Deep Reinforcement Learning:Pong Player

Shravya Rani Thatipally, Gopi Krishna Chaganti, Nikhil Junneti

Abstract

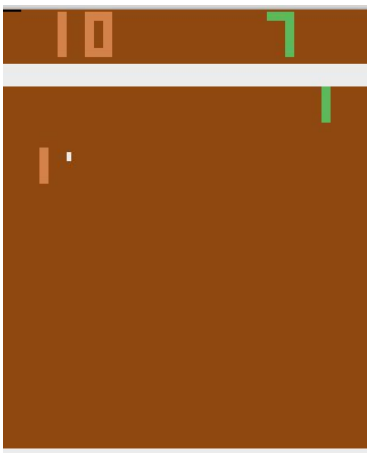
Deep Reinforcement learning is an area which is concerned with how agents take actions in an environment so as to maximize the cumulative ***reward***. In this project we trained a pong player using **Deep Q-learning** and **policy gradient algorithms.** We found that initially policy networks model gave better performance compared to the Q-learning implementation but after running for long time Q-learning outperformed policy networks.

Introduction

i. Problem Definition:

Our goal is to train an agent to play pong game at human level without explicitly handpicking features. In this project we have implemented two types of reinforcement learning techniques to teach our agent how to take actions just based on the raw pixels it received from the environment.

At the low level, we build an AI agent which takes decisions of actions to be performed in every state. States are latest images we get from the game. Each image gives us information about where the pong paddle is on the screen, and in which direction the ball is moving. To enable the agent to learn this information, we feed difference of the latest two frames to our learning model. Our agent can take two actions UP or DOWN from any state. Action UP moves the paddle up and DOWN moves it down. And our agent gets a reward of +1 if the ball passes opponent's paddle without the opponent hitting it and agent gets a reward of -1 if it misses the ball i,e. Ball goes past our agent's paddle. And a reward of 0 is awarded if our agent hits the ball sending to the other side. Our objective is to train our agent to learn to win the games. A game ends when one of the players scores a reward of +1. For each game, our agent may gets a reward of +1 if it wins or gets a reward of -1 if it loses. An episode ends whenever one of the opponents scores 21 points.



ii. Motivation:

Pong player is just a first step towards learning to build complex AI engines. A simple system such as a pong player enables us to learn about the state of the art approaches towards reinforcement learning and build systems that solve complex, high-dimensional problems like robot manipulation, assembly and navigation. AI systems which can learn without hard coded rules perform much better than traditional hard coded programs after sufficient learning time. Such systems can be widely used across all fields ranging from disease diagnostics to space applications for solving complex problems.

iii. Contributions:

We implemented specifically two approaches towards solving reinforcement learning problems in this work

namely Policy gradient method and Q-Learning.

Policy gradient methods are end-to-end algorithms that directly learn policy functions mapping states to actions. An approximate policy could be learned directly by maximizing the expected rewards. Here rewards are the points our agent scores at the end of every game. The parameters of a policy function could be trained and learned under the guidance of the gradient of expected rewards. In other words, we can gradually tune the policy function via updating its parameters, such that it will

generate actions from given states resulting in higher rewards.

Deep Q-Learning (DQN): Deep Q-Learning method is an alternative to policy gradients. It is based on Q-Learning that tries to learn a value function (Q function) mapping states and actions to some value. DQN employs a deep neural network to represent the Q function as a function approximator. The training is done by minimizing temporal-difference errors. A neurobiologically inspired mechanism called "**experience replay**" is typically used along with DQN to help improve its stability caused by the use of non-linear function approximator.

iv. Concepts learnt:

Initially the results from policy networks achieved a higher level of performance compared to Deep Q learning model but after running for approximately 20 hours Q learning model started to show high performance and was ultimately able to learn a far superior policy. Early results of policy networks may be because of modified state representation and reward structure than the change in algorithm. In Q learning, we set up the reward structure to reinforce exactly the behaviour we wanted to achieve, but we did not have to tell the agent how to achieve the goal.

Description

Policy Gradients Algorithm: Our policy network consists of two layer neural network taking image input and giving us the probability of going up (since our action space is two, probability of going down is just 1 minus probability of going up). We then sample an action from this distribution and act upon it.

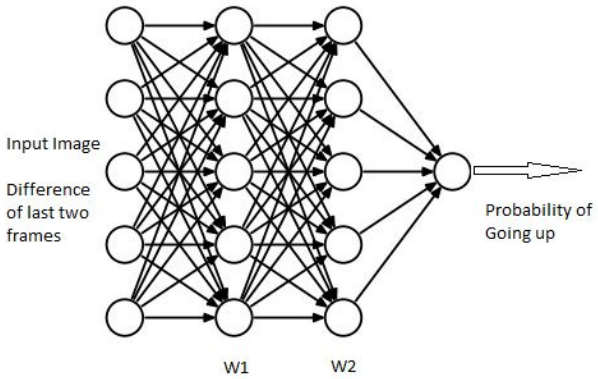


Figure 1 Two layer Policy network

We specifically perform the following computations:

h = W1.x

logp = W2.h

Where h represents hidden layer neuron activations obtained by the dot product of weight matrix for layer 1 and x which is the input matrix (difference frame). And we calculate log probability in the second step which is the dot product of h and weight matrix for layer 2, W2. We then apply sigmoid function on the log probability to get exact probability of going up. W1 and W2 are initialized to random matrices and we update W1 and W2 from the agent's learning experience. Now our problem is to learn W1 and W2 matrices using **Stochastic Gradient Descent**.

In gradient step ideally our objective is to maximize probability of taking good actions. But in this case we don't know whether we are taking right actions yet, but we do receive reward after each step, so we will multiply our action probability with the reward and maximise the resultant function(shown below).

$$\sum_i R_i \log p(y_i \mid x_i)$$

Here Ri is the reward we received from ith step and yi is the action we have taken and xi is the input at step i. Maximising this function encourages all good actions and discourages the network to take bad actions. But this requires reward for all good actions to be positive and bad to be negative. From openAI gym environment we receive -1 reward in case of loss and 1 in case of win and 0 in all other cases. We have assigned the game final reward to all the actions taken before for reaching that state using discounted reward function. Here e denotes final step (win or lose step).

$$R_t = \sum_{k=0}^{e} \gamma^{e-k} r_{t+k}$$

For each episode we will store all input images, hidden layers and rewards received in each step and at the end we apply backpropagation by updating the weight vectors according to the gradient policy function discussed above. For example, at the end of a 21 point game, we get 10 points which means, in 10 of the 31 games, we had reward of +1 and in the other 21 games we lost. Assuming each game has 100 frames, we made 3100 decisions of going up and down and of them, 1000 decisions resulted in a reward of +1 and the remaining resulted in a -1 reward. So we now enter gradients and backprop to update the parameters of the weight matrices such that all those 1000 decisions resulting in a win are taken more often and the decisions resulting in losing are less likely to be taken. This is one policy update and we expect our agent to now play better than before.

On the whole, this method updates weight matrices at the end of every game using the gradients for particular actions and this results in changing weights for actions in a way that actions resulting in higher rewards are more likely to be taken in the future and actions resulting in losing or lower rewards are less likely to be taken in the same game setting.

PreProcessing: In this step, we calculate the difference between two input frames at the start of a game.

Training: We initialized the weight matrices to random numbers and played for 36 hours (roughly 14000 episodes). For each game, depending upon the number of games(experience), our agent made decisions of going up or down. At the end of each episode, we have assigned reward to each decision based on discounted reward method discussed above and ran back propagation to update the weights matrices.

Q learning:

In this method, each action taken in a state is assigned a value (Q-value). Q-value represents the quality or importance of the action taken from a particular state. Our objective is to find these values associated with each state and actions. Once we have these values we can take an action which has higher Q-value in that state. Difference between policy gradient and Q-learning is, in policy gradient we define a policy of reward and update policy periodically but here, we explore the available states and assign Q-values for each state on the basis of rewards received. After considerable exploration, a policy for actions will be formed.

If we are in a state just one action away from the goal state which is the state of highest reward (in pong it is +1 meaning win), action resulting in the goal state from current state has more Q-value compared to other actions.
Our game looks like this,

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here, ri+1 is the reward we get after we perform action ai from state si and action ai from state si results in a game state si+1
In this paper from Deep Mind, Q-value is calculated as follows

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') \middle| s, a \right]$$

In the above equation, s and a are current state and action and upon performing action a, we are in state s'. For calculating Q-value for (s,a) we consider the maximum Q-value of all possible actions from state s' and add it to the reward we get in state s'. Max Q-value is multiplied by gamma(discounted value), the learning rate we assign for our algorithm.

Our input is a 80x80 difference frame. So, each input is 6400 data points and each data point can be in 2 states(Black/White) which means there are 26400 possible screen states. This number is too big for any computer to reasonably deal with. So, to solve this problem, we used deep convolutional nets to compress this image matrix to a space of just 256 points. We then learn Q-values from this output.

When a Q-network is initialized randomly, we will get random predictions initially. If we now pick an action randomly, it will just be a brute force exploration of all states. As a Q-function converges, it returns more consistent Q-values and the exploration part decreases. As time proceeds, it learns all the Q-values for all states and takes actions depending upon the learned Q-values i,e. it takes good actions. So we can say that Q-learning incorporates exploration as the initial part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds. This may lead to the algorithm getting stuck at a local minima or the local best state.

An effective fix for the exploration problem is ε-greedy exploration. The idea is to decrease the exploration space as the learning time increases. In ε-greedy exploration, with probability ε, a random action is chosen otherwise, it goes on with the greedy action with highest Q-value. We first assign a high ε value close to 1 encouraging the model to explore more and as time increases we decrease ε to limit exploration and depend upon Q-value.

Implementation:

We used tensorflow for creating convolutional neural network and we defined our loss as follows:

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s',a')}_{\text{target}} - \underbrace{Q(s,a)}_{\text{prediction}}]^2$$

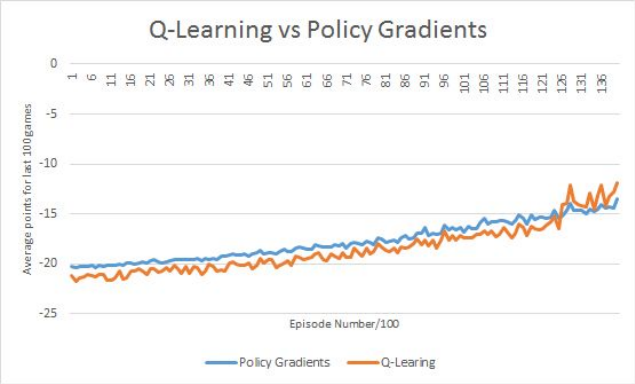
Objective for the network is to minimize the loss function. Tensorflow network takes input and loss function and returns points scored by our agent at the end of every episode. We saved the learning data at regular intervals to enable us to start from that particular point if anything unwanted happens.

Evaluation

We ran both the algorithms for 36 hours and recorded average points gained by both agents for every 100 episodes. In this time roughly 14,000 episodes have been played by both the agents against OpenGym AI player.

In learning curve graph X-axis represents game number/100 and Y-axis represents the average number of points scored at the between (y-1)*100th episode and the y*100th episodes. If our agent loses an episode, he gets (number of games won -21) points. If score is -20 in an episode means, our agent scored one point winning only one game among all games of the episode.

Learning rate for Q-Learning and Policy Gradients:

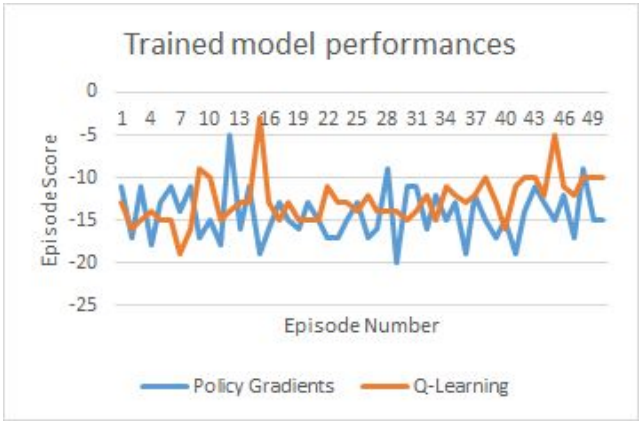


Comparison shows the learning rate difference between Q learning and policy gradient algorithm. We have plotted episodes on X-axis and Average points for 100 games on Y-axis. Here we can observe in the first 2000 episodes, average points is near to -21 indicating our agents learnt almost nothing in both implementations losing almost all games apart from scoring 1 or 2 points in some games. From then onwards their scores increased and it starts learning. Initially, we observe that policy gradients perform well compared to Q learning but after certain time Q-learning outperforms policy gradients and the trend continues. If we see at the end average points over 1000 episodes for Q-learning agent from 13000 to 14000 is almost -11.47 meaning our agent scored more than 9 points on average in every episode from episode number 13901 to 14000.

We can see our agent's consistent performance over time. If given some more time around 3 days it can play very well.

Performance of the models:

Finally our trained models are played against same OpenAI gym agent each for 50 episodes. Following graph shows the comparison between both the algorithms in each episode. We can observe that on an average Q learning dominates policy gradient in most of the episodes.



Conclusion

We trained both the models for around 30000 episodes. Initially the results from policy networks achieved a higher level of performance compared to Deep Q learning model but after running for approximately 20 hours Q learning model started to show high performance and was ultimately able to learn a far superior policy. Early results of policy networks may be because of modified state representation and reward structure than the change in algorithm. To use policy gradients efficiently it was necessary to use some prior knowledge to set up a reward structure that presented the agent with a gradient but this could lead to certain errors. A poor step could collect more rewards initially but fail in achieving the primary task.

In Q learning, we set up the reward structure to reinforce exactly the behaviour we wanted to achieve, but we did not have to tell the agent how to achieve the goal. Thus, less prone to errors resulting from inappropriate reward configurations.

Credits

https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

https://studywolf.wordpress.com/2012/11/25/reinforcement-learning-q-learning-and-exploration/

http://www.danielslater.net/2016/03/deep-q-learning-pong-with-tensorflow.html

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

http://karpathy.github.io/2016/05/31/rl/