COMP 767 (Reinforcement Learning) Assignment 3

Arna Ghosh (260748358), Arnab Kumar Mondal (260906419)

April 2020

1 Theory Part

Answer 1. Off-policy Reinforcement Learning

i.

We start with the definition of R_t^λ and simplify it:

$$\begin{aligned}
R_t^\lambda &= (1-\lambda)\sum_{k=1}^{\infty}\lambda^{k-1}R_t^k \\
&= (1-\lambda)\sum_{k=1}^{\infty}\left(\lambda^{k-1}\sum_{i=1}^{k}\left(\gamma^{i-1}\prod_{j=1}^{i-1}\rho_{t+j}r_{t+i}\right)+\lambda^{k-1}\gamma^k\prod_{j=1}^{k}\rho_{t+j}v(s_{t+k};\theta_t)\right) \\
&= \sum_{i=1}^{\infty}\left(\gamma^{i-1}\lambda^{i-1}\prod_{j=1}^{i-1}\rho_{t+j}r_{t+i}\right)+(1-\lambda)\sum_{k=1}^{\infty}\left(\lambda^{k-1}\gamma^k\prod_{j=1}^{k}\rho_{t+j}v(s_{t+k};\theta_t)\right) \\
&= v(s_t;\theta_t)+\sum_{k=0}^{\infty}\tilde{\delta}_{t+k}\gamma^k\lambda^k\prod_{i=1}^{k}\rho_{t+i} \qquad (1)\\
\textit{where,}\quad \tilde{\delta}_{t+k} &= r_{t+k+1}+\gamma\rho_{t+k+1}v(s_{t+k+1};\theta_t)-v(s_{t+k};\theta_t) \\
\textit{Hence,}\quad R_t^\lambda - v(s_t;\theta_t) &= \sum_{k=0}^{\infty}\tilde{\delta}_{t+k}\gamma^k\lambda^k\prod_{i=1}^{k}\rho_{t+i} \qquad (2)
\end{aligned}$$

Since we restrict our analysis to linear function approximation cases, $v(s_t;\theta_t)=\theta^T\psi_t$. Using equation 2 to simplify the update rule:

$$\begin{aligned}
\Delta_t &= \alpha(R_t^\lambda-\theta^T\psi_t)\psi_t\prod_{i=1}^{t}\rho_i = \alpha(\sum_{k=0}^{\infty}\tilde{\delta}_{t+k}\gamma^k\lambda^k\prod_{i=1}^{k}\rho_{t+i})\psi_t\prod_{i=1}^{t}\rho_i \\
&= \alpha(\sum_{k=0}^{\infty}\tilde{\delta}_{t+k}\gamma^k\lambda^k\prod_{i=1}^{t+k}\rho_i)\psi_t \qquad (3)
\end{aligned}$$

In order to show the required result, we simplify the net parameter update equations and derive a forward view for them. In doing so, we can easily show that the two updates are similar across expectations under respective policies. We use equation 3 to derive a forward equivalence view of the net parameter update equation.

$$
\begin{aligned}
\Delta\tilde{\theta} &= \sum_{t=1}^{\infty}\Delta_t = \sum_{t=1}^{\infty}\alpha(\sum_{k=0}^{\infty}\tilde{\delta}_{t+k}\gamma^k\lambda^k\prod_{i=1}^{t+k}\rho_i)\psi_t \\
&= \alpha\sum_{t=1}^{\infty}\sum_{k=t}^{\infty}(\tilde{\delta}_k\gamma^{k-t}\lambda^{k-t}\prod_{i=1}^{k}\rho_i)\psi_t \\
&= \alpha\sum_{k=1}^{\infty}\sum_{t=1}^{k}\gamma^{k-t}\lambda^{k-t}\psi_t\tilde{\delta}_k\prod_{i=1}^{k}\rho_i \\
&= \alpha\sum_{k=1}^{\infty}\left(\gamma^k\lambda^k\tilde{\delta}_k\prod_{i=1}^{k}\rho_i\sum_{t=1}^{k}\left(\gamma^{-t}\lambda^{-t}\psi_t\right)\right) \\
&= \alpha\sum_{k=1}^{\infty}\left(\tilde{\delta}_k z_k\right) \qquad (4) \\
\textit{where,}\quad z_k &= \gamma^k\lambda^k\prod_{i=1}^{k}\rho_i\sum_{t=1}^{k}\left(\gamma^{-t}\lambda^{-t}\psi_t\right)
\end{aligned}
$$

Using a similar algebraic manipulation for the on-policy TD(λ) parameter update equation, we can write:

$$
\begin{aligned}
\Delta\theta &= \alpha\sum_{k=1}^{\infty}\sum_{t=1}^{k}\gamma^{k-t}\lambda^{k-t}\psi_t\delta_k \\
\textit{where,}\quad \delta_k &= r_{k+1} + \gamma v(s_{k+1};\theta_t) - v(s_k;\theta_t) \qquad (5)
\end{aligned}
$$

Furthermore, we would need a couple more relations to prove the required result. Given $t \leq k$, we can write the following relations involving importance sampling ratio.

$$
\begin{aligned}
\mathbb{E}_\pi[\psi_t\delta_k \mid s_0, a_0] &= \sum_{s,a}\prod_{j=1}^{k+1}P(s_j \mid s_{j-1}, a_{j-1})\prod_{j=1}^{k}\pi(a_j \mid s_j)\psi_t\delta_k \\
&= \sum_{s,a}\prod_{j=1}^{k+1}P(s_j \mid s_{j-1}, a_{j-1})\prod_{j=1}^{k}\rho_j\mu(a_j \mid s_j)\psi_t\delta_k \\
&= \sum_{s,a}\prod_{j=1}^{k+1}P(s_j \mid s_{j-1}, a_{j-1})\left(\prod_{j=1}^{k}\rho_j\mu(a_j \mid s_j)\psi_t r_{k+1}\right) \\
&+ \sum_{s,a}\prod_{j=1}^{k+1}P(s_j \mid s_{j-1}, a_{j-1})\left(\gamma\prod_{j=1}^{k+1}\rho_j\mu(a_j \mid s_j)\psi_t v(s_{k+1};\theta_t) - \prod_{j=1}^{k}\rho_j\mu(a_j \mid s_j)\psi_t v(s_k;\theta_t)\right) \\
&= \mathbb{E}_\mu[\psi_t\tilde{\delta}_k\prod_{j=1}^{k}\rho_j \mid s_0, a_0] \qquad (6)
\end{aligned}
$$

Using equation 6, we can relate equations 5 and 4

$$
\begin{aligned}
\mathbb{E}_\mu[\Delta\tilde{\theta} \mid s_0, a_0] &= \alpha\sum_{k=1}^{\infty}\sum_{t=1}^{k}\gamma^{k-t}\lambda^{k-t}\mathbb{E}_\mu[\psi_t\tilde{\delta}_k\prod_{i=1}^{k}\rho_i \mid s_0, a_0] \\
&= \alpha\sum_{k=1}^{\infty}\sum_{t=1}^{k}\gamma^{k-t}\lambda^{k-t}\mathbb{E}_\pi[\psi_t\delta_k \mid s_0, a_0] \\
\implies \mathbb{E}_\mu[\Delta\tilde{\theta} \mid s_0, a_0] &= \mathbb{E}_\pi[\Delta\theta \mid s_0, a_0] \qquad (7)
\end{aligned}
$$

It is important to note that this relation holds iff the initial weight vectors are equal so that δ_k's are similar for both cases.

ii.

We can easily derive a recursive relation for z_k in equation 4 and compute it using eligibility traces.

$$
\begin{aligned}
z_k &= \gamma^k \lambda^k \prod_{i=1}^{k} \rho_i \sum_{t=1}^{k} \left(\gamma^{-t}\lambda^{-t}\psi_t\right) \\
&= \gamma^k \lambda^k \prod_{i=1}^{k} \rho_i \left(\gamma^{-k}\lambda^{-k}\psi_k + \sum_{t=1}^{k-1} \left(\gamma^{-t}\lambda^{-t}\psi_t\right)\right) \\
&= \prod_{i=1}^{k} \rho_i \left(\psi_k + \gamma^k\lambda^k \sum_{t=1}^{k-1} \left(\gamma^{-t}\lambda^{-t}\psi_t\right)\right) \\
Define \quad a_k &= \prod_{i=1}^{k} \rho_i = \rho_k a_{k-1} \\
and \quad b_k &= \gamma^k\lambda^k \sum_{t=1}^{k} \left(\gamma^{-t}\lambda^{-t}\psi_t\right) = \psi_k + \gamma\lambda\gamma^{k-1}\lambda^{k-1} \sum_{t=1}^{k-1} \left(\gamma^{-t}\lambda^{-t}\psi_t\right) \\
&= \psi_k + \gamma\lambda b_{k-1} \\
s.t. \quad z_k &= a_k b_k
\end{aligned} \tag{8}
$$

Algorithm 1: Online off-policy TD(λ) algorithm (updates done at end of episode)

```
Result: $\theta$ to calculate $v_\pi(s)$
Input: Target policy $\pi$ and behavior policy $\mu$
Input: a feature function $\psi$: $\mathbf{S}^+ \longrightarrow \mathbb{R}^d$
Algorithm parameters: step size $\alpha > 0$, trace decay $\lambda \in [0, 1]$
Initialize value-function weights $\theta \in \mathbb{R}^d$ (e.g., $\theta = 0$)
while For each episode do
    Initialize state and obtain initial feature vector $\psi_0$ for state $s_0$.
    Take a defined action $A_0$.
    $s \longleftarrow s_0$
    $a \longleftarrow 1$
    $b \longleftarrow 0$
    $\Delta\theta \longleftarrow 0$
    while For each step in episode do
        Choose action $A \sim \mu$
        Take action $A$ and observe $r'$, $\psi'$ (from next state $s'$)
        $\rho \longleftarrow \frac{\pi(A|s)}{\mu(A|s)}$
        $V \longleftarrow \theta^T \psi$
        $V' \longleftarrow \theta^T \psi'$
        $\delta \longleftarrow r' + \gamma\rho V' - V$
        $a \longleftarrow \rho a$
        $b \longleftarrow \psi + \gamma\lambda b$
        $z \longleftarrow ab$
        $\Delta\theta \longleftarrow \Delta\theta + \delta z$
        $\psi \longleftarrow \psi'$
    end
    $\theta \longleftarrow \theta + \alpha\Delta\theta$
end
```

Answer 2. Part A

i). As mentioned in the question, experience replay is a technique where agent stores it's experience as a tuple $e_t = (s, a, r, s')$ in a replay memory $\mathcal{D} = e_t, ..., e_{t-N+1}$ of fixed size N. (s is the current state the agent is in and it takes an action a to reach state s' and get a reward r). Imagine it as a sliding window of size N over the sequence of experiences. Note that at any time step t in an episode when the agent is at state s it selects a ϵ-greedy policy based on the current approximation by the Q network. Also note that the replay buffer stores some preprocessed $\phi(s)$ instead of s in DQN but for our arguments we are going to consider just s. To update the Q network an experience tuple $e \sim \mathcal{D}$ is sampled uniformly at random from the replay buffer. The first advantage is data efficiency by using each step of experience potentially many times in weight update. To better understand this consider a rare experience which would be useful later on but without experience replay it would be used for just a single update which will make our learning slow. We are effectively reducing the amount of experience required to learn by adding more memory and some extra computation which is a fair trade-off given they are cheaper resources compared to the agent's experience in RL. The second advantage is that by sampling from experience buffer, by mixing both more and less recent experience, for updates breaks the temporal correlation between the experience samples and therefore reduces the variance. Learning directly from strong correlated consecutive experience samples is inefficient and possibly unstable as stochastic gradient descent works best with independent and identically distributed data samples in supervised learning. Experience replay essentially smooths out learning and circumvents oscillations or divergence in the parameters of the deep Q network.

ii). The key idea of prioritized experience replay is that an RL agent can learn more effectively from some experience or transition tuples. It proposes to replay transitions with high expected learning progress more frequently. To further deal with the loss of diversity due to prioritization and the bias introduced to alleviate that, it introduces stochastic prioritization and importance sampling. Coming back to the importance of each transition, the magnitude of a transition's TD error δ is a reasonable proxy for the amount the RL agent can learn from a transition and is suitable for incremental online algorithms like Q-learning which already uses TD error for updating the parameters. It takes priority of a transition in the replay memory to be proportional to the TD error of that transition and hence needs to store the TD error along with the transition in the memory. Now, if it just uses the greedy TD prioritization algorithm to select experiences or transitions from the replay memory, which has a maximum TD error, then there are several limitations. Out of all the transitions stored in the replay memory, only the TD errors are updated for transitions that are replayed to avoid expensive sweeps over the memory. This makes the transitions that have low TD error on the first visit not replayed with a sliding window replay memory. This means a small subset of the experience with high TD error will keep getting replayed as the error shrinks slowly in function approximation. This exacerbates the diversity of updates and makes the function approximator overfit. Hence it further proposes a stochastic sampling from the replay memory with the probability of being sampled proportional to transition's priority while maintaining a nonzero probability even for the lowest priority transition. The probability of sampling transition i is given by:

$$P(i) = \frac{p_i^\alpha}{\sum_i p_k^\alpha}$$

where $p_i > 0$ is the priority of transition i. The exponent α is a hyperparameter and gives degree of prioritization where $\alpha = 0$ corresponds to uniform sampling. Consider δ_i is the TD error of transition i, then there are two variants of priority: $p_i = |\delta_i| + \epsilon$ where $\epsilon > 0$ ensures the diversity and $p_i = \frac{1}{rank(i)}$ where rank is determined after sorting the replay memory according to $|\delta_i|$. Although both are monotonic, the second one is more robust. At last, the prioritized sampling, as mentioned above, introduces a bias because it changes the distribution over which we were taking the expectation and therefore, would change the solution the Q values would converge to. This can be corrected by using importance sampling and weighing the TD error by $w_i = \frac{1/N}{P(i)}$, i.e., $w_i\delta_i$. The original paper uses weighted importance sampling $w_i = (\frac{1/N}{P(i)})^\beta$ where β is another hyperparameter which controls how much prioritization to apply. It starts small and anneals towards 1 over the episode to get an unbiased estimate near convergence.

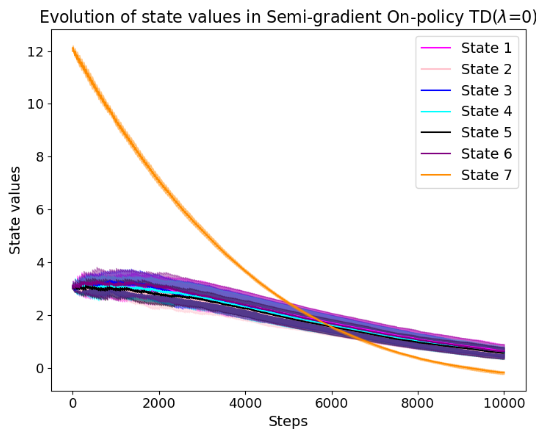
iii). After the first two parts, it can be argued why prioritized experience replay outperforms original experience replay, and this has been empirically established. Consider the cases where rewards are rare, like a bipedal robot falling where there are a few relevant transitions from rare success in a huge pool of highly redundant failure cases. Although using original experience replay memory to store this rare transition, which will give rise to a high TD error δ for this but the parameters of the Q network will be updated a few times before it moves out of the experience replay window. This makes learning slow compared to prioritized replay memory where this same rare experience will have a high probability of getting sampled and will result in more number of updates. Note that the replay memory window is the same for both the techniques, but the difference lies in the way we sample the stored experience. In contrast to ordinary experience replay which liberates online learning agents from processing transition in their temporal experience order, prioritized replay additionally liberates agents from considering transitions with the same frequency for training.

Although prioritized experience relay sought to alleviate most of the practical issues of ordinary experience replay, there are still a couple of downsides to it. Firstly, the issue about not replaying experiences with low TD-error. In the discussions section, the authors mention that in some cases (reduced to a large extent by incorporating stochastic prioritization), a certain fraction of the visited transitions are never replayed before dropping out of the sliding memory window. This can happen if during the experience, the TD-error was very small. However, as the DQN is trained using other experiences the TD-error on this experience could increase. This phenomenon is closely related to the catastrophic forgetting behavior of deep networks. Therefore, we would drop certain experiences hoping that the TD-error on the same is still low, although the network may have significantly changed from the time we calculated the TD-error. Contrary to prioritized experience replay, ordinary experience relay samples uniformly from the bag of visited transitions and therefore, even transitions from low TD-error will be replayed and trained on. Secondly, the implementation introduces two new hyperparameters, namely α and β. These hyperparameters are needed to control the stochastic behavior of experienced relay and incorporate the importance sampling ratio correction respectively. However, the added hyperparameters need to be tuned for the problem at hand and therefore engender practical issues for the experimenter.

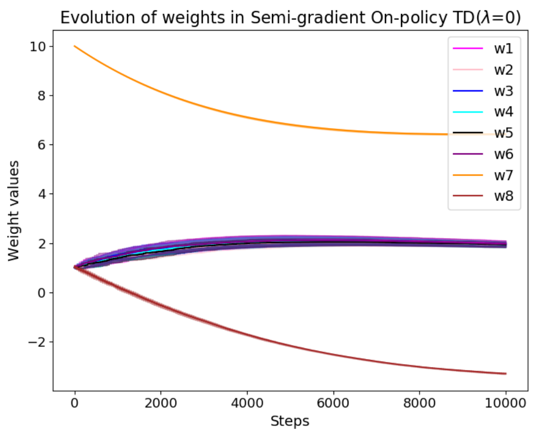
2 Coding Part

2.1 Question 1. Baird's counterexample

We initially defined a class to match the task specifications of Baird's counterexample. The task has 7 discrete states, the reward for each transition is 0 and there are 2 actions that the agent can take from each state. Action 0 takes the agent to one of the first 6 states, with uniform probability. Action 1 takes agent to the 7^{th} state. The environment's discount factor (γ) is set to 0.99 for all experiments. There was a slight discrepancy between the task descriptions of the book and the assignment, namely the 7^{th} state being the terminal state. We kept this as a setting that can be changed in the class definition. The implementation can be found here.



(a) Estimated state values



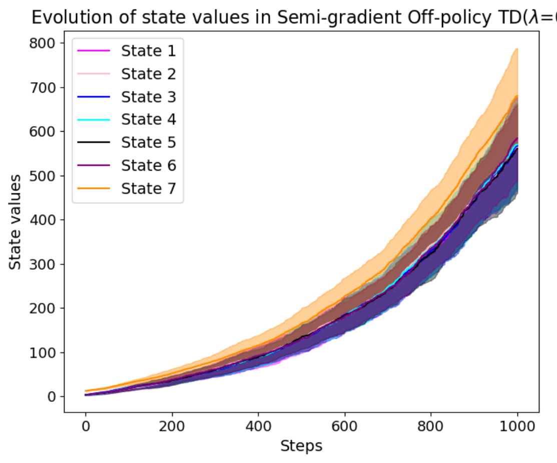
(b) Estimated parameter weight values

Figure 1: State values and corresponding parameter weights estimated by the on-policy TD(λ) algorithm for the Baird's counterexample task for the described target policy (π). Results are averaged over 20 seeds and shaded region depicts standard deviation.

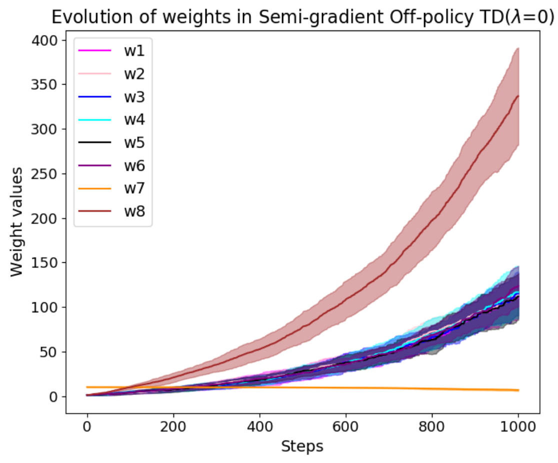
Furthermore, we created a class for off-policy semi-gradient TD(λ) algorithm. We use a linear function approximator with 8 weights. The TD(λ) implementation uses accumulating traces. The importance sampling ratio is incorporated while doing parameter updates in order to correct the TD-error. The implementation can be found here. The function approximator weights are initialized to non-zero values, specifically all weights are set to 1 except the 7^{th} weight, which is set to 10. The step size is set to 0.01. Finally, the off-policy TD class is flexible to run both on-policy and off-policy experiments. We ran the on-policy experiments with both behavior and target policies set to be the same. We observe both the state values and parameter weights evolve through time during the learning procedure. The results were averaged over 20 different runs with different random seed for each run. The results for both behavior and target policy set to policy π is shown in Figure 1. The code for plotting and the corresponding plots for the results for policy b can be found here.

Figure 1 shows the mean estimated state value and parameter weight values across 20 different seeds. The shaded region indicates the standard deviation for the corresponding quantity. As expected, the on-policy TD algorithm converges to true state values that is 0. Since the problem is over-parametrized, there are multiple values of weight parameters that can lead to this solution. Here, the weights converge to one such solution. It is important to note that these experiments were performed by ignoring the terminal state property. This allows all runs to have equal number of steps for equal number of episodes (here, 200).

Furthermore, we evaluate the off-policy behavior of TD for this problem. As described in book, we used behavior policy b and target policy π. Corresponding parameter weight values and estimated state values are shown in Figure 2. Here, the agent is trained for 20 episodes (each episode consisting of 5 steps) in order to match the plot in Fig 11.2 of the book. The corresponding code can be found here.



(a) Estimated state values



(b) Estimated parameter weight values

Figure 2: State values and corresponding parameter weights estimated by the off-policy TD(λ) algorithm for the Baird's counterexample task for target policy π using behavior policy b. Results are averaged over 20 seeds and shaded region depicts standard deviation.

Per-iteration/episode Time Complexity	$\mathcal{O}(n)$
Per-iteration/episode Space Complexity	$\mathcal{O}(\lvert S_f \rvert)$

Table 1: Time and Space Complexity of the off-policy semi-gradient TD(λ) implementation. n denotes the average length of trajectory/episode and S_f denotes the sparse state representation space.

An interesting observation is that the weight values do not diverge if the discount factor is reduced. This is because the algorithm assigns more importance to immediate rewards, which are always zero than the value of the next state. Therefore, the algorithm converges to the correct values. Also, this effect is a gradual effect wherein lowering γ from 0.99 increases the number of steps required for the values to explode. Finally, for some value it doesn't explode anymore and instead converges to 0. For completeness, we present the time and space complexity for the implemented algorithm in Table 1.

2.2 Question 2. Part B.

In this part, we used the Cartpole environment from Open AI gym and set the discount factor γ to 0.99. In this problem, the agent can provide a force of either $+1$ or -1 to balance the upright pendulum for as long as it can. For every time step, it gets a reward of $+1$. We have used version 1 in which the episode terminates when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center, or it has already run 500 time steps. So the maximum reward the agent can collect in each episode is 500 as the even the reward for termination step is $+1$. Note that we have set gamma different from what was asked in the question because in this problem, we would like to keep the horizon as long as possible. Using a discount factor of 0.9 makes the agent learn from timesteps earlier in the episode and focus more on them while we want to keep it upright as long as possible, which needs us to concentrate equally on rewards we get later in the episode. We have further used a wrapper class $Cartpole()$ for the environment.

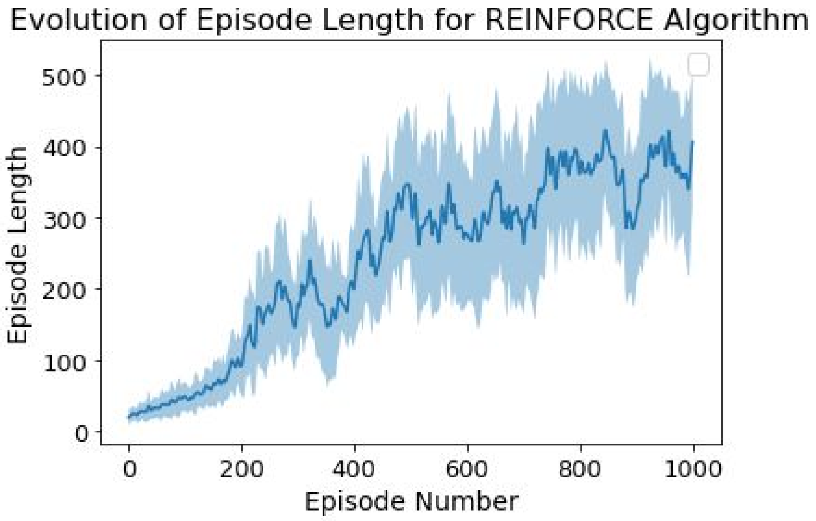


Figure 3: Performance (number of steps per episode) of the REINFORCE algorithm for the Cartpole environment. Results are averaged over 5 seeds and shaded region depicts standard error.

The first part of this problem needs us to implement the REINFORCE algorithm, which is basically a Monti Carlo Policy Gradient algorithm. We have defined a two-layered multi-layer perceptron using Pytorch, which takes the states as input and gives a soft policy as out. The networks can be understood as: Linear-LeakyRelu-Linear-Softmax. The implementation of the policy network can be found here. Note that we have also attached a static member function $fit\_model\_REINFORCE()$ to this class $PolicyNet()$ which given the history of an episode updates the network according to Monti Carlo policy gradient algorithm. To efficiently implement that, we have defined a class $Buffer()$ to store all the transitions and then later use that to compute the losses at each step in an episode. The $REINFORCE()$ function simulates the environment and calls the $fit\_model\_REINFORCE()$ to fit the model. It finally returns the episode count and the number of steps per episode. The result of running it for 1000 episodes using 5 different seeds can be found in Figure 3. Note the hyper-parameter here is the learning rate. We provide run the algorithm with the same seed for different learning rates to figure out which one works the best. As can be seen from Figure 4 of cumulative reward over 1000 episodes, any value in the range of $(0.001, 0.004)$ seems to work well. But the plot isn't the best metric as with higher learning rates, we start to get more rewards first(which results in more cumulative rewards) but then the episode length drops while with lower learning rate the improvement is more stable which made us choose 0.001 learning rate for our experiments. We notice that when the network starts to learn even when we start at a random position with the policy, in starts to manage to stay

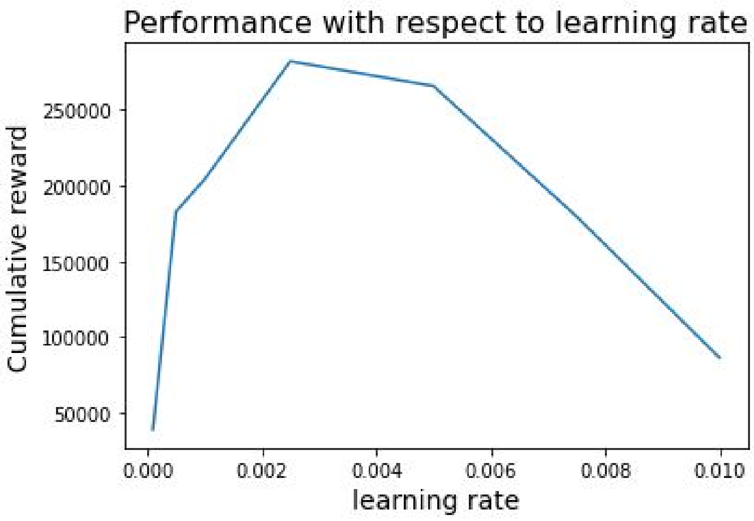


Figure 4: Cumulative reward over learning rates for the Cartpole environment to help choose the best learning rate.

upright as long as possible. As the initial start state is random for each episode, so for bad starting position results in less reward even though the network is doing, it's best with respect to the policy. This makes the plots very spiky with high variance over episodes. To fix this, we used a 1D Gaussian filter to display our results.

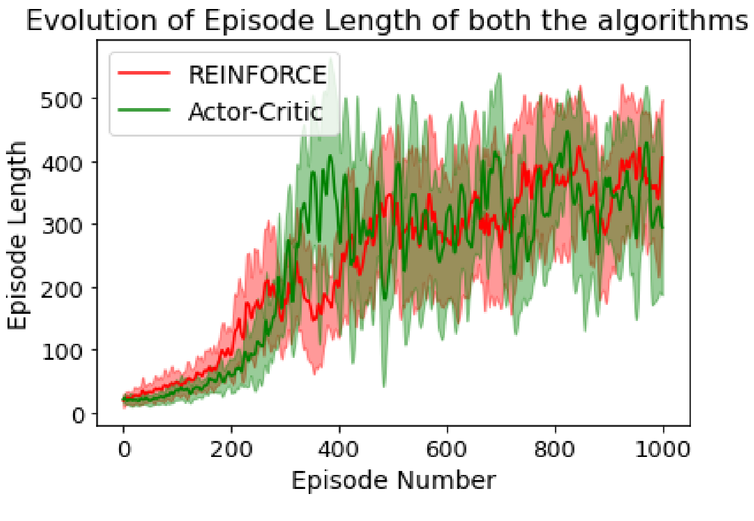


Figure 5: Performance (number of steps per episode) of both the algorithms for the Cartpole environment. Results are averaged over 5 seeds and shaded region depicts standard error.

Finally, we implement the Actor Critic algorithm where we add another critic network which is basically a value function estimator. We keep the actor the same Policy Network we used in the previous algorithm for fair comparison between both the algorithms. The main difference is rather than using Monti Carlo estimation of the the target we use 1 step bootstrapping. Note that we didn't try bootstrapping after multiple steps which could be another hyperparameter due lack of resource and time. Both the implementation of REINFORCE and Actor Critic closely follows the pseudocode given in the Sutton and Barto book and can be found here. The implementation of actor critic doesn't require the buffer as we update our network every step. One interesting point to note here is that in REINFORCE we freeze the network over an episode(which is a bit different than the algorithm given in the book) then update it after gathering all the gradients from each step which has made the training faster and stable. But in Actor Critic we update at every step as we don't have to wait for the entire episode to end. Although this combined with the low variance due to bootstrapping makes our Actor Critic converges faster than the REINFORCE as we note from Figure 5. But one thing to note is that it has increased the runtime of the algorithm as we backpropragate through both the Value and Policy Network to update their weights every time step. We can possibly mitigate this by freezing the network, using a buffer and training in minibatches during every episode which will exploit the Pytorch framework. Another important advantage of Actor Critic over normal REINFORCE is that it can be used for non terminating environments which we didn't get to explore in this problem. While experimenting with the algorithm we notice that it is sensitive to the learning rate. We run similar loops like REINFORCE to figure out the best set of learning rate for both Actor and Critic. It is worth mentioning that the Critic network(Value function network) needs higher learning rate compared to the Actor(Policy network). The learning rate we took for Critic is 0.002 while for the Actor is 0.0001. Figure 5 gives a comparison between both the algorithms. We conclude that though in theory Actor Critic is better than REINFORCE in most of the cases the difference in performance in this particular problem isn't that noticeable.

Contributions

AKM worked on Q2 of theory and coding. AG worked on Q1 of theory and coding. Both AG and AKM contributed equally to writing the report.

Reproducibility Checklist

- ☒ Description of algorithms and environments
- ☒ Time and space complexity for off-policy TD(λ)
- ☒ Link to code – added link to Colab notebook where applicable
- ☐ Description of data collection process - **not applicable**
- ☒ Link to download dataset or environment
- ☐ Explanation of excluded data – **not applicable**
- ☒ Explanation of train/test splitting
- ☒ Range of hyperparameters and method to select best hyperparameter
- ☒ Number of evaluation runs
- ☒ Description of experiments
- ☒ Definition of metrics used
- ☒ Error bars in plots – plots indicate standard deviation
- ☒ Result description with mean and standard deviation – shown in plots
- ☒ Description of computing infrastructure used – Colab notebook provided to enhance reproducibility of results