Class Project Report on Implementation of Double Deep Q-Network with Prioritized Replay.

Anuj Doshi
ME-GY 7973 - Optimal and Learning Control for Robotics
Dept. of Mechanical Engineering
New York University
ad5092@nyu.edu

***Abstract*—Throughout the duration of the course, we have come across many interesting algorithms for optimal control of robots. In this project, we have implement one such algorithm, Deep Query Network (DQN) with a minor change by adding another DQN agent and prioritizing selection of values from the replay buffer. [1].**

I. Introduction

The use of Reinforcement Learning (RL) algorithms, its applications and the abundance of benefits it provides in the multitude of industries are already widely known. Assuming familiarity with such algorithms, we will directly focus on Q-Learning and more specifically, by using neural networks for estimation of weights of the model at every step, making the network a deep Q-network. DQN is an off-policy, bootstrapped and model-free algorithm. As we proceed with the project we will understand the derivation of each of these terms with respect to DQN. After realizing the shortcomings of Q-learning based on a Q-Table growing out of proportion exponentially for each adding variable, scientists came up with a with DQN, where the Q values would be derived from a neural network. The DQN agent, much like any other online RL agent, updates its parameters incrementally as it observers newer experiences. In this process, data is discarded immediately after a single update. As we have seen in our lectures, this presents itself with two major problems: *a* We may be sampling strongly co-related values over and over and *b* the rapid forgetting of possibly rare experiences that would be useful later on.

The solution to both of these problems is demonstrated in the DQN algorithm (Mnih et al., 2013; 2015 [2][3]) which uses the *experience replay* (Lin, 1992 [4]) to stabilize the training of the value function. In this project we try to implement the prioritized experience replay that can try to make experience replay more efficient and effective by prioritizing the transitions.

II. Tools and libraries

A. OpenAI gym and model parameters

To demonstrate it, we implement the algorithm on the Acrobot-v1 environment, a double inverted pendulum model from the OpenAI gym environments. OpenAI gym is an open source interface to reinforcement learning tasks. It acts as a toolkit for developing and comparing reinforcement learning algorithms. As per its description, “”Acrobot is a 2-link pendulum with only the second joint actuated. Initially, both links point downwards. **The goal is to swing the end-effector at a height at least the length of one link above the base.** Both links can swing freely and can pass by each other, i.e., they don’t collide when they have the same angle. This height has been set by a black link on the rendered output, which, our algorithm must cross.

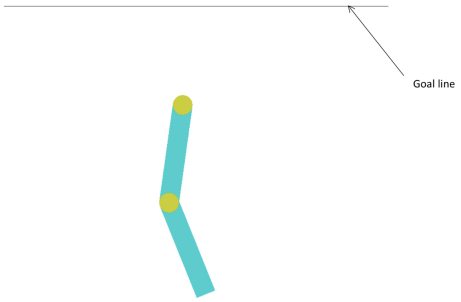

Fig. 1. A rendered image of the environment with the goal line

OpenAI gym makes it very simple to deal with the environment. The environment is a representational system of the physical model with its animations and all governing parameters. It takes in the action for the current state and gives back the next state value in the form of *observation*, the reward achieved at that stage and if the episode is done. Done is a boolean which gets true for when either a 500 time steps period is over or the link reaches the required height for each episode in this environment. The reward is accrued as -1 for every step for which the goal is not reached and 0 for the terminal step. The Acrobot model has 6 states for the two links, represented as the state of the $\sin()$ and $\cos()$ of the two rotational joint angles and the joint angular velocities: $[\cos(\theta 1)\sin(\theta 1)\cos(\theta 2)\sin(\theta 2)\dot{\theta}1\dot{\theta}2]$. For the first link, an angle of 0 corresponds to the link pointing downwards. The angle of the second link is relative to the angle of the first link. An angle of 0 corresponds to having the same angle

between the two links. A state of $[1, 0, 1, 0, ..., ...]$ means that both links point downwards. The action is either applying +1, 0 or -1 torque on the joint between the two pendulum links. The entire environment and its performance can be visually seen by the render command of the environment (*env.render()*).

B. Tensorflow and other libraries

The code is run on a jupyter notebook that is attached with this report. For the development of the neural network Tensorflow library is used. Tensorflow is an open source platform for machine learning that helps in easy creation and management of neural networks in an algorithm. We see definitions of various model, classes and function in the attached jupyter notebook, therefore we will not focus on the code in this report. To track the computational errors and the step by step running of the code, we have used the *IPython* debugger (ipdb). This has served as the most important tool in overcoming all the errors.

III. Algorithm

This is a typical RL problem that involes an Agent , interacting with an Environment with a goal of selecting the actions that will maximize its long term reward, for each episode of interaction with its environment. The algorithm that we will be implementing to try out Double DQN with prioritized experience replay can be seen as follows.

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Fig. 2. Double DQN with prioritized learning algorithm. *Tom Schaul et.al,2016*

Before we dive into the algorithm and it's working, lets first focus on the why we select two networks in the algorithm Focusing on the Bellman's equation for calculation current $Q(state, action)$ pair. That is represented by:

$$Q(s_t, a_t) = r_{t+1} + \gamma * \max(Q(s_t + 1)) \tag{1}$$

where query Q is a pair of state s and action a values at time step t in an episode and γ is our discount factor. As we can see, in this equation for every incremental update of function Q at time t, there is a change in $Q(t_1)$, therefore this leads to the function being dependent on itself. Therefore in cases where there is an instantaneous change in parameters of the variable at time t, there is a change in time $t+1$, but since our next state values depend on this state and so on, we may have sometime miss out on values because of this quick update. To deal with this problem we come up with a target Q network as we have seen during the course. This model mimics the design of the local Q network but updates itself from the local network only after few iterations and not at every time step. The Q-values from this state are then fed to the environment.

This double network arrangement also solves another very important problem: Over-estimation of the Q-values. As we see from our Bellman's equation in (1), our current state Q value is derived from an estimate of the $\max$ of the next time step Q value. But at the start, we initialize this to be completely random. This will make the the state Q values to be very inaccurate and we could have an overestimation of the true Q value. Therefore we implement another network to avoid this. In this setup, one network provides the action for the next state in the form $Q(s_{t+1}, a_{t+1})$ and the other provides us with the Q values for the for the next state $Q(s_{t+1})$. This way we put a secondary check on the selected action leading to the next Q value. The selection of a network for either of the tasks can be understood from this paper on Deep Reinforcement Learning with Double Q-learning [5]. Without running into too much details we can see understand intuitively that this network selection then changes our agent too, making it a Double DQN agent instead of a normal DQN agent.

Now that we have a better understanding of the dual network idea and the use of a Double DQN agent, we can focus on the other highlight of this paper, prioritized learning. We already know the importance of have an experience replay, but what is even more important is the selection of the experiences that we wish to replay. We know that not all experiences carry the same importance for our output. Some experiences can cause bad actions and some can withhold rare information of a very rare state that might lead to a better solution by application of the correct action. Therefore, it is important to prioritize the experiences that we use for training.

Let's take a lot at this in the context of our OpenAI gym environment. We now know that the environment withholds certain information about the model. Upon application of an action, the system changes and the environment produces change in corresponding information. This information in our case in the form of the next state, rewards, done and info. This helps us in understanding what action to chose best. A tuple of this information can also be used to prioritized experience. In this context, we could get the learning value of each tuple and use that value to prioritize the experience that we will replay next.

To get a better understanding of this idea, lets focus on the error that we get for predicted Q value for the state action pair and the Q-target value calculated from the Bellman's equation. The absolute value of this error can help us in deciding the priority of each experience in the replay buffer. We can select this by taking the Priority = absolute(error) + offset. The use of offset over here is to make sure that when we have a 0 zero, the priority does not tend to 0. This makes the

selection more realistic. Furthermore, we can convert these weights for every tuple to be equal to the probability of choosing that tuple from the batch. Which can be done by selecting the weight of that tuple over all others. But due to the approximate nature of the error, this probability could be too high or too low. This can be seen in the Prioritize Experience Replay paper mentioned earlier. Therefore we introduce another hyper-parameter α such that the sampling method interpolates between pure greedy prioritization and uniform random sampling. The probability of such a sampling for transition i can be shown as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2}$$

where $p_i > 0$ is the priority of transition i. The exponent α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case.

Even though the prioritizing looks good so far, it does comes with a few limitations. Primarily, the estimation of the expected value with random updates on those updates corresponding to the same distribution as its expectation. Prioritized replay introduces bias because it changes this distribution in an uncontrolled fashion, and therefore changes the solution that the estimates will converge to (even if the policy and state distribution are fixed). We can correct this bias by using importance-sampling (IS) weights that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$.

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta \tag{3}$$

Therefore, we have our prioritized experience replay. The selection of $\alpha and \beta$ can be a major influencing factor in the speed and performance of the whole algorithm and is discussed in much detail in the paper but it out of the scope of this report.

IV. Implementation and Results

The given algorithm was implemented as presented in fig(2) in the attached jupyter notebook. We can see that this policy is *model-free*. Also, in this case the previous estimates are based on other learned estimates up to that point, i.e. they *bootstrap* and since there is always some exploration going on because of the hyper-parameter epsilon, which balances the exploitation/exploration ration, this algorithm also becomes and *off-policy algorithm*.

Upon the implementation of the algorithm we see that the agent slowly starts to interact with environment, exploring many possible options initially. We find absolutely no improvement in the initial few episodes as epsilon is very high and therefore it will probably only take random steps to initialize an episode. As and when epsilon reduces it starts to exploit it possible options and we can visually see the agent's performance improving quickly. The same can also be noted in the graph for one of the 100 episode instances below.

As can be seen from the graph, the agent learns quickly but takes sometime to master the environment. This is just one of the random instances that we are tried. Since the starting position is always random there is no guarantee that the agent will reproduce the same result twice. But upon trying the same algorithm multiple times for the same values we were able to get very close results.

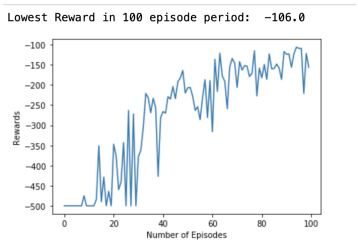

Fig. 3. Rewards at every episode for a 100-episode instance.

Overall results: Highest reward reached: -68.
Highest Reward in a 100- Episode Instance: -91.
Average Reward for 100-Episode Instance: -258.

These result show very satisfactory performance. We can also see the results in the attached video.

References

[1] Schaul, T., Quan, J., Antonoglou, I., Silver, D. (2016). Prioritized Experience Replay. CoRR, abs/1511.05952.

[2] Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A, Veness, Joel, Bellemare, Marc G, Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K, Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wier- stra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] Lin, Long-Ji. Self-improving reactive agents based on reinforcement learning, planning and teach- ing. Machine learning, 8(3-4):293–321, 1992. K. Elissa, "Title of paper if known," unpublished.

[5] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double Q-Learning. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16). AAAI Press, 2094–2100.