Reinforcement Learning for Atari Games

Group33: Data Vaders 2 — Mentor: Nishit Asnani
CS771 Project Report
Indian Institute of Technology, Kanpur

Abhishek Agarwal
150027
abhishag@iitk.ac.in

Anubhav Mittal
150116
anubhavm@iitk.ac.in

Aarsh Prakash Agarwal
150004
aarshp@iitk.ac.in

Bhuvi Gupta
150191
bhuvig@iitk.ac.in

Chinta Hari Chandana
150205
hchinta@iitk.ac.in

Abstract

The main aim of our project was to use Deep Reinforcement learning to learn a model which can achieve super-human performance on the popular Atari game Flappy Bird. Two of the commonly used approaches for Reinforcement Learning are - DQN(Deep Q-Learning) and Policy Gradient(PG). We train two separate models for Flappy Bird using both approaches and draw a comparison between our findings.

1 Problem Statement

The goal of this project is to learn a model that has super-human performance on the game 'Flappy Bird'.

Flappy Bird:
Flappy Bird is a game in which the player tries to keep the bird alive for as long as possible. The bird automatically falls towards the ground due to gravity, and if it hits the ground, it dies and the game ends. The bird must also navigate through pipes. The pipes restrict the height of the bird to be within a certain specific range as the bird passes through them. If the bird is too high or too low, it will crash into the pipe and die. Therefore, the player must time jumps properly to keep the bird alive as it passes through these obstacles. The game score is measured by how many obstacles the bird successfully passes through. Therefore, to get a high score, the player must keep the bird alive for as long as possible as it encounters the pipes.
Training an agent to successfully play the game is especially challenging because our goal is to provide the agent with only pixel information and the score. The agent is not provided with information regarding what the bird looks like, what the pipes look like, or where the bird and pipes are. Instead, it must learn these representations and interactions and be able to generalize due to the very large state space.

In our project, we attempt to train our model using two approaches - Deep Q-Learning and Policy Gradient. We then compare the results obtained from both models - including the difficulties encountered in training. We then explore the possible future scope of work.

2 Problem Motivation

Reinforcement Learning for an Atari Game closely resembles an approach of a human - reinforcing positive outcomes and discouraging negative outcomes. As in case of human, the algorithm also takes in a high dimensional input(in this case game screen) and processes to decide what action to take next. This is consistent with the philosophy of learning in the real world wherein an agent learns from its mistake rather than a predetermined policy for all states.

3 Existing Work

The field of Reinforcement Learning, especially Atari is one of the more explored fields currently. Googles Deepmind has implemented and achieved impressive results. The approach used by Deepmind [1] The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. The inputs and outputs stated above are facilitated by toolkits like Open AI Gym and Arcade Learning Environment, which are compatible with Tensorflow. There are some implementations [2] of flappy bird and similar games that are open source. Most of them implement Q-Learning.

4 Methodology

As we have discussed, Reinforcement Learning imitates the way a human learns from the environment: primarily by trial and error. The crux can be explained using **Markov Decision Process**. It has three main components: state s_t, action a_t and reward r_t. The process states transitions are independent. If we define $p(s_{t+1}|a_t, s_t)$ as the probability of reaching the state s_{t+1} given that you have taken action a_t on observing state s_t. Thus, we have $p(s_{t+1}|a_t, s_t)$ is independent of $p(s_t|a_{t-1}, s_{t-1})$.

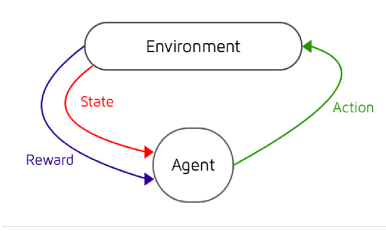

Figure 1: Reinforcement Learning

The agent on observing state s_t takes the decision/action a_t and receives the reward r_t and the state s_{t+1}. The process is repeated in the same manner till the episode ends. The above formulation of the problem can be used to achieve two functions that can be optimized: **action value** based function and **policy** based functions.

4.1 DQN - Deep Q-Network Approach

4.1.1 Q-learning

Consider one episode of the game:

$$s_0, a_0, r_0, s_1, a_1, r_1, ..., s_n, a_n, r_n$$

Here, s_n is the terminal state. The total reward from time t onwards can be calculated as :

$$R_t = r_t + ... + r_n$$

Because the game is stochastic in nature, we apply an increasing discount factor to the future rewards because the same actions may not result in same rewards in future:

$$R_t = r_t + \gamma r_{t+1} + ... + \gamma^n r_n$$
$$\Rightarrow R_t = r_t + \gamma R_{t+1}$$

Now, we define a **Q-function** $Q(s, a)$ representing the maximum discounted future reward we can get after we perform action a on state s :

$$Q(s_t, a_t) = maxR_t$$

If we have this function for our game, deciding the action for the next state is trivial:

$$a_{t+1} = \arg\max_a Q(s_{t+1}, a)$$

This is the function we try to learn in a Q-learning setting. Also,observe that:

$$Q(s_t, a_t) = maxR_t = r_t + \gamma \max R_{t+1} = r_t + \gamma \max_a Q(s_{t+1}, a)$$

This is taken as the required target value of the Q-function, and its difference with the value output by the Q-function directly is used in loss functions to obtain the required Q-values.

4.1.2 Deep Q network

For a simple setting with finite(small) set of states, Q-function can be implemented as a table. However, for a setting such as ours which has possibly infinite no. of states, we use a Neural net to approximate our Q-function known as DQN. As the inputs are images, we used CNN in our model.

Two distinguishing characteristics of the DQN based learning method are:

- Experience Replay: If we update our network by getting the Q-values for the current state, performing the action with highest Q-value, getting the target Q-values as defines earlier, and performing a gradient step on the loss function, the learning is unstable and converges much slowly as the closer inputs are highly correlated. To prevent this, instead, we store the values $\{s_t, a_t, r_t, s_{t+1}\}$ after each iteration in a 'replay buffer'. For training the network, we sample a batch of these sets from the network randomly and perform mini-batch gradient step to update the values. As the inputs are now taken randomly from a large memory, they are mostly uncorrelated, and thus we get a much more stable learning.
- Exploration-Exploitation: Initially, as we randomly initialize the network, there is a sense of randomness to how the actions are decided using the Q-values. Thus, the network is able to explore the possibilities and decide which actions would yield better overall reward. However, we can see that the approach is somewhat 'greedy' in nature - the network settles down after some time and the randomness is lost, neglecting other possible actions that may have resulted in a better reward. To tackle this, we introduce another variable ϵ and for each state, execute a random action with ϵ probability, and otherwise execute the action with highest Q-value. This ϵ is reduced over time to allow the network to explore the possible rewards.

4.1.3 Algorithm

The overall algorithm can be summarized in the following pseudo-code:

Algorithm 1: DQN algorithm

Initialize replay memory D to size N
Initialize the Q-function randomly
for episode 1,...,B **do:**
  Initialize initial state s_1.
  for time t=1,...,T **do:**
    Either Select action a_t with probability ϵ **OR** Select $a_t = \underset{a}{max} Q(s_t, a)$
    Perform action a_t and note the reward r_t and state s_{t+1}
    Store the transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample a minibatch of transitions (s_j, a_j, r_j, s_{j+1})
    Set $y_j = r_j$ (if s_{j+1} is the terminal state) **OR** $r_j + \underset{\bar{a}}{max}$ Q$(s_{j+1}, \bar{a})$ (otherwise)
    Perform gradient step on $(y_j - Q(s_j, a_j))^2$
  end for
end for

4.2 Policy Gradient Approach

4.2.1 Policy Gradient method

The method involves the learning of policy function $\pi_\theta(a_t|s_t)$, i.e., learn to choose from a set of actions, given a state that maximizes the value of **expected reward function**. Thus it omits any kind of action value based functions. We are using **sigmoid** function to determine the probabilities of various actions from the action space $\mathbf{A}$. In our case action space simply consists of two actions: to jump or not to jump.

$$p(jump|s_t) + p(notjump|s_t) = 1$$

We are choosing a deep neural network to model our policy function $\pi_\theta(a_t|s_t)$. Consider a typical episode, the policy network receives s_t and outputs the probability of the bird jumping, thus a decision a_t is made based on that probability and reward r_t is given and our game reaches to state s_{t+1}. Now the state s_{t+1} is fed to the network and the process goes on till a terminal state s_n (when the bird collides with pillar or the bird has crossed 1000 pillars) is achieved. The goal is to maximize the expected reward $E[R_t]$ for the episode. The entire sequence of action $a_1, a_2.....a_n$ are evaluated on the basis of Reward for the overall episode. In order to learn our policy $\pi_\theta(a_t|s_t)$, i.e., weights of our deep policy network, we will try batch gradient descent algorithm. The gradient of expected Rewards with respect to policy π_θ can be written as

$$\nabla_\theta E[R_t] = E[R \cdot (\nabla_\theta \sum_{i=1}^{i=n} log(\pi_\theta(a_t|s_t)))]$$

The parameter θ of the network, i.e., weights are updated using back-propagation.

4.2.2 Policy network

The states are actually the frames of the game. The raw frame that we are receiving is RGB (288*512). We are resizing it to grayscale (80*80*1) and feeding it our feed-forward policy network.

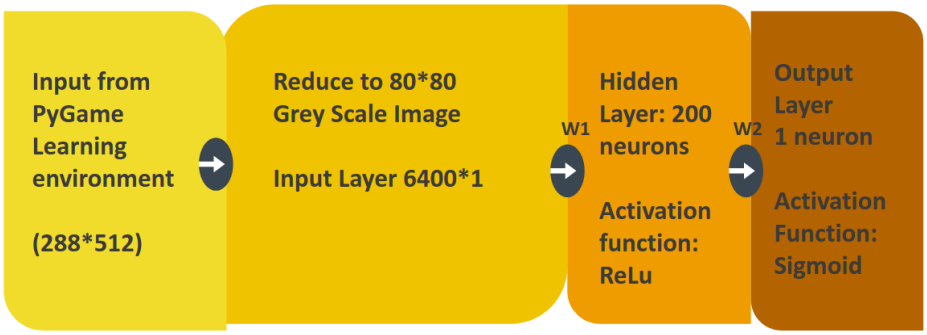

Figure 2: Policy network

The policy network is the feed-forward neural network that comprises of a hidden layer of 200 neurons with ReLU activation function, the output is fed to single neuron with sigmoid function at the end to compute the probabilities of going up.

5 Data

We use a Pygame implementation of flappy bird, which provides us with game image pixels(512*288), positive rewards for staying alive,for crossing a pipe, and negative rewards for dying.

6 Experimental Results

For DQN, the metrics used for evaluation are average score and Q values over episodes. For PG, the metrics used for evaluation are average score over episodes.

6.1 Deep Q Learning

6.1.1 Preprocessing

As done in the previous related papers, the image input was converted to grayscale and converted to size 80X80. This encourages faster convergence as colour information add to the noise and thus, cause slower learning. For input to the network, last four images were stacked together to give the sense of motion to the images.

6.1.2 Structure of CNN

Tensorflow was used to implement the CNN. Our final CNN had the following structure, which was majorly derived from the implementations in related papers:

Layer	Input size	Filter size	filters	stride	output
conv1	80x80x4	8x8	32	4	20x20x32
conv2	20x20x32	4x4	64	2	10x10x64
conv3	10x10x64	4x4	16	2	4x4x16
fc	256x1	-	-	-	256x1
Output	256x1	-	-	-	2x1

6.1.3 Tuning of parameters

- In the first approach, the ϵ was reduced from 0.5 to 0.01 over the course of 1,500,000 iterations. It was observed that due to the nature of the game, a 50% chance of random actions means a 25% chance of selecting the action to go up, which is much higher considering the game is running at 30 fps. This caused the bird to remain at the top of the screen for a large interval of training time, and thus the effective training time was much smaller. To counteract this, now ϵ was set to uniformly reduce from 0.1 to 0.001 over the course of 1,500,000 iterations.
- The reward system was +1 for crossing the pipe and -1 for dying. After tuning for ϵ, it was observed that there was close to no improvement in game play for a large number of iterations. It was concluded that as

model is behaving entirely random at the start, it is a rare event for the bird to cross over the pipe and give the positive reward. It was then even rarer for that event to be present in the batch extracted for training the network. Hence, new reward system was introduced where we also provided a reward of +0.1 for each frame where the bird is alive.

- Squared loss function was used and Adam adaptive learning rate method was used for adjusting the learning rate with $eps = 1e - 6$.
- The training was done with mini-batches of size 32 and the replay memory was set to a maximum size of 50,000 experiences. For the first 10,000 iterations, no training was done and the experience were used to fill up the replay memory.
- Discount factor γ was set to 0.99.

6.1.4 Training time and overall performance

Two modes were used in training of the model: one with $PIPEGAPSIZE = 150$ (easy) and the other being for $PIPEGAPSIZE = 100$ (hard).

- Hard settings:

  The model was initially trained on the hard setting, as the thought was if the model can pass through the hard settings, it should do excellently in easy settings too.

  The model was trained on hard setting for about 1,000,000 iterations and the results were quite encouraging:

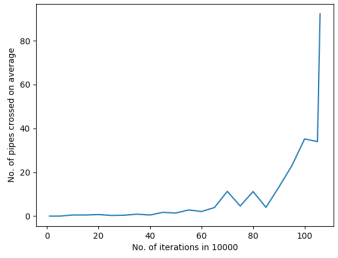

Figure 3: Averaged Mean Score vs no,. of training iterations

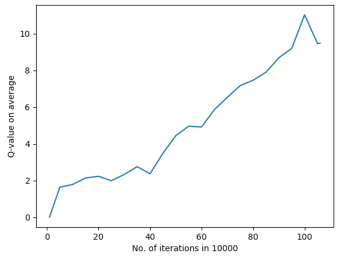

Figure 4: Averaged Q-value vs no. of training iterations

As we are taking average over only 10 episodes for each model, this result is somewhat fluctuating and not really accurate, but it gives a good enough indication that the model will converge to a state which has infinite (> 1000) average score. The link for the video of final model: `https://youtu.be/yVVeGVXKhzQ`

- Easy settings:

  The performance of the hard settings learned model was now tested in the easy mode game:

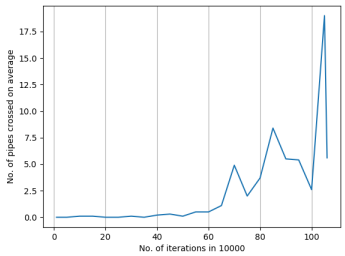

Figure 5: Averaged Mean Score vs number of training iterations

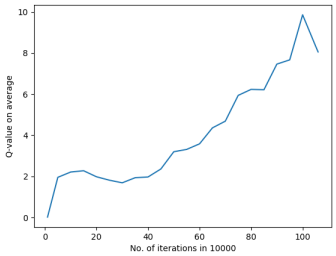

Figure 6: Averaged Q-value vs no. of training iterations

It was seen that a good score in hard settings does not directly translate into a good score for easy settings. Now the learned model on hard settings for 1,00,000 iterations was again put on training, but this time it was trained on the easy settings. **The model converged in the next 150,000 iterations to infinite score** (> 1000). The link for the video of the model: `https://youtu.be/GqiQvZHU8hQ`

6.2 Policy Gradient

6.2.1 Tuning

Approach 1 In our initial approach we implemented the net as described in Section 4.2. We trained the model for around 1500 episodes. We observed that the bird learns to just shoot up to the roof since that bought more time. Falling to the floor terminated the game as is, but shooting up to the roof added more reward every passing frame till the bird hits a pillar. The graph of averaged mean score vs number of episodes is as shown below. However, we used this attempt to tune hyper parameters like learning rate, discount factor and batch size.

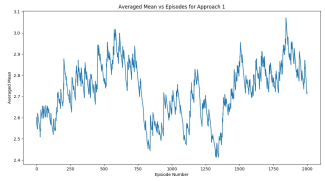

Figure 7: Averaged Mean vs Episodes for Approach 1

Video: `https://youtu.be/2t9dTif-9G0`
Approach 2 One of the reasons for the bird to go up was the high frequency with which it was making decisions. Since jumps on three consecutive frames meant that it would hit the roof. Therefore, we reduced the frequency to match that of a human (around 0.3 seconds). Training the model, we saw that the bird started to go towards the opening of the

pillars, following which the model just stuck to this local optima. The algorithm still wasn't exploring enough and we also realized that in an ideal game play, the bird flaps with lesser frequency. Hence the action of 'not flapping' must be promoted during training phase too.

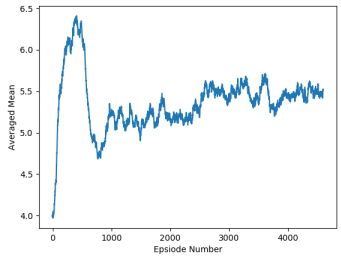

Figure 8: Averaged Mean vs Episodes for Approach 2

Approach 3 Initially our reward set was 0.1, -1, 1 for staying alive in a frame, dying, passing a pillar. From the results of our first approach it is clear that staying alive was taking precedence over passing a pillar. Hence we tried tweaking the reward set and finally settled on 0.1, -1.5, +3.

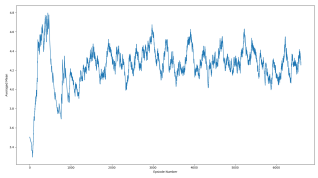

Figure 9: Averaged Mean vs Episodes for Approach 3

Video (Approach 2 and 3 combined): `https://youtu.be/zezgsqWqtlw`
Approach 4 From the results of approach 2, we wanted to promote 'not flapping'. Hence we added a gaussian noise, mean centered at a suitable negative value to the probability of 'flapping' obtained from forward pass of the net. The results improved slightly but still the algorithm was not exploring enough and getting stuck on local optima.

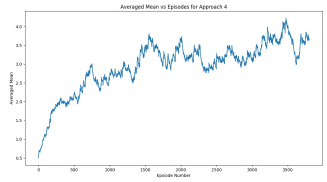

Figure 10: Averaged Mean vs Episodes for Approach 4

Video:
Approach 5 In order to increase exploration by the algorithm, we adopted the following approach for training in the first 1000 episodes - 90% of the time the bird randomly chose to flap or not, while the rest 10% times it acted on the output of the net.

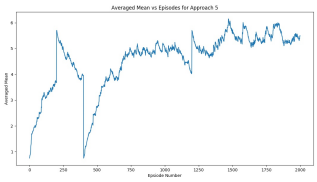

Figure 11: Averaged Mean vs Episodes for Approach 5

Video: `https://youtu.be/hhqxa6lTWxs`
The final results were not as impressive as DQN. The bird only passed through 3 pillars at max in some episodes as opposed to almost super human performance of DQN, The possible reasons behind this are discussed in 6.3.

6.3 Benchmarks

As stated in our aim, our goal was to pit the two approaches - DQN and PG against one another. As stated in the results above, it is evident that the model trained by DQN has a far superior result that PG. PG methods, although more sought after, have seen to be not working on Atari Games as well (as also stated in [3]). Following arguments form the basis of the above stated claims:

- It is evident in literature that although PG methods have faster convergences than DQN, they have a tendency to converge to a local maxima. This is evident in our case and more detailed explanation can be found in the Section 6.2.1.
- One major drawback of PG is the high variance associated in estimating the value of $E[R_t]$ Every time a gradient update is performed, it is an estimate based on a B number of episodes, where B is batch size. It is very likely that this estimate is very noisy and this in turn can have a major impact on the stability of the learning algorithm. DQN tries to determine the associated expected reward for all possible actions in state since it calculates $Q(s, a)$ values. While, PG tweaks the probability of a certain action based on how good/bad the reward of the episode(s) was. As a result PG might be making bad updates because the reward was positive for that episode too. This is clearly seen as the PG model Flappy Bird learns not to crash to the floor (since every passing frame has a positive reward), but is not so good at learning to pass through pillars (since episodes in which that happens are scattered and less frequent).

7 Novel Contributions

- Policy Gradient is not a very explored area in the field of Reinforcement Learning for Atari Games. We tried to implement Policy Gradient for Flappy Bird. We investigated a lot of possible reasons behind the under performing model obtained from PG approach as compared to DQN model. For training the model, to encourage exploration, we tried a variety of solutions including adding Gaussian noise to the output, scaling the reward(similar effect of adding weights) and taking random actions 90% of the time.
- We have also tried to compare the models learnt from DQN and PG approaches. We found that DQN performs better than PG in this case. We have argued in the favor of DQN as stated in Section 6.3.

8 Future Work

- The DQN model is trained on the game with constant distance between pipes on both axes (although the pipe height changes, the pipe gap remains the same) . Consider the game where the distance between the pipes on both axes is random. The trained model was found to give horrible performance on such a game. Training on this random model did not yield much fruit either, as the performance was mostly bad after training for 1,000,000 iterations. An extension can thus be achieving a model that can perform a good enough performance on the said game.
- For policy gradient, an approach to reduce variance in gradient implements Advantage Actor Critic (A2C) algorithm, which optimizes both policy and action value based function
- CNN can also be implemented in PG approach instead of directly giving input to the net

- For PG, in the initial iterations, the policy could be given feedback from some source (such as DQN trained model) to reduce the variance. this would be a mix of Active-Reinforcement learning.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra & Martin Riedmiller (2013) *Playing Atari with Deep Reinforcement Learning* NIPS Deep Learning Workshop 2013

[2] Chen *Deep Reinforcement Learning for Flappy Bird* https://pdfs.semanticscholar.org/b56c/7703337cb9db008422b9b3410c97fff8bb54.pdf

[3] Cai, Ionescu, Jhaveri & Levy *Popular Approaches to Deep Reinforcement Learning* https://github.com/GordonCai/Project-Deep-Reinforcement-Learning-With-Policy-Gradient/blob/master/Report/Project%20Final%20Report.pdf

[4] Shu, Sun, Yan & Zhu *Obstacles Avoidance with Machine Learning Control MeAndrej Karpathy blog thods in Flappy Birds Setting* http://cs229.stanford.edu/proj2014/Yi%20Shu,%20Ludong%20Sun,%20Miao%20Yan,%20Zhijie%20Zhu,%20Obstacles%20Avoidance%20with%20Machine%20Learning%20Control%20Methods%20in%20Flappy%20Birds%20Setting.pdf

[5] Andrej Karpathy *Deep Reinforcement Learning: Pong from Pixels* http://karpathy.github.io/2016/05/31/rl/

[6] Tambet Matiisen *Demystifying Deep Reinforcement Learning* https://www.intelnervana.com/demystifying-deep-reinforcement-learning/