RL Agent for Google Dino

Mikkael Dumancas

Introduction

This project explores the application of Reinforcement Learning (RL) to train an agent to autonomously play Google's offline Dino Game. Reinforcement learning is an area of machine learning where an agent learns to perform actions within an environment to maximize cumulative rewards. The central concepts in RL include the agent, reward, environment, and action which interact continuously until the environment reaches a terminal state.

Project Setup

This project was run completely locally. The PyTorch implemented did not use GPU acceleration, only CPU (for this setup, the CPU used is a Ryzen 7 3700). The following dependencies were used:

- **Stable-baselines3**: Facilitates reinforcement learning algorithms, specifically DQN in this project.
- **Gymnasium**: Provides the environment API for RL interactions.
- **MSS**: Efficient screen capture for real-time observation extraction.
- **Pydirectinput**: Enables direct keyboard interactions for executing actions.
- **Numpy**: Manages array operations, primarily for image processing.
- **OpenCV**: Handles image preprocessing and analysis.

Methodology

The approach of this project boiled down to 4 key steps:

1. Building the environment

In this project, a custom environment was built using OpenAI's Gymnasium API to handle agent-environment building and interactions. The key methods of the environment are:

- `__init__`: Initializes the environment with defined observation and action spaces. For this project, it also handles bookkeeping for reward shaping done in the step() method
- `step`: Implements the iterative interaction cycle, handling the agent's observation, selected action, reward calculation, and next observation.

- `get_observation`: Uses mss to capture an image of the game environment and handles preprocessing the image via grayscaling and resizing.
- `get_done`: Uses mss to capture an image of the game over screen and uses pixel color recognition via OpenCV to determine environment termination
- `restart`: Uses pyautodirect to move mouse to the left side of the game and click, triggering the game to start again.

2. Training the model

The RL model used was a Deep Q-Network (DQN), a combination of Q-learning and the framework of a neural network. Q-learning functions by learning Q-values, which represent the expected cumulative reward of taking a particular action in a given state. The project is coded using PyTorch and stable-baselines3, which is a PyTorch library that supports and handles DQN. Callbacks provided by Stable-baselines3 are integrated for performance monitoring and logging during training.

3. Testing the model

Utilizing the logs created from each callback, previous runs can be tested by passing them into the model. For simplicity's sake, the model's testing criteria are their top scores from training and within a testing stage of 5 episodes.

4. Optimizing the model

Since this project focuses on DQN, I explored different optimizations that could be made to the model within these constraints:

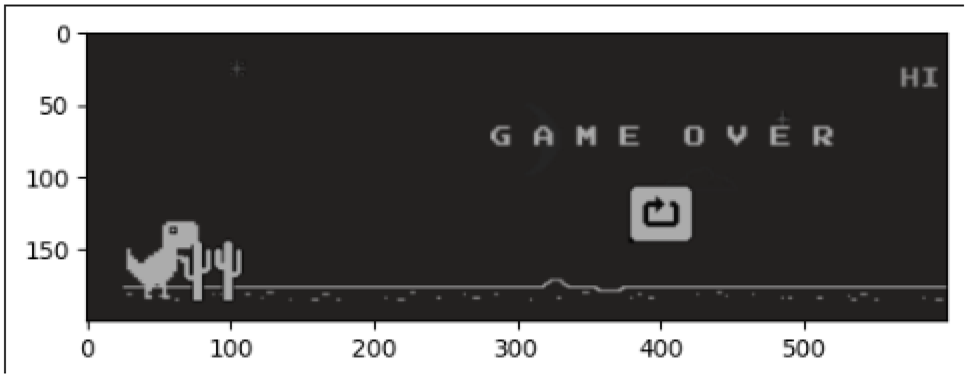
- Training rounds of 50,000 episodes
- A replay buffer of 300,000

This limited the training periods with my setup to around 30-45 minutes, which worked out for testing different optimizations. This project focuses on input optimization, reward shaping, and hyperparameter optimizations.

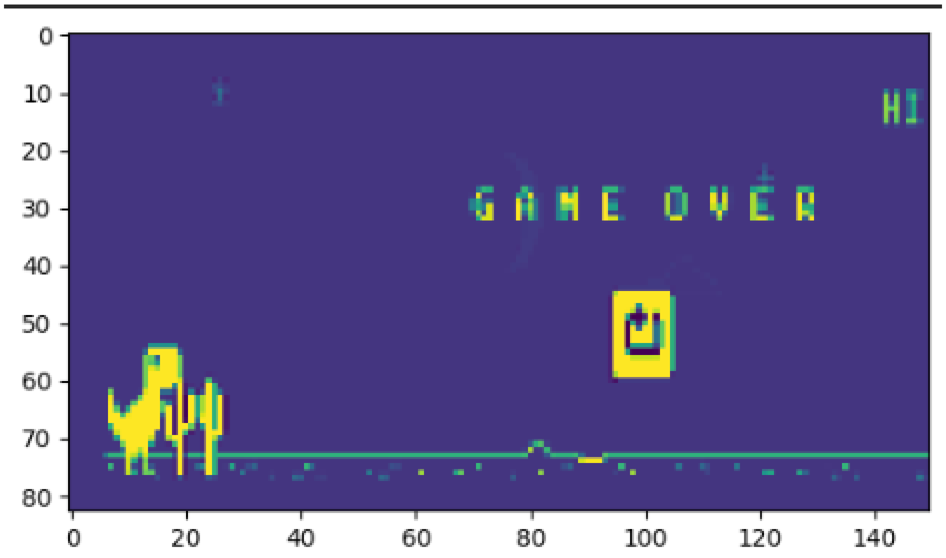
Dataset Description

In this RL setup, traditional static datasets are not utilized. Instead, the dataset consists of real-time observations captured as screenshots during gameplay interactions. Each screenshot represents the current state of the game environment. These observations are continuously generated through the MSS library, processed using OpenCV, and fed directly into the DQN for immediate decision-making.

This is an example of the raw image that is captured for the game state:



This is what the image fed into the DQN looks like after preprocessing:



Optimization (Complexity of project)

Building the environment and training the model is the baseline of the project. To make this project more complex, I wanted to explore optimizations that could be made. These are the optimizations that I explored:

Input optimization

The game input turned out to be a major bottleneck in how much frames were processed per second. I explored using different methods in pyautodirect, and successfully increased fps from an initial 3 fps to an average of 8-10 fps. I also explored more direct inputs, such as the keyboard library. However, the keyboard library made processing nearly instantaneous, increasing the fps to 3 digits. This caused training to become super difficult, because so many actions were being spammed, and ultimately I chose to stick with pyautodirect for more stability.

Reward shaping
My initial reward setup was simply to reward the agent for every frame it was alive. In testing, this proved to work but only over super long training runs. My goal was to ultimately find a reward shape that worked best at optimizing the agent within 50,000 episodes.

I found that the best reward shape was giving the model a very small dense reward for each frame survived, a sparse reward for each time it would clear an obstacle, a penalty for every time the environment terminated (game over), and a small penalty for taking an action. Conceptually, this was how each part of my reward was shaped:

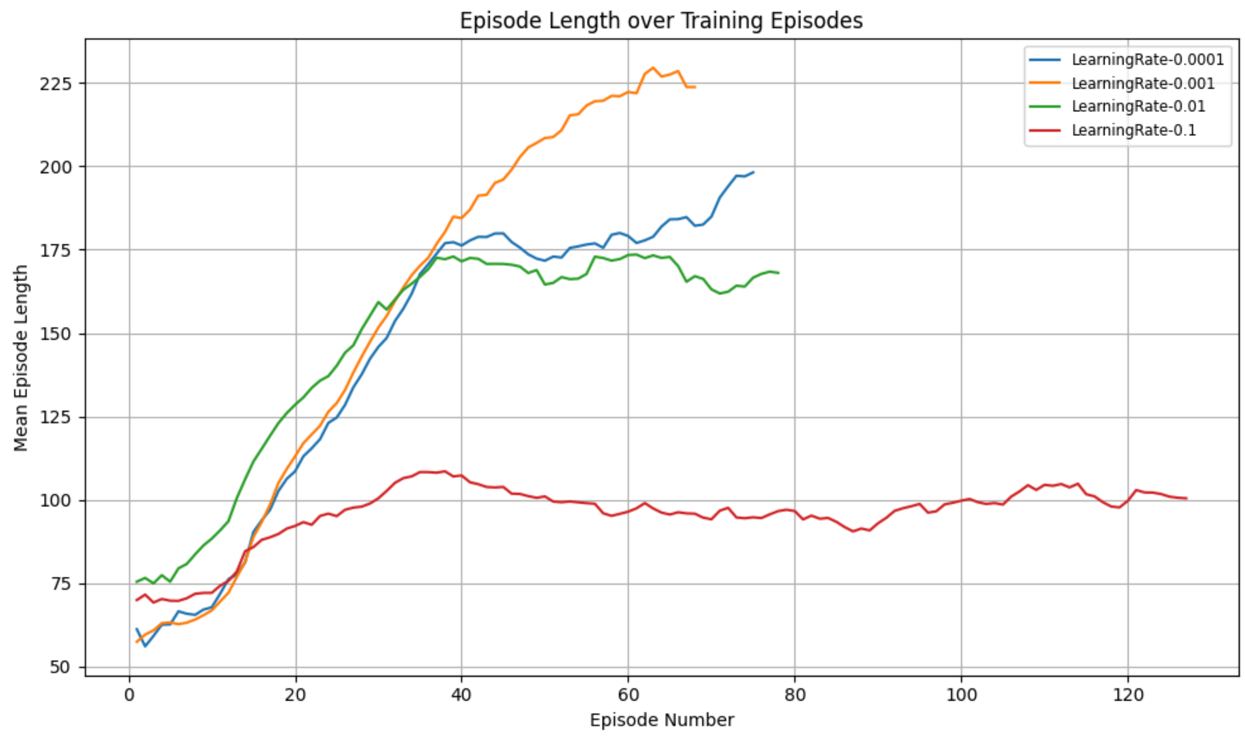
$$r_{\text{survive}}(t) = 0.01$$
$$r_{\text{clear}}(t) = 2$$
$$r_{\text{crash}}(t) = -1$$
$$r_{\text{act}}(a_t) = \begin{cases} -0.02, & a_t = \text{jump}, \\ 0, & \text{otherwise.} \end{cases}$$

Hyperparameter tuning
For this project, I tested out different buffer sizes, learning rate, discount factor, and exploration rate. At first, I trained the model using the default setting and a buffer size of 300,000. By default these were the hyperparameters:

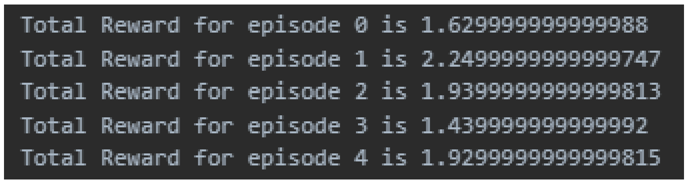
- Learning rate = 0.0001
- Gamma (discount factor) = .99
- Exploration rate = Starts at 1 and decreases to .05 over 10% of training

These are good starting parameters, specifically Gamma and Exploration for initial training. I first focused on finding a good learning rate while keeping everything else default. Here were my findings:

Learning Rate



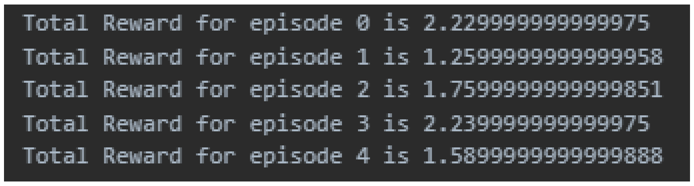
At .1:



```
Total Reward for episode 0 is 1.629999999999988
Total Reward for episode 1 is 2.2499999999999747
Total Reward for episode 2 is 1.9399999999999813
Total Reward for episode 3 is 1.439999999999992
Total Reward for episode 4 is 1.9299999999999815
```

Training High Score: 87
Testing High Score: 47
Couldn’t make it past 1st obstacle after 50,000 episodes

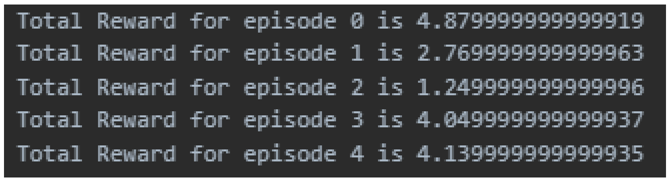
At .01:



```
Total Reward for episode 0 is 2.229999999999975
Total Reward for episode 1 is 1.2599999999999958
Total Reward for episode 2 is 1.7599999999999851
Total Reward for episode 3 is 2.239999999999975
Total Reward for episode 4 is 1.5899999999999888
```

Training High Score: 98
Testing High Score: 47
Still couldn’t make it past 1st obstacle after 50,000 episodes

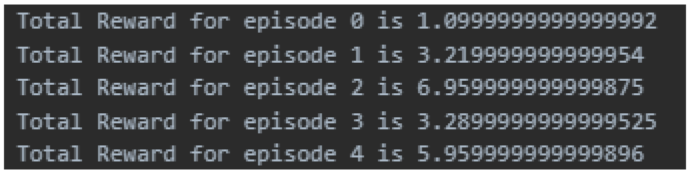
At .001:



```
Total Reward for episode 0 is 4.879999999999919
Total Reward for episode 1 is 2.769999999999963
Total Reward for episode 2 is 1.249999999999996
Total Reward for episode 3 is 4.049999999999937
Total Reward for episode 4 is 4.139999999999935
```

Training High Score: 266
Testing High Score: 102

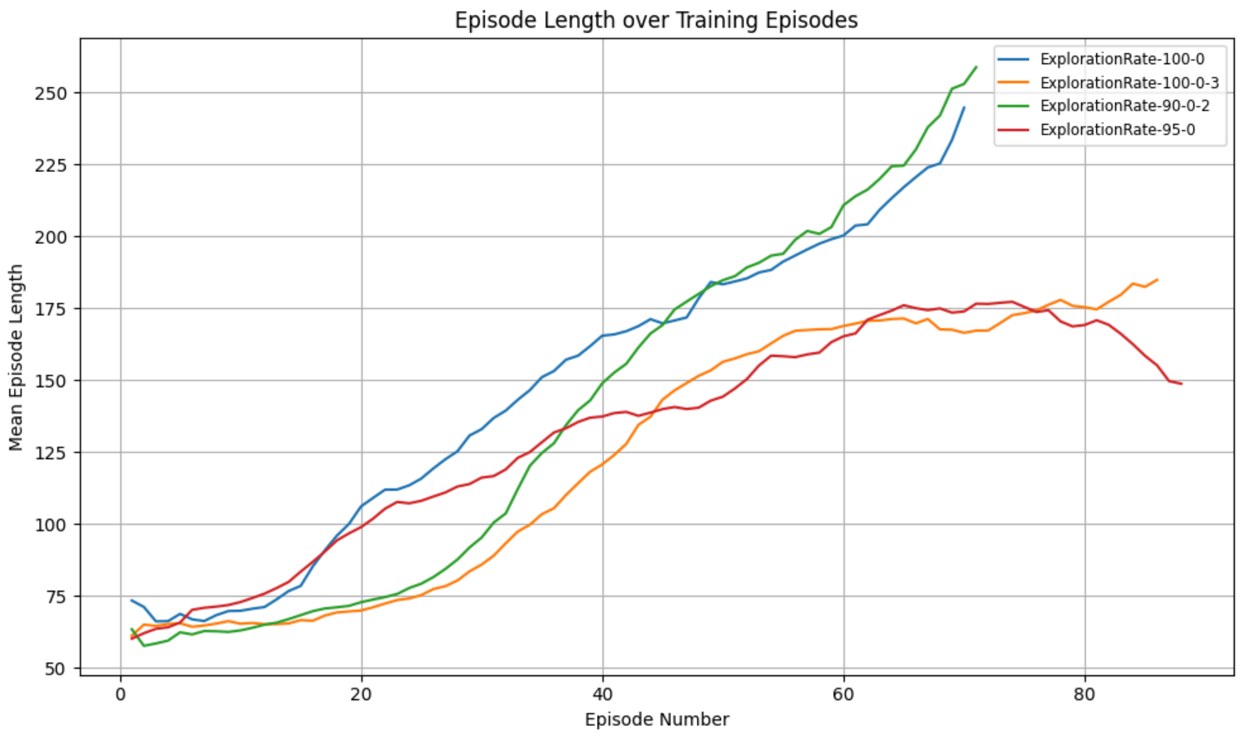
At .0001:



```
Total Reward for episode 0 is 1.0999999999999992
Total Reward for episode 1 is 3.219999999999954
Total Reward for episode 2 is 6.959999999999875
Total Reward for episode 3 is 3.2899999999999525
Total Reward for episode 4 is 5.959999999999896
```

Training high score: 152
Testing High Score: 250

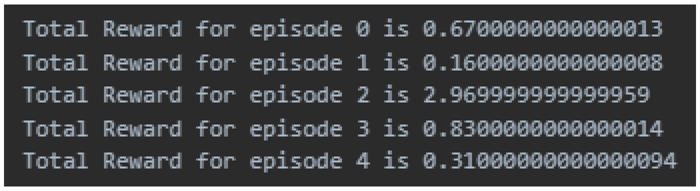
Exploration rate



With this hyperparameter I wanted to test the following scenarios:

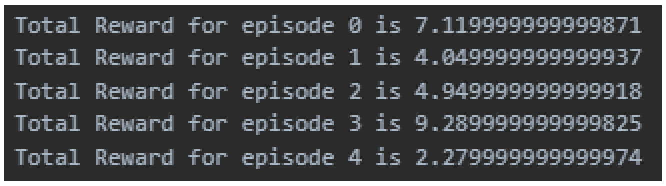
- Lower the frequency of exploration overall to possibly lower jump spam (Slightly less initial exploration -> no exploration)

At .95 initial and .0 final:


```
Total Reward for episode 0 is 0.6700000000000013
Total Reward for episode 1 is 0.1600000000000008
Total Reward for episode 2 is 2.969999999999959
Total Reward for episode 3 is 0.8300000000000014
Total Reward for episode 4 is 0.31000000000000094
```
Training High Score: 180
Testing High Score: 150

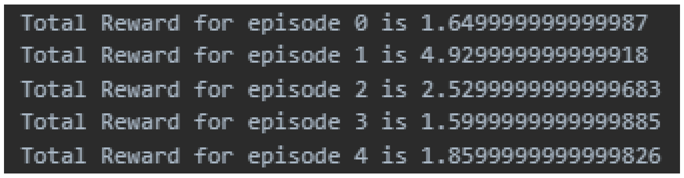
- Remove randomness from the model's training at the end to try out what it knows (No exploration at the end)

At 1 initial and .0 final:


```
Total Reward for episode 0 is 7.119999999999871
Total Reward for episode 1 is 4.049999999999937
Total Reward for episode 2 is 4.949999999999918
Total Reward for episode 3 is 9.289999999999825
Total Reward for episode 4 is 2.279999999999974
```
Training High Score: 287
Testing High Score: 232

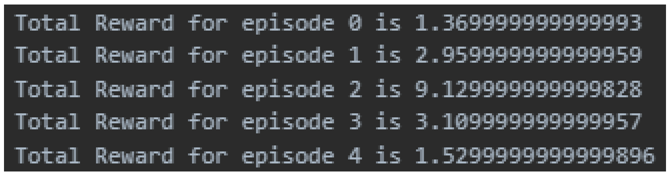
- Allow model to explore a larger fraction of the episodes (increase exploration fraction)

At 1 initial and .0 final with a .3 exploration fraction:

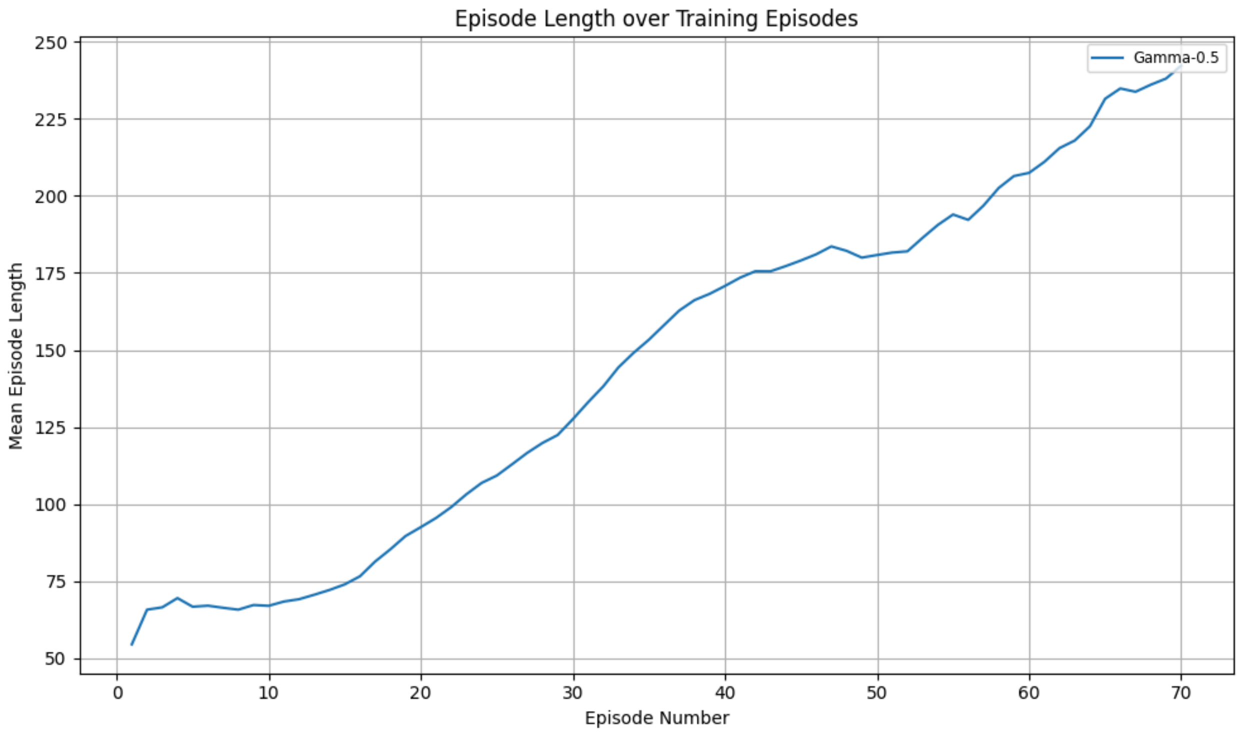

```
Total Reward for episode 0 is 1.649999999999987
Total Reward for episode 1 is 4.929999999999918
Total Reward for episode 2 is 2.5299999999999683
Total Reward for episode 3 is 1.5999999999999885
Total Reward for episode 4 is 1.8599999999999826
```
Training High Score: 222
Testing High Score: 171

- Expand the episodes with exploration but make exploration more cautious (less frequent exploration within episode + more episodes to explore)

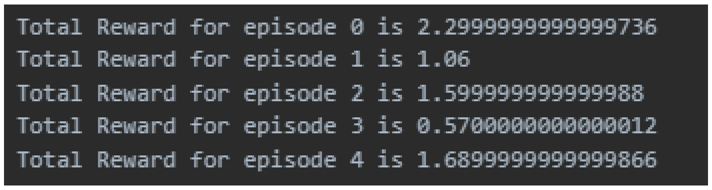
At .90 initial and .0 final with a .2 exploration fraction:


```
Total Reward for episode 0 is 1.369999999999993
Total Reward for episode 1 is 2.959999999999959
Total Reward for episode 2 is 9.129999999999828
Total Reward for episode 3 is 3.109999999999957
Total Reward for episode 4 is 1.5299999999999896
```
Training High Score: 302
Testing High Score: 435

Gamma (Discount Factor)



For gamma, I wanted to only explore one specific scenario. By default, the DQN model prioritizes long-term rewards. However, I wanted to test if making the model favor immediate rewards is beneficial. This goes against the logic of maximizing survival time, but I thought it would be interesting to see how it changed the behavior of the agent in testing. To test this, I lowered gamma to .5. The results are:



```
Total Reward for episode 0 is 2.2999999999999736
Total Reward for episode 1 is 1.06
Total Reward for episode 2 is 1.599999999999988
Total Reward for episode 3 is 0.5700000000000012
Total Reward for episode 4 is 1.6899999999999866
```

Training High Score: 257
Testing High Score: 94

Analysis and Conclusions

Optimizing preprocessing showed that input image gray scaling and resizing gave a baseline frames per second (FPS) of 3. The main bottleneck was determined to be the game input, which was optimized through different method calls for less delay, resulting in a 3x increase of FPS, upwards of 10 FPS. Further preprocessing optimizations proved to result in too much FPS, which led to the agent spamming unnecessary actions which would have needed medium refactoring of code to tame. It was concluded that image preprocessing and somewhat minimal game input delay was sufficient.

The resulting reward shape is somewhat inconclusive of its impact on the model training. This is because of the sparse rewards used for every 20 time steps after the initial 60 time steps. These sparse rewards were ultimately assumptions of when the dino would clear an obstacle. In a sense, the sparse rewards mimicked the role of the discount factor of the reward function by favoring future rewards over immediate rewards. In the future I would like to test the discount factor, as well as implement game object recognition, such that it can explicitly define when to reward the dinosaur for clearing an object.

For learning rates, .0001 is likely the best option to use for training with this model. It had the highest mean of 4.1 total reward and a high score in testing of 266. Even though it showed more variance compared to .001, it could be considered neglected as the model almost always fails at the first obstacle on the first test run. This is possibly just an issue of the replay buffer not rendering fully at first or just a difference in time steps from initial start up to the restarted environment.

Exploration rate showed a unique relationship between how much is initially explored, how much is explored towards the end, and how much of the training episodes are actually explorable. Generally speaking, lowering the exploration towards the end of training did not have a direct correlation with the score. When just this value was lowered, the results were similar to the control (learning rate=0.0001), and when it was tested elsewhere, other factors had more effect. That being said, if both probabilities of exploration are lowered while maintaining a small fraction of explorable episodes, then training produces worse results. The model was underfitted and did not learn how to actually jump an obstacle. What seemed to have a drastic effect on training was actually an inverse relationship between initial exploration and the fraction of explorable episodes. Increasing the fraction of explorable episodes while lowering the probability of initial observations showed that the model was able to explore more while being cautious (not spamming the jump button). This led it to making more reasonable decisions and boosted both training and testing scores by ~2x (302 in training and 435 in testing).

Gamma, or discount factor, showed to impact the way the model went about training. Generally speaking, favoring long-term rewards is ideal within the context of this project, because the goal is to survive as long as possible. Turning gamma down to 0.5 made the model favor immediate rewards. During training, this showed the model to initially spam jump. It also learned very quickly how to get over the first obstacle consistently. Over the course of training, however, the favoritism towards immediate rewards showed a logical clash with the reward shape. Since I added a small penalty to jumping to discourage spamming, the model eventually began interpreting inaction as the best action. This resulted in the model favoring to not jump an obstacle due to its perspective on immediate penalties. Thus, despite the training looking good for a low gamma, it overfitted and on average performs poorly. Overall, .99 seems to be a good value for reinforcement learners that are trying to maximize the environment life cycle.

To conclude, the project gave valuable insight into how Q-learning builds the foundation of reinforcement learning, as well as how DQNs are actually implemented and it gave exposure to the key building blocks of an environment. In addition, preprocessing and learning rate proved to be influential within a simple environment like this where there are few actions to take. Other hyperparameters did not have as much impact, but it can be further investigated how their importance fluctuates within more complex environments. I hope to explore more reinforcement learning in the near future.