Submitted by:
Akshay Singh Rana - 260963467
Harmanpreet Singh - 260962547

1 Theory

1. **TRACK A: Bias-Variance trade-off**

   (a) **Monte-Carlo (MC) methods are defined by forming estimates of returns (Gt) by performing rollouts in the environment (to collect trajectories), usually without using concepts from Bellman equations. Temporal Difference (TD) methods are similar to MC methods, except they use a "bootstrapped" target, i.e. the target value depends on the current estimate of the value function. Explain how these differences affect the learning process, specifically the bias- variance trade-off. For each method, comment how the estimated value function depends on the amount of data available to the agent. Also, briefly explain the scalability of these approaches with experience. Relate your answer to TD() methods.**

   In Monte Carlo, the return G_t is the discounted sum of rewards from state S_t until the end of the episode. In TD method instead of waiting till the end of the episode, the target is the sum of immediate rewards and the estimate of future rewards. This is called bootstrapping, as we are using an estimated value to update a similar estimated value.

   Monte Carlo Return:

   $$G_t = R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T$$

   Temporal Difference Return:

   $$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$$

   Due to the bootstrapping of the value function of the next state, we induce a bias in the system as the V value is an estimate of the value function and not the true value function so the TD target is biased. Whereas in MC, we wait till the end of the episode and achieve a the reward from the environment without estimating it and thus the return for MC is an unbiased estimate for $v_\pi(s)$

   But because we waited till the end of the episode, our return is affected by all the actions we took along the way. Change in any action will alter the path and give a different return thus inducing variance in the process. In contrast, the variance

for TD is quite low as the target depends on the immediate next state and thus have lower variables to create variance.

In terms of G_t^λ, one could produce a substantial new range of algorithms by controlling the value of λ.

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t$$

The equation makes it clear that for $\lambda = 1$, it becomes a Monte Carlo algorithm. On the other hand, if $\lambda = 0$, then it becomes a one-step TD method. Therefore, the bias and variance of the algorithm can also be controlled using λ. Increasing the λ will increase variance and reduce bias wheres decreasing it will increase bias and reduce the variance.

In regards to the amount of training data, if we are using longer trajectories without bootstrapping values, it means you need more samples before the estimates converge. One can also reduce the variance in MC once more data is added to it. Whereas, the main disadvantage of bootstrapping is that it is biased towards whatever your starting values of Q(s,a) are. Those are are most likely wrong, and the update system can be unstable as a whole because of too much self-reference and not enough real data. Despite the problems with bootstrapping, if it can be made to work, it may learn significantly faster, and is often preferred over Monte Carlo approaches.

Monte-Carlo estimation though simpler cannot be used in MDPs with large state-spaces as it will require really long trajectories to learn from. In contrast, the TD methods can we easily scaled to problems with bigger state-action spaces.

(b) **MC methods are known to have bounds that are independent on the size of the state space: Explain why this is the case, along with the advantages and disadvantages of it. In your answer, focus on how one would learn the optimal policy using MC methods and the sample complexity of such an approach.**

$$|\hat{v} - v^\pi(s)| \leq \frac{R_{max}}{1-\gamma}\sqrt{\frac{1}{2n} ln \frac{2}{\delta}}$$

Monte Carlo algorithms upper confidence bound can be found using Hoeffding's inequality. This algorithm achieves logarithmic asymptotic total regret. The returns returned from these methods do not depend on the state space. We can basically compute V(s) without the need to know anything about other value

functions of the states as our method depends on the real reward at the end of the episode.

Compare to the Dynamic Programming where we need to explore all the state space before evaluating the value function, it is indeed an advantage that the error bound is independent of the state space. We can parallely explore different states of the same problem to speed up the process.

Unlike TD, we might still need to draw trajectories from a state to the end of the episode and we still need to know all the states in the trajectory. In the absence of that, there could be an error while calculating the value function of those states.

The error bound depends instead of the number of samples used to train, and based on the value of 'n' we can achieve the error with 1-δ probability.

(c) **Explain why/how TD methods can be viewed as learning an implicit model of the world and solving for it at the same time. Describe why this leads to different solutions for MC and TD**.

Batch Monte Carlo method always find the estimates that minimize mean-squared error on the training set, whereas batch TD always finds the estimates that would be correct for the maximum-likelihood estimate of the Markov process. Here the maximum-likelihood estimate is the model of the Markov process formed from the observed episodes. So TD methods can be viewed as learning an implicit model of the world, as if the model were correct, we will be estimating the value function that would be exactly correct. This is called the certainty-equivalence estimate because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated. This is the reason why TD methods converge more quickly than Monte Carlo methods and leads to different solutions. Batch TD are able to perform even better than Monte Carlo methods because TD is optimal in a way that is more relevant to the predicting returns.

2. **TRACK A:**

   (a) **Outline a SARSA algorithm with function approximation that uses Boltzman exploration instead of the standard -greedy policy.**

   Boltzmann softmax policy assigns the following probability to each action.

$$\pi(a|s) = \frac{e^{\beta Q(s,a)}}{\sum_a e^{\beta Q(s,a)}}$$

Algorithm 1 SARSA with Boltzman exploration

Input: a differentiable action-value function parameterization $\hat{q} : \mathbb{S} \times \mathbb{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha \in (0, 1)$, Boltzman coefficient β
Initialize value-function weights w ∈ R arbitrarily (e.g., w = 0)
for *each episode* **do**
  Initialize S
  Choose A from S using policy derived using Boltzman with β
  for *each step of episode until terminal state* **do**
    Take action A and observe R and S'
    Choose A' as a function of q(S',.,w) using policy derived using Boltzman with β
    $w \leftarrow w + \alpha[R + \gamma\hat{q}(S', A', w) - \hat{q}(S, A, w)]\Delta\hat{q}(S, A, w)$
    $S \leftarrow S'$;A $\leftarrow A'$
  end
end

   (b) **Explain why SARSA with ϵ-greedy policy can fail to converge and why Boltzman exploration can help in this respect**

   Refrences from:
   1. On the Existence of Fixed Points for Q-Learning and SARSA, Perkins 2002
   2. Chattering in SARSA. Gordon 1996

   Convergence problems stems from discontinuous action selection strategies as employed in ϵ-greedy policy. Discontinuous action selection strategies can result in a lack of fixed points in the space of action-value functions making convergence impossible.

   As stated (*Gordon, 1996*), SARSA fails to converge because the probability of visiting a given state can change discontinuously when the Q function fluctuates slightly. For an MDP described in figure 1, the agent alternates between the upper and lower paths because the values of lower path keep decreasing initially and then starts rising when the value function of upper path decreases.

   This above cycle remains substantially the same for any sufficiently small values of α and ϵ. Currently in the above example, we can divide the space into small

regions where SARSA follows a constant policy and we can see that these regions will be convex in weight space and will have their own local greedy policy. So here, we had two different greedy regions and two greedy points and due to which the SARSA oscillates because these points lie on the boundary between the greedy regions. As stated (*Perkins 2002*), although it may converge to the boundary between the regions, they will forever chatter between them. It does mean that it is impossible for the agent to choose a fixed policy. The agent will be forced to move from one region to another indefinitely, and so the agent's policy must switch among π_1, π_2

In general, the existence of a fixed point does not guarantee that action-value function converges but however that is true for SARSA and Q-Learning. The agent if behaves according to one policy π, its action values do converge (*Singh et al. 1994*). There exists only one fixed point for SARSA and we can say that it converges. (Theorem 3, Perkins 2002). Therefore, if an agent employs any continuous action selection strategy, such as Boltzman, where we can get a distribution for actions, then action-value and policy can be converged.

Adding a figure from the paper (*Perkins 2002*) to shed more light into the empirical results.

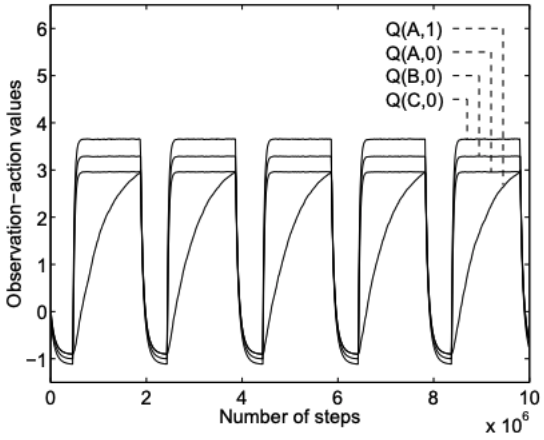


Figure 3. Evolution of observation-action values under ϵ-greedy action selection.

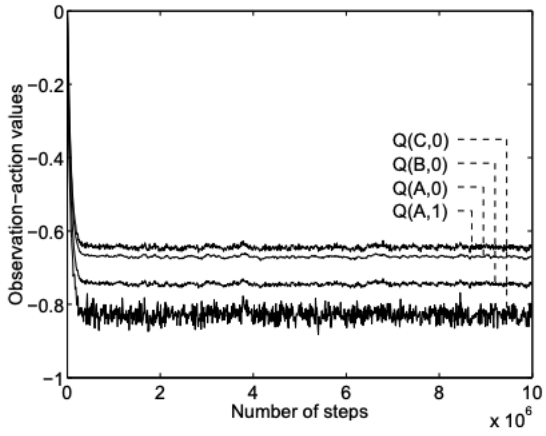


Figure 4. Evolution of observation-action values under modified softmax action selection.

As evident in the graph, under ϵ-greedy action selection, the action values followed a definitive cyclic pattern of rise and fall. Whereas in the continuous exploration strategy like Boltzman, the values quickly converged and hovered around a fixed point.

Moreover, if there exists a function approximator powerful enough to represent every Q function exactly, then each greedy region in a cycle must have the same greedy point and it may converge to the optimal policy.

(c) **Explain the connection between SARSA and Q-learning. Show how Q-Learning can be viewed as a special case of Expected Sarsa.**

SARSA - On Policy TD Control:

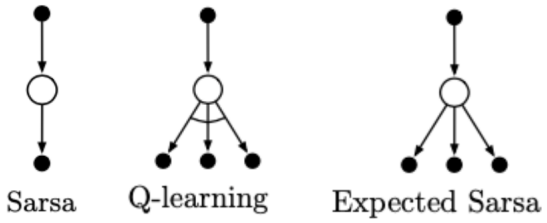
$$Q(S_t, A) \leftarrow Q(S_t, A) + \alpha[R + \gamma Q(S_{t+1}, A') - Q(S_t, A)] \tag{1}$$

Q-Learning - Off Policy TD Control:

$$Q(S_t, A) \leftarrow Q(S_t, A) + \alpha[R + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A)] \tag{2}$$

Expected Sarsa - On Policy TD Control:

$$Q(S_t, A) \leftarrow Q(S_t, A) + \alpha[R + \gamma \Sigma_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A)] \tag{3}$$



The most important difference between SARSA and Q-Learning is that the former picks actions by following a policy like ϵ-greedy whereas the latter uses the maximum Q values over all action in the next step. Q-learning is an off-policy algorithm which doesn't follow any policy but instead chooses the action in a greedy way. Unlike SARSA where an action A' is chosen by following a certain policy, here the action A' is chosen in a greedy fashion by simply taking the max of Q over it. Since this is independent of the policy being followed, this dramatically simplifies the analysis of the algorithm and enables early convergence proofs. Since SARSA takes the action selection into account and learns the longer but safer paths unlike Q-learning which directly learns the values for optimal policy.

Expected Sarsa can be viewed as a On-learning version of Q-Learning. In case of a greedy policy, $\pi(s, a) = 0$ for all 'a' except for the one with maximum action values, Q-learning becomes a special case of Expected Sarsa. In this case, the above two equations become equal. Although the Expected Sarsa is different from Q learning because it is on policy and the latter is off-policy.

$$Q(s, a) = \Sigma_a \pi(s|a) Q(s|a)$$

if $\pi(s|a) = 0$ for all a except the one with maximum Q

$$\implies Q(s, a) = maxQ(s|a)$$

$\therefore$ It is a special case of Q-learning

2 Coding

1. **Continuous Random Walk (Prediction task)**
   https://colab.research.google.com/drive/1qYVGs9YNybXadT1pI1qk0bG99fkvyuvf

2. **Control task:**
   Cartpole:
   https://colab.research.google.com/drive/1OpXcjpeHtn8pAGHiu-gMOXD4YbtPQb76