Atari *Breakout* with Reinforcement Learning

Bryan McKenney

December 15, 2021

1 Introduction

1.1 Problem

The goal of this project was to program an artificial intelligence that uses reinforcement learning techniques to play the Atari 2600 game *Breakout* and get high scores. The OpenAI Gym provides the environments for several Atari games, including *Breakout*, so all that was needed was the reinforcement learning algorithm. The object of *Breakout* is to move a paddle at the bottom of the screen to bounce a ball upwards and destroy blocks at the top of the screen. Letting the ball go off the bottom of the screen costs a life, and the game is over if all 5 lives are lost. A frame from the game can be seen in Figure 5. The Gym *Breakout* has the actions NOOP (do nothing), FIRE (spawn a new ball if there is not one on-screen already), RIGHT (move the paddle right), and LEFT (move the paddle left). In the BreakoutDeterministic-v4 environment, which was used for this project, a chosen action is taken for 4 frames before the observation is returned. The observation is the current screen (frame), which is represented as an RGB array of size (210, 160, 3). Rewards are given whenever a block is destroyed, and mirror the score in the game (higher blocks are worth more than lower blocks).

1.2 Motivation

My motivation for choosing this project is that I am quite fond of games, and learning how to use reinforcement learning to "solve" a video game seemed like an interesting task. I also may try to get a job as a video game programmer after college, so knowing how to train game AI would be useful. Needless to say, another motivation I had was getting a good grade in CS 780.

This is a challenging problem because the algorithm has to learn based on visual input alone (tied to actions taken and rewards received). This challenge makes the problem important to the fields of AI and reinforcement learning in general. Learning to understand visual input is particularly important for certain

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 1: The deep Q-learning algorithm from "Playing Atari with Deep Reinforcement Learning" [2].

kinds of robots, such as autonomous vehicles. Of course, this problem has already been worked on and I was not attempting to add anything new to the field.

2 Prior Work

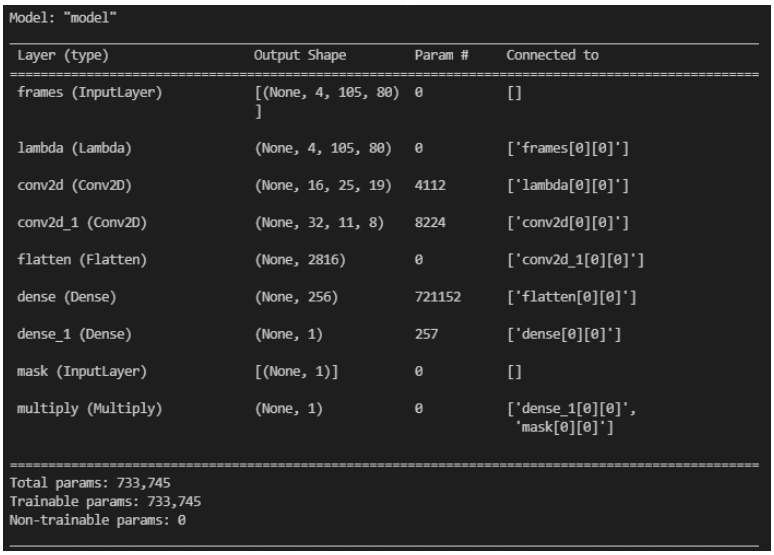
The DeepMind Technologies paper "Playing Atari with Deep Reinforcement Learning" [2] was one of the first publications introducing an algorithm that could play Atari games. The ones that came before are listed in the paper. This paper introduced a strategy called *deep Q-learning*, which is shown in Figure 1. It uses a deep neural network that learns to predict Q-values (called a *Deep Q-Network*, or DQN) and a *replay memory* which stores tuples each containing a frame and its associated action, reward, and next frame. I decided to implement this algorithm for my project, and I used an article called "Beat Atari with Deep Reinforcement Learning! (Part 1: DQN)" [1], which walks through how to implement certain parts of the DeepMind paper, for guidance. The author, Adrien Ecoffet, makes some slight modifications to the algorithm, and I decided to follow those.

3 Implementation

This project is implemented in Python 3.7, as using an older version is necessary to get the Gym's Atari environments to work.

Keras with a TensorFlow backend is used for the DQN. Keras' Functional API was used to create the DQN so that it can have two inputs (states and an action mask). Given a state, it outputs a Q-value estimation for each action from that state. The layers in the DQN can be seen in Figure 2.

For the replay memory, 4 parallel circular queues (a custom class) of capacity 1,000,000 store the related



```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #     Connected to
=================================================================================================
 frames (InputLayer)         [(None, 4, 105, 80)       0           []
                             ]

 lambda (Lambda)             (None, 4, 105, 80)        0           ['frames[0][0]']

 conv2d (Conv2D)             (None, 16, 25, 19)        4112        ['lambda[0][0]']

 conv2d_1 (Conv2D)           (None, 32, 11, 8)         8224        ['conv2d[0][0]']

 flatten (Flatten)           (None, 2816)              0           ['conv2d_1[0][0]']

 dense (Dense)               (None, 256)               721152      ['flatten[0][0]']

 dense_1 (Dense)             (None, 1)                 257         ['dense[0][0]']

 mask (InputLayer)           [(None, 1)]               0           []

 multiply (Multiply)         (None, 1)                 0           ['dense_1[0][0]',
                                                                    'mask[0][0]']

=================================================================================================
Total params: 733,745
Trainable params: 733,745
Non-trainable params: 0
_________________________________________________________________________________________________
```

Figure 2: The Deep Q-Network (DQN) that was implemented.

actions, rewards, next frames, and terminal statuses. The initial frame of each game is not stored. Before training begins, the replay memory is partially filled using data from a certain number of frames (Ecoffet used 50,000) that are obtained from taking random actions.

To choose an action, an ϵ-greedy policy is used, where a random action is chosen with probability ϵ and the "best" action is chosen otherwise. ϵ starts at 1.0 and anneals linearly to 0.1 over EPSILON_MAX (1,000,000 was used in the DeepMind paper) frames of gameplay. Then the action is taken and the next frame, reward, and whether or not the game is over are returned. This data is preprocessed (frames are scaled down by a factor of 2 and converted to grayscale, and rewards are simplified to 1, 0, or -1) and added to the replay memory, and then the memory is sampled for 32 random states, which are fed into the DQN. A state is 4 consecutive frames.

This process continues for a set number of frames (the DeepMind paper uses 10,000,000). After that number of frames is reached, the agent will continue playing to finish the current game and also play one more game, this time saving each frame to a list which will be used to create a GIF of the final game. This is how the plots in this report are able to have the number of episodes (games) on the x-axis and why they say that the algorithm was run for "around" a certain number of frames – that was the target, which will always be exceeded by some amount.

In addition to creating a GIF of the final game and a plot of the total rewards for each episode, the program outputs its progress and timing to the console. Figure 3 shows a sample console output.

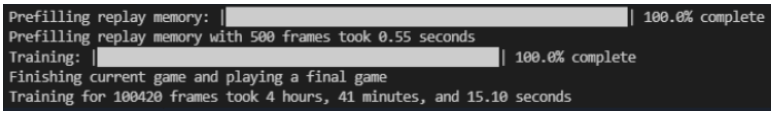
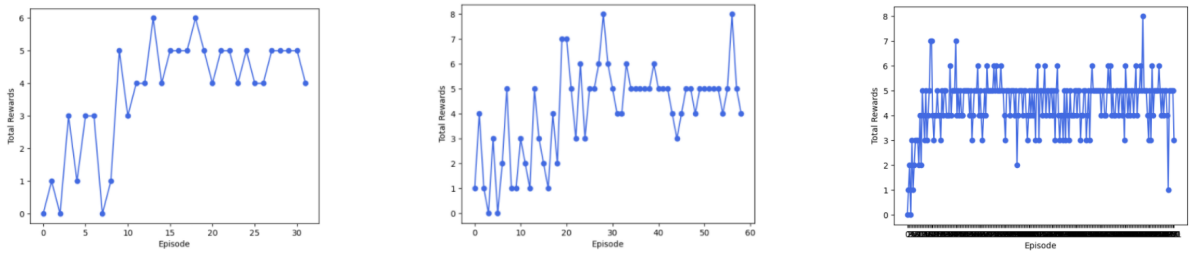


Figure 3: An excerpt of the console output of the program after playing for around 100,000 frames.



(a) Around 10,000 frames **(b)** Around 20,000 frames **(c)** Around 100,000 frames

Figure 4: Test runs of learning to play *Breakout*. Each test was run for a different number of frames.

4 Results

4.1 *Breakout*

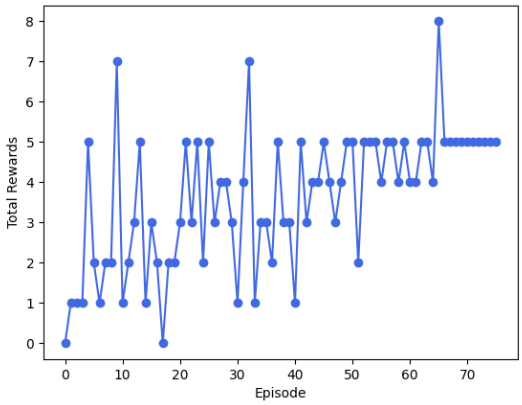
Originally, the plan was to follow the DeepMind paper and run a test for 10,000,000 frames, but the program turned out to be far too slow for this (despite GPU acceleration) – it would have taken around 13 days to complete. Instead, the algorithm was trained for 10,000 frames, 20,000 frames, and 100,000 frames, with the initial memory replay size and EPSILON_MAX scaled down as well. The results of these tests can be seen in Figure 4. Figure 4a shows that for 10,000 frames, the rewards gained per episode quickly climbed to an average of 5 and then stayed there. Figure 4b shows that with twice the number of frames, the result was about the same. And with 100,000 frames, as shown in Figure 4c (apologies for the illegible episode numbers), there was no further improvement. 4 and 5 were the most common total rewards, and the highest reward was only 8. A frame from the final game of this test can be seen in Figure 5. The resulting GIFs from all three tests showed the same thing: the agent was sending the paddle to the left side of the screen and keeping it there (with some jitters back and forth). This enabled it to hit the ball 5 times with minimal effort, but would also cause it to lose after those 5 hits (because it would let the ball fall and respawn after each). Perhaps this is simply because there was not enough training done to learn a better policy – after all, EPSILON_MAX was supposed to be 1,000,000, but in these tests it was reduced to 5,000, 10,000, and 10,000, respectively, which caused the AI to think that it had learned enough to be making the best actions all the time when it still did not have a lot of data. It is also possible that something was implemented improperly.

Figure 5: A screenshot of the final game played during the test run of around 100,000 frames. This screenshot was captured right before the agent moved the paddle to the right, letting the ball get past and losing its last life.

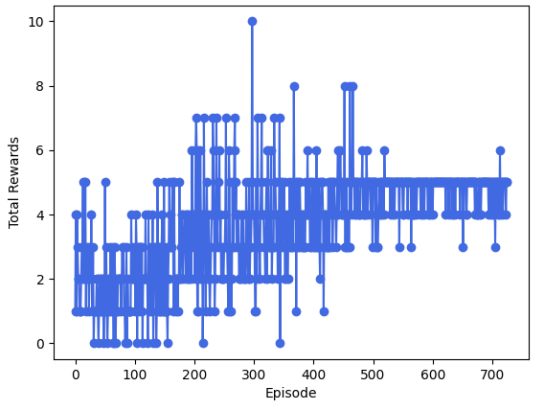
In hopes of better results, I modified the implementation so that for every action, 4 steps are taken (each of which were already 4 frames) before the final frame is saved into the replay memory (along with the total rewards gained during those 4 steps). This is an improvement that Ecoffet mentions is suggested in another DeepMind paper, "Human-level control through deep reinforcement learning." The results of two tests using this strategy, one run for 10,000 frames and the other for 100,000 frames, can be seen in Figure 6. Although more episodes are able to be played in the same number of frames, there is no improvement in rewards gained, and the agent learns the same policy as before – to stay in the left corner. This was to be expected, as skipping frames will speed games up but not change the way that the algorithm learns.

4.2 *Pong*

In the DeepMind paper, the same algorithm is applied to seven Atari games, including *Breakout* and *Pong*, all with good results. Though the results for my implementation on *Breakout* were not good, I wanted to see how the algorithm would fare on *Pong*. The results of 50 games spanning 10,000 frames are shown in Figure 7 – and they are worse than the results on *Breakout*. The agent does not seem to learn anything at all, getting the worst possible score on 49 out of the 50 games and only a slightly better score on the remaining game (which was towards the beginning, not the end, of training). In hindsight, this is not surprising, as the number of frames is low and *Pong* is more complex than *Breakout* in that it has the moving enemy paddle as well as the moving ball to deal with.



(a) Around 10,000 frames



(b) Around 100,000 frames

Figure 6: Test runs of learning to play *Breakout* with 4 steps taken per action (when possible). Each test was run for a different number of frames.

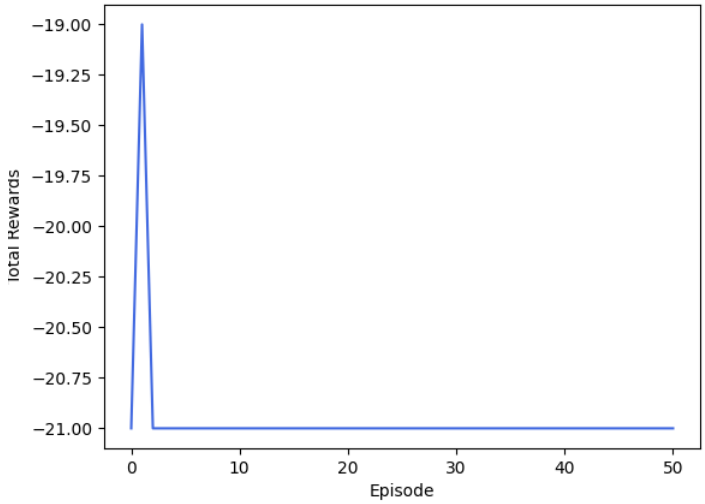


Figure 7: Rewards gained per episode when playing *Pong* for around 10,000 frames.

5 Reflection

Although it did not work out the way that I wanted it to, this project was fun and I learned quite a bit. I learned about convolutional neural networks, RMSProp, deep reinforcement learning, and how to implement these things in Python. I also learned how to make console loading bars and create GIFs, and got some more practice making plots (I found that markers and static x-ticks are a bad idea when there is a lot of data). In addition, I learned how to set up NVIDIA CUDA properly for GPU acceleration and how to code a robust circular queue in Python.

I hope to return to this project in the future and find a way to get it to work, whether that means following a better tutorial, running the program for 13 days, or starting with a pre-trained neural network (although that is not as fun). I would also like to read "Human-level control through deep reinforcement learning" and Part 2 of Ecoffet's tutorial, as those might be helpful.

References

[1] Adrien Ecoffet. Beat atari with deep reinforcement learning! (part 1: Dqn). *Becoming Human: Artificial Intelligence Magazine*, 2011.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.