Study on Deep Q-Learning

CS 6140 Machine Learning Spring 2018 Final Project
Brittany Haffner, Bryan Fuh, Atharva Jakkanwar, Apar Singhal

Abstract

The goal of this project was to study model-free reinforcement learning, with emphasis on deep Q-learning. This was accomplished by training an agent to learn a control policy in a single-input environment, specifically the game Flappy Bird. By completing this exercise, we were able to successfully apply theoretical concepts to a practical application in a way that allows us to expand to other environments.

Introduction

Training artificially intelligent agents to successfully play video games is a great way to demonstrate the utility and power of reinforcement learning. There are many facets of reinforcement learning, so this report will be focusing on a very specialized case of reinforcement learning - deep Q-learning. We chose this particular case because of its model-free nature; the same algorithm can be used to train an agent to play several different video games, despite differing objectives, rules/physics, and number of inputs.

Deep Q-learning is a form of reinforcement learning that utilizes q-learning, convolutional neural networks (CNNs). It is best used for big environments with many different potential states, which is why it is a popular algorithm to use for training agents to play video games. The concept of deep Q-learning was first introduced in the paper “Playing Atari with Deep Reinforcement Learning”, where the researchers introduced the notion of using CNNs to learn successful control policies in complex reinforcement learning environments [1]. The idea is that the complex environment can be simplified by means of convolution, while the use of experience replay alleviates challenges posed by highly-correlated states usually present in reinforcement learning.

Once we formally define deep Q-learning, we apply the methodology to training an agent to play the single-input game Flappy Bird. The original code used to train the agent can be found here: https://github.com/yenchenlin/DeepLearningFlappyBird. We chose to use this existing code for several reasons, including that it demonstrated all of the concepts we are studying here and allowed us to further our understanding of how deep Q-learning works in practice. We experimented by tuning hyperparameters and recording the effects after several hours of training.

Defining Deep Q-Learning

Reinforcement Learning

Reinforcement learning is a unique form of machine learning that is neither supervised nor unsupervised. It does not require training data in any form, so it relies on the exploration and exploitation of the environment it interacts with to learn the optimal solution or policy. The environment provides the agent with rewards, which the agent then uses to continually update the policy. [2][3]

Markov Decision Process

The Markov decision process (MDP) is the framework used to make decisions on a stochastic environment. It is more useful than simple planning because it gives an optimal action, even if something goes wrong. Whereas simple planning does not allow for mistakes or errors and the plan is followed *after* finding the best strategy. [4]

The major driver of this process is the Markovian property, in which the effects of an action taken in a state depend only on that state and not on prior history [5]. This allows for the use of a discrete time stochastic control process that leverages rewards to determine the next actions. This can be put together in a transition model as follows:

Equation 1: $T(s, a, s') \sim Pr(s'|s, a)$

where *s* is the current state, *s'* is the next state, *Pr(s'|s,a)* is the probability of reaching the desired state *s'*, and *a* is an action in the set of all possible actions *A(s)*. The goal of MDP is to find the optimal policy *π** that determines the best next action to take in that current state based on the maximum amount of reward, *R(s),* gained in the long term.

There are two ways of implementing MDP: infinite horizon and finite horizon. For infinite horizon, the agent can move an infinite number of steps and still find a viable solution (i.e., win the game). In finite horizon, there is a finite number of time steps an agent can take before losing or ending the game. This changes the policy to sometimes take riskier paths to the solution because of the need to finish before time steps run out. For example, let's say there is a game environment in the form of a finite grid where the actions are up, down, left, and right. The agent starts in one corner of the grid and has to move to the "goal" tile without accidentally moving to the "lose" tile. If there is an 80% probability that the agent will move in the desired direction, an infinite horizon MDP agent would take the path that best avoids the "lose" tile, which could be the longest path to the goal. However, if we assume that the number of allowed steps is much less than the longest path, the finite horizon agent will lose the game by taking the longest path and must assume some risk of passing close to the "lose" tile in order to find a solution.

For each state in the environment, there is an associated utility of sequences such that

Equation 2: $U(s_0, s_1, s_2, \ldots) = \sum_t \gamma^t R(s_t)\,, 0 \leq \gamma \leq 1$

Where γ is the discount factor. The utility of sequences allows the agent to go infinite distance in a finite amount of time. This concept is applied in the Bellman equations to solve for the optimal policy in a given environment.

Bellman Equations

The key to solving MDP in reinforcement learning are the Bellman equations. This is the fundamental set of recursive equations that define the true value of a particular state and determine the best action to take. As mentioned in the previous section, we want to find the optimal policy *π** to determine the best next action *a* to take in state *s* given the transition function and a utility value as defined in previous equations. The optimal policy over the course of timesteps can be defined as follows:

Equation 3: $\pi^* = argmax_{\pi}\mathbb{E}[\sum_t \gamma^t R(s_t)|\pi] \rightarrow \pi^*(s) = argmax_a \sum_{s'} T(s,a,s')U(s')$

To further understand this concept, we must expand about the utility score (value), which is the expected maximum cumulative reward by following the policy given by state *s.* The utility score can be defined as

Equation 4: $U(s) = \alpha[R(s,a) + \gamma max_a \sum_{s'} T(s,a,s')U(s')], 0 < \alpha < 1$

where *α* is the learning rate. The optimal policy can then be rewritten as

Equation 5: $\pi^*(s) = argmax_a U(s,a)$ [6]

which allows the utility score to be rewritten in terms of the optimal policy:

Equation 6: $U(s) = R(s) + \gamma\pi^*(s)$

Using these relations, we can compute the expected maximum cumulative reward from following the current policy in state *s* and taking action *a*. This is known as the quality score or Q-value and can be computed as follows:

Equation 7: $Q(s,a) = \alpha[R(s,a) + \gamma max_{a'} \sum_{s'} T(s,a,s')Q(s',a')]$

The Q-value can be calculated recursively such that

Equation 8: $Q(s',a') \leftarrow Q(s,a) + \alpha[R(s,a) + \gamma max_{a'} Q(s',a') - Q(s,a)]$

These Q-values are then used to find the policies at each state. The process for finding policies is to define π_0 as the initial guess. Then, the policy is evaluated and used to calculate the corresponding Q-value. Then, using the transition function, the policy is improved upon as follows:

Equation 9: $\pi_{t+1} = argmax_a \sum_t T(s,a,s')Q_t(s',a')$

This leaves us with the final policy equation that will be used in Q-learning:

Equation 10: $Q_t = \alpha[R(s) + \gamma \sum_{s'} T(s,\pi_t(t),s')Q_t(s')]$

Q -Learning

Q-learning is a branch of reinforcement learning that uses Q-values to select the best actions to take. The general idea is that there is a table that holds the maximum expected future reward for each state (rows of the table) and action (columns of the table), which is known as the Q-table. The Q-table is updated throughout training by measuring the reward generated from each action. If the Q-value generated is greater that the value already in that cell of the table (assuming Q-table is initialized as a zero matrix), then that is the new maximum expected reward and the table will be updated.

The major drawback of Q-learning is that the Q-table can grow exponentially in large environments, so it is ill-equipped to process games such as Flappy Bird, where snapshots of the environment are the inputs to the algorithm. To mitigate this issue using deep Q-learning, the Q-table is replaced with a deep neural network that intakes the current state and returns the Q-value associated with each possible action. The action with the highest Q-value is then chosen to update the policy.

Convolutional Neural Networks

By replacing the Q-table with a convolutional neural network (CNN), we are able to apply the basic concept of Q-learning in a large image-based environment, such as most computer games. Figure 1 below shows an example of a CNN in the context of Flappy Bird.

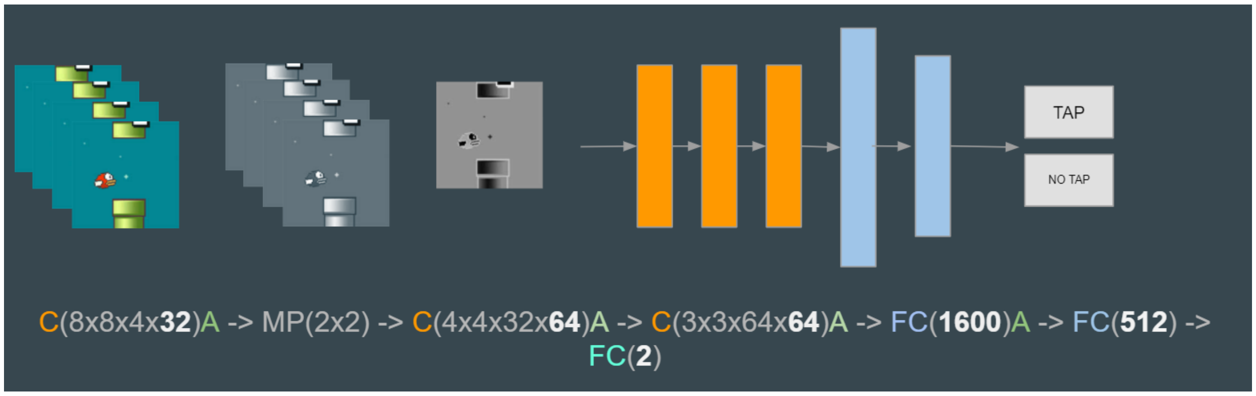

TAP
NO TAP
C(8x8x4x32)A -> MP(2x2) -> C(4x4x32x64)A -> C(3x3x64x64)A -> FC(1600)A -> FC(512) -> FC(2)

Figure 1. CNN for Flappy Bird

In order to create a successful CNN, pre-processing the image data is crucial. Tasks such as converting the image to gray-scale, shrinking the image size, and stacking frames (in our case, we've stacked 4 frames) reduces the amount of irrelevant data, allowing the CNN to run faster.

The concept of convolution takes advantage of the idea that an object is the same no matter where it appears in an image [7]. Instead of feeding entire images into the neural network, the image can be broken into equally sized overlapping image tiles. Each of these tiles is fed into a small neural network where the weights for every single tile are the same as in the original images; however, if there is something of interest in that tile, it is marked as such. The results are passed into a new array that keeps track of the arrangement of the original tiles. This smaller array records which sections of the original image that were most "interesting" and is then down-sampled using max-pooling to shrink the array further (keeping only the most interesting bit in a grid square). This final array is used as an input to another neural network that will make any further decisions, in our case deciding if the agent should "tap" or not.

Experience Replay

In reinforcement learning, we receive sequential samples from interactions with the environment, which are highly correlated and can lead to overfitting a solution. This is where the concept of experience replay comes to the rescue in training the deep Q-network. The use of experience replay allows the agent to avoid forgetting previous experiences while reducing the correlations between experiences. It also averages the distribution over many previous states, which essentially smooths out the learning process. All of this is accomplished by utilizing a replay buffer to update a batch of experiences.

The replay buffer stores *N* number of past events, which include the state, action, and reward observed by the agent in that environment. Once the buffer is full, the agent then randomly samples from the buffer and stores that sample in a batch of experiences which is then filled and passed to the deep Q-network for training. As more states are reached, the replay buffer acts as a queue that constantly replaces the oldest information with the newest, allowing for the continued updating to the batch of experiences. Randomly sampling from the replay buffer reduces correlations between experiences while still holding on to past information as an aid in learning the environment. [8]

Experimental Methods and Results

Experimental Methodology

The experimentation was done by tuning the Q-learning hyperparameters of the algorithm and calculating the resulting cumulative rewards and Q-values. There has already been work done to create a deep Q-learning agent that can play Flappy Bird, so our goal was to characterize the learning based on the effects of different hyperparameters. The algorithm used to train an agent to play Flappy Bird can be seen in Figure 2.

```
Algorithm:
Initialize environment $E$
Initialize replay memory $M$ with finite capacity $N$
Initialize the DQN weights $w$
for episode in max_episode:
  for steps in max_steps:
      $s$ = Environment state
      Choose action $a$ from state $s$ using probability $\epsilon$ greedily
      Take action $a$, get reward $R$ and next state $s'$
      Store experience $<s_t, a_t, r_t, s_t'>$ in $M$
      Get random minibatch of experience $<s_p, a_p, r_p, s_p'>$ from $M$
      Set optimal Q-value $Q(s_p)^* = r_p + \gamma max Q(s'_p)$, where $\gamma$ is discount value
      Update $Q$ by taking gradient descent step on $( Q(s_p)^* - Q(s_p) )^2$
```

Figure 2. Pseudocode for deep Q-learning algorithm

Mapping Deep Q-Learning Concepts to Flappy Bird

The goal of the deep Q-learning agent is to continually keep the bird alive while it is moving forward in the Flappy Bird environment. The basic architecture for this agent can be seen in Figure 3. The position of the bird in the environment as determined by the physics of that game constitutes the different states of the game. The actions are limited to what is essentially a Boolean input – either the agent “taps” the bird to make it fly up or doesn’t. The original configuration of the rewards was as follows:

1. +0.1 points every timestep the bird lives

2. +1 point for every pipe the bird crossed
3. -1 points every time the bird dies (terminal state)

The death of the bird signals the end of a single episode, and the game restarts. The agent, however, is constantly updating its weights and never stops playing the game. The rewards structure of the game was a hyperparameter that was tuned during experimentation. We experimented with different positive and negative reward values.

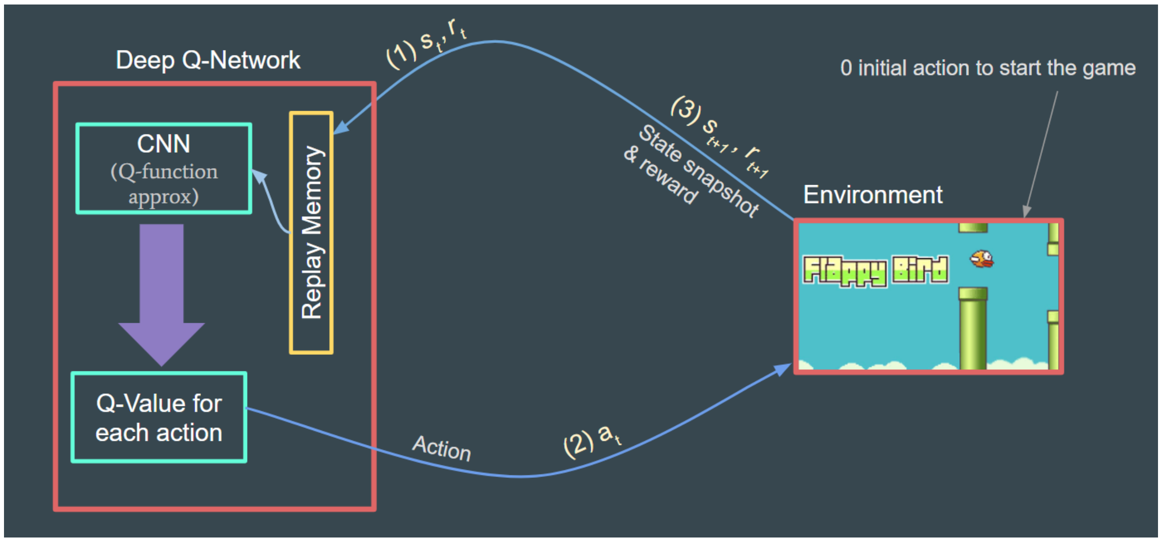
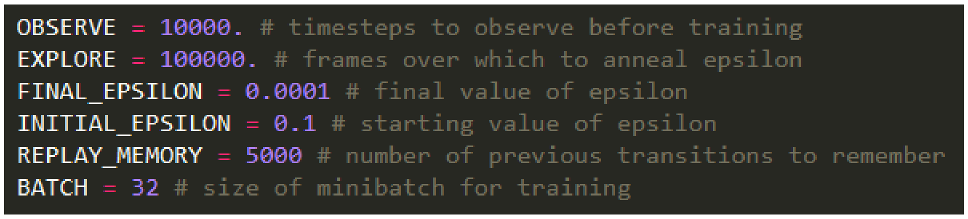


Figure 3. Architecture for Flappy Bird Deep Q-learning Agent

The hyperparameters that most affected the way the agent converges are shown in Figure 1. The *observe* parameter is the number of timesteps for which the agent does not update the Q-functions (i.e., the CNN weights). After the *observe* phase until the *explore* phase, the epsilon values are annealed or reduced from *initial epsilon* to *final epsilon*. The *replay memory* is the number of latest experiences that the learner stores for future use. Each time the agent trains, the *batch* number of experiences are picked up at random from the *replay memory* and fed to the CNN.



```
OBSERVE = 10000. # timesteps to observe before training
EXPLORE = 100000. # frames over which to anneal epsilon
FINAL_EPSILON = 0.0001 # final value of epsilon
INITIAL_EPSILON = 0.1 # starting value of epsilon
REPLAY_MEMORY = 5000 # number of previous transitions to remember
BATCH = 32 # size of minibatch for training
```

Figure 4. Hyperparameters for the Learning Agent

Results and Analysis

Figure 5 shows the data from a 20+ hour training session completed by our agent. There is clearly a gradual increase in both Q-values and rewards initially, but then the values start to level off near the end. When the training starts, the agent takes random actions by passing the input images through the CNN without backpropagating or updating the weights. Since the weights are not updated, there is no update in the Q-values as is evident in Figure 8. This is the *observation* phase of the learning part (Figure 6). The sudden spike in the Q-values shows that the learning has just begun, then it recedes when the agent starts making taking better decisions. This is the *exploitation* phase where the learner exploits the data stored in the replay memory. The overall trend shows a gradual increase in the Q-values as the agent completes more and more episodes.

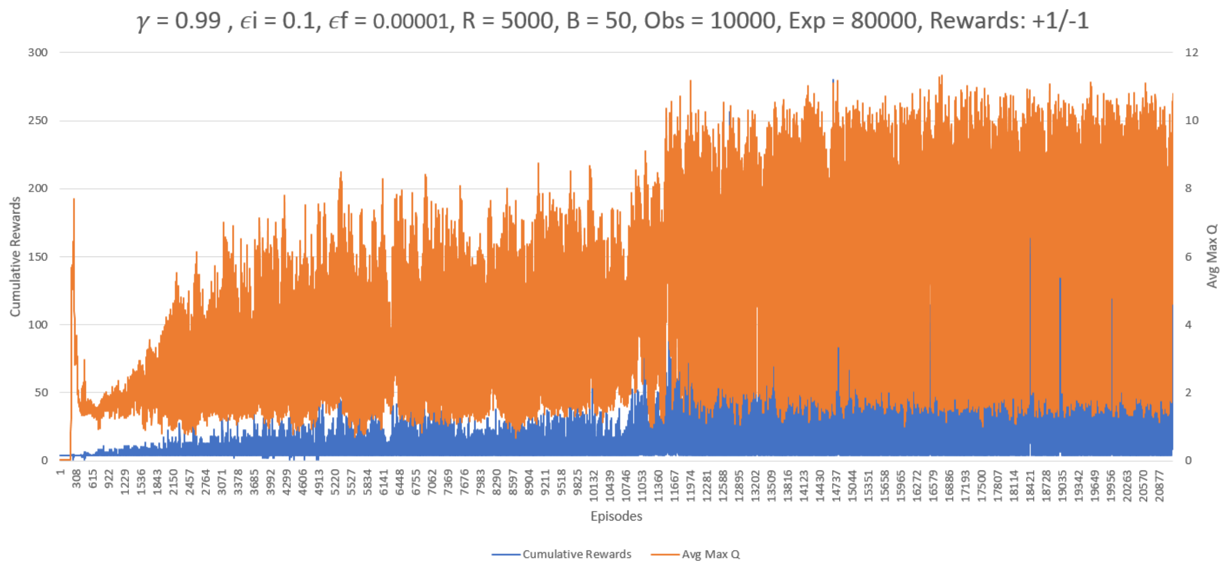


Figure 5. Average Maximum Q-values and Rewards vs. Episodes

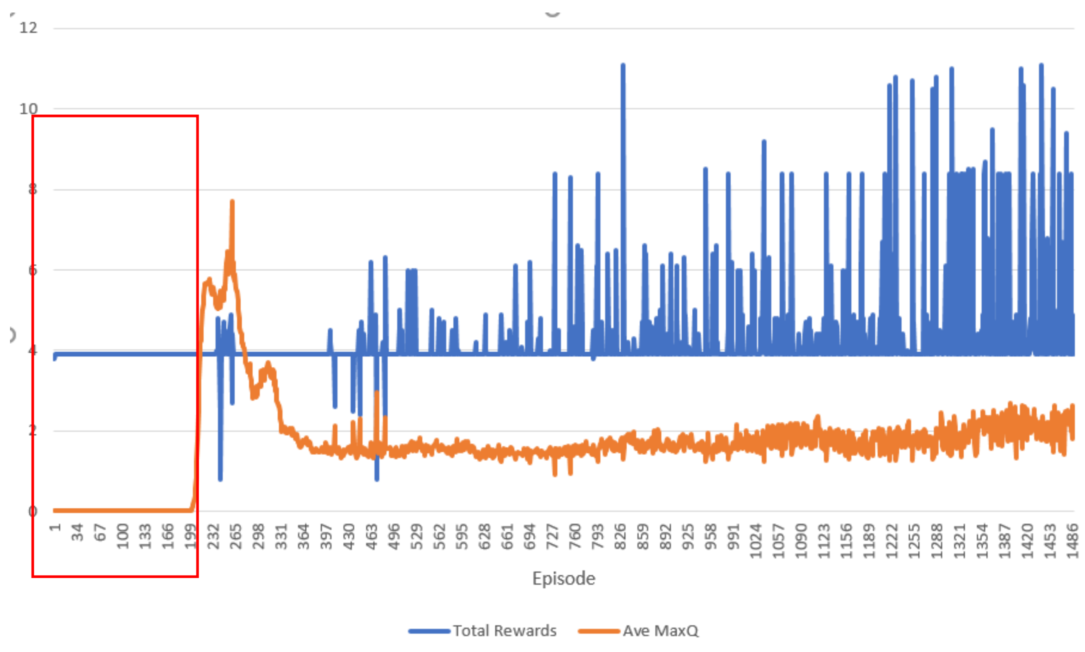


Figure 6. Initial training phase, with the observe phase marked in the bounding box

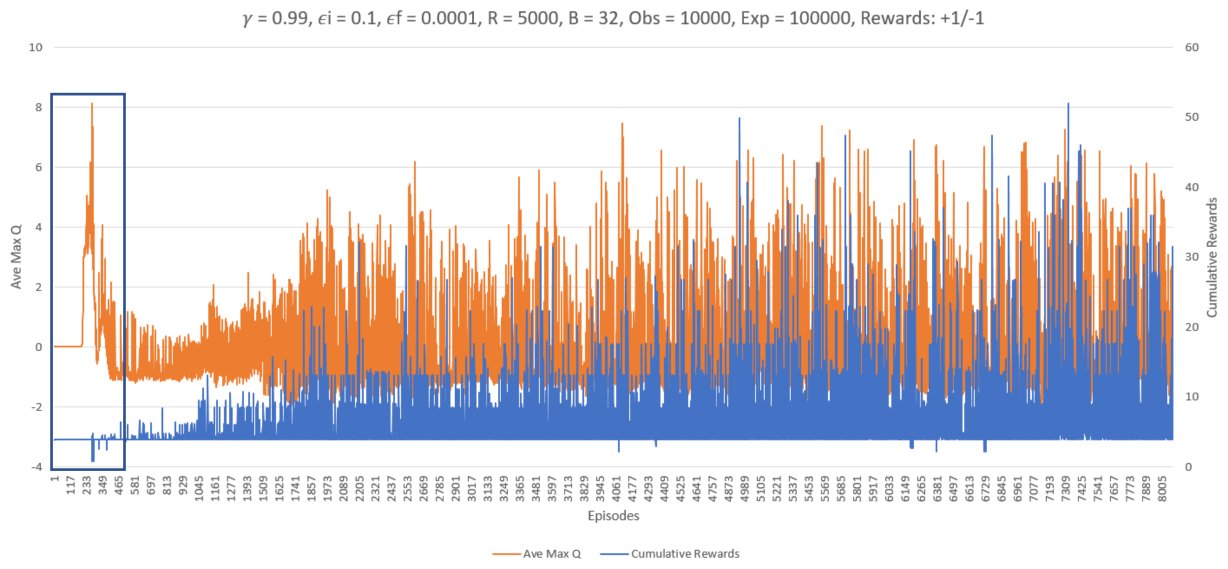


Figure 7. Average Maximum Q-values and Rewards vs. Episodes. The bounding box is zoomed in on in Figure 8

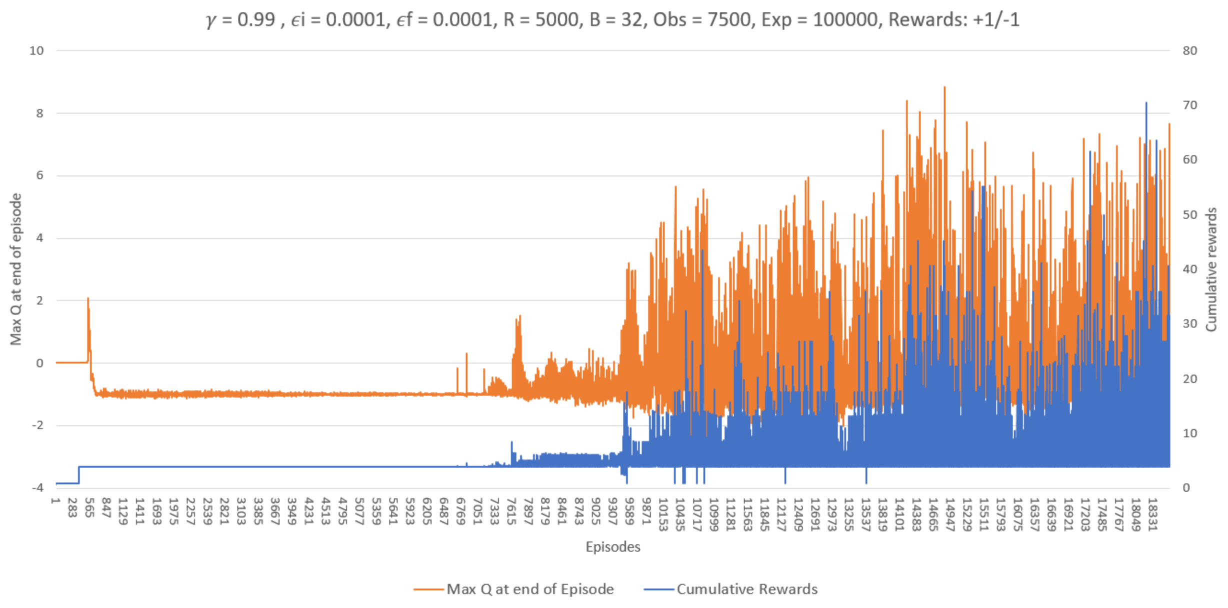


Figure 8. Maximum Q-values and Rewards vs. Episodes

To consider the effect of change in rewards, we tried experimenting with high positive rewards and high negative rewards. Figure 9 shows the plot of the number of pipes the bird crossed as time passed when the reward for crossing a pipe was +2 while the reward(penalty) for dying was -20. Figure 10Figure 9 shows the same with positive reward as +20 and negative as -2. Although the high negative case looks slightly better than the high positive case, the difference is too insignificant to draw any conclusions.

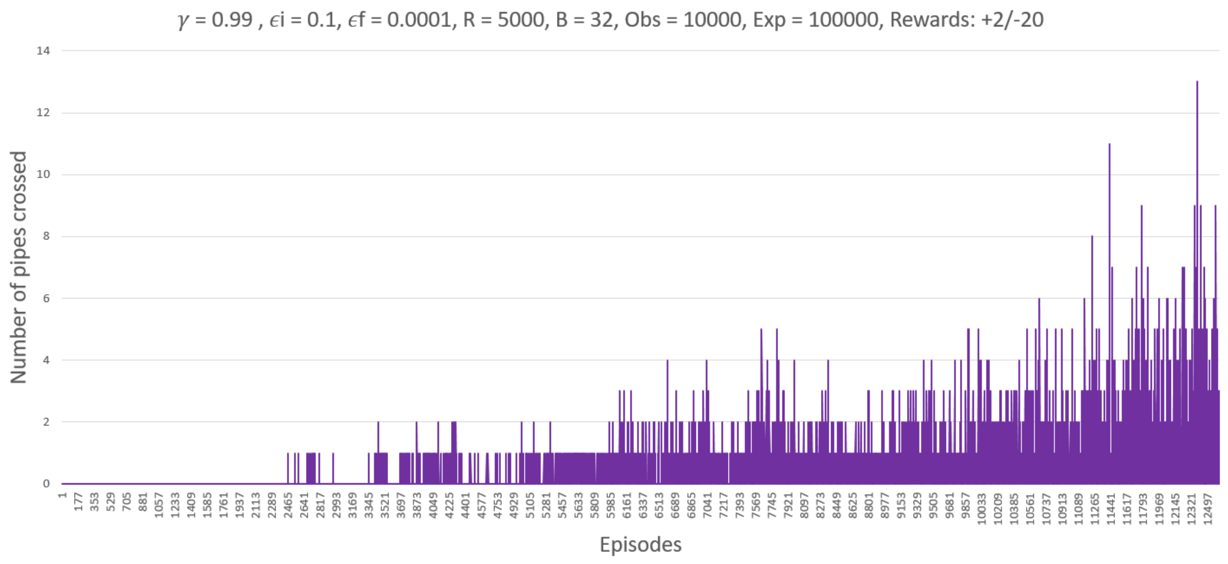


Figure 9. Number of pipes crossed vs Episodes (high negative reward)

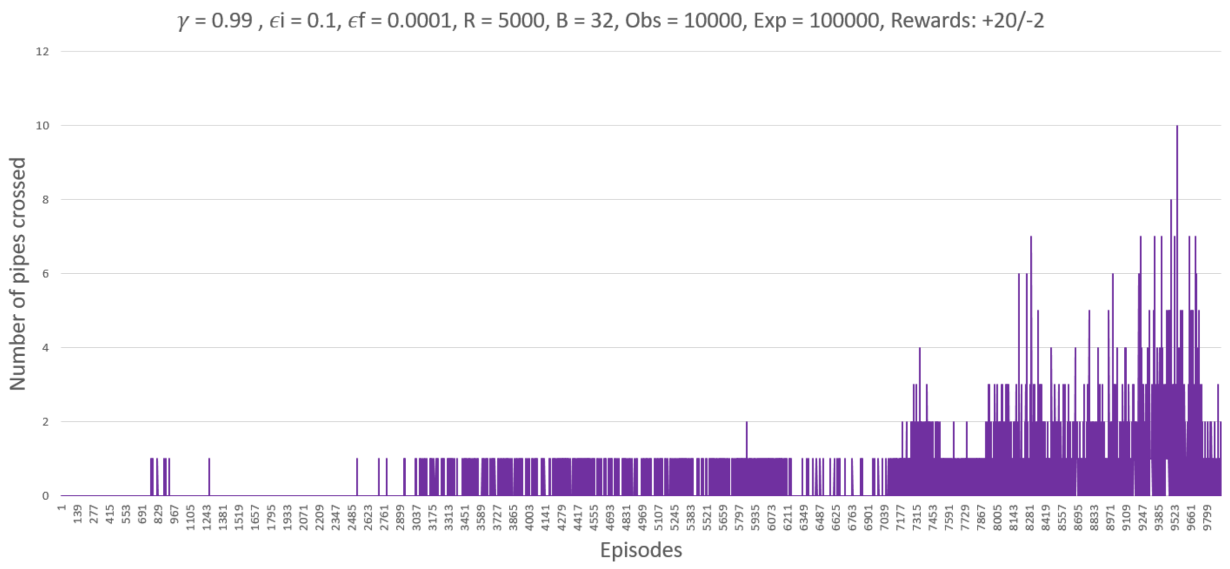


Figure 10. Number of pipes crossed vs Episodes (high positive reward)

We also tried reducing the size of the *replay memory* to be equal to the *batch* size, which is equivalent to not having a greedy epsilon approach. For each iteration, the learner will use only its last 32 (*batch*) experiences and nothing else. The *observe* state is also reduced to 32, so effectively there is no random action to add to the *replay memory*. This resulted in the agent not learning anything at all – there is no evident increase in Q-values or rewards, and its behavior resembles that of a random policy, as seen in Figure 11.

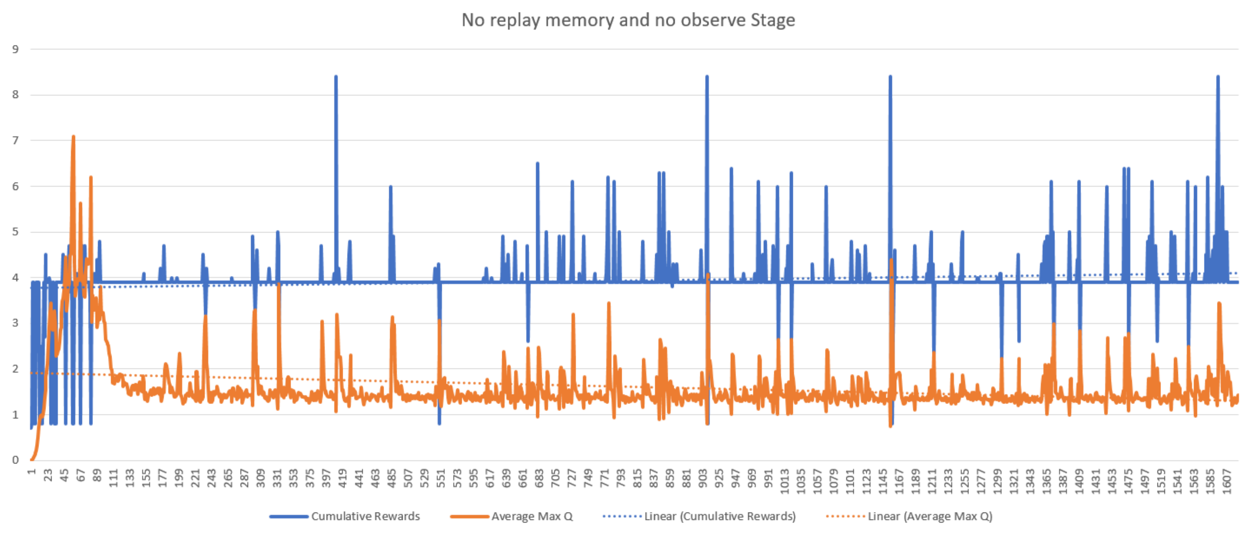


Figure 11. Maximum Q-values and Rewards vs. Episodes for agent that is not learning

Discussion

Q-values and the rewards obtained are two important metrics that are available to analyze Q-learning algorithms. The results show a clear increasing trend for the maximum Q-values and rewards as the agent trains for longer durations. This indicates that the agent learns a policy better with training than by merely executing random actions. The results also demonstrate the importance of experience replay – without it, the agent is unable to learn. In tuning the hyperparameters, we realized that increasing the *replay memory* size, increased the agent’s ability to learn. We also initialized *epsilon* to a relatively small value, as increasing epsilon increased the probability of random action and decreased agent’s ability to learn.

Related Work

Reinforcement learning is an area of machine learning inspired by behaviorist psychology that concerns itself with how any agent ought to make decisions in any given environment. Using video input of real world tasks, such as the motion of a robot arm or the movement of a car on a street, proves to be computationally intensive. Video games, on the other hand, provide a simple environment that mimic real world scenarios (to a certain extent). Games, which themselves use code, also provide an interface to interact with the environment. This makes it easy to send and obtain signals to and from the environment. We selected a pygame version of Flappy Bird as the learning environment because the dimensionality of the input space is only two: tap, no tap. With increasing dimensionality of the input space, training becomes more computationally expensive, which we avoided due to time restrictions. The following related works are where we drew our inspiration from:

1. Human Level Control using deep Reinforcement learning: nature.com/nature/journal/v518/n7540/full/nature14236.html

2. Playing Atari with Deep Reinforcement Learning: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

3. The Arcade Learning Environment: An Evaluation Platform for General Agents: https://arxiv.org/pdf/1207.4708.pdf

4. Temporal Difference Learning and TD-Gammon: https://cling.csd.uwo.ca/cs346a/extra/tdgammon.pdf

Future Work

The most immediate way to further our knowledge on deep Q-learning would be to apply these concepts to other games, including multi-input environments. We can also check the credibility of the current network by testing it with other single-input games. Once we are satisfied with our ability to apply these concepts, we would look into other forms of deep reinforcement learning and applying them in a similar manner and compare results. Better computational power will result in faster training and allow more experimentation, which is why we would make the switch from running on CPUs to running on GPUs in the future.

Final Remarks

Although this project took the form of a study as opposed to experimental research, there was a lot of work done to generate data to use as a means of understanding the underlying concepts. Recognizing which hyperparameters had the most effects on the agent's ability to learn and why they had such effect forced the researchers to acquire a low-level understanding of various components. The resulting discussions and experiments were incredibly beneficial to the overall study.

Works Cited

1. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
2. https://classroom.udacity.com/courses/ud600/lessons/4100878601/concepts/6512308540923
3. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf
4. https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/markov_decision_process.html
5. https://www.cs.rice.edu/~vardi/dag01/givan1.pdf
6. https://articles.wearepop.com/secret-formula-for-self-learning-computers
7. https://medium.com/@ageitgey/machine-learning-is-fun-part-3-deep-learning-and-convolutional-neural-networks-f40359318721
8. https://medium.freecodecamp.org/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8

References

1. https://github.com/yenchenlin/DeepLearningFlappyBird
2. https://en.wikipedia.org/wiki/Reinforcement_learning
3. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture14.pdf and https://www.youtube.com/watch?v=lvoHnicueoE
4. https://deepmind.com/blog/deep-reinforcement-learning/
5. http://rll.berkeley.edu/deeprlcoursesp17/docs/ng-thesis.pdf
6. https://piazza-syllabus.s3.amazonaws.com/jc964ooqwwu6jx/CourseSyllabus.pdf
7. https://classroom.udacity.com/courses/ud600