Machine Learning Project: Playing Pong with Deep Reinforcement Learning

Azhar Shaikh-01FB16EEC370
Akshay Singh-01FB16EEC365
Jayanth L-01FB16EEC372

April 23, 2019

1 Introduction

In 2013,Volodymyr Minh, a academic at DeepMind,published a paper with associated co-collaborators at DeepMind which caught the attention of the both the press and the machine learning community. In the paper they developed a system that uses Deep Reinforcement Learning(Deep RL) to play various Atari games, like Breakout and Pong. The system was trained purely from the pixels of an image / frame from the video-game display as its input, without having to explicitly program any rules or information of the game. More unusually, the system detailed in the paper beat human performance across various Atari games using an (almost) indistinguishable architecture across the games, and is based on comparatively simple concepts which have been identified and studied in the Reinforcement Learning (RL) and Machine Learning (ML) fields respectively for decades.
Our aim in this project is to use two very popular RL methods belonging to two very different classes of algorithm namely Policy gradient belonging to Policy based methods and Deep Q Learning(DQN) belonging to value based methods to learn optimal policies to win at a game of pong.

2 Related Work

For policy gradient methods section 13.1 through to 13.4 of chapter 13 of Reinforcement Learning an Introduction by Sutton and Barto served as the source for understanding and helped in the implementation of the algorithm.Another paper Policy Gradients with Pong analyses different neural network architectures for use in solving the game of pong.

For implementing DQN we used the the DeepMind DQN paper as our reference, In their

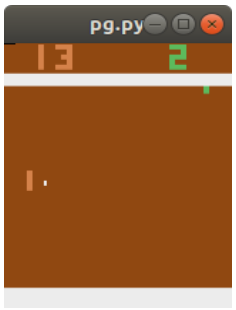

Figure 1: Snapshot of a game state. The green represents the autonomous agent while the orange represents the hard coded AI agent. The goal is to get the ball past the opponent.

implementation they use a deterministic Pong environment while we use a stochastic one.

3 Methods

3.1 Policy gradient

In our implementation we use Monte-Carlo-Policy Gradients also known as the Reinforce Algorithm as the pong environment is episodic in nature. Reinforce tries to maximize the expected return of an episode. Mathematically it can written as follows:

$$arg_\theta max \ E_{\pi\theta}\Big[\sum_{t=1}^{T-1}\gamma^t r_t\Big]$$

Here π is the policy that maps states to actions and is parameterised by θ,T is the length of the sampled episode and γ is the discount factor. The above expression can be represented as maximizing the cost function shown below

$$J(\theta) = E_\pi\Big[\sum_a q_\pi(S_t,a)\pi(a|S_t,\theta)\Big]$$

To maximize the expected return the parameters θ have to found out, this is done by finding the gradient of the cost function with respect to parameter θ and performing gradient ascent using the found out gradient.The gradient of the cost function is given by the following equation:

$$\nabla J(\theta) = E_\pi\Big[G_t\frac{\nabla_\theta\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\Big]$$

Above equation is commonly written as follows:

$$\nabla J(\theta) = E_\pi\Big[G_t\nabla_\theta ln\pi(A_t,S_t,\theta)\Big]$$

Algorithm 1 Monte Carlo Policy Gradient[3]

procedure REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\}$ **do**
    for $t = 1$ to $T-1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
  return θ

The gradient update is performed by

$$\theta_(t+1) = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}$$

Here G_t is the return at time step t, since we are sampling an episode of length T the update is applied for each time step that is for every action taken in that particular episode. We have used a Convolutional Neural Network(CNN) to represent our policy in this project.

3.2 Deep Q-Learning

The elementary idea behind many reinforcement learning algorithms is to estimate the action value function, by using the Bellman equation as a recursive update. In Q learning a non-linear function approximator is used in its place, such as a neural network,in our case we used a convolutional neural network.

$$Q(s,a) = r + \gamma * maxQ(s', a')$$

To obtain the Q values of a particular state we use the screen pixels,they implicitly contain all the usefull game information. To account for speed and direction of the ball we feed in a group of frames at a time,because of this the total number of possible states becomes too big to handle. This is the point where we use deep learning. Neural networks are really good at coming up with good features for highly structured data. So, we have used three convolutional network layers.Input to the network are four 8484 grayscale game frames(screenshots). Outputs of the network are Q-values for each possible action. Q values are real values which makes it a regression task,which is enhanced by using squared error loss,

$$L = (1/2) * [r + maxQ(s', a') - Q(s,a)]$$

The most significant trick is experience replay. During gameplay all the experiences are kept in a replay memory. When training the network, random minibatches from the replay memory are used as an alternative of the most recent transition. This breaks the similarity of following training samples, which otherwise might drive the network into a local minimum.
When we initialize a Q-network randomly, then its predictions are primarily random as well. If we pick an action with the highest Q-value, the action will be random and the agent executes crude exploration. As a Q-function converges, it returns more reliable Q-values and the amount of exploration decreases.

Given below is the algorithm for dqn.

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

4 Models and Data Used

The data used in this project is OpenAI gym namely the atari Pong-v0 environment emulator, the emulator outputs an 210X160 RGB image. We process the image by gray-scaling and reducing it to a 80X80 image and feed the difference of the current frame and previous frame to capture relative motion and increase correlation between images feed into the policy network.The architecture for the CNN used in Policy gradients method consists of 6 layers one convolutional layer followed by flattening layer which is fed into 2 hidden layers with neurons 64 and 32 respectively and output layer with 2 neurons representing action of going up and down.The CNN has 32 filters of size 6X6. All the layers have RELU activation except the last layer which has softmax activation.

For deep q learning we are using the input as 4 , 84*84 images. we pass it through 3 layers of convolutional neural network. First hidden layer is having 32 neurons followed by a relu activation function. Second and third hidden layers have 64 neurons each,followed by a relu activation function.The filter size of first hidden layer is 8*8 , second is 4*4, third is 3*3. Then we have a 512 fully connected neurons, hidden layer,with softmax activation function. Finally we have a output layer which gives out the Q value of every action to be taken.

5 Results and Discussion

Both DQN and Policy gradients were trained on a NVIDIA 740MX GPU. We Observed that DQN takes a lot lesser time to train to beat the hardcoded bot as compared to

Policy Gradients.DQN took 6 hours to train while policy gradients took 20 hours, this could be attributed to the high variance in estimating the gradient of the expected return of an episode in Policy gradient, this leads to instability in training and slows it down.Often times the Policy Gradient agent would get stuck at a local minima and not improve it's score when using epsilon-greedy algorithm for exploration and exploitation, this problem was solved by sampling actions from the softmax distribution obtained at the output of the network.

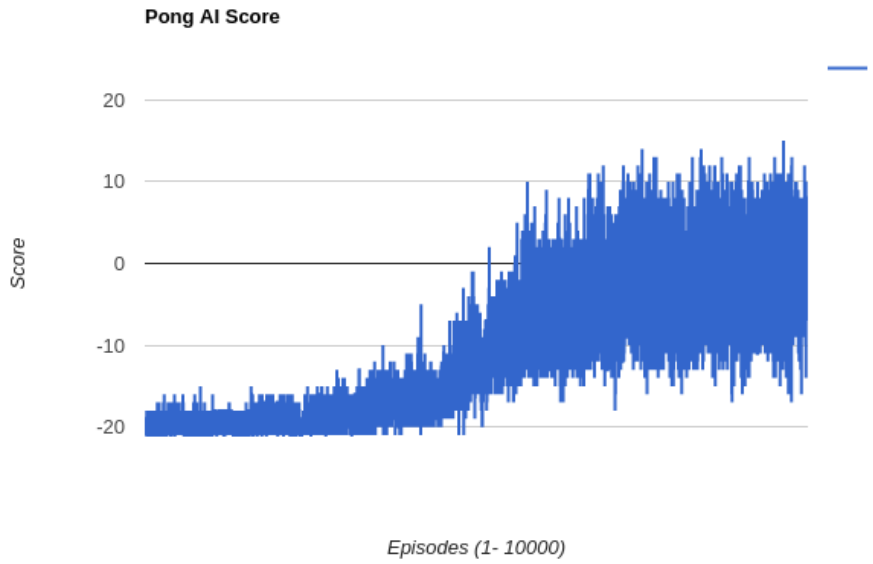

Figure 2: plot of reward vs episodes for policy gradients.We can observe the variation and instability in the received reward per episode in the above graph

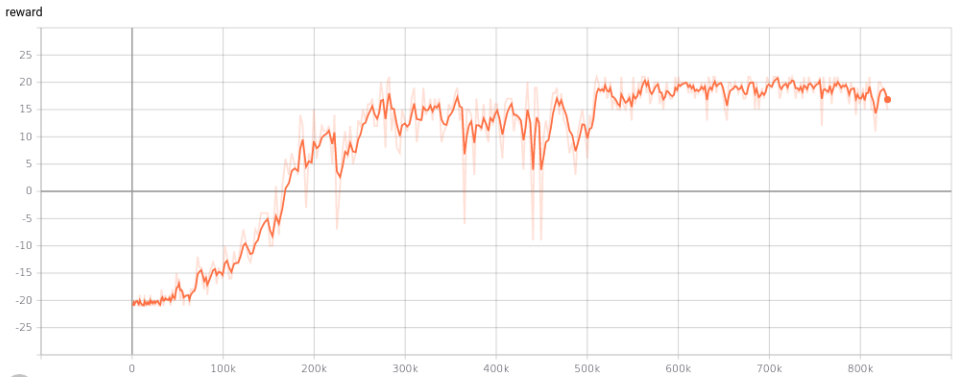

Figure 3: plot of reward vs timesteps for DQN.The training is much more stable in the case of DQN

6 References

6.1 Papers referred

Playing Atari with Deep Reinforcement Learning
Policy gradient method for Pong

6.2 Books referred

Reinforcement Learning: R.S Sutton and A.G Barto