Reinforcement Learning - Gym Environments

Sharmistha Swasti Gupta

1 Introduction

Reinforcement learning algorithms are widely used in sequential decision making tasks. In this work, we explore two reinforcement learning algorithms - REINFORCE with Baseline, and Deep Q-Learning (DQN) with Experience Replay, on two environments - CartPole (self implemented) and Acrobat-v1 (through OpenAI Gym). We evaluate the performance of these algorithms through the standard metrics specified in the environments, and present appropriate analysis and learning curves over 20 runs.

2 Environments

2.1 CartPole-v0 (Self Implemented)

CartPole [2] is a benchmark environment used for reinforcement learning tasks. Though it is available for use through the OpenAI Gym Classic Control environments, we implement it from scratch and use version v0 [3]. Figure 1 provides a code snippet of the environment implemented in Python 3.

```
def transition(s, a):
    x, v, angularx, angularv = s[0], s[1], s[2], s[3]
    #defining constant values
    g = 9.8 #gravity
    mc = 1 #mass of cart
    mp = 0.1 #mass of pole
    lp = 0.5 #length of pole
    F = 10 #force magnitude
    dt = 0.02

    #dynamics
    m = mc + mp
    F = F if a == 1 else -F #syntax
    temp = (F + mp*lp*(angularv**2)*np.sin(angularx))/m
    angularacc = (g*np.sin(angularx) - np.cos(angularx)*temp)/(lp * (4/3 - mp*np.cos(angularx)**2/m))
    acc = temp - mp*lp*angularacc*np.cos(angularx)/m

    #updates
    x += v * dt
    v += acc * dt
    angularx += angularv * dt
    angularv += angularacc * dt

    s_ = (x, v, angularx, angularv)
    return s_

def terminal(s):
    s = s.view(-1).tolist()
    x, v, angularx, angularv = s[0], s[1], s[2], s[3]
    if s[0] <= -2.4 or s[0] >= 2.4:
        return True
    if s[2] <= -0.2095 or s[2] >= 0.2095:
        return True

def reward(s, a, s_):
    return 1
```

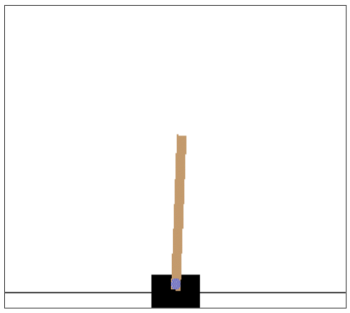

Figure 1: (left) Code Snippet of CartPole in Python 3 (right) Thumbnail of the rendered CartPole environment

2.2 Acrobot-v1 (OpenAI Gym)

Acrobot [1] is also a benchmark environment used for reinforcement learning tasks. For our experiments, we have used the version v1 available for import into Python 3 through the gym package. Figure 2 shows a thumbnail of the Acrobot environment, when rendered.

Figure 2: Thumbnail of the rendered Acrobot environment

3 Algorithms

In this project, we choose to explore policy parameterization methods as opposed to tabular methods. Our initial experiments using tabular methods did not perform well on these environments because discretization of the large state spaces led to huge value tables, thus increasing the runtime of the algorithms substantially and lowering the scope for hyperparameter tuning given the time and resource constraints. Parameterizing the policies eliminates the need to maintain a table, consequently eliminating the need to discretize the state space. This led to better observed performances in both the environments.

3.1 REINFORCE with Baseline

The original REINFORCE algorithm, is in some sense, a Monte Carlo method for learning policy parameters, because it is episodic in nature. A differentiable policy parameterization is initialized, and an episode is generated following this policy. At the end of the episode, returns for each state in the trajectory are generated. We know, that performance in the episodic case is defined as: $J(\theta) = v_{\pi_\theta}(s_\theta)$ [6]. The gradient update for this performance metric is modified such that an update at a time step involves only the action taken at that time step, as opposed to all possible actions, resulting in a final update of:

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

REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 3: Pseudocode for REINFORCE with Baseline [6]

Thus, a policy parameter update in the algorithm looks like:

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

In case of REINFORCE with a Baseline, as the name suggests, additionally, there is a baseline in the form of state value estimates. These estimates are learnt in a manner similar to the policy parameters, but by minimizing the mean squared error loss.

$$\delta^2 = (G - \hat{v}(S_t, w))^2$$
$$\frac{d\delta^2}{dt} \propto \delta \bigtriangledown \hat{v}(S_t, w)$$

Thus, the update rule becomes:

$$w_{t+1} = w_t + \alpha^w \delta \bigtriangledown \hat{v}(S_t, w)$$

The policy parameter update rule in this algorithm becomes:

$$\theta_{t+1} = \theta_t + \alpha^\theta [\delta \frac{\bigtriangledown \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)}]$$

These updates are performed over multiple episodes, until the environment is considered solved. Figure 3 shows the pseudocode for this algorithm. For our experiments, discount factor, $\gamma = 1$ always.

3.2 Deep Q-Learning (DQN) with Experience Replay

The Deep Q-Learning algorithm, with Experience Replay, is a non linear function approximation algorithm, that aims to tackle instabilities in Q function approximation. We use a neural network as our approximator [4].
We initialize a memory queue, with a recall size of 128. This forms a mini-batch over which we perform updates. We initialize a policy and a target network, with the aim of performing soft updates on the target network, using parameters from the policy network, with a certain delay.
We then generate trajectories and append them to the memory queue. Next, we randomly sample a mini-batch, and generate estimates of the Q function as per the current policy. We then perform an update on the target estimates using Q function of the next states. Finally, we perform a gradient descent step to minimize the loss between the current Q function estimates and the target Q function estimates. The target estimates (target network) are then updated softly using parameters of the current Q function estimates (policy network), with a delay of τ.
These updates are performed over multiple episodes, until the environment is considered solved. Algorithm 1 shows the pseudocode for this algorithm [4][5]. For our experiments, discount factor, $\gamma = 1$ always.

Algorithm 1 DQN with Experience Replay

```
Initialize experience replay vector, memory
Initialize current Q function, policy
Initialize target Q function, target
while True do
    Initialize s (first state of episode)
    while s not terminal (for each time step) do
        With probability ϵ, select a random action a
        else select a = argmax_a policy(s)
        Perform a and observe the next state s_ and reward R
        Add (s, a, s_, R) to memory
        Sample a random mini-batch from memory
        Store current Q function estimates from policy(s)
        Store target Q function estimates using Q function of next states
        Perform a gradient descent step on the loss between estimates
        Perform a soft update on the target Q function estimates
    end while
end while
```

This algorithm attempts to remove correlation between observations by randomly sampling data from the experience buffer. Also, performing delayed updates as the final step ensures that the parameters change slowly [4]. The parameters change with a probability of τ and retain the old parameters with a probability of $1 - \tau$. This greatly improves stability [7].

4 Experiments and Results

4.1 REINFORCE with Baseline on CartPole-v0

Hyperparameters: $\alpha^{\theta} = 1e-3, \alpha^{w} = 1e-3$
These learning rates are the standards used in many implementations on this environment. We tried a higher rate, but it returned results with instabilities. We were able to arrive at reasonable hyperparameters for this experiment in a short amount of time.
Figure 4 shows the learning curves plotted. CartPole v0 is considered solved when the pole stays within the desired range for an average of 195 steps or higher out of the maximum allowed 200 steps. The first graph shows a decreasing slope indicating an increase in the time steps with an increase in episodes. This supports our experiment because it indicates that with time, the pole is staying upright for an increasing number of time steps. The second graph shows that the average number of steps in an episode gradually increases to 200, converging to an average of 197.564 over 20 runs. Thus, we have successfully solved CartPole-v0 using REINFORCE with Baseline.

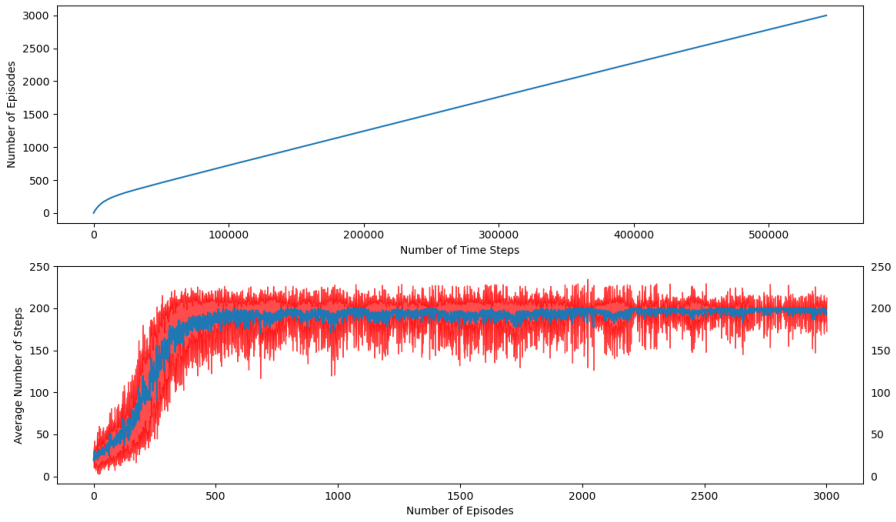

Figure 4: (above) Number of episodes, as a function of the average cumulative actions (time steps) taken up to that episode, across 20 runs (below) Average number of steps in an episode, as a function of number of episodes, across 20 runs. Also shown is the standard deviation (in red).

4.2 REINFORCE with Baseline on Acrobot-v1

Hyperparameters: $\alpha^{\theta} = 8e-4, \alpha^{w} = 8e-4$
We began with the learning rates of 1e-3 as in the previous experiment. However, that returned results with fluctuations and instabilities. We then tried with 5e-4, but the learning seemed to be too slow. Finally, we increased the learning rate to 8e-4, and were able to achieve good performance.
Figure 5 shows the learning curves plotted. Acrobot v1 is considered solved when the lower link reaches the desired height in an average of 100 steps or lower out of the maximum allowed 500 steps. The first graph shows an increasing slope indicating a decrease in the time steps with an increase in episodes. This supports our experiment because it indicates that with time, the link is being able to reach the desired height in less number of time steps. The second graph shows that the average number of steps in an episode gradually decreases to 100, converging to an average of 93.395 over 20 runs. Thus, we have successfully solved Acrobot-v1 using REINFORCE with Baseline.

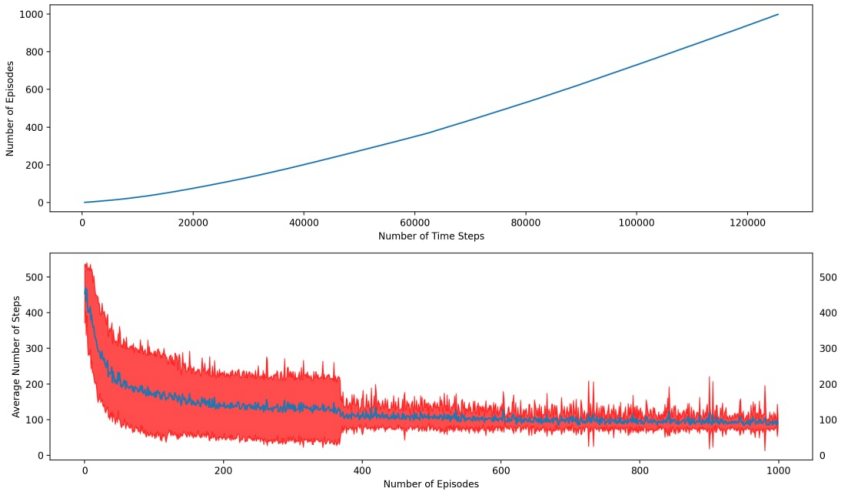

Figure 5: (above) Number of episodes, as a function of the average cumulative actions (time steps) taken up to that episode, across 20 runs (below) Average number of steps in an episode, as a function of number of episodes, across 20 runs. Also shown is the standard deviation (in red).

4.3 DQN with Experience Replay on CartPole-v0

Hyperparameters: lr = 1e-4, $\epsilon = 0.9$ with a decay of ϵ*.999 at every time step until a value of 0.05, $\tau = 0.005$, number of hidden units = 64
We chose a learning rate of 1e-4 because it seemed to be working reasonably well for this environment previously. We chose the value of ϵ in a manner that a run begins with a higher exploration rate, and at every time step, since it observes and learns something about the environment, the rate decays by a small amount. This happens till the value touches 0.05, at which point the exploitation is largely preferred. The value of τ was referred from [5]. Since the state tuple was of length 4, we chose a smaller number of hidden units.
Figure 6 shows the learning curves plotted. As mentioned previously, CartPole v0 is considered solved when the pole stays within the desired range for an average of 195 steps or higher out of the maximum allowed 200 steps. The first graph shows a decreasing slope indicating an increase in the time steps with an increase in episodes. This supports our experiment because it indicates that with time, the pole is staying upright for an increasing number of time steps. The second graph shows that the average number of steps in an episode gradually increases to 200, converging to an average of 198.7285 over 20 runs. Thus, we have successfully solved CartPole-v0 using DQN with Experience Replay.

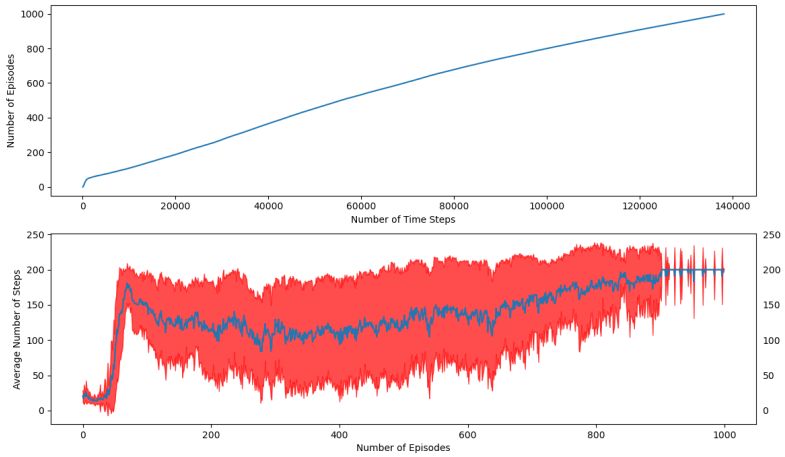

Figure 6: (above) Number of episodes, as a function of the average cumulative actions (time steps) taken up to that episode, across 20 runs (below) Average number of steps in an episode, as a function of number of episodes, across 20 runs. Also shown is the standard deviation (in red).

4.4 DQN with Experience Replay on Acrobot-v1

Hyperparameters: lr = 1e-4, epsilon = 0.9 with a decay of epsilon*.999 at every time step until a value of 0.05, τ = 0.005, number of hidden units = 128
We chose a learning rate of 1e-4 because it seemed to be working reasonably well for this environment as well. We chose the value of ϵ in a manner that a run begins with a higher exploration rate, and at every time step, since it observes and learns something about the environment, the rate decays by a small amount. This happens till the value touches 0.05, at which point the exploitation is largely preferred. The value of τ was referred from [5]. Since the state tuple was of length 6, we chose a larger number of hidden units.
Figure 7 shows the learning curves plotted. Acrobot v1 is considered solved when the lower link reaches the desired height in an average of 100 steps or lower out of the maximum allowed 500 steps. The first graph shows an increasing slope indicating a decrease in the time steps with an increase in episodes. This supports our experiment because it indicates that with time, the link is being able to reach the desired height in less number of time steps. The second graph shows that the average number of steps in an episode gradually decreases to 100, converging to an average of 95.1725 over 20 runs. Thus, we have successfully solved Acrobot-v1 using DQN with Experience Replay.

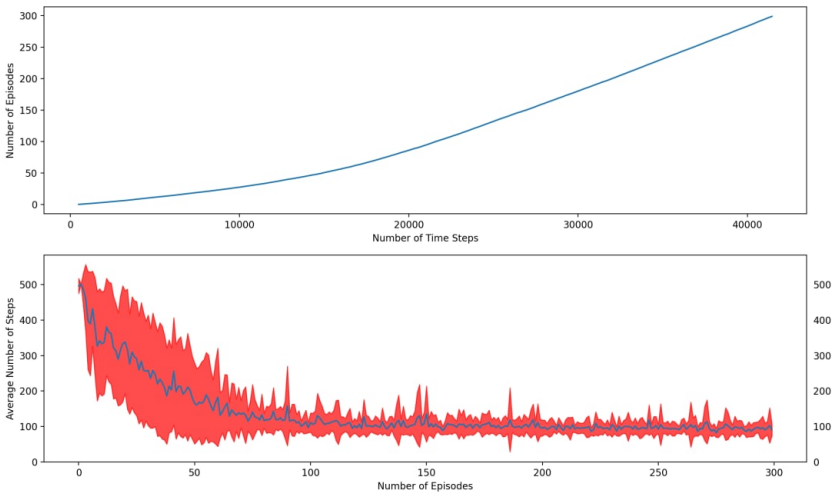

Figure 7: (above) Number of episodes, as a function of the average cumulative actions (time steps) taken up to that episode, across 20 runs (below) Average number of steps in an episode, as a function of number of episodes, across 20 runs. Also shown is the standard deviation (in red).

5 Conclusion

We can observe that DQN with Experience Replay performed similarly to REINFORCE in terms of returns, but drastically better in terms of number of episodes, in both the environments. This corroborates reinforcement learning literature which says that DQN with Experience Replay is sample efficient as it learns from its experience, and is more stable in terms of performance. The experiments can be improved with more time and compute.

References

[1] *Acrobot - Gym Documentation — gymlibrary.dev*. https://www.gymlibrary.dev/environments/classic_control/acrobot/. [Accessed 26-Dec-2022].

[2] *Cart Pole - Gym Documentation — gymlibrary.dev*. https://www.gymlibrary.dev/environments/classic_control/cart_pole/. [Accessed 26-Dec-2022].

[3] *gym/cartpole.py at master · openai/gym — github.com*. https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py. [Accessed 26-Dec-2022].

[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236.

[5] *Reinforcement Learning (DQN) Tutorial x2014; PyTorch Tutorials 1.13.1+cu117 documentation — pytorch.org*. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. [Accessed 26-Dec-2022].

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[7] Yi-Han Xu et al. "Deep Deterministic Policy Gradient (DDPG)-Based Resource Allocation Scheme for NOMA Vehicular Communications". In: *IEEE Access* PP (Jan. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2968595.