. The environment was created with Mamba package manager. The GPU used was a 7900XTX. The total training time was 16.6 hours over 2000 episodes. This includes the 5 validation episodes that were run every 100 episodes.

Agent Parameters The hyperparameters were scheduled to change over time. The learning rate started with 1e-04 and decreased to 6.25e-06. The epsilon value started at 1.0 and decayed to 0.01 with a decay factor of 0.995. A 100k experience replay buffer was used with a batch size of 32.

3.2 Performance Metrics

3.2.1 Training Rewards Progression

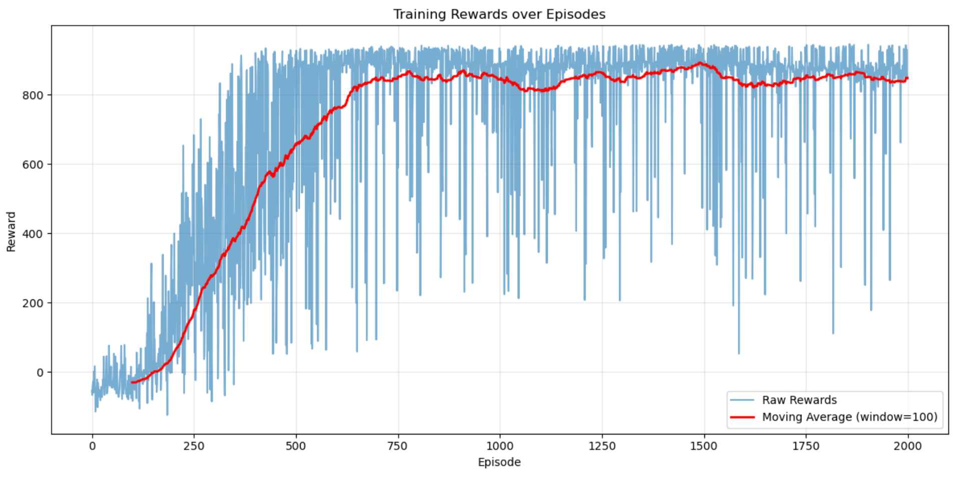

Figure 1: Training Rewards Over Time

The training rewards progression is shown in Figure 1 above. The training rewards diagram showed three distinct phases of improvement.

The moving average of the training rewards steadily increased over time from episode 0 where it started in the negatives to episode ∼150 where it breached 0. This phase was marked by steady improvement.

The second phase was marked by a sharp exponential growth from episode ∼150 to episode ∼700. The moving average rose from breaching positive values to reaching above 800.

The third phase was marked by slow and steady improvement from episode ∼700 to episode 2000. The moving average rose from 800 to 850 where it seemed to plateau. The agent's plateau can be attributed to the fact that it learned a policy that consistently achieved a higher score, leading it to adopt a strategy where it would stop driving altogether to maintain its current score. It believed that the score loss was not worth the risk of driving. This was evident when episodes were manually reviewed.

In terms of the values rather than the moving average, intial spike from episode 0 started from a training reward of -54. The spiked reached a maximum of 946.30 at episode ∼1857. The dips in the training rewards after hitting the peak were much more severe than those seen before the peak. However, the dips never fell below a score of 0. This can be attributed to the hyperparameter scheduling. The learning rate was halved when the agent plateaued. After 2000 episodes, the training reward settled to 845.85. The average reward across the entire training process was 704.44. This shows that the agent was able to learn to navigate the track and achieve a score that met the critieria set in above sections despite the initial -54 reward.

3.2.2 Validation Performance

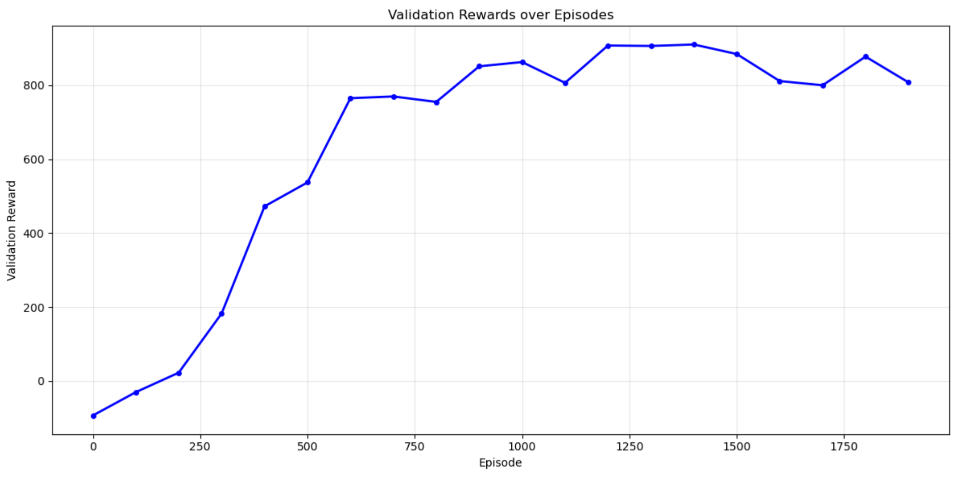

Figure 2: Validation Rewards Over Time

The validation rewards progression that was measured every 100 episodes is shown in Figure 2 above. The validation rewards showed three distinct phases of improvement.

The first phase was marked by a sharp increase from episode 300 to episode 700. When tested in the validation environment, the trained agent showed significant improvement. It went from -54 to 769.73. The giant leap in performance was most likely due to the agent learning the track and the environment.

The second phase was marked by a steady increase over time. From episode 700 onwards, the validation rewards steadily increased and maintained a reward over 800. This phase was marked by the agent steadily improving its policies.

The third phase was marked by a hitting a peak at episode 1300 with a reward of 910.23 and plateauing before steadily decreasing until it hit a reward of 807.82. This phase was marked by the agent's policy plateauing and the agent's policy no longer improving. The average validation reward was 640.31.

3.2.3 Dummy Agent vs Trained Agent

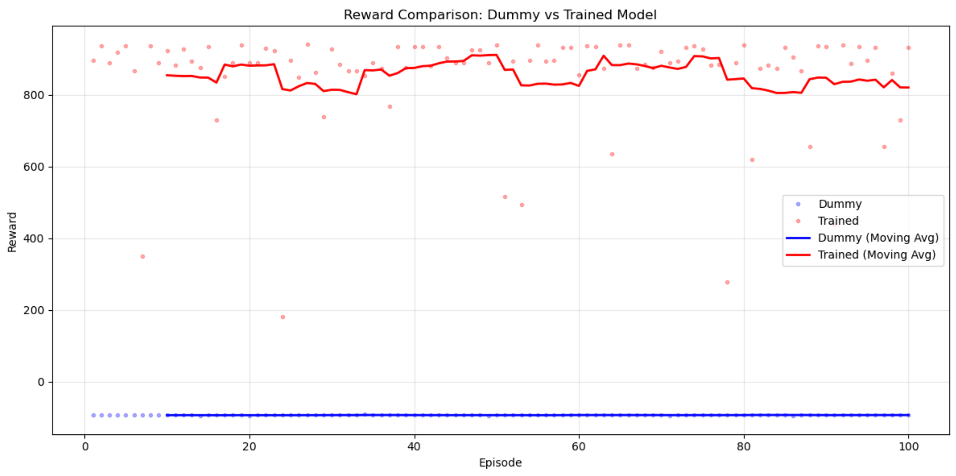

Figure 3: Dummy Agent vs Trained Agent

The dummy agent's performance is shown in Figure 3 above. The figure charts the moving average of the rewards gained by both the dummy agent and the trained agent over 100 episodes.

The dummy agent utilized Python's random library to randomly select a number between 0 and 4 inclusive, the number was then used to select one of the 5 actions. The dummy's performance was used as a baseline to compare the trained agent's performance. This aided in ensuring that the trained agent had learned a policy that was better than random / a baseline one.

The dummy agent's performance over 100 episodes averaged -93.13 with a standard deviation of +- 0.49. It achieved a minimum reward of -93.30 and a maximum reward of -91.30.

The trained agent's performance over 100 episodes were significantly better than the dummy agent's performance with the condition of a higher standard deviation. The trained agent's performance averaged a reward of 855.74 with a standard deviation of +- 141.07. It achieved a minimum reward of 181.90 and a maximum reward of 940.50.

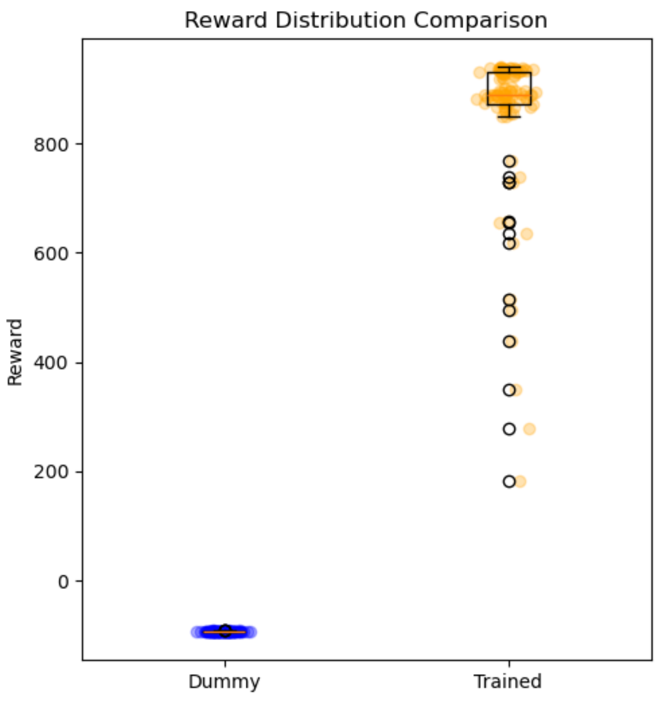

Figure 4: Reward Distribution

Closer analysis of the figure above 4 shows the dummy agent's rewards were tightly clustered around the mean of -93.13. There was very little variance in the dummy agent's rewards. In contrast, the trained agent's rewards were spread out over a much larger range but in a positive direction. Majority of the trained agent's rewards were above 800. This distribution further supports the evidence that the trained agent had developed a policy that was more effective than random behavior.

Based on the gathered information, it can be concluded that the trained agent's performance was better than the dummy agent's performance. The evidence showed that the trained agent has learned a policy that was better than random / a baseline.

3.2.4 Completion Rate

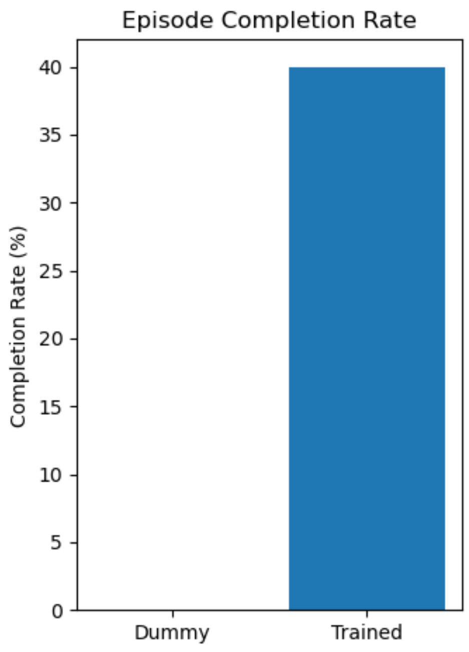

Figure 5: Completion Rate

The completion rate of both the dummy and trained agent is shown in Figure 5 above. The completion rate is the percentage of episodes that the agent was able to complete the 1000 tile track across 100 episodes.

The dummy agent was unable to complete the track even once as it was randomly selecting actions. It can be seen that the completion rate was 0% across all episodes.

The trained agent was able to complete the track 40% of the time. This was a significant improvement over the dummy agent's performance. The agent may have been able to complete the track more often if it did not adopt a strategy where it would stop driving altogether to maintain its current score. Due to the adopted strategy, the trained agent was only able to average 683 steps per episode. Since the agent was able to complete the track 40% of the time, it can be asserted that the trained agent was better than a random agent.

3.2.5 Success at Reward Rates

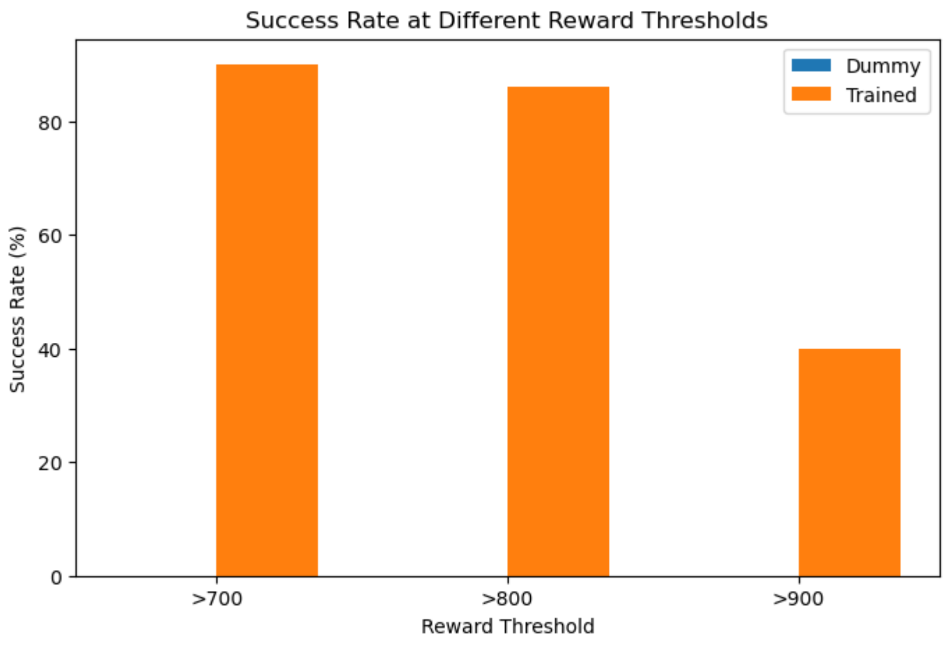

Figure 6: Success at Reward Rates

The success at obtaining reward rates at thresholds of 700, 800, and 900 is shown in Figure 6 above. The dummy agent was unable to reach any of the thresholds. The trained agent was able to reach 700 reward at an extremely high rate. The reward rate at 800 resembled a similar success rate. However, the reward rate at 900 was significantly lower. This was likely due to the agent's variance in performance.

4 Discussion

4.1 Key Findings

4.1.1 Emergent Behavior

The most notable finding during the testing process of the trained agent was an unexpected emergent behavior. The agent an policy where it would stop driving altogether to maintain its a high score (900+). It would then let the score decay until the game ended. This typically resulted in a score of 820-850. This risk averse behavior while technically optimal, reveals a limitation in the current reward structure. The current reward structure does not incentivize the agent continue driving to complete the track. It does not penalize the agent for stopping. It allows the agent to stop and avoid taking further risks.

4.1.2 Discrete Action Space

The performance improvement of the trained agent was significantly better when utilizing a 5 action discrete action space compared to a continuous action space. The training process never reached a high enough reward to meet the criteria set in the objectives when utilizing a continuous action space within the same time frame. When the action space was simplified, the agent was able to learn at a much faster rate. This was evident when the agent was able to reach metrics set in the objectives within 2000 episodes.

The difference in improvement rates can be attributed to the fact that a continuous action space is much more complex than a discrete action space. While it provides more fine grain control, it may give the agent too many variables to consider. I suspect that under a continuous action space, the agent will perform better if given more training episodes.

4.1.3 Dummy vs Trained Agent

The dummy agent's performance was used as a baseline to compare the trained agent's performance. The dummy agent's performance was significantly worse than the trained agent's performance. It averaged a reward of -93.13 with a standard deviation of +- 0.49. This means the dummy agent's actions comsistently led to the same result. There was little to no variance.

The trained agent's performance showed a significant improvement over the dummy agent's performance. It averaged a reward of 855.74 with a standard deviation of +- 141.07. This means the trained agent's actions were not consistent and had a high variance in the agent's actions, but it was able to average a much higher reward. It showed that its learned policy was better than random.

The high variance in the trained agent's performance can be attributed to a variety of potential factors. Due to the agent's risk averse strategy, some scores may have been artificially underrepresented. The agent may have been able to achieve a higher score if it continued driving. There is also a possibility that some of the randomly generated tracks may have been easier than others, leading to a performance variance. Moreover, limitations of a discrete action space may have also contributed to the variance. There potentially may have been randomly generated tracks that were extremely difficult to navigate without more precise control.

4.2 Implementation Insights

4.2.1 Hardware Configuration

ROCm and WSL were the chosen environment for this project. ROCm is a GPU acceleration library that is only compatible with AMD GPUs and acts as an alternative to CUDA. WSL stands for Windows Subsystem for Linux and is a Linux subsystem that allows for Linux to be run on Windows. This combination allowed for the training process to be completed in under 24 hours. The training time would likely be even faster if the agent was able to run on a cloud GPU service like SageMaker or utilize CUDA with a 4090 GPU.

WSl was utilized over a non virtualized Linux environment because my primary machine is a Windows machine. The other machine that was available was a Macbook Pro with an M1 Pro chip. The Macbook Pro would not be able to leverage ROCm or CUDA to accelerate the training process. ROCm was chosen over CUDA because I only have access to an AMD GPU on my Windows machine.

4.2.2 Debugging

The training process had a checkpoint system that allowed for the agent to be tested in a validation environment. This allowed for the agent's performance over time to be monitored and analyzed. The system was setup to check every 100 episodes since it was a small enough interval to not be at risk of some unfortunate event happening.

There was a failsafe in the event that there was a error. When an error occurred, the code would attempt to save the model and exit gracefully. If it was unable to save the model, the code would print a message to notify the user. This prevented the training progress of the agent from being lost.

4.2.3 Tracking

The training progress was tracked using a CSV file. The CSV file recorded the performance of the agent during each episode. It tracked metrics such as episode number, train reward, validation reward, epsilon, and learning rate. This allowed me to monitor the agent's training progress and catch issues early. The validation rewards were recorded every 100 episodes at the checkpoints. The average reward over 5 episodes was recorded.

With the tracking system, I was able to create key visualizations that allowed me to compare the agent's performance over time. The training rewards progression and validation rewards progression are shown in Figure 1 and Figure 2 respectively.

5 Future Work

5.1 Reward Structure Adjustment

In future work, adjusting the reward structure to favor track completion would potentially yield a better result. The current reward structure does not have a sufficient penalization for stopping. While the score is negative affected, it is very minimal when compared to the potential score loss from navigating off the track. With a higher penalty, the agent would be incentivized to continue driving to complete the track and abandon its risk averse strategy.

For example, the agent could be penalized on a time based basis. Depending on the time stationary, the agent could be penalized exponentially. This would incentivize the agent to not remain stationary for too long and encourage it to continue driving when possible.

Another approach would be to implement a multi-objective reward function. The reward function could be designed to balance the agent's performance between completing the track, driving fast, and avoiding penalties.

5.2 Action Space Refinement

While the discrete action space proved sufficient for the agent to learn to navigate the track, it could potentially be improved. The current action space is a 5 action discrete space, but a hybrid approach of a discrete and continuous action space may yield a better result. This approach may allow for the benefits of a continuous action space while still maintaining similar complexity to a discrete action space.

Implementation of this approach could utilize a tuning mechanism to determine the optimal action space. The tuning mechanism could be implemented by running the agent through a series of episodes and recording the performance. The performance could be averaged over a series of episodes and the action space could be tuned to utilize one of the two action spaces more based on the performance.

6 Conclusion

The project successfully implemented a Deep Q-Learning agent that was able to navigate the CarRacing-v3 environment. The agent was able to learn to navigate the track and achieve a score that met the criteria set in the objectives. The agent's performance was significantly better than a dummy agent's performance as demonstrated in the Performance Metrics section.

The project demonstrated the potential of reinforcement learning in the context of autonomous driving. Despite the 2000 episode training time, the agent was able to learn and achieve a score of 900+ with a 5 action discrete action space with no prior knowledge of the environment.

This project highlighted the considerations for tradeoffs that must be made when implementing a reinforcement learning agent. The choice of action space, reward structure, and hyperparameter scheduling all played a role in the agent's performance.