AI Agent for Playing Breakout on Atari 2600

Jai Gupta (jaigupta@standford.edu) **Wenjia Hou (hwjsu@stanford.edu)**

Ritesh Reddy (riteshr@stanford.edu)

1 Task

The objective of this project is to build an AI agent to play the Atari game *Breakout* [1]. In Breakout, the objective is to destroy all the bricks on the screen with the ball before losing all lives. The controls are moving the paddle left, right or not moving at all.

1.1 Input-Output

The input for this problem consists of the raw pixels of the game screen of dimensions (210,160,3), the reward for taking a particular action, and whether the game has ended after taking a particular action.

The output will be the best action to take to based on the given input state in order to improve our score.

1.2 Baseline-Oracle

The baseline for this project was an agent that simply moved at random. We achieved an average reward of 0.36 when we ran this agent for 30 episodes.

The oracle for this project will be a high score achieved by a Reinforcement Learning algorithm of 760[3]

2 Infrastructure

We used the openAI Gym[2] reinforcement learning environment to interact with the game. The engine provides a simple API to get the pixel values for the screen by first taking in an action to perform and then performing that action in the environment.

To build the neural networks and learn the models needed to predict the Q values, we used the TensorFlow[9] framework to do the actual learning.

3 Literature Review

The first Deep Q Networks Learning Model is introduced by DeepMind in 2013[5]. They trained a convolutional neural network to learn best policies from raw video data using reinforcement learning. The same algorithm was applied to 7 different Atari 2600 games including Breakout. The results outperformed all the results before this work. Since then, there have been some improvements, such as experience replay, and frame-skipping. This algorithm was later applied to many other Atari games[8].

To reduce the overestimation problem in DQN, the Double DQN algorithm, which combines the double Q-learning and deep Q Networks, is introduced in 2015[6]. Double DQN

decomposes the max operation in the target into action selection and action evaluation. The results on Atari games showed that double DQN not only reduced overestimation, but also achieved better performance on several games.

4 Approach

4.1 Model

We have modeled the problem as a Markov Decision Process (MDP) with the following definitions:

1. **State**: A stack of four frames representing the last 4 screen frames of the screen. This is modified from the original RGB frame (210, 160, 3) from the game environment to gray-scale shape of 84x84x1 pixels. The final shape combining all 4 four such frames is (84, 84, 4). We need to use multiple frames because a single frame does not provide enough information to determine the direction of motion and velocity of the ball which are critical to taking the right action.
2. **Actions**: There are three actions in any state: NO-OP, MoveLeft, MoveRight.
3. **Reward**: The reward is straightforward and is returned by the environment when a new state is generated based on an action. The reward system in the game results is 1 point per destroyed block.
4. **End State**: The end state is also signalled by the game environment. A game ends when all the blocks are destroyed or there are no more lives left.

The standard approach to find the optimal policy for MDPs would be to use Value Iteration. However, for this game, the state space is very large and we don't know the state transition probabilities. Therefore, we cannot use simple Value Iteration to find the optimal Q Value for each state and choose the best action for each state based on that. Instead, we have to play the game to generate (state, action, reward, new-state) tuples as examples and learn the Q values from them.

We chose a model-free modelling solution to deal with the large state space, and to generalize well to new, unseen states with similarities to older states. We use deep learning to learn an approximation of the Q value for each state and action pair and then derive the optimal policy based on these values.

4.2 Algorithms

This solution can be broken down into different parts, specifically:

1. Generating Data
2. Learning a model from Data
3. Evaluating the learnt model

4.2.1 Generating Data

To generate the data, we play the game by following a simple epsilon-greedy strategy where we pick a random action with probability ϵ and pick the best action given the current model's Q-values for each action with probability $1 - \epsilon$

$$\pi_{act}(s) = \begin{cases} arg\max_{a \in Actions} Q(s,a) & \text{with proba } 1-\epsilon \\ \text{random from Actions} & \text{with proba } \epsilon \end{cases} \quad (1)$$

We start with an epsilon of 1.0 and slowly reduce this value by small amounts to 0.07 in 1,000,000 steps. This ensures we explore more to start with, but are more cautious and exploit the best solutions at later stages.

4.2.2 Learning the model from the data

Based on the above data generation technique we are able to generate many tuples of the form (state, action, reward, next-state) where state is a stack of the three previous frames and the current frame. Specifically we learn the function: $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[R(s,a) + \gamma \max_{a'} Q(s',a')]$ where (s,a) is the current state and action pair, s' is the succeeding state based on action a, a' is the best possible action for this state representing the best future reward. $\gamma = 0.99$ is the discount factor for future reward and $R(s,a)$ is the reward for the current (s,a) pair.

However, in order to learn this from the given state representations without explicitly extracting features, we relied on convolutional neural networks. We tried three variants of our algorithms described below, starting with a simple DQN and slowly enhancing this algorithm with various improvements to speed up the learning.

Simple Deep Q Learning
Our first learning algorithm is a simple deep neural network in order to learn the Q value for each state, action pairs. The architecture of this neural network is as follows:

1. 32 unit Convolution Layer with relu activation
2. 64 unit Convolution Layer with relu activation
3. 64 unit Convolution Layer with relu activation
4. Flattening layer
5. 512 unit Dense Layer
6. 3 unit with linear activation

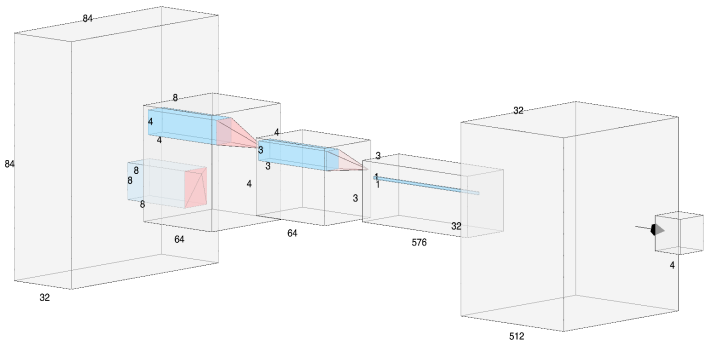

Figure 1: Structure of the Neural Net

Each unit in the output layer maps to each possible action in the game. The unit with the highest value will represent the action with the max Q value for the given state.

After taking an action based on the above epsilon-greedy algorithm, we observe the new game frame, and then combine it with the previous three frames to create the next game state. We then feed it through the above neural net to obtain 3 Q values, one per each possible action that can be taken at the new state. We then take the max of these 3 Q values as $\max_{a'} Q(s',a')$. This value will be part of our target Q value for state, action (s,a) and the predicted value is simply the Q value for the current state, action with the current weights. These two values are then used to compute the loss and minimize it by changing the weights using the RMSProp[4] algorithm. The results from this approach are detailed below in the results section.

Deep Q Learning with Experience Replay, Double Q and Fixed Target Network
The network above shows moderate results, but in general there are some problems with it:

1. A high degree of correlation between the many different states explored in the above approach resulting in a biased data set.

2. Since we train on a single new state based, we forget the previous experiences and the weights learn from them.
3. Since the target value keeps updating on each iteration, the algorithm may not converge as we are chasing a moving target.
4. Due to the inherent max function in computing the target value, the algorithm tends to prefer overestimated values to underestimated values and result in bias.

To mitigate these problems we used three improvements: Experience Replay, Fixed Target Network and Double DQN (Double Q).

Experience Replay[5] tackles the first two problems and works by decoupling the data generation and learning aspects. In experience replay, we simply record the last X (10000 in our case) state, action, reward tuples in a cache and then sample from this recording and learn on this mini-batch of tuples.

Fixed Target Network[8] tackles the third problem. It's implemented as a second neural network that mirrors the above network. Its weights are fixed for a number of learning iterations (1000 in our case) and is used to generate the target Q value for a given state. After the 1000 iterations, the weights for this network are updated from the learning network. This reduces the variance and oscillations caused by a moving target and hence improves the chances of convergence.

Double Q[6] tackles the fourth problem. It's an enhancement of the Fixed Target Network and works by decoupling the action selection and action evaluation. Instead of having the Q network, which has constantly changing weights, return the Q value of the highest action, we have the Q network only return the action with the highest value, and compute the Q value of this s', a' pair using the target network, which is more stable and changes weights every 1000 iterations.

4.2.3 Evaluating the Model

We evaluated the models that we learnt by running the game for 10 episodes after every 100 episodes of learning and computing the average reward over all 10 episodes.

4.3 Implementation

The implementation of the above algorithms can be found here: `https://gist.github.com/riteshreddyr/7d9fba477c5b1a976a91b4f2782945c2`

5 Results and Analysis

5.1 Deep Q Learning

5.1.1 Performance

Please refer to Figure 2.

5.1.2 Analysis

We found that the simple DQN preformed better than our baseline. However, the training took a very long time, around 12 hours to get to 18000 episodes. We expect that this model will continue to improve if we let it train for longer. However, we found this untenable and attempted to improve it by applying well known improvements to speed up this learning.

We trained each model for 18000 episodes. It is challenging to train the model for a longer time due to limited compute resources, hence we use this as our benchmark for comparing the various variants of DQN.

The large oscillations in the average reward in Figure 2 is because we were chasing a moving target as the target was also computed using the neural network with constantly changing weights. We will see how this can be improved in the next variant.

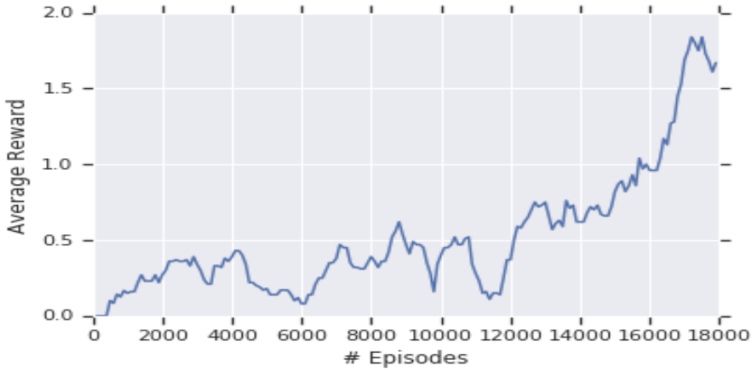

Figure 2: Simple DQN Result

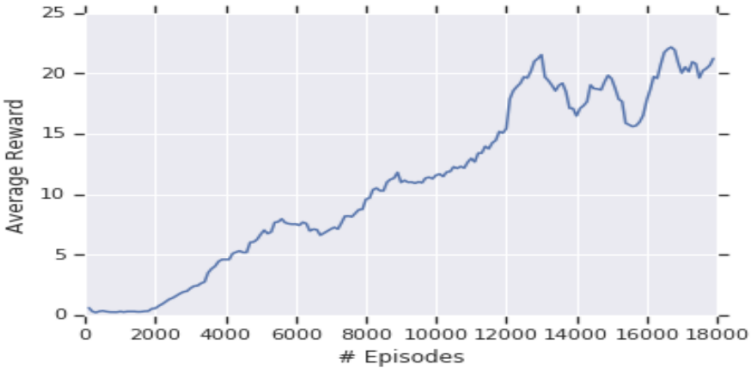

Figure 3: DQN with Experience Replay and Fixed Target Result

5.2 Deep Q Learning w/ Experience Replay and Fixed Target

5.2.1 Hyper-Parameters

This variant can be tuned further via two additional hyper-parameters: The Experience Replay Buffer size and the Target Neural Network Update Frequency. Due to limited resources we stored only 10000 steps in our memory buffer and updated the target network every 1000 steps.

5.2.2 Performance

Please refer to Figure 3.

5.2.3 Analysis

This variant took approximately 14 hours to train and it already showed a significant improvement over the simple DQN within 18000 episodes.

This variant performed extremely well compared to the simple DQN because of the Experience Replay buffer and also the Fixed Target. By sampling from the memory instead of online learning, we were able to re-learn from the same experiences multiple times, including those which have a small probability of being discovered via our explore-exploit strategy. The fixed target helped to reduce the large oscillations that are inherent when chasing a moving target in the loss minimzation process. It can be seen clearly in Figure 3 that the average reward up to 2.0 is achieved much more linearly than in Figure 2 owing to the reduction in oscillations due to the Fixed Target network architecture.

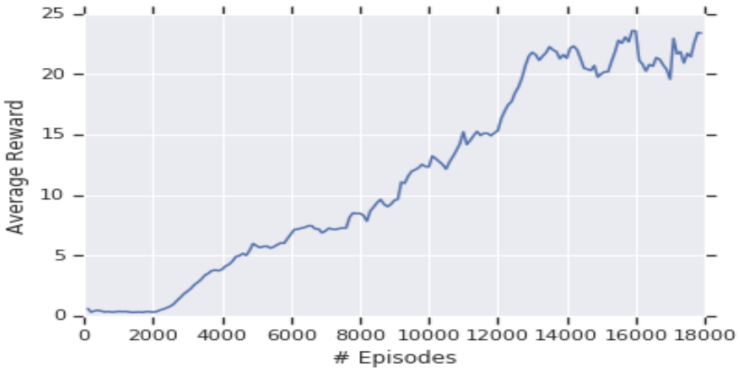

Figure 4: DQN with Experience Replay, Fixed Target and Double DQN Result

5.3 Deep Q Learning w/ Experience Replay and Fixed Target Network and Double Q Learning

We wanted to explore another enhancement that shows promising speed and accuracy improvements in existing literature: the double-q network that reduces overestimation biases when using the natural fixed target architecture.

5.3.1 Performance

Please refer to Figure 4.

Experience Replay as shown helps significantly in speeding up the training process. But latest literature showed that more improvement can be gained by using a Double-Q net as it further tackles the problem of a biased target introduced by the $\max$ operator.

5.3.2 Analysis

We can see a clear gain in the speed of the training at the very beginning of the process where the weights of the target network are still in flux. This is expected as the double-q network uses the evaluation network to find the action with the $\max$ operator but pick the Q value for this action using the fixed network. This allows us to remove the bias in always choosing the action with the current, and possibly incorrectly ranked, highest Q value from the target network, while retaining the fixed nature of the target's actual Q value. We observed an overall improvement of approximately 10 times in the $18,000$ episodes compared to the simple DQN algorithm: while the simple DQN implementation achieves a score less than 2, the modified version here achieves a score over 22 within a much shorter time frame.

6 Challenges

There were a number of challenges in building an agent to play Breakout. Some of the prominent ones were:

1. We start to see an improvement in scores only after approximately 5000 episodes, and up to 10000 episodes most models perform similarly. Therefore, the edit-run cycle was quite long to be able to debug and improve the models given our limited time and compute resources.
2. The performance of an algorithm is greatly influenced by the hyper parameters, for example, the fixed target network update frequency, buffer size for experience replay. However, finding the right combination of all the hyper-parameters is challenging especially given the long edit-run cycle.

A simple solution to both of these problems is to use specialized learning components such as Graphics Processing Units or Tensor Processing Units[10] to speed up the edit-run cycle, however, this can be very expensive. An alternative is to save the model to disk at various points and reload

from certain checkpoints while debugging specific issues that occur close to some checkpoint. These can be explored in future work to further improve the speed of the algorithms use in this project.

7 Conclusion and Future Work

The central idea in this project was to use Deep Q-Learning to learn the Q values of Breakout state, action pairs to compute the best policy for any given state. Simple Deep Q-Learning is promising, and performed better than our baseline, however, it is not very efficient. To learn more efficiently, we enhanced the simple DQN with Experience Replay[5], Fixed Target Networks[8], and Double Q Learning[6]. The improved network was able to achieve an average score of 35 in approximately 25000 episodes. To further improve the efficiency, we can incorporate prioritized experience replay [7] as an improvement over the uniform experience replay.

A different approach would be to explore different types of Reinforcement Learning algorithms such as the Asynchronous Advantage Actor-Critic[11] model that parallelizes and learns the model much faster as described in the oracle[3].

References

[1] *Breakout (Atari 2600).*
`https://en.wikipedia.org/wiki/Breakout_(video_game)`

[2] *OpenAI Gym*
`https://gym.openai.com/`

[3] *Breakout Reinforcement Learning High Score*
`https://gym.openai.com/evaluations/eval_NiKaIN4NSUeEIvWqIgVDrA/`

[4] Geoffrey Hinton *RMSProp Algorithm*
`https://www.cs.toronto.edu/t̃ijmen/csc321/slides/lecture_slides_lec6.pdf`

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning, 2013; arXiv:1312.5602.

[6] Hado van Hasselt, Arthur Guez and David Silver. Deep Reinforcement Learning with Double Q-learning, 2015; arXiv:1509.06461.

[7] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver. Prioritized Experience Replay, 2015; arXiv:1511.05952.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness etc. Human-level control through deep reinforcement learning, 2015; Nature 518, 529–533 (2015).

[9] *TensorFlow*
`https://www.tensorflow.org/`

[10] *Tensor Processing Unit* `https://cloud.google.com/tpu/`

[11] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, 2016, ICML 2016; arXiv:1602.01783.