Exploring Deep Q Network for Atari Game

Sumanyu Garg
Data Science
Indiana University
Bloomington, IN 47408
sumgarg@iu.edu

Abstract

In this project, I experiment with the Deep Q Networks on Atari Environment. These networks are able to learn policies from the input using reinforcement Learning. The network is trained with a variant of Q-learning, with input as raw pixels from the screen and the output is action value function which estimates future rewards for each action.

1 Introduction

Learning control policies directly from images or some other high dimensional sensory input was not possible before the advent of deep learning.

However, even with deep learning at our disposal, there are several challenges in applying to solve reinforcement learning problems. Deep Learning methods require large amounts of training data and how that can be mapped to a reinforcement learning problem is not immediately clear. One might argue to learn a model of the environment by estimating transition probabilities and reward functions but that is almost impossible for most environments due to very large state space and stochasticity in the system. Another issue with deep learning-based methods is that we need our training samples to be independent whereas in reinforcement learning, we get a sequence of states which have high correlation between them. Deep Q Networks [1], along with a variant of Q-learning [2] helps us to tackle these challenges.

2 Background

In this project, I will be experimenting with Atari environment [3], specifically the *SpaceInvaders-v0'* environment.(However, the implementation is well organized to be able to tackle other environments as well with minor modifications that will be needed as per the dynamics of different environments.)
In general, any reinforcement learning problem with single agent consists of an environment, and at each time step, the agent selects an action a_t from the agent's action space. (There are 6 actions in *SpaceInvaders-v0* environment, {*0:no action}, {1:fire}, {2:move_right}, {3:move_left}, {4:move_right_fire}, {5:move_left_fire}).* The agent gets to only observe the images of the current screen x_t.

The goal of the agent is to interact with the environment by selecting actions in a way that maximizes future rewards. I have considered discounted rewards such that the discounted *return* at time *t* can be written as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ where *T* is the time-step at which the game ends.
We can also define the optimal action value function, which can be defined as the maximum expected reward achievable by following an optimal policy, using *Bellman equation* as follows.

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

In this project, this action value function is estimated using a neural network as a function approximator. We can use one neural network for each action or alternatively use one single neural network that will approximate the action value function for each action.

$$Q(s,a;\theta) \approx Q^*(s,a).$$

This network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i.

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right] ,$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \middle| s, a \right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences *s* and actions *a* that the authors of the original paper referred to as *behaviour distribution.* One important thing to note is that target depends on the network weights which is in contrast with what the target is in supervised learning, which are fixed for all iterations during training. Now, if we differentiate the loss function with respect to the weights, then the gradient can be written as follows.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

However, how can we compute these gradients practically? It is computationally infeasible to calculate the full gradients and hence we use stochastic gradient descent to compute them This algorithm is same as the *Q-Learning* algorithm. Some important things to note are that this algorithm as *model-free*, which means that we don't estimate the dynamics of the environment and instead solve directly from the samples. Another thing to note is that this algorithm is *off-policy*, which means it learns an optimal policy by following a behaviour distribution. Now how to select a behaviour distribution? In practice, behaviour distribution is chosen to be an ϵ-greedy strategy which allows exploration by choosing the optimal action with probability 1- ϵ and random action by probability ϵ.

3 Deep Reinforcement Learning

Deep Reinforcement Learning uses something called as *experience replay* to store the agent's experience at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in a dataset $\mathcal{D} = e_1, ..., e_N$ during each episode. The weights of the network are updated using Q-learning updates using samples of

experience, $e \sim \mathcal{D}$, drawn at random from the experience replay dataset. After performing experience replay, the agent selects and executes action according to ϵ-greedy strategy.
This allows for the experiences to be used in many weight updates and allows for greater data efficiency. Moreover, the samples are not correlated as the correlation is broken due to randomized sampling which acts as a training dataset for updating the network. This in turn reduces the variances of the updates. The most important thing is that, when learning on-policy, the current parameters determine the next data sample that the parameters are trained on. Due to this, it is possible that the parameters might stuck in some local minima and do not converge to the desired optimal solution. Therefore, we use off-policy learning, which in fact becomes necessary due to the use of experience replay to update the weights and not the correlated samples. The diagram below, taken from the original paper, shows the complete algorithm.

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

4 Pre-Processing and Model Architecture

OpenAI gym environment for Atari games returns a RGB image as the state of the environment. However, we don't really need colored frames to capture the information in those frames. Therefore, each frame is converted into grayscale as the first preprocessing step. Cropping of the frame, to get a 185x95 size frame, is also done in order to capture only the relevant information.

The network takes input as the processed image and outputs the action value function for each action. The network architecture consists of one convolution layer and 2 linear layers due to limited computational resources (however, one can experiment with a much more complex network in order to get much better results). The first layer is a *convolutional layer* with 64 output channels, each kernel of size 3x3 and stride as 1. The second layer is a *linear layer* with 64 neurons. The final layer is the output layer with number of outputs as the number of actions, the agent can perform (6 in the case of '*SpaceInvaders*').

5 Experiments

I have conducted experiments only on the 'SpaceInvaders' environment due to limited computational resources and strict timeline. However, the network architecture is robust enough to be used for other games as well. The original paper scales all the positive rewards to be 1 and all negative rewards to be -1. However, in my implementation, I haven't done so. I wanted to experiment with the original reward setting instead of modifying it. However, one important change that I have done, is to give a large negative reward when the episode ends. This will avoid the agent to actions which lead to end of the games or episodes. This has been done only during the learning part of the algorithm. While storing the experience into replay buffer, I have stored the original rewards without modifying them.

During the training, I have used the RMSProp algorithm with minibatches of size 32. The behaviour policy during training was ϵ-greedy with ϵ varied exponentially from 0.95 to 0.05 over a course of 500 episodes. Since the number of episodes is very less (restricted to this number only due to limited computational resources), the epsilon decreased only till 0.1. Also, the original paper decreased epsilon across each frame, whereas I have experimented by decreasing epsilon for each episode.

I have also used a frame-skipping technique which is generally used for Atari games. More precisely, the agent selects actions on every k^{th} frame instead of every frame, and the last action is repeated for k skipped frames. I have used $k=3$ which is same as the one used by the original paper.

6 Results

Figure 1 shows the results during training of the network. We can see from the plots that the agent is learning to behave in the environment in order to maximize its total expected reward.

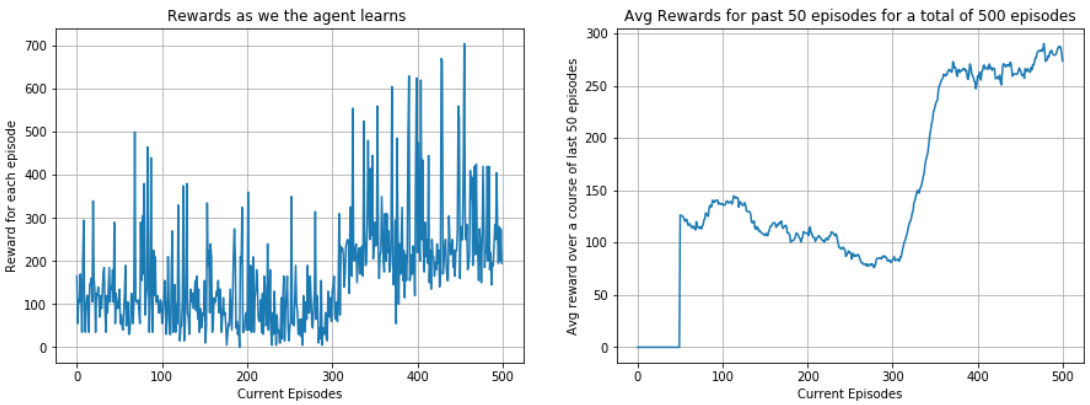

Figure 1: The plot on the left shows total reward per episode on *Space Invaders* environment during training. The plot on the right shows average reward for the past 50 episodes.

The plot on the left in figure 1 shows the total reward that agent is able to get during each episode. The agent manages to achieve a maximum reward of 705 at 456th episode. The

agent starts with taking random actions and gradually it starts choosing actions which lead to better states and hence better states. However, we can see lot of randomness in the total reward for an episode. This is due to the extreme stochastic nature of the *SpaceInvaders* environment. In plot on the right-hand side, average rewards for past 50 episodes has been plotted. Initially, I tried to evaluate policy after 10 episodes. But this was computationally expensive. Therefore, I plotted the average reward for the last 50 episodes. Important thing to note here is that the policy is changing with each episode and therefore it is not same as evaluating policy after every few episode. Nonetheless, we can see that the as the number of episodes is increasing, the average reward is also increasing which is indicating that the agent is trying to learn the optimal policy.

7 Conclusion

In this project, I explored Deep Q Networks on Atari environment. The agent was able to learn a good enough policy and was taking actions which will lead to better rewards instead of just taking actions randomly. There were many difficulties that I had to face during this project. I read the paper on Deep Q Networks to gain insights of how these networks work. In the process, I also learned about the importance of experience replay and off-policy learning in the case of deep q networks. Theoretical understanding is one thing and actually implementing is another. I faced many problems during implementation, mainly on how to divide the code into different sections to make it self-explanatory and extendable to other environments as well. If I were to start this project from scratch, I would try to extend my work to use deep double Q learning [3] which is an extension of double Q Learning using neural networks. I will also try to compare the results of different algorithms across different environments. I was unable to do this in this project due to limited resources and a time constraint. But I would surely like to try out these in future.

8 References

[1] *Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller*. Playing Atari with Deep Reinforcement Learning.

[2] *Christopher JCH Watkins and Peter Dayan*. Q-learning. Machine learning, 8(3-4):279–292, 1992

[3] *Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba.* OpenAI Gym

[4] *Hado van Hasselt, Arthur Guez, David Silver*. Deep Reinforcement Learning with Double Q-Learning.