Reinforcement Learning In A Car Driving Enviroment

George Lawton, Benjamin Kataoka, Hyeonwoo Lee

***Abstract*—This project focuses on the implementation of deep reinforcement learning within the OpenAI's Gym environment to train an autonomous vehicle to drive and avoid obstacles in a 2D space. With the rapid advancements in artificial intelligence and the increasing prevalence of autonomous driving technologies, understanding the foundational principles of how reinforcement techniques are applied in training vehicle models are invaluable in today's world. This project, while not directly applicable to real-world vehicles, provided valuable practical experience in applying deep reinforcement learning techniques to address complex challenges in a 2D space, and showed how different techniques apply better to certain scenarios then others.**

I. Introduction

In recent years, artificial intelligence, reinforcement learning techniques and their applications have seen a significant growth both in terms of the rate at which the field is changing, and it's general integration into society. The expansion into the public eye has been particularly focused on autonomous vehicles, and driving technologies with major players like Google and Tesla each developing their own driving models. This surge of interest in autonomy in driving emphasizes the potential of these technologies and is the motivation behind our project's focus and exploration into the applications of deep reinforcement learning in autonomous vehicle navigation. With this project we hoped to better understand the mechanisms of specific reinforcement learning algorithms, and better appreciate its potential in creating more advanced driving systems.

To achieve this, we chose to integrate a simplified yet still applicable simulation environment, such as OpenAI's car_racing.py in Gym. This choice allowed for us to investigate and refine our control over reinforcement learning techniques and the environment itself in a uncomplicated setting. In learning how to apply reinforcement learning algorithms in even a simple simulation, we continuously improved the model's performance, and built an understanding that could be the foundation of an eventual transition to more realistic driving scenarios, and potentially even real-world driving.

II. Related Work

As autonomous vehicle technologies advance, understanding the algorithms that govern their decision-making processes becomes critical in developing even the most simple car-driving models. In this section, we review the studies that we found relevant to our developing our vehicle navigation, focusing on the application of reinforcement learning techniques in the environment that we used. Reinforcement learning applied in similar OpenAI gym environments. As well as the original paper which discusses a new approach to reinforcement learning, which is now called Proximal Policy Optimization.

We organize the related works into two main sections: Reinforcement learning algorithms, and other applications of reinforcement learning in both the car_racing.py environment and other environments from OpenAI gym.

A. Relevant Reinforcement Learning Algorithms

1) Proximal Policy Optimization Algorithms by John Schulman et al.: This paper by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, introduced the idea of Proximal Policy Optimization algorithms to the world, as well as explained how PPO addressed the various issues that other reinforcement learning algorithms failed to address. PPO was and remains a reinforcement learning algorithm that offers simple implementation, efficient learning, and generally better performance than the reinforcement learning techniques that preceded it.

Key Notes:
- Overview of Proximal Policy Optimization, its principles, and the general effectiveness of this branch of reinforcement learning techniques.
- Demonstrated how Proximal Policy Optimization is an algorithm that has high stability, high sample efficiency, and far less complexity (especially when compared to its direct predecessor trust region policy optimization or TRPO).

Application to Our Work: Our project makes use of the core algorithm from Schulman et al.'s Proximal Policy Optimization paper. Which we decided to use, after learning about PPO's more sturdy performance and suitability for efficient learning. By integrating Proximal Policy Optimization, we aimed to enhance the learning process, improve decision-making, and achieve more effective navigation in our modified car_racing environment.

2) Playing Atari with Deep Reinforcement Learning (DQN) by Volodymyr Mnih et al.: This work by Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou Daan Wierstra, and Martin Riedmiller is the first paper on Deep Q-Learning. This paper introduced the concept of DQNs as a reinforcement learning technique in teaching a model how to play the game Atari.

Key Notes:
- This paper was the introduction of the DQN algorithm, showing the algorithm's effectiveness in learning from pixel inputs.

Application to Our Work: Our project initially made use of the core algorithm from Mnih's paper. This algorithm was the one we initially decided to use in creating our model, as we knew about DQN's early inception, and had the conception that DQN may work fairly well in our modified environment. Unfortunately, we saw little success with this algorithm, with more details to come in the results section.

B. Application of Reinforcement Learning Algorithms in Similiar Works

1) Applying a Deep Q Network for OpenAI's Car Racing Game by Ali Fakhry: This article from Ali Fakhry is an article on how Fakhry solved the car_racing.py environment.

Key Notes:
- This article introduced the idea of modifying the environment to try to achieve a more favorable result.
- It also served as a sort of proof of concept for how we may begin to approach the problem at hand.

Application to Our Work:
While this paper wasn't anything groundbreaking in the grand scheme of things in reinforcement learning. For our project in particular it served as a sort of proof of concept for modifying the environment.

2) Learning to Play CartPole and LunarLander with Proximal Policy Optimization by Youness Mansar: This article by Youness Mansar attempts to solve the CartPole and LunarLander environments.

Key Note:
- This article served as a proof of concept for applying algorithms like Proximal Policy Optimization in OpenAI's environments.

Application to Our Work:
Similar to the last work listed, this was another source that served as a proof of concept when researching what we would do for our project.

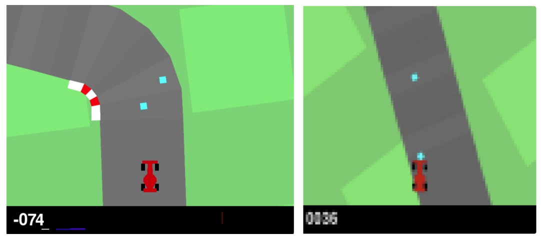
C. Conclusion

The exploration of reinforcement learning algorithms and their applications in similar works has laid the foundation for our project. The information that we gained from the original works on Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN), as well as the practical experiences, guided our decisions in what algorithms to use, and how to make use of them.

III. Problem Statement and Methods

A. Environment

To mimic a simplified version of the reality of driving, we use OpenAI's Car Racing v2 Environment. The environment provides the user with a car agent and the track that the car will be driving. Furthermore, for every episode of the simulation, a new track is generated. To train the model, a 96 × 96 RGB pixel frame of the environment is created for the agent to observe. The pixel frame provides a bird's eye view of the car and the area surrounding the car. Based on these frames, the agent decides an action. For simplicity sake, a discrete action space was decided on, allowing only 5 possible moves, that being accelerate, brake, turn right, turn left and do nothing. Of course, to sufficiently train the model, a reward was given for certain behaviors. First and foremost, a positive reward is given for simply staying on the track and moving forward. However, to prevent the car from simply staying still, this action is penalized and an episode is terminated if the agent stay still for too long. Furthermore, a small negative reward is also given for each step to further mitigate this issue. Since the goal is to complete a given track in the fastest way possible without leaving the track, the more tiles, as in the track, the agent visits the more reward it receives. Another implementation we decided to add to more efficiently teach the car was that an episode would be terminated if the track was in less than 10% of the frame the agent was observing, or if the car was off the track for excessive time. After seeing the effectiveness of the algorithm in this modified yet simple version of OpenAI's environment, we also decided to add obstacles to the track to see how well the agent could be trained given this dilemma.



The figure on the left shows what the user will see when running the simulations. The figure on the right shows what is fed into the agent for it to be trained. As it can easily be seen, the image on the right is much more pixelated compared to the left. The small cyan squares are the obstacles that we implemented.

B. Problem Statement

The objective of this project was to learn and compare the efficacy of different reinforcement learning algorithms, specifically policy methods and value methods that dictate the best action for the car agent to take at each state to avoid obstacles in the track and successfully complete the entire track autonomously. In detail, using Deep Q Network (DQN) and Proximal Policy Optimization (PPO) the car agent will learn what direction and the appropriate speed is required to navigate the track successfully without deviating off the path for each state.

C. Reinforcement Learning Algorithms: Policy and Value Methods

There are two fundamental approaches in reinforcement learning: Policy methods and Value methods.

1) Policy Methods: The method explicitly builds a representation of the policy and keeps it in memory during the learning decision.

$$\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}$$

Where: The equation denotes the probability of taking action a given state s and parameter θ, representing the current policy.

2) Value Methods: The method only stores the value function, so the policy here is implicit and it is able to derive directly from the function. In other words, we are able to pick the action with the best value.

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}[R(s,a) + \gamma \cdot V^{\pi}(s')]$$

Where: $R(s,a)$ is the immediate reward received upon taking action a in state s. γ is the discount factor determining the importance of future rewards.s' represents the successor state resulting from action a in state s.

D. Methods: Deep Q Network

Before we look into Deep Q Network, we first need to understand Q-learning. The basic strategy for our car agent is that no matter what state, take the action that gives the highest cumulative reward. So similar to the greedy algorithm, it takes the action that is judged to be the highest at every moment. To apply this strategy, what Q-learning does is to use a table of all possible state-action combinations and write down all Q-values there. After that, it can gradually update the table using the Bellman equation:

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

Here, the equation calculates the new Q-value for a state-action pair based on the immediate reward obtained from taking action a in state s ($r(s,a)$) and the maximum expected future reward discounted by γ ($\max_a Q(s',a)$), where s' is the resulting state after taking action a.

However, the greedy algorithm has one serious problem. If the algorithm always insists on making the 'best choice', it will never try new choices, and if this is repeated, it will never know about the existence of unknown rewards that have not yet been received. To solve this problem, a strategy called ϵ-greedy is applied. By using values between 0 and 1 for ϵ, the agent alternates between exploration and exploitation appropriately.

With a probability of $p = 1 - \epsilon$, the agent chooses to act greedily as it originally did, and with a probability of $p = \epsilon$, the action is taken randomly. By doing this, the agent has the opportunity to explore new spaces and learn about unknown rewards and the approach for this algorithm is called Q-learning.

However, in Q-learning, we face the challenge of having a massive number of possible states and actions for each problem. Even in the very simple game of Tic-Tac-Toe, there are over hundreds of possible 'states', and considering that there are up to 9 possible 'actions' in each state, it's a huge amount. This is where Deep Q Network comes in, the combination of Q-learning and deep learning. The idea is simple. Instead of using a Q-table in Q-learning, a neural network is used and the neural network model is trained to approximate the Q value.

Furthermore, the same occurrence happens when the volum of states becomes overwhelming for Q-learning. Thus this is where the integration of Convolutional Neural Networks comes in. The key is to preprocess the raw frames into the CNN. Each frame's pixel data is inputted into the network where it contains convolutional layers for learning the hierarchies and patterns from the visual data. Then the result is flattened to form a representation that captures the learned features.

This transformed representation from the CNN becomes the input for our DQN structure, allowing the network to predict the Q-values. The DQN here updates with the CNN to handle visual inputs and updates the policy respectively.

The expression for the model is $Q(s,a;\theta)$, where θ represents the weights to be learned by the neural network. The expression for the model is $Q(s,a;\theta)$, where θ represents the weights to be learned by the neural network.

We aim for the Q-values to converge towards their true values as the training progresses. When these values become perfectly equal, it signifies that our algorithm has reached its final destination of learning. Thus, in DQN, the learning process involves minimizing the disparity between these two values.

DQN learns by minimizing the difference between the predicted Q-values and the target Q-values derived from the Bellman equation as mentioned earlier.

This minimization objective aims to reduce the disparity between the predicted Q-values obtained from the neural network and the expected Q-values derived from the environment's dynamics. Ultimately, the goal is for the network to learn an accurate estimation of the Q-values, approximating the optimal policy and converging towards the true values for each state-action pair.

The convergence of Q-values signifies that the DQN has learned an effective strategy to make decisions in the given environment, approaching the optimal policy that maximizes cumulative rewards.

Now let's take a closer look at the learning process. In reinforcement learning, there is no separate training data set at first. As the agent takes action, the dataset gradually accumulates. At each moment, the Agent takes actions determined to be optimal through the network learned up to that point, accumulating data on states, actions, rewards, and the next state until the end of the episode

E. Methods: Proximal Policy Optimization

Proximal Policy Optimization is a policy gradient method for reinforcement learning which involves optimizing a surrogate objective function. PPO has become popular because it strikes a balance between complexity, sample efficiency, and ease of tuning.

The core idea behind PPO is to take small steps to update policies to improve performance while maintaining a stable learning process. PPO restricts the policy update at each iteration, ensuring the new policy is not far from the old policy, hence the term 'proximal'. This is realized through the use of a clipped probability ratio, ensuring the updates do not deviate too much from the current policy.

Let π_θ denote the policy parameterized by θ, and let $\hat{A}_t$ be an estimator of the advantage function at time step t. The probability ratio is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Where $\pi_{\theta_{old}}(a_t|s_t)$ is the probability of taking action a_t in state s_t under the old policy, and $\pi_\theta(a_t|s_t)$ is the probability under the new policy.

PPO optimizes an objective function $J(\theta)$ that is the minimum of an unclipped and clipped objective:
$L^{CLIP}(\theta) = \hat{\mathbb{E}}[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$

The clip function in the equation prevents the ratio $r_t(\theta)$ from moving outside of the interval $[1-\epsilon, 1+\epsilon]$, with ϵ being a hyperparameter, typically chosen to be a small value.

Furthermore, PPO often includes a value function penalty or entropy bonus to encourage exploration, which can also be included in the objective.

When applying PPO to our car driving environment, estimating the advantage function, $\hat{A}_t$, is critical. This is often computed using Generalized Advantage Estimation (GAE):
$\hat{A}t = \sum_{l=0}^{T}(\gamma\lambda)^l \delta_{t+l}^V$
where $\delta_{t+l}^V = r_{t+l} + \gamma V(s_{t+l+1}) - V(s_{t+l})$ represents the temporal difference error, γ is the discount factor for future rewards, and λ is the GAE parameter.

The training process iteratively collects data through interaction with the environment, evaluates the advantage estimator and policy gradient, and applies PPO updates to the policy weights θ. Specifically for this environment, the aim is to maximize the cumulative reward, which is achieved by the car staying on the track and minimizing lap time. The state s_t, could be raw pixel data from the environment, physics parameters of the vehicle, and other inputs like velocity. The action space a_t includes discrete values representing left, right, accelerate, brake and do nothing.

To train using PPO, the following iterative process was applied:

- Collect a set of trajectories by running the current policy in the environment.
- Compute rewards to go and advantage estimates for the collected trajectories.
- Update the policy by optimizing the PPO-Clip objective.

Although we understand the algorithm's approach and core idea, we decided to implement StableBaselines's PPO algorithm instead of creating our own due to time constraints and complexities in the algorithm that were out of our scope.

IV. Experiments and Results

Building on the environment and the algorithmic foundations outlined in the previous section, we now present the implementation and outcomes of our experiments. The primary focus of these experiments was to prove the performance of the selected reinforcement learning algorithms—Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN)—within the customized car_racing environment.

A. Results

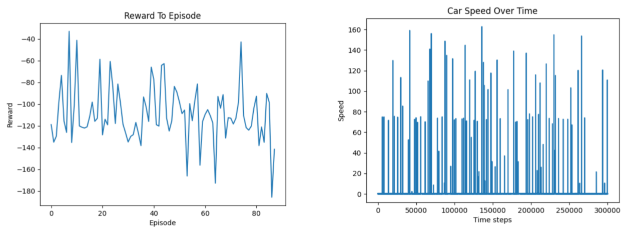


Fig. 1. DQN Graphical Results

The graphs above illustrate various metrics measured during iterations of training using the DQN approach within the modified car_driving environment. The left graph represents the reward per episode, while the right graph depicts the car speed over time.

In observing the graph above, it becomes apparent that the DQN approach to this problem maintained a high instability. Even when making use of StableBaselines DQN algorithm the results didn't seem to improve significantly. One of the conclusions we came to was that we did not give the model enough time to train for it to come up with a stable solution in traversing through the course using the DQN model. One potential reason behind the instability itself was the reward function we applied to the model and the environment, where the harsher negative rewards that we implemented in an attempt to improve training, may have interfered with effective training using the DQN approach. This instability aligns with a problem that is often associated with the DQN which is that the algorithm is often susceptible to fluctuations in learning and may require more training periods to converge to a stable solution when compared to something like PPO. This attribute of the DQN reinforcement learning algorithm highlighted the importance of tuning and extending training cycles when applying this algorithm to more complex tasks, such as vehicle navigation/operation.

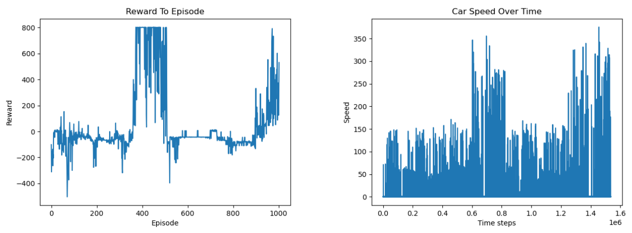


Fig. 2. PPO Graphical Results

The graphs above illustrate various metrics measured during iterations of training using the PPO approach within the modified car_driving environment. The left graph represents the reward per episode, while the right graph depicts the car speed over time.

Looking at the results, it can be seen that the graph looks much more stable compared to the DQN results. However, there is an abnormality in the graph, as in the large spike between episodes 300 and 500. Although at first the spike seemed inexplicable and we just observed it as a strange result, as we investigated further and compared it to the speed over time graph, we saw a clear correlation between these two. During the same time period as the spike, there was an increase in speed as well. So, the conclusion we came to was that before episode $\tilde{3}00$, there was a change in the policy that promoted acceleration. Due to the car being rewarded for staying on the track and visiting new tiles, the policy must have promoted more speed as it could visit more tiles in a shorter amount of time. However, when we looked closer we observed that although the policy was able to come up with an optimal solution for certain track types, there were also various points in this peak in which there existed severe dips in reward during the same time period. Nearing the 500-episode range, we assumed that there must have been another policy change as seen by the dip in reward as well as speed, meaning the PPO algorithm may have realized this was not a true optimal solution. As more episodes occurred, the graph began to see a gradual increase in not only reward but also speed. Furthermore, during this increase, the reward does not seem to deviate much compared to the initial increase in reward, meaning the policy was reaching a more stable and perhaps more optimal solution to this problem of traversing the track.

B. Abnormalities In The Behavior of The Car Agent

In this section, we go into an analysis of abnormalities/strange behaviors observed while training the car agent. Our exploration here attempts to shed some light onto what may have improved the training times based on specific abnormalities seen while training the model.

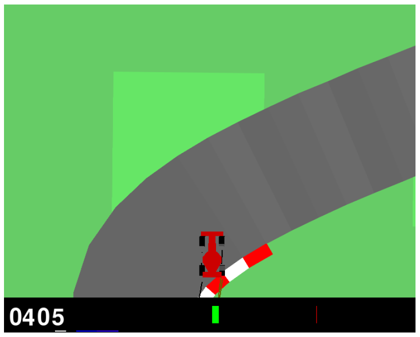


Fig. 3. Right Stopping

The figure above shows one of the more interesting behaviors seen while training the model on driving through the course. While it's difficult to see from a static image, this photo displays an instance in which the car model decided to stop instead of proceeding to take a right turn.

Interestingly enough this behavior presented itself fairly frequently throughout the training process, which prompted us to take a deeper look into why this may be. The conclusion that we came to for this particular behavior was related to a bias in the generation of the track. After a significant amount of time observing the generation, we began to notice a trend in the track generation, and the location of the car. It seemed as if the car was often placed on the right side of what is a circular track, thus it seemed as if the number of left turns often overwhelmed that of the number of right turns. Thus this behavior could be been attributed to the fact that the model may have not had enough training on right turns in the earlier iterations, as supported even further by the fact that later iterations often did better at performing on right turns. Therefore a possible solution to this problem may be to modify the track generation more such that the overall amount of left and right turns over all episodes are about the same.

Another interesting behavior that we encountered was the maintenance of the decelerated speed post passing an obstacle. As in after the car passed an obstacle, it often didn't accelerate while the obstacle was on the screen regardless of whether it had passed it. In general, this behavior makes sense to some degree, it may simply be that the model slows down when it sees an obstacle anywhere, which may have given significant rewards and the model has not attempted to learn to speed up after passing an obstacle. In general, we believe that after more iterations of training this type of behavior may have disappeared, however, a faster solution may have been to delete the obstacles after the car passed it.

V. Discussion and Conclusion

To conclude let's first look at the path forward if we were to continue working with this project we can easily envision the path we would take to get to the real world. One example is of course making the environment 3D instead of 2D, allowing the model to better mimic driving in real life. For example, using the CARLA library instead of OpenAI's Car Racing environment will most definitely allow our results to become more applicable to reality. We would also like to implement other different training models and other reinforcement techniques to better understand the true power of such technology. Other than that, we would simply train the agent more, allowing them to achieve more accurate and optimal results. Furthermore, we would want to move from a discrete setting into a more continuous action space to adapt to real-world scenarios where the model is not only based on moving left, right, forward, and backward. Using continuous action space means we will have the opportunity to explore different algorithms other than DQN and PPO, thereby enhancing the adaptability and performance of our model in real-world applications.

Overall, we have recognized that the environment that we have used here really does not map to real-life situations, but the big thing that we have learned here is that it does give a sense of scale to the efficiency and power of AI and reinforcement learning techniques, under the right conditions.

VI. Source Code

Link to the source code: GitHub Repository

Acknowledgment

The authors would like to thank **Professor Raj** for an amazing semester!

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Aug. 2017.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, ”Playing Atari with Deep Reinforcement Learning,” in *NIPS 2013: Neural Information Processing Systems*, Lake Tahoe, NV, USA, 2013, pp. 1-9.