Resource-Restricted Deep Reinforcement Learning

Brent DeVries, Ryan Friberg
University of Chicago, CMSC 25500 Final Project
December 11th, 2021

1 INTRODUCTION & PROJECT OBJECTIVE

The primary objective of this project was to build a resource-restricted Deep-Q Network for deep reinforcement learning. We experiment with how to build a model that is capable of generalizing across multiple related but distinct tasks, while observing significant restraints on computation and memory resources. We define our tasks to be three retro video games, each with their own particular environment and objectives, and we use the specific context of each game to define the rewards that drive reinforcement learning. In this paper, we will outline the methods and procedures by which we built our resource-constrained, generalizable deep reinforcement network, and we will reflect on its success.

2 METHODS & MODELS

Agent

In this paper, we apply Q-learning to solve reinforcement tasks in video game playing. We use a deep neural network to model the Q-value function, representing the expected cumulative reward for a given state-action pair, and we use the frames of each game as our network input. Our neural network employs three convolutional layers with ReLU activations to map game states into feature space. The state's feature representation is then mapped into action space via a pair of fully-connected linear layers in order to identify the action the learned Q-value function expects to yield the greatest cumulative rewards based on the given state.

Experience Replay: We further augment our deep Q-learning agent using Experience Replay in order to avoid passing our agent highly dependent, consecutive game states and thus improving the quality and efficiency of learning. On each parameter update, we randomly sample a batch of 8-32 transitions from the DQN agent's replay memory buffer, and we use the loss on these randomly sampled batches to update the DQN parameters. For each application, we employed a memory replay buffer that stored either 10 or 30 thousand previously observed transitions.

Loss: We define our loss using the equation:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s,a;\theta_i))^2]$$

where y_i represents the target Q-value under the optimal function Q^* and is given by

$$y_i = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$$

We represent y_i using a latent target DQN that is identical to the agent network in structure, and we define the agent's loss relative to the output of the target network. In particular, we compare the output from the agent network and the target network using Huber (Smooth L1 Loss), which is then used to update the network parameters.

Optimizer: We use an Adam optimizer to perform updates on our DQN. As opposed to performing parameter updates on each step the agent takes, parameters are learned once every three steps for the sake of computational efficiency. Rather than updating the target network on the regular update steps, we delay updates to the target network by synchronizing it with the agent network at a frequency defined by some number of steps taken by the agent (usually between 1,000 and 5,000 steps).

Epsilon Decay: We utilized exponential epsilon decay to adjust our agent's exploration rate of new, random actions over our training iterations, i.e. at each step, epsilon is updated according to the formula:

$$\varepsilon_{i+1} = \varepsilon_i * \alpha$$

where $\alpha \in [0,1]$ represents the decay rate.

Data Preprocessing

We employ a series of data preprocessing methods on the game frames input to our DQN with the goal of improving computational efficiency.

Frame Skipping: For each application, we define a frame skip of 4, meaning only one of every four game frames is input to our model.

Grayscale Conversion: We map each RGB frame to a graysale frame, reducing the number of input channels for each frame from 3 to 1.

Cropping: We crop each image from its initial spatial extent to an 84 by 84 pixel spatial extent.

Frame stacking: For each of our applications, we define a frame stack of 4, passing 4 frames as input into our network at a time.

3 EXPERIMENTAL PROCEDURES

During our study, we applied our model to different situations with various parameters and hyperparameters. This section explains both the changes and environments we chose as well as the reasoning behind them.

Emulated Games

We trained our DQN over three different games with fundamentally different designs, gameplay, objectives and platforms. Specifically, we trained on Frogger for the Atari 2600, the original Super Mario Bros. for the Nintendo Entertainment System (NES), and the mobile game Flappy Bird. As mentioned before, we selected these games as they covered a fairly diverse set of features including graphics (both in terms of color pallets and resolutions) and action systems in addition to their

simplicity which limits the runtime of the DQN training compared to more complex games. Another requirement we had was that our selected games needed to have both an in-game score and relatively fast-paced game sessions. The related, yet distinct natures of these tasks yields unique insight into the behavior and generalization of the DQN.

Reward Systems

The games we used each had unique reward and scoring systems. For example, in Frogger, the network was rewarded for the further up the screen its character traveled. In addition to having positive reinforcement (the in-game score), all of our games also had a form of negative reinforcement in the form of player death. In our study, we tried both substantially boosting the reward for increasing the in-game score, but also harshly penalizing player death. Somewhat surprisingly to us, our results of this experiment revealed that this approach was not always beneficial for training. One demonstration was when our model was training on Flappy Bird, the system would get adjusted over a new range of rewards. Even though the reward was very very low, the model would just find its actions within this new lower range and thus not affect the training significantly. As such, we elected to have more normal range for incentives and punishments in our training.

Hyperparameters

In our efforts to reduce training costs without too much of an expense to the accuracy of the model, we experimented varying several values that our model was dependent on.

Frame skip: We tried a wide range of frame skip values based on an internal emulated frame rate of 24-60 frames per second. Training on a frame skip of less than 4 significantly increased the total time to train the model but provided a noticeable increase in average reward per episode over many episodes. Training with frame skips of 6, 10, 20, or even 30 would unsurprisingly drastically decrease the runtime but unfortunately, even after several tens of thousands of episodes, there would be no upward trend in reward. Through trial and error, we determined a frame skip of 4 provided the best balance between quality learning and resource constraints.

Experience Buffer Size: At the beginning of our study, a severe bottleneck to our training was total memory usage. Starting with a buffer size of greater than 100k, we would quickly run out of VRAM after only roughly 40-60k steps, which is far too few to achieve substantial learning. To counteract this, we used a range of buffer sizes between 10k and 30k, which allowed us to stay within our hardware's limits. (Similarly we tried varying the batch size between 8 and 32 with mixed results on improved RAM usage.) Without the capability to train with a greater memory capacity, it is difficult to determine how the truncated memory may have impacted the performance of the DQN.

Epsilon: We observed that the optimal epsilon value varied based on the particular application of our agent. For Super Mario Bros and Frogger, a standard epsilon value of 1.0 sufficed. However, Flappy Bird's gameplay proved to be a unique case with respect to the epsilon value. While taking a random action in a game like Mario is unlikely to have significant consequences, it is far more likely to cause an episode to be terminated in Flappy Bird due to the precision necessitated to be successful in the game. Therefore, even if a majority of actions are determined by the agent's DQN, occasional random actions may be detrimental to game play and prevent the agent's success. Therefore, the epsilon value needed to be dropped. With a high epsilon, even within the first couple episodes of the training loop, the in-game bird would immediately rapidly repeat the flap button to gain as much altitude as possible and thus end the game on the upper part of the first pipe or plummet to the ground at some point before the first pipe. A lower epsilon of 0.1 allowed the network to drive less aggressive action, allowing the agent to find more success.

Decay-rate: We observed that the epsilon value was extremely sensitive to the value of the decay rate where changes on the order of magnitude of less than 0.00001 would have significant effects on the epsilon only after a handful of episodes. Because of this behavior, we largely left the decay rate at values extremely close to 1.0, only changing it slightly in specific situations. One such situation was in Flappy Bird with the intention of having the network eventually decide nearly all of its actions as opposed to relying on a higher percentage of random inputs (for similar reasons as described in the section on the epsilon value above).

Action Space

Each of our games' emulation environments had a different input space but we surmised that not all of each game's available actions were beneficial to increasing the score and in fact, some were completely useless or even detrimental to the reward. To counteract this, while increasing the speed of training at the same time, we decided to limit the scope of actions that the network had in certain games (Super Mario Bros and Frogger). Super Mario Bros can be played with only two actions, running to the right and jumping to the right. Stopping Mario, moving left, jumping straight up, and jumping left all were not necessary to play the game. With these two actions, we were still able to see Mario occasionally complete the first level of the game. Frogger's emulation environment had access to all 18 of the possible Atari inputs when many of them did nothing in game so instead we greatly restricted the possible actions. One of the restrictions we contemplated was to only allow the frog to move forward or perform no operation, however, we wanted the model to be able to get to the other side of every time. Unlike in Super Mario Bros, there may be situations where such a heavily restricted input space would be unwinnable so in the end, we allowed the model to move in all four directions in addition to no operation and were able to see fairly continuous progress in the training of the model.

4 RESULTS

Frogger

Our model was trained on roughly 30 thousand episodes and with an overall number of steps between 2 and 3 million. Its ending epsilon value was just over 0.488 and it was approaching an average reward of 15 (after starting with a reward around 4). Comparatively, a model that randomly

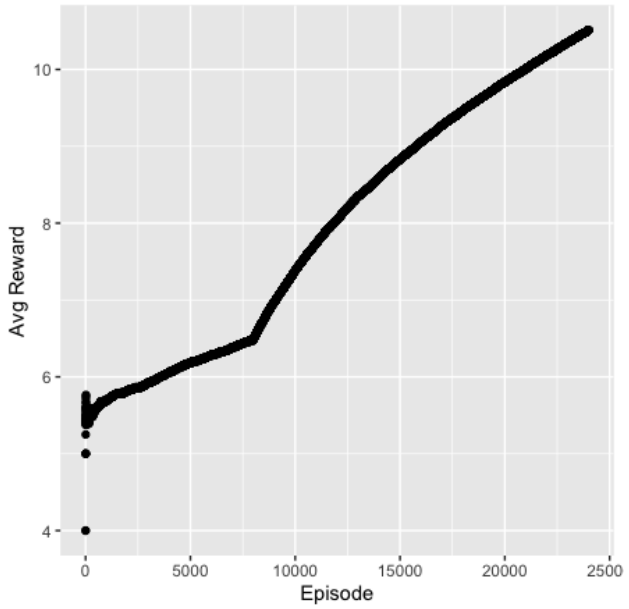

Figure 1. ***FROGGER - Reward vs Episodes***
(Please note that there is a a jump in the values due to an unfortunate batch of data being overwritten)

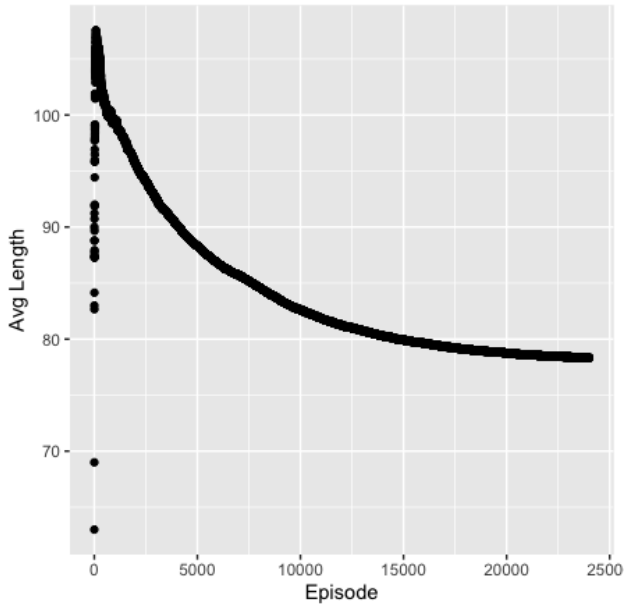

Figure 2. ***FROGGER - Reward vs Length***

inputs data, tested over 50 episodes consistently had a reward of 0, with a very rare instance of a reward of 1. As referenced in Figure 1, our DQN saw a steady, noticeable increase in reward over the time of testing and as such, was our most successful of the three games. Figure 2 demonstrates that the average length of each episode decreases with time, implying that each action it takes becomes more meaningful.

With the rendered videos, the model can be seen regularly crossing the road section of the game with relative ease, but when the game changes to the river section the model is still figuring out what to do. This is likely because the frog must only advance to the moving objects when it had to strictly avoid moving objects in the prior section. However, as it does sometimes advance to the logs, it is still learning and with more training, it will likely be able to further increase its score and eventually get to the other side.

Flappy Bird

Flappy Bird ended up posing an unusual challenge for our model. After 50,000 episodes and around 1.5 million steps, ending with an epsilon value of just under 0.0731 (starting from 0.1), the model's reward rarely reached over the value of 101 (or 21 in our second training environment) or the in-game equivalent of passing through the first pipe. Compared to the random baseline, which never exceeded a reward of 5, our model still clearly beats out random actions, but the design of the game ended up being a severe and immediate bottle-neck. In the rendered video, our model sends the bird high up in y-direction and then attempts to fall back down to the opening between the first set of pipes. After the agent dies repeatedly early in training as a result of hitting the floor in the empty space before the first pipe, the model likely eventually believes that the only harmful thing in the environment is the floor. To counteract this, it logically tries to move its bird as far away from that danger as possible and realizes that it needs to correct itself too late. However, because it falls back down at all, and it roughly lines up with the first opening, we believe that the model will eventually overcome this hurdle with more training. This behavior is different from when we ran the model with a high epsilon value because in that situation, the bird would immediate be sent upwards only after a very small number of episodes and never think to fall back down, whereas in this case, the bird fell to the ground many times and eventually learned to position itself higher up and then it does in fact try to correct itself when it needs to.

One method we attempted to use to mitigate this issue was to hard-code in the actions of the bird for the first handful of steps to get it closer to the pipes before handing over control to our DQN. The idea would be to not give the agent enough time to move the bird up too high. In setting up this second training environment, we did observe a very slight increase in reward over time but because there were now far fewer steps per episode, the learning would still need substantially more time for more drastic improvement. Another method we would look further into would be to add a punishment for a higher y-value to directly combat the problematic behavior we observed.

Both of these training sessions saw an increasing reward with increasing episodes but the difference was the shape of the curve. Figure 3 shows how the original model learns fairly quickly how to get to the x position of the first set of pipes and begins to reach a horizontal asymptote in reward whereas Figure 4 displays how the model with the head start sees a steadier but slower increase in reward. As expected, as the number of steps in-game is directly proportional to

the reward, the plots of the lengths per episode have almost exactly the same shape as the reward.

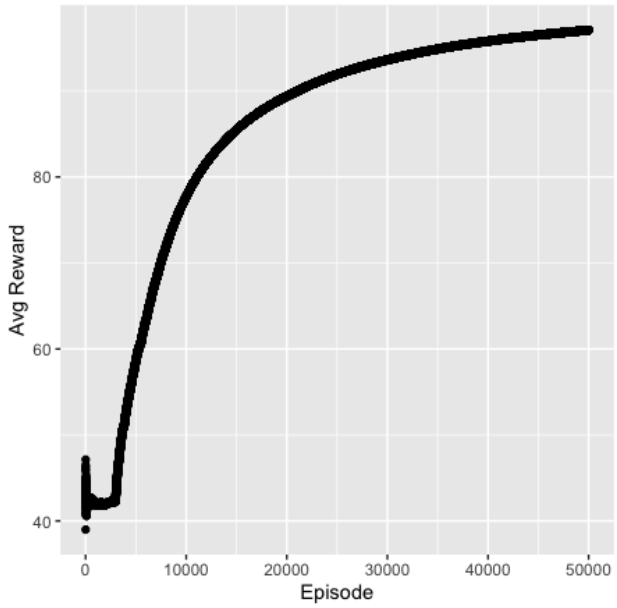

Figure 3. ***FLAPPY BIRD (VERSION 1) - Reward vs Episodes***
This is model trained to start from the beginning of the game

Super Mario Bros

We trained our DQN agent for 8,000 episodes on the first level of world 1 in the original NES game of Super Mario Bros with a final epsilon value of approximately 0.357, and we restricted our agent's action space to include only forward movement and forward jumping. Training time was a significantly inhibiting factor for this application of our model, and more substantial training would likely require a significant increase in computational and memory resources. Nonetheless, we observed a modest improvement in rewards over the course of the training episodes and relative to the random baseline. The fully trained agent achieved an average score of 658.88 with an average of 181.2 steps per episode over 50 episodes, while a random choice of actions at each step was able to achieve an average score of 649.74 on 202.82 steps per episode over 50 episodes. Therefore, our trained agent outperformed the random baseline by a rather modest margin of 9.14 points per episode. However, it is important to consider that Mario is relatively forgiving to a random choice of actions relative to the other applications, i.e. random choices of actions are less likely to terminate an episode or have serious consequences. Furthermore, restricting the action space to exclusively forward movement greatly improved the performance of the random baseline, while simultaneously presenting challenges to our trained agent by preventing backward movement to collect on opportunities that may have boosted the agent's score.

Referring to Figure 5, it is evident that after a period of unstable rewards, our trained agent displayed a consistent gradual increase in rewards acquired per episode, indicating that further training would be likely to yield substantial improvement. Furthermore, Figure 6 indicates that while average rewards increased steadily throughout training, the average number

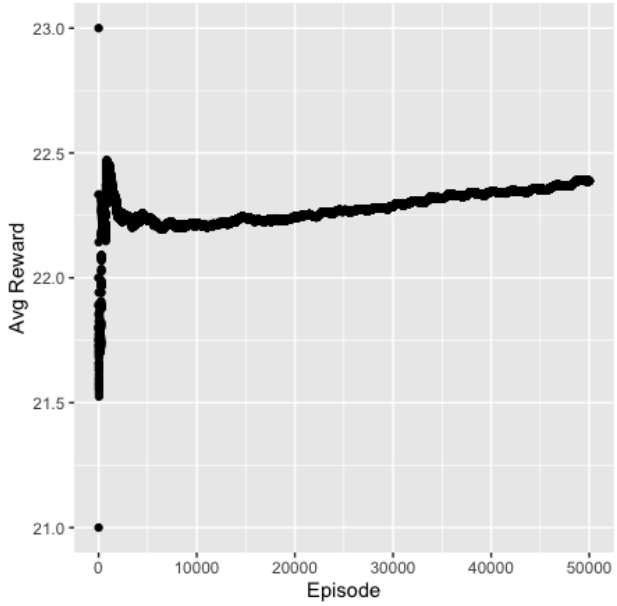

Figure 4. ***FLAPPY BIRD (VERSION 2) - Reward vs Episodes***
This is model trained to start closer to the pipes

of steps per episode gradually decreased. This suggests that while the trained agent displayed only modest improvements in rewards relative to the random baseline, it displayed more significant improvements in regard to the rewards acquired per step taken by the agent, reflecting greater efficiency relative to the random baseline. This same phenomenon was observed in our training for Frogger.

In a more qualitative assessment, the rendered video of our trained agent's game play sheds some light on the behaviors learned through training. The agent displays a couple of promising learned behaviors, such as jumping to acquire points for hitting bricks and jumping to acquire points for killing adversaries such as goombas and koopas. It is also clear from the rendering that the limited action space inhibits the agent from capitalizing on frequent point opportunities that would require waiting or moving backward. Finally, it is evident that the agent is less effective in navigating the episode past a certain point due to limited experiences, which further highlights the value of more extensive training.

Results Discussion

One trend that we noticed in our results from Mario and Frogger was that, while the model does fairly well at improving on the cumulative reward per episode in both cases, it seems to be significantly better at improving the reward per step taken. This is reflected in the fact that the average reward per episode consistently increased while the average number of steps per episode decreased. This may not come as a surprise as it it easier to predict from a single step to the next as opposed to the actions over an entire episode, but it is a promising finding nonetheless. Somewhat counter-intuitively, however, as shown in Figure 7, Flappy Bird's average length increases as the model starts traveling further through the game and this is still a good thing. The game autoscrolls, meaning the

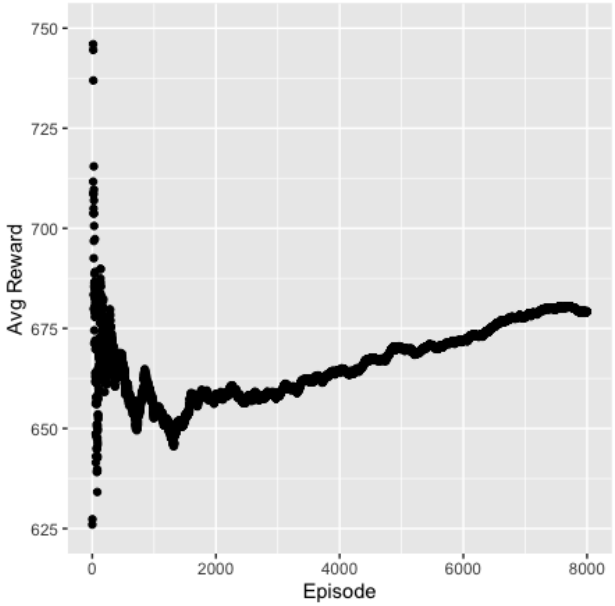

Figure 5. ***MARIO - Reward vs Episodes***

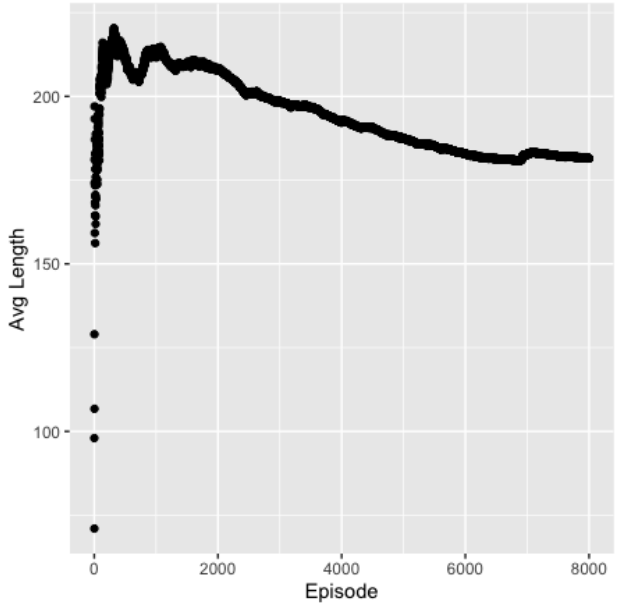

Figure 6. ***MARIO - Length vs Episodes***

more steps you are in the game the better. With Frogger, for example, the player can move up and down and side to side, some of these actions are bad and some good depending on the surrounding context, thus seeing a decrease in length is a net-positive if it means the model is determining the more efficient paths. Flappy Bird has no such route-planning and the player has no choice but to constantly move forward, making larger lengths always indicative of learning.

Through the application of our model to multiple tasks, it is clear that certain situations and obstacles create bigger challenges for the agent than others, even if they may not seem particularly significant to a human player. In each of our games, we observed that the introduction of an unfamiliar environment (e.g. the water in Frogger, the first pipe in Flappy Bird, and the later parts of the level in Mario) can make it very difficult for the agent to adjust, especially if they have spent a substantial amount of time learning one environment in the earlier part of the game. Nonetheless, our agent certainly displayed the ability to generalize. In each task, the agent showed signs of significant learning, although learning occurred more smoothly in some situations than others. This gives us reason to believe that given more resources and training time, the model would perform successfully across each of these tasks.

Note: You can find the rendered videos of our trained agents for each game in the appendix.

5 FUTURE STUDY

There are multiple areas for future study that we would like to explore given access to stronger hardware. These include the effect on both runtime and performance of several parameters. Some examples testing and comparing grayscale and full-RGB inputs, non-cropped images and croppings of various sizes, and many more values for our hyperparameters,

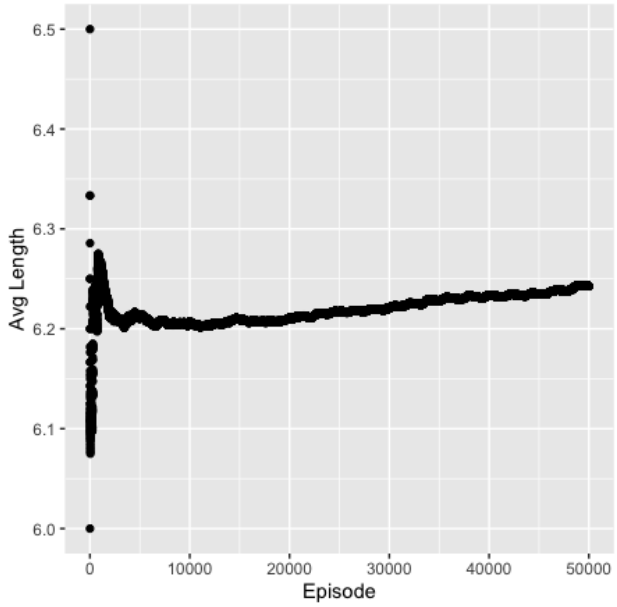

Figure 7. ***FLAPPY BIRD (VERSION 2) - Length vs Episodes***
This is model trained to start closer to the pipes

including experimenting with variable decay and learning rates. Additionally, our research of other studies suggested an ideal number of training steps to be excess of 40 million, and we would love to train and evaluate our model on a number of steps of that magnitude.

Perhaps the most interesting area for further study would be in regards to the incidence of catastrophic forgetting or interference. We mentioned in the experimental procedures section that our memory constraints required that we truncate our agent's replay memory size from 100 thousand transitions

to a number between 10 and 30 thousand transitions, a very significant decrease. The concern is that, even if we were to train our agent for an arbitrarily large amount of time, the limited size of the memory buffer may cause the agent to completely lose previously learned information over time, deeming further training drastically less effective. Catastrophic forgetting is an obstacle that has the ability to prevent the development of a successful model, even in the absence of restraints on computational resources. Thus, a systematic experiment and study on the occurrence of catastrophic forgetting relative to buffer size would be of great interest.

Finally, a further area of interest would be comparing the performance of a Q-learning approach to reinforcement learning with algorithms that may be more sophisticated or founded upon a different paradigm. For instance, Actor-Critic algorithms leverage both policy gradients and Q-learning, so it would be very interesting to study how much that added complexity improves model performance. Another point of comparison would be with proximal policy optimization (PPO), which is the state of the art policy gradient-based approach to reinforcement learning. It would be interesting to systematically compare these approaches to a DQN with respect to results, resource requirements, and generalizability across tasks.

6 APPENDIX

All appendix materials are contained in the zip file included with this submission. For each game, these materials include our Jupyter notebooks (ipynb and pdf), CSV's containing all data, and videos of our trained agents in game play.

7 REFERENCES

[1] *Playing Atari with Deep Reinforcement Learning* (Mnih et al., 2013)
https://arxiv.org/pdf/1312.5602.pdf
[2] OpenAI Gym: Documentation
https://gym.openai.com/docs/
[3] OpenAI Gym: Environments
https://gym.openai.com/envs/atari
[4] OpenAI Gym
https://github.com/openai/gym
[5] Flappy Bird Gym
https://github.com/Talendar/flappy-bird-gym
[6] Super Mario Bros Gym
https://github.com/Kautenja/gym-super-mario-bros
[7] PyTorch Documentation
https://pytorch.org/docs/stable/index.html
[8] *Overcoming catastrophic forgetting in neural networks* (James Kirkpatrick et al., 2017)
https://www.pnas.org/content/114/13/3521