Crash Now, Don't Crash Later:
Reinforcement Learning for Autonomous Vehicles

Peter Adam Aboud
Ryerson University
adam.aboud@ryerson.ca

Christian Wang
Ryerson University
christian.wang@ryerson.ca

Adam Srebrnjak Yang
Ryerson University
a1yang@ryerson.ca

Mohammad Saleh Joonghani
Ryerson University
msalehjo@ryerson.ca

***Abstract* — This paper reports various processes and implementations of Deep Q Learning on the 2D Open AI Gym environment "CarRacing-v0". The report will detail the problem, challenges, failures, and successes associated with our exploration into reinforcement learning.**

I. Introduction and Motivation

A. Significance of the Challenge

For decades, fully autonomous vehicles have been out of reach due to the exceptional hardware and software requirements associated with large scale sensing, localization, and decision-making processes. Various technologies have been posited as potential solutions. One of these is reinforcement learning, which aims to simulate the process of experiential learning which all humans do.

This new concept comes with a new set of challenges. Choosing the appropriate algorithm for autonomous driving is a difficult task as the algorithm needs to reach a significant amount of trustworthiness with respect to the uncertainties, convergence rate, and new data interpretation. Nonetheless, with all the challenges and difficulties comes a great reward since the most important factor in traffic incidents is the human driver. An increase in road safety and a commensurate decrease in fatalities are what autonomous vehicles promise [1].

B. Overview

The project began with initial research and exploration into a multitude of different algorithms. Our team settled on Deep Q Learning (DQL) for its robustness and ease of implementation. We created four different implementations, each of which varied either in architecture, hyperparameter values, or code implementation.

To evaluate our models, videos were generated of an agent traversing a random track under the network weights created from the training process. Plots of the total reward, episodic duration, and losses were also generated to rate the modifications made to the basic implementation. The remainder of this report will detail the journey taken with DQN to attempt to solve the car racing environment.

II. Problem Statement and Environments

A. Problem Statement

The purpose of the exploration was to ascertain which combination of DQL hyperparameters, optimizers, and methodologies resulted in the best possible outcome for the car racing environment. This was gauged based on factors such as the convergence rate, total reward, episode completion, episode duration, and heuristic analysis of the method of driving. The details of said environment are described below in detail.

B. Environments and Libraries

We used a top-down racing environment from OpenAI known as CarRacing-v0 [2]. The environment utilized a continuous action space over 3 controls, namely braking, acceleration, and turning. These were bounded between the value of 1 and -1 inclusively. The environment applied rudimentary methods to simulate real world motion. Such methods included, but were not limited to implementations of friction, variable traction, and the use of dynamics associated with inertia and momentum. Due to the nonlinearities associated with the environment, multiple assumptions and a thorough

understanding of the environment was required, which has been detailed below.

The action space of the car racing environment was continuous between the values of 1 and -1. For braking and acceleration, it was determined that the values were effectively between 0 to 1, where 0 represents an idle motion and 1 represents the full application of the action. A value of -1 in turning represented a left turn and 1 represented a right turn. Note, braking was applied gradually unless a value above 0.9 was applied. In that instance, a wheel lock was initiated putting the agent into a skid. These values were organized into a 3x1 vector, ordered by turning, acceleration, and braking. There was a significant distinction between the operation of acceleration versus speed, namely that an acceleration of 0 did not guarantee the agent was not in motion.

The environment utilized both positive and negative rewards along with extended rewards upon access of a termination state. A reward of -0.1 was associated with every frame in the environment, note this meant that at any time step, a negative reward was added to the immediate reward of the agent. A positive reward of 1000/N, where N was the number of unique track tiles, was awarded to the agent on traversal of a new track tile. Note in the environment, there were track tiles, grass tiles, and black or out of bound tiles. A termination state was associated with exiting the bounds of the environment, with an immediate step reward of -100 added to the total reward. A second termination state was associated with completing or traversing the entire track. Note, no additional rewards were associated with completion of the track.

The track was generated randomly after every environment reset. Resets occurred when a termination state was reached. The details of track generation are unimportant aside from the fact that the agent did not meet the same consecutive track through consecutive episodes. The position of the car was calculated using the dynamics equations and momentum associated with the program, this is stated for purposes of understanding and the details are not crucial to operation of the learning algorithm.

Due to limitations in memory and time, the action space of the system was discretized to accelerate the convergence rate of the system. The action space was limited to the normal operations of a vehicle, namely accelerate, brake, turn left, and turn right. This basic space was then extended to take advantage of features such as gradual braking, turning while accelerating, turning while braking, and applying middle values of 0.3 to 0.5 to each of the action space parameters.

III. Methods and Models

A. Preprocessing

The original pixel space was down sampled from 96x96 to 40x60. Trimming excess peripheral space and action displays allows the model to focus on the pixels that really matter. Gray scaling was not applied, so the 40x60 pixel space has three channels representing the RGB values. Finally, 3 frames were stacked and fed into the model. Frame stacking was crucial as it provided a transient understanding of the environment across a period of time. This provided the agent with more meaningful changes in the environment, which positively affected the operation of Deep Q Network.

B. Deep Q Learning (DQL)

In brief, DQL uses neural networks to approximate a nonlinear function that maximizes expected reward based on a state-action space. A replay memory stores state-action pairs (“what it did and when it did it”), along with the reward it got, and the subsequent state. This acts as a type of local memory which the agent will randomly sample from during training. For instance, when a human is faced with a new scenario and is forced to act they may draw on past experiences to guide them, even if those past experiences do not map perfectly to the current situation. A loss function is used to evaluate agent actions for a given state, subsequent minimizing of the loss function by the optimizer drives the agent toward an ‘optimal policy’. In this exploration we used an L1 loss function, which is the sum of all the absolute differences between true and predicted values. This function is more tolerant of outliers, compared to the L2 loss function. The agent is forced to ‘explore’ various actions by way of an epsilon value. This is referred to as an epsilon-greedy algorithm.

The epsilon value represents a likelihood that the agent will take a random action as opposed to the stored "best so far" action for a given state. This epsilon value decays over time, in the hopes that as the agent trains its learning will become more valuable than randomized action sampling. Because this algorithm uses a neural network, the learning can be unstable or divergent. In order to overcome this, disjointed weight updating schedules are applied. A target model will only have its weights updated after some predefined interval, this in essence prevents overfitting. Finally, other hyperparameters affect the performance of this algorithm, specifically: learning rate, discount factor, and epsilon decay rate.

Basic 1

Our first Deep Q implementation was a failure. Some of the key features of the implementation were:

- *Frame difference input*

The input was a representation of the difference between the current frame and the previous frame. Essentially using the change in pixels as input for the network. This is clearly a significant error, both on a conceptual level and a practical one. The agent is supposed to mimic a human user, and we take in visuals as we receive them, not as a differential representation of past states.

- *RMS prop optimizer*

The Pytorch tutorial that we based this implementation on suggested using RMSprop as the optimizer. RMSprop uses adaptive learning rates generated by the root mean square of the gradient.

- *Lack of episode termination*

In this model we allowed the agent to continue in the environment for a predetermined maximum number of time steps. Early on it was clear that the agent wasn't acting productively, and having it remain in the environment accruing negative rewards was not helping to converge to an optimal policy.

- *Epsilon decay resetting*

The single most important shortcoming of this implementation was that we were unknowingly resetting the epsilon value after each episode. Coupled with a starting epsilon value of 1, this effectively means that the agent was throwing out everything it learned in favor of exploration after each episode.

Basic 2

After consulting with Dr. Farsad and Mr. Kowal, we identified the failure points of our implementation and fixed them.

- RMSprop optimizer was replaced with ADAM optimizer. ADAM optimizer was found to be more favorable due to slower momentum in learning, which helps in volatile environments.
- Refactored code
- Fixed the scope issue with epsilon resetting
- Implemented LR Scheduler for ADAM
- Implemented an episodic termination
    - If the agent acquires 30 negative rewards consecutively it terminates the episode

Refer to appendix Figure 8 for the architecture of this model.

The changes above resulted in an immediate improvement and a learning model that converged on an optimal policy. Despite this success, we decided to try and improve our results by making some changes to the underlying architecture. Models 3 and 4 represent those efforts. They are both very similar in terms of hyper parameter configuration and input conditions, and as such we will only discuss the differences at the neural network architecture level.

Deep 1

For this experiment we decided to add a max pool layer, but simultaneously removed a convolutional layer. Max pooling is a process whereby an input representation has its dimensionality reduced, while making some assumptions about the features that are not kept. Figure 1 is a simple rendering of the max pool function. Refer to the appendix for the architecture of this model.

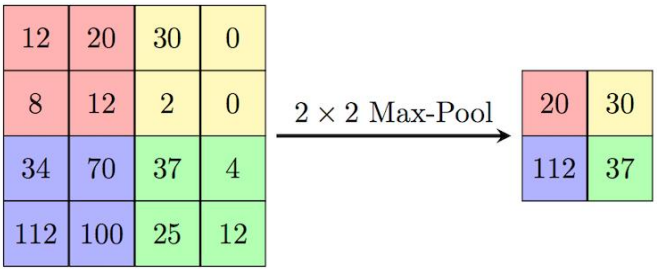

Figure 1. Max pool Operation

Deep 2

Building on Deep 1, we added back the third convolutional layer, and added an average pooling layer. Average pooling is like max pooling, but instead of taking the maximum value in the input it takes the average value. Refer to appendix Figure 8 for the architecture of this model.

IV. Results and Discussion

Basic 1

The results of our first implementation showed no convergence to an optimal policy after a thousand training episodes. Figure 2 shows the accrued rewards over the training process

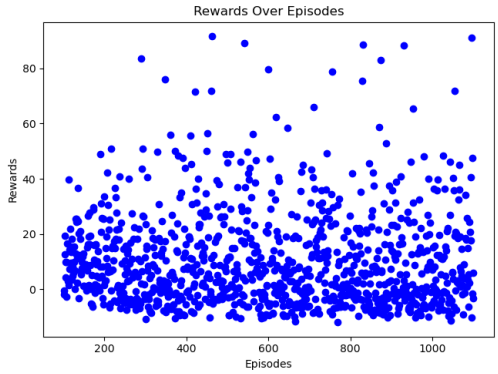

Figure 2. Basic 1 - Rewards over Episodes

Clearly there is no improvement in the agent's received reward over time. Confirming that the problems discussed in the previous section prevented this implementation from converging on the optimal policy. Videos of the agent produced after a thousand training episodes still show the car spinning in circles at the starting line.

Basic 2

After correcting the mistakes of our first implementation we were immediately met with significant improvements in the results. Figure 3 shows the episode/reward plot for this implementation.

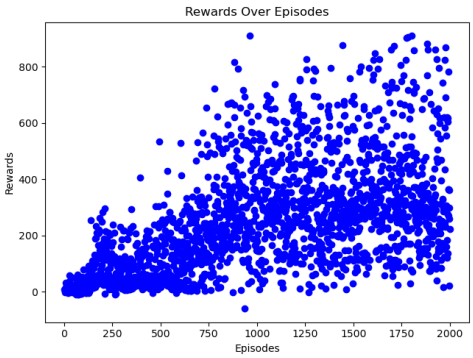

Figure 3. Basic 2 - Rewards over Episodes

As you can see, the agent begins improving almost immediately, and by a thousand training episodes is averaging approximately 400 rewards per episode. A closer look at the graph reveals that that average however does not increase much after 1000 episodes.

The generated videos of this model show that the agent has learned to accelerate early and to stay on the track. If the agent happens to get off track it will navigate its way back. Please see the video reference for a detailed demonstration.

Deep 1

This model learned fast but hit a ceiling just as quickly. Moreover, the average reward gain was smaller than that of Model 2. We speculate that this is because we removed one of the convolutional layers. Adding the max pool layer allows the model to learn in a more general sense, quickly identifying features that represent large changes to expected rewards but missing the smaller features that represent a lesser proportional value. Figure 4 shows the episode/reward plot for this model.

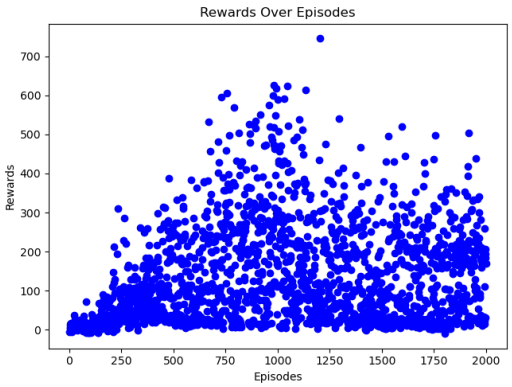

Figure 4. Deep 1 - Rewards over Episodes

The videos generated from this model show the agent has a limited 'understanding' of the game, making significant errors in strategy. For instance, over-braking at upcoming turns and over-correcting turns. Please see the video reference for a more detailed demonstration

Deep 2

Our final experiment with architecture modification yielded some of the most interesting results. The reward plot showed a cyclical pattern of learning. The agent would become proficient at solving the problem, but then performance would degrade over a period before improving again. These 'learning peaks and valleys' are most likely due to the 'deep' nature of the architecture and the role of the nonlinear function approximator. Despite this, the model weights extracted during a 'learning peak,' result in the agent conducting itself quite effectively around the track. We speculate that additional training time would show that the performance of this model would continue to increase on average. Indicating that this architecture holds the most promise for this and other environments. Figure 5 shows the reward/episode plot for this model.

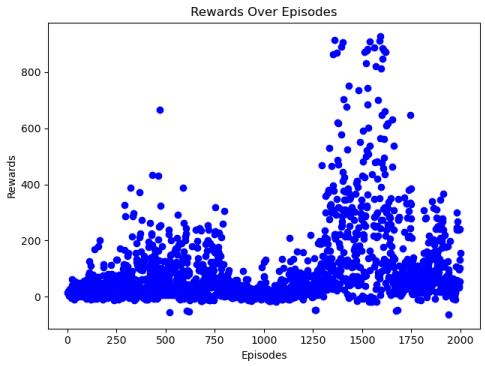

Figure 5. Deep 2 - Rewards over Episodes

Comparison

In order to compare these implementations, we extracted model weights that corresponded to the best performance for that model, and then tested them each over the same 50 randomly generated tracks. Figure 6 shows the accrued rewards for each model over the testing period, the red line represents the average rewards over the entire period.

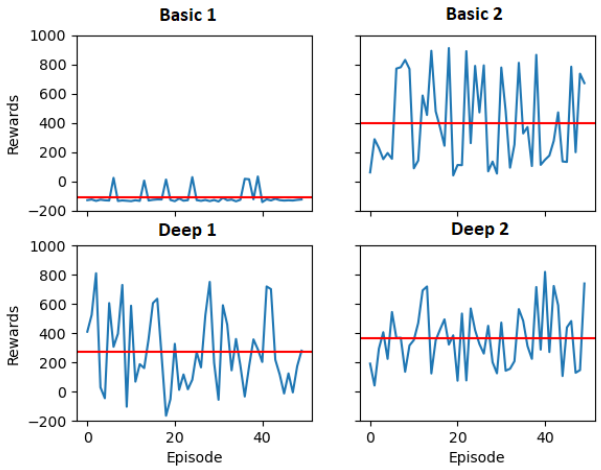

Figure 6. Rewards Comparison between Models

This figure clearly demonstrates the failure of our first implementation, showing an average of below zero for reward gain. Surprisingly though, Deep 2 had a slightly worse average performance than Basic 2. In the future an experiment could be run to ascertain why Deep 2's performance was so poor. Perhaps there are track features that stumped the agent at these points, or

perhaps the agent got turned around and began driving the wrong way around the track.

Average reward gain is not the only way to evaluate our models. Figure 7 shows the reward accuracy of each model over an average time step. A positive value indicates that on average the model takes an action that results in positive reward. The red line represents the average accuracy for the model.

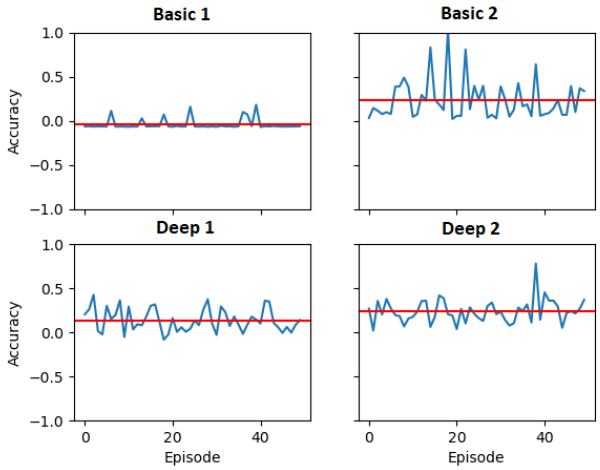

Figure 7. Accuracy Comparison Between Models

The performance difference between Basic 2 and Deep 2 is much smaller here, with both having excellent accuracy. Additionally, Deep 2 exhibits much tighter values around the average, indicating higher precision than Basic 2.

Besides just number-based metrics, we also took a qualitative approach to evaluation by loading each of the models and pitting them against one another on the same randomly generated track. Basic 2 starts off strong but gets turned around, while Deep 2 drives more cautiously and accurately, ultimately winning the race. Please refer to the accompanied video for a white-knuckled ride.

V. Implementation and Code

The main libraries used in this project are gym, an open-source reinforcement learning environment package, specifically the Atari and CarRacing-v0 environment. PyTorch was used to build and run neural network implementations. Matplotlib was used to generate visualizations of our results. Our implementation used the PyTorch Deep-Q learning tutorial as a starting point. Please refer to the Github repo for a detailed README on our implementation, the link can be found in the references.

VI. Future Work

Our Basic 2 model is a great starting point for further exploration of Deep Q learning as a solution for this problem. Deep 1 and Deep 2 are good first steps to finding the best solution. Both of these models would be well served by an increase in training time, particularly Deep 2. This would allow us to verify our speculations about the cyclical learning pattern of this architecture, and hopefully show a continual increase in average performance. Further experimentation and exploration is required in order to increase the reliability, performance, and accuracy of these models. Such experiments might include:

- Additional preprocessing
- Hyper parameter optimization
- Action space modifications

Larger changes to the architecture are worth exploring as well. The integration of Long Short-Term Memory (LSTM) might allow the agent to anticipate future states and act predictively. Similarly, a Dueling Deep Q implementation might yield interesting results. By having separate estimators for the state values and state-action advantages, the agent would be able to learn when it is most important to act.

Finally, transplanting these and future models into other environments would allow for our team to evaluate the ‘generalized’ performance of our work. Specifically, the OpenAI highway-env would be a natural successor to CarRacing-v0.

VII. Conclusions

Though much work remains, this exploration of Deep Q learning yielded very positive results, ultimately solving the CarRacing-v0 problem. We maintain that although dependent on custom environments for training and evaluation, reinforcement learning is a robust and viable path to success for the autonomous driving problem.

VIII. References

https://github.com/OneHandKlap/CPS824-Project

https://www.youtube.com/watch?v=YS-fcZ05Z9Y

[1] M. F. Lohmann, “Liability Issues Concerning Self-Driving Vehicles,” European Journal of Risk Regulation, vol. 7, no. 2, pp. 335–340, 2016.

[2] https://gym.openai.com/envs/CarRacing-v0/

[3] Brockman, et al., “A toolkit for developing and comparing reinforcement learning algorithms,” *Gym*. [Online]. Available: https://gym.openai.com/. [Accessed: 17-Apr-2021].

[4] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., & Lerer, A. (2017). Automatic differentiation in PyTorch. In NIPS-W

[5] J. D. Hunter, “Matplotlib: A 2D graphics environment,” Computing in Science & Engineering, vol. 9, no. 3, pp. 90–95, 2017.

[6] V. Bushaev, “Understanding RMSprop - faster neural network learning,” *Medium*, 02-Sep-2018. [Online]. Available: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a. [Accessed: 17-Apr-2021].

[7] R. Karim, “Intuitions on L1 and L2 Regularisation,” *Medium*, 05-Oct-2020. [Online]. Available: https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261. [Accessed: 17-Apr-2021].

[8] A. Paszke, “Reinforcement Learning (DQN) Tutorial,” *Reinforcement Learning (DQN) Tutorial - PyTorch Tutorials 1.8.1+cu102 documentation*, 2017. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#. [Accessed: 17-Apr-2021].

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv.org*, 19-Dec-2013. [Online]. Available: https://arxiv.org/abs/1312.5602. [Accessed: 17-Apr-2021].

IX. Appendix

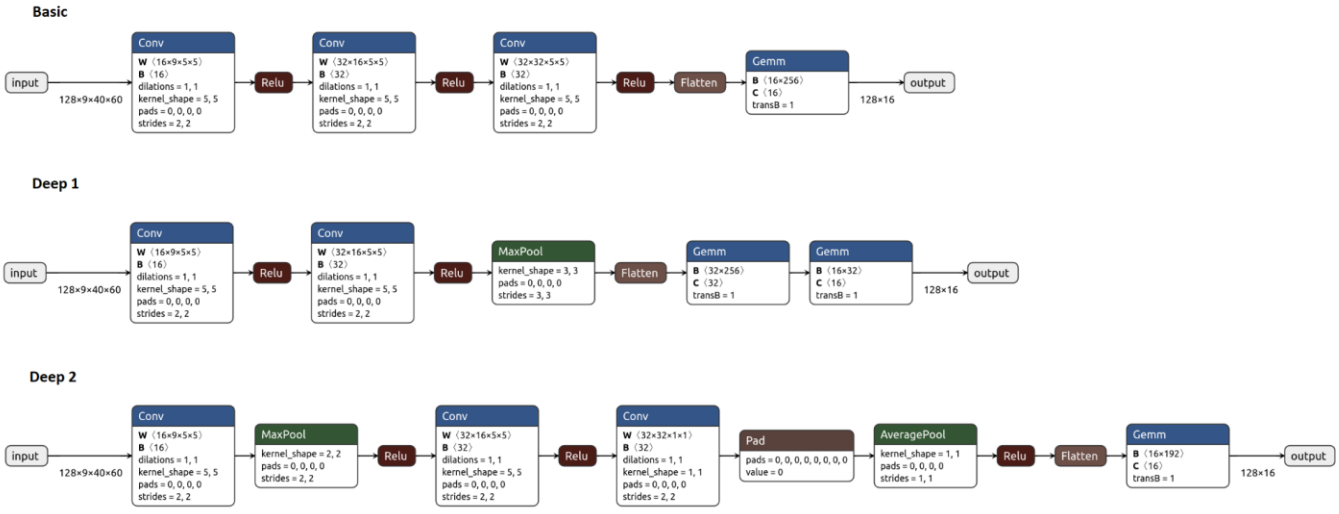

Figure 8 - Comparison of various implemented models