687 Class Project: Implementation of REINFORCE with Baseline, One-Step Actor-Critic, and Episodic semi-gradient n-step Sarsa

Sagar Palao
spalao@umass.edu

Shriya Natesan
snatesan@umass.edu

1 Introduction

In this project we implemented three Reinforcement Learning algorithms (a) REINFORCE with Baseline (episodic), (b) One-Step Actor-Critic (episodic), and (c) Episodic semi-gradient n-step Sarsa. We experimented these algorithms in two environments: (a) 687-Gridworld and (b) a classical control environment - Acrobot. The report presents these algorithms, our experiment procedures, the results from these experiments, how we adapted the algorithms to tackle the problems faced while training the agent and our learnings along the way.

Contributions: The implementation of all three algorithms were done in pair programming. For experimentation, we divided the tasks. Sagar Palao performed experimentation for the algorithms REINFORCE with Baseline (episodic) and One-Step Actor-Critic (episodic) for both the environments. Shriya Natesan performed experimentation for algorithm Episodic semi-gradient n-step Sarsa for both the environments. As we discovered challenges in training or found unsatisfactory results, we visited TA hours and together tackled the problems by making adaptation to the algorithms.

2 Environments

We provide a brief description of the environments we experimented with in our project.

2.1 687-Gridworld

687-GridWorld domain is the classic RL environment where the agent starts in state $(0, 0)$ in the 5×5 grid and has to reach a Goal State in order to win a reward. It can be described as follows:

- **State**: This problem consists of a 5×5 grid world where each state $s = (r, c)$ describes the current coordinates/location of the agent. There are two *Obstacle states* in this domain: one in state $(2, 2)$ and one in state $(3, 2)$. $(4, 2)$ is a *Water state* and $(4, 4)$ is a *Goal state*. Initial state is state $(0, 0)$
- **Actions**: There are four actions: AttemptUp (AU), AttemptDown (AD), AttemptLeft (AL), and AttemptRight (AR).
- **Dynamics**: This is a *stochastic* MDP:
  - Agent moves in the specified direction with 0.8 probability.
  - Agent veers to the right with respect to the intended direction with 0.05 probability.
  - Agent veers to the left with respect to the intended direction with 0.05 probability.
  - Agent temporarily breaks and does not move with 0.1 probability.

  The agent cannot enter the *Obstacle states* or cross the border. If it attempts to go in these states, it remains in same the state.
- **Rewards**: The reward is always 0, except when entering the Goal state, in which case the reward is 10; or when transitioning to (entering) the *Water state*, in which case the reward is -10. The goal is to have the agent reach the Goal state in as few steps as possible, avoiding the *Water state*.
- **Terminal State**: The Goal state is terminal.
- **Discount Parameter** $\gamma = 0.9$
- **Episode End**: The Episode ends when the agent reaches the goal state or when the episode length exceeds 100.

We wrote our own code for this environment.

2.2 Acrobot

The system in this environment consists of two links which are linearly connected to form a chain, with one end of the chain fixed. The joint between these two links is actuated and the goal is to apply torques to this joint in order to reach a certain target height. It can be described as follows:

- **State**: In this environment a state is defined through two angles θ_1 and θ_2 and their angular velocity. θ_1 is the angle of the first joint, where an angle of 0 indicates the first link is pointing directly downwards. The angle θ_2 is the relative angle of the first link, where an angle of 0 indicates that the two links have the same angle. At any point the observation space has 6 values: $[\cos(\theta_1),\ \sin(\theta_1),\ \cos(\theta_2),\ sin(\theta_2)$, angular velocity of θ_1, angular velocity of $\theta_2]$. There are designated values for the high and low observation space values. Initial state: uniformly sampled between -0.1 and 0.1 for θ_1, θ_2, angular velocity of θ_1, angular velocity of θ_2.

- **Actions**: There are 3 actions: apply -1 torque to the actuated joint, apply 0 torque to the actuated joint, apply 1 torque to the actuated joint.

- **Rewards**: The reward is always -1, except when the free end reaches the desired target height it gets a reward of 0. The goal is to have the free end reach the target height in as few steps as possible.

- **Terminal State**: The step when the free end reaches the target height is terminal.

- **Discount Parameter** $\gamma = 1$

- **Episode End**: The Episode ends when the free end reaches the target height, which is constructed as: $(-\cos(\theta_1) - \cos(\theta_2 + \theta_1)) > 1.0$ or when the episode length exceeds 500.

We used the Acrobot environment[1] provided in the Gym library which is a standard API for reinforcement learning.

[1] www.gymlibrary.dev/environments/classic_control/acrobot/

3 REINFORCE with Baseline (episodic)

REINFORCE with Baseline is a Policy Gradient method to learn a parameterized policy directly instead of learning an intermediate action-value function and then deriving policy from it. Policy gradient methods have an objective to maximize performance, $J(\theta) = \mathbf{E}[G|\theta]$, by learning parameters of the policy θ which maximizes $J(\theta)$. To do this, it performs gradient ascent step $\theta_{i+1} = \theta_i + \alpha \nabla J(\theta)$.

By Policy Gradient Theorem, for all finite MDPs with bounded rewards, $\gamma \in [0, 1)$, and unique deterministic initial state, s_0:

$$\nabla J(\theta) = \sum_s d^\theta(s) \sum_a \frac{\partial \pi(s, a; \theta)}{\partial \theta} q^{\pi_\theta}(s, a)$$

where, $d^\theta(s) = \sum_{t=0}^{\infty} \gamma^t P(S_t = s|\theta)$.

We can rewrite the above equation to get,

$$\nabla J(\theta)$$
$$= \sum_{t=0}^{\infty} \sum_s P(S_t = s|\theta) \sum_a \pi(s, a; \theta) \gamma^t q^{\pi_\theta}(s, a)$$
$$* \frac{\partial \ln(\pi(s, a; \theta))}{\partial \theta}$$

Now, if we drop gamma related terms from $d^\theta(s)$ to obtain a proper probability distribution, we can express it as:

$$\nabla J(\theta) \propto \mathbf{E}\big[\sum_{t=0}^{\infty} \gamma^t q^{\pi_\theta}(S_t, A_t) \frac{\partial \ln(\pi(S_t, A_t; \theta))}{\partial \theta}\big]$$

We don't know $q^{\pi_\theta}(S_t, A_t)$ in advance. REINFORCE is a Monte-Carlo method which estimates $q^{\pi_\theta}(S_t, A_t) = G_t$. So the algorithm will run an episode, observe G_t and compute $\nabla J(\theta)$. Then perform the gradient ascent update on the parameters as below:

$$\theta = \theta + \alpha \sum_{t=0}^{T} \gamma^t G_t \frac{\partial \ln(\pi(S_t, A_t; \theta))}{\partial \theta}$$

Monte Carlo method has very high variance. To control the variance REINFORCE with Baseline, uses Control Variates. Control Variates are variables which is correlated to the estimate and we subtract it from the estimate of the variable. This, in turn, reduces the variance. Now, the update equation is:

$$\theta = \theta + \alpha \sum_{t=0}^{T} \gamma^t (G_t - b(S_t)) \frac{\partial \ln(\pi(S_t, A_t; \theta))}{\partial \theta}$$

REINFORCE with baseline uses the state-value function $b(S_t) = v^{\pi}(S_t)$ as a control variate as it is correlated to G_t. Since REINFORCE is a Monte Carlo method, it is natural to use Monte Carlo method to estimate $\hat{v}^{\pi}(S_t)$ and learn the policy and value function together.

As update is performed only once after the end of episode, it will be very slow. To resolve this, we perform update at every step. This is not a true stochastic gradient ascent, but works in practise. Thus, update in each time step t will be:

$$\theta = \theta + \alpha\gamma^t(G_t - \hat{v}^{\pi}(S_t)\frac{\partial \ln(\pi(S_t, A_t; \theta))}{\partial\theta}$$

The pseudocode is as shown in Algorithm 1. We adopted the pseudocode in the Reinforcement Learning Book (Sutton and Barto, 2018) with specific details of our implementation.

Algorithm 1 REINFORCE with Baseline (episodic)

Input: Neural Network for policy $\pi(a|s, \boldsymbol{\theta})$
Input: Neural Network for state-value function $\hat{v}(s, \mathbf{w})$
Input: Learning rate for policy $\alpha^{\boldsymbol{\theta}} > 0$
Input: Learning rate for state-Value function $\alpha^{\mathbf{w}} > 0$
Initialize: Neural Network weights $\boldsymbol{\theta}$ and $\mathbf{w}$ to small random numbers.
  for EACH EPISODE **do**
    Follow the policy $\pi(.|., \boldsymbol{\theta})$ and generate an episode: $S_0, A_0, R_0, ..., S_{T-1}, A_{T-1}, R_{T-1}$
    $t \leftarrow 0$
    while $t < T$ **do**
      $G \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t} R_k$
      $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} * \delta * \nabla\hat{v}(S_t, \mathbf{w})$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} * \gamma^t * \delta * \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
      $t \leftarrow t + 1$
    end while
  end for

3.1 Experimentation

For the Neural Network representation of our policy and state-value function we used a single hidden layer Neural Network. For policy network, input is the state (for 687-Gridworld it is a one-hot encode of row and column number, for Acrobot it is 6 neurons for 6 elements of state) and output is one neuron for each discrete action. For state-value function input is state and output is state-value. Based on our experimentation we found that using a simple stochastic gradient descent update was slow and not performing very well. We used an adaptive learning rate optimization scheme where we build momentum and velocity for each gradient and update the gradient. Particularly, we implemented the Adam optimizer (Kingma and Ba, 2014) based gradient update instead of stochastic gradient update. We referred the pseudocode given in the pytorch documentation[2]. The optimizer was initialized with the given learning rate as the initial learning rate. We ran the algorithm for 1000 episodes as the return converged in 1000 episodes.

Hyper-parameters we tuned: $\alpha^{\mathbf{w}}, \alpha^{\boldsymbol{\theta}} \in \{10^{-2}, 10^{-3}, 10^{-4}\}$, hidden layer size of both neural networks $\in \{10, 32, 64\}$

For hyper-parameter tuning, we did a grid search on the parameter space. We selected a parameter and did a complete run of the algorithm for 2 runs. If the results were good, we ran the algorithm with the hyper-parameter for 20 times to assess the hyper-parameter and generated the plots. We then found the hyper-parameter which performed best (converged to the highest return) on average of the 20 runs of the algorithm.

3.2 Results for 687-Gridworld

The hyper-parameters which gave the best performance (highest average return) was: $\alpha^{\mathbf{w}} = 10^{-3}$, $\alpha^{\boldsymbol{\theta}} = 10^{-3}$, hidden layer size of both neural networks $= 10$. The mean return at the end of 1000 episodes was: 3.96

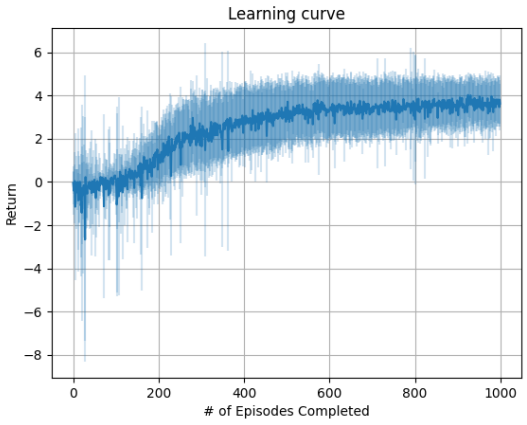

Figure 1: Episodes completed vs. Return for REINFORCE with baseline on 687-Gridworld

[2]https://pytorch.org/docs/stable/generated/torch.optim.Adam.html

As we can see in Figure 1, as the agent completes more and more episodes the return of the agent is increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 2, as the agent completes more and more actions, it continues to learn and it starts taking fewer and fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken. Once the agent has learned, the slope becomes constant.

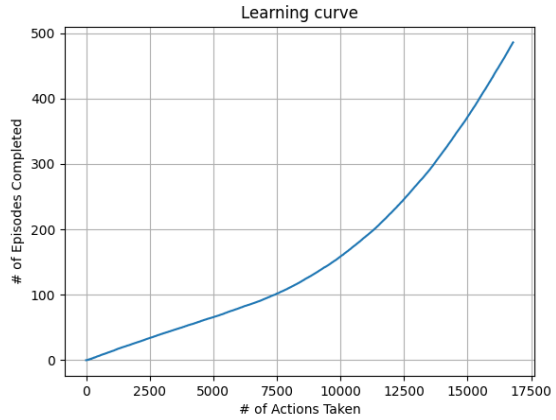

Figure 2: Actions taken vs. Episodes completed for REINFORCE with baseline on 687-Gridworld

3.3 Results for Acrobot

The hyper-parameters which gave the best performance (highest average return) was: $\alpha^{\mathbf{w}} = 10^{-3}$, $\alpha^{\boldsymbol{\theta}} = 10^{-4}$, hidden layer size of both neural networks $= 10$. The mean return at the end of 1000 episodes was: -148.32

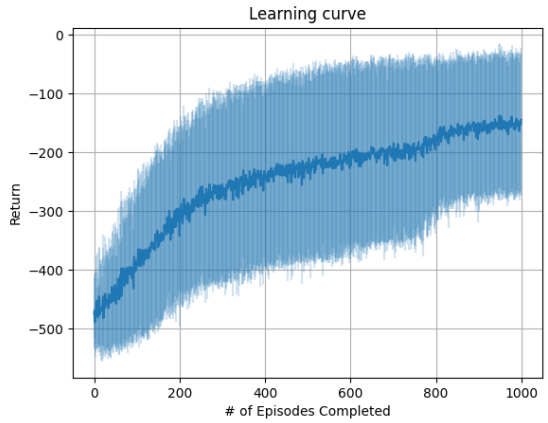

Figure 3: Episodes completed vs. Return for REINFORCE with baseline on Acrobot

As we can see in Figure 3, as the agent completes more and more episodes the return of the agent starts increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 4, as the agent takes more and more actions, it continues to learn and it starts taking fewer and fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken.

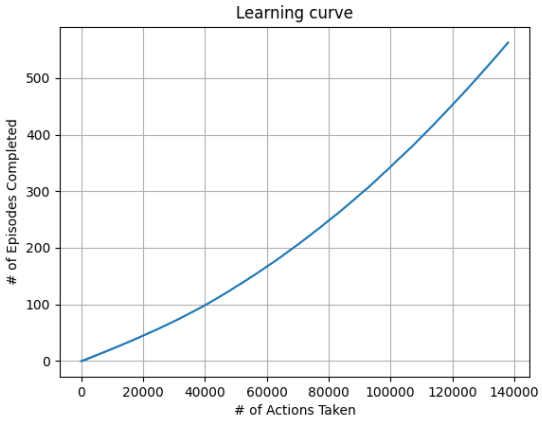

Figure 4: Actions taken vs. Episodes completed for REINFORCE with baseline on Acrobot

In Figure 3, we can see that the standard deviation is pretty high. It starts to reduce slightly as the agent continues to learn. We observed that some runs of the algorithm the return were consistently poor (it was learning, i.e., returns were increasing, but the returns were low), while in some runs the returns were very good. This may be due to the neural network getting stuck in a local optima. In average, across runs, we did saw consistent learning behavior.

4 One-Step Actor-Critic (episodic)

One-step Actor-Critic is a Policy Gradient method which is designed as a set of extensions on the REINFORCE with Baseline algorithm.

In REINFORCE with baseline, the learned state-value function estimates the value of the current state of each state transition. It is made prior to the transition's action and thus cannot be used to assess that action. In actor–critic methods, on the other hand, the state-value function is applied also to the second state of the transition. It then estimates the return (one-step return) as current reward plus the discounted state-value function of next state. Since it takes the action into considera-

tion, it is a useful way to assess the action.

One-step return has high bias but very low variance and in practise is superior to the actual observed return.

When the state-value function is used to assess actions in this way it is called a Critic, and the overall policy-gradient method is termed an Actor–Critic method. Actor refers to the policy who is taking actions.

The main appeal of one-step methods is that they are fully online and incremental. Thus, update of the policy based on one-step Actor-Critic method on a single time-step is given as:

$$\delta_t = R_t + \gamma \hat{v}^{\pi}(S_{t+1}, \mathbf{w}) - \hat{v}^{\pi}(S_t, \mathbf{w})$$

$$\theta = \theta + \alpha \gamma^t \delta_t \frac{\partial \ln(\pi(S_t, A_t; \theta))}{\partial \theta}$$

Similar, to REINFORCE with baseline the state-value function and policy learns simultaneously. Since Actor-Critic is learning online, it is natural to learn state-value function online too using semi-gradient TD(0). The pseudocode is as shown in Algorithm 2. We adopted the pseudocode in the Reinforcement Learning Book (Sutton and Barto, 2018) with specific details of our implementation.

Algorithm 2 One-step Actor–Critic (episodic)

Input: Neural Network for policy $\pi(a|s, \boldsymbol{\theta})$
Input: Neural Network for state-value function $\hat{v}(s, \mathbf{w})$
Input: Learning rate for policy $\alpha^{\boldsymbol{\theta}} > 0$
Input: Learning rate for state-value function $\alpha^{\mathbf{w}} > 0$
Initialize: Neural Network weights $\boldsymbol{\theta}$ and $\mathbf{w}$ to small random numbers.
for EACH EPISODE **do**
  Initialize S
  $I \leftarrow 1$
  while S is not terminal **do**
    $A \sim \pi(.|S, \boldsymbol{\theta})$
    Execute action A and observe S' and R
    $\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
    Perform critic update:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} * \delta * \nabla \hat{v}(S, \mathbf{w})$
    Perform actor update:
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} * I * \delta * \nabla \ln \pi(A|S, \boldsymbol{\theta})$
    $I \leftarrow I\gamma$
    $S \leftarrow S'$
  end while
end for

4.1 Experimentation

For the Neural Network representation of our policy and state-value function we used a single hidden layer Neural Network. For policy network, input is the state (for 687-Gridworld it is a one-hot encode of row and column number, for Acrobot it is 6 neurons for 6 elements of state) and output is one neuron for each discrete action. For state-value function input is state and output is state-value. Based on our experimentation we found that using a simple stochastic gradient descent update was slow and not performing very well. We used an adaptive learning rate optimization scheme where we build momentum and velocity for each gradient and update the gradient. Particularly, we implemented the Adam optimizer (Kingma and Ba, 2014) based gradient update instead of stochastic gradient update. The optimizer was initialized with the given learning rate as the initial learning rate. We ran the algorithm for 1000 episodes as the return converged in 1000 episodes.

Hyper-parameters we tuned: $\alpha^{\mathbf{w}}, \alpha^{\boldsymbol{\theta}} \in \{10^{-2}, 10^{-3}, 10^{-4}\}$, hidden layer size of policy and state-value function neural networks $\in \{10, 32, 64\}$

For hyper-parameter tuning, we did a grid search on the parameter space. We selected a parameter and did a complete run of the algorithm. If the results were good, we ran the algorithm with the hyper-parameter for 20 times to assess the hyper-parameter and generate the plots. We then found the hyper-parameter which performed best (converged to the highest return) on average of the 20 runs of the algorithm.

With Actor-Critic algorithm we found that for most of the runs of our algorithm the agent not exploring enough. For e.g., in the 687-Gridworld problem, the agent was continuously performing AttemptRight action and the algorithm was stuck as the episode never completed. To mitigate halting, we truncated the episode length to 100. Then, we observed that in most of the episode run the agent had return as 0 and was not learning. We wanted to encourage exploration so that the agent reaches the goal. To encourage exploration in the Softmax Layer of the Neural Network we introduced a parameter σ and modified the Softmax equation to the Boltzam distribution as taught in

the class.

$$\pi(s,a) = \frac{e^{\sigma\theta(s,a)}}{\sum_{a'} e^{\sigma\theta(s,a')}}$$

where, $\theta(s,a)$ is the output of the neural network before applying softmax equation. This allowed us to explicitly control exploration. As $\sigma \to 0$, the policy becomes more stochastic and as $\sigma \to \infty$ the policy becomes more deterministic. We tuned the $\sigma \in \{0.6, 1\}$

4.2 Results for 687-Gridworld

The hyper-parameters which gave the best performance (highest average return) was: $\alpha^{\mathbf{w}} = 10^{-3}$, $\alpha^{\boldsymbol{\theta}} = 10^{-3}$, hidden layer size of both neural networks $= 10$, and $\sigma = 0.6$. The mean return at the end of 1000 episode was 4.28.

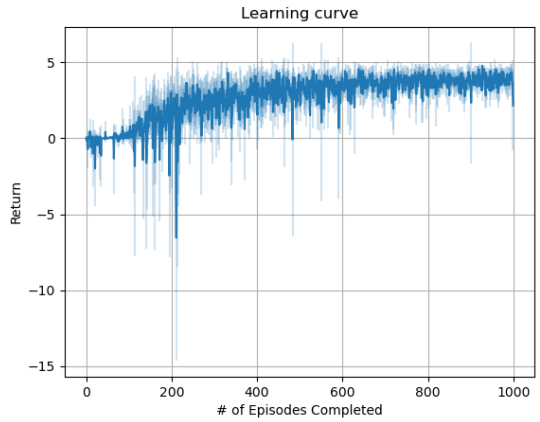

Figure 5: Episodes completed vs. Return for One-step Actor–Critic on 687-Gridworld

As we can see in Figure 5, as the agent completes more and more episodes the return of the agent is increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 6, as the agent takes more and more actions, it starts to learn and it takes fewer and fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken.

In this environment we particularly struggled when $\sigma = 1$, i.e., when we were not explicitly controlling exploration and relying on the Softmax distribution to dictate exploration. We observed that with $\sigma = 1$, in most of the runs the agent was not exploring enough and was getting stuck in the environment without reaching the goal, or terminating after episode truncated in 100 steps. To promote exploration, we set $\sigma = 0.6$. Figure 5 and 6 shows that initially due to high exploration the agent was performing very poorly in the environment. But after sufficient exploration, it found an some path to the goal state and started to learn. As the agent learned, the Softmax distribution started assigning more and more weights to the optimal action and even with $\sigma = 0.6$, the distribution continued to take the optimal discovered action instead of randomly exploring.

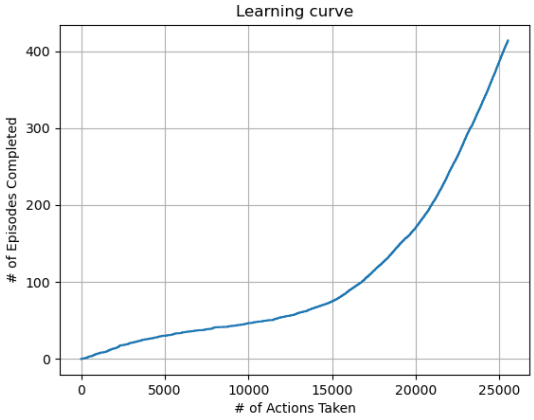

Figure 6: Actions taken vs. Episodes completed for One-step Actor–Critic on 687-Gridworld

4.3 Results for Acrobot

The hyper-parameters which gave the best performance (highest average return) was: $\alpha^{\mathbf{w}} = 10^{-3}$, $\alpha^{\boldsymbol{\theta}} = 10^{-3}$, hidden layer size of both neural networks $= 10$, and $\sigma = 1$. The mean return at the end of 1000 episodes is -178.64, but the max mean return in the 1000 episodes was -97.12.

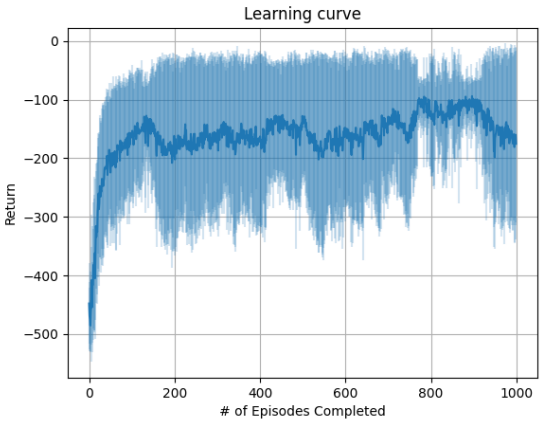

Figure 7: Episodes completed vs. Return for One-step Actor–Critic on Acrobot

As we can see in Figure 7, as the agent completes more and more episodes the return of the agent is increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 8, as the agent takes more and more actions, it continues to learn and it takes fewer and fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken. When the agent has learned the slope becomes constant.

In Figure 7, we can see that the standard deviation is pretty high. However, the standard deviation is lower when compared to the algorithm REINFORCE with baseline for the same environment. We observed that while running the algorithm, in some runs of the algorithm the return were poor (it was learning, i.e., returns were increasing, but the returns were low), while in some runs the returns were very good. We believe this may be due to the neural network getting stuck in a local optima. In average across runs, we did saw consistent learning behavior.

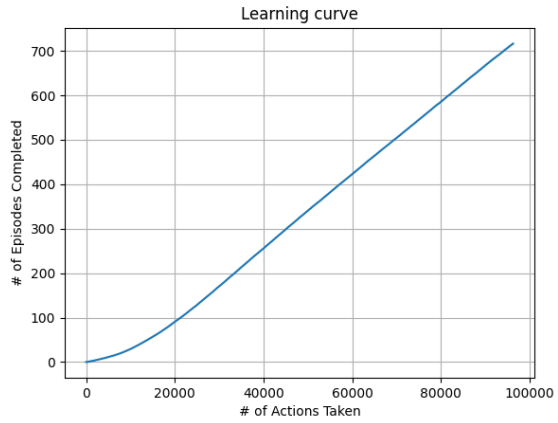

Figure 8: Actions taken vs. Episodes completed for One-step Actor–Critic on Acrobot

5 Episodic semi-gradient n-step Sarsa

Episodic semi-gradient n-step Sarsa is a Value Function Approximation algorithm which is used as a control algorithm. In these algorithms we construct a function $\hat{v}_{\mathbf{w}}$ or $\hat{q}_{\mathbf{w}}$ and train it as a neural network. It is parameterized by a vector of weights $\mathbf{w} \in \mathbb{R}$ such that in the end the neural network's estimation of $\hat{v}_{\mathbf{w}}(s)$ or $\hat{q}_{\mathbf{w}}(s,a)$ is approximately equal to the $v^{\pi}(s)$ or $q^{\pi}(s,a)$. The mean squared value error which is of importance is these algorithms is given as:

$$\overline{VE}(\mathbf{w}) = \sum_{s} \mu(s)[v^{\pi}(s) - \hat{v}_{\mathbf{w}}(s)]^2$$

The objective here is to find a local optimum $\mathbf{w}^*$ such that $\overline{VE}(\mathbf{w}^*) \leq \overline{VE}(\mathbf{w}) \; \forall \; \mathbf{w}$ in some neighbourhood of $\mathbf{w}^*$. We approach these problems using the Stochastic Gradient Descent to update the weight vector. For action-value:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[q^{\pi}(S_t, A_t) - \hat{q}_{\mathbf{w}}(S_t, A_t)]\nabla \hat{q}_{\mathbf{w}}(S_t, A_t)$$

In case of TD algorithms the $q^{\pi}(S_t, A_t)$ is not an independent of $\mathbf{w}$, but still we assume to not depend on $\mathbf{w}$ and use it in the update equation thus, this becomes a Semi-Gradient approach. Thus the update equation for a episodic Semi-gradient 1-step Sarsa method would be:

$$\begin{aligned}&\mathbf{w}_{t+1} = \\ &\mathbf{w}_t + \alpha[R_t + \gamma * \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) \\ &- \hat{q}(S_t, A_t, \mathbf{w}_t)]\nabla \hat{q}(S_t, A_t, \mathbf{w}_t)\end{aligned}$$

For an n-step version of episodic semi-gradient Sarsa we use an n-step return as the update target in the semi-gradient Sarsa update equation like:

$$\begin{aligned}&\mathbf{w}_{t+n} = \\ &= \mathbf{w}_{t+n-1} + \alpha[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})] \\ &\nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\end{aligned}$$

such that $0 \leq t < T$ and where, the n-step return $G_{t:t+n}$ is for $t + n < T$:

$$\begin{aligned}&G_{t:t+n} = R_t + \gamma * R_{t+2} + ... + \gamma^{n-1} * R_{t+n-1} \\ &+ \gamma^n * \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1})\end{aligned}$$

and $G_{t:t+n} = G_t$ for $t + n \geq T$ where T is the length of the episode.

To form control methods, we need to couple this action-value prediction methods with techniques for policy improvement and action selection. For each possible action a available in the next state S_{t+1}, we can compute $\hat{q}(S_{t+1}, a, \mathbf{w}_t)$ and then find the greedy action $A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t)$. Policy improvement is then done by changing the estimation policy to a soft approximation of the greedy policy such as the ϵ-greedy policy. Actions are selected according to this same policy.

The pseudocode is as shown in Algorithm 3. We adopted the pseudocode in the Reinforcement Learning Book (Sutton and Barto, 2018) with specific details of our implementation.

Algorithm 3 Episodic semi-gradient n-step Sarsa

Input: Neural Network for action-value function $\hat{q} : S \times A \times \mathbb{R}^d \rightarrow \mathbb{R}$
Input: Learning rate for policy $\alpha > 0$ and $\epsilon > 0$.
Input: Positive integer $n > 0$
Initialize: Neural Network weights $\mathbf{w}$ to small random numbers.
  for EACH EPISODE **do**
    Initialize S, ensure S_0 is not terminal
    Select and Store action A_0 based on ϵ-greedy action selection w.r.t $\hat{q}(S_0, ., \mathbf{w})$
    $T \leftarrow \infty$
    for t = 0, 1, 2... **do**
      if $t < T$ (current time step less than the end of episode) **then**
        Execute action A_t
        Observe and Store next reward R_t and next state S_{t+1}.
        if S_{t+1} is terminal **then**
          $T \leftarrow t + 1$
        else
          Select and Store action A_{t+1} based on ϵ-greedy action selection w.r.t $\hat{q}(S_{t+1}, ., \mathbf{w})$
        end if
      end if
      $\tau \leftarrow T - n + 1$ (Estimate of the τ time-step is being updated here)
      if $\tau = T - 1$ **then**
        break
      end if
      if $\tau \geq 0$ **then**
        Compute the discounted reward from the next n time step: $G \leftarrow \sum_{i=\tau}^{\min(\tau+n-1, T-1)} \gamma^{i-\tau} R_i$
        if $\tau + n < T$ **then**
          Update estimate of G with discounted reward from the next n time step and q-value function of time step at $n + 1$ as : $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$
        end if
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha * [G - \hat{q}(S_\tau, A_\tau, \mathbf{w})] * \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
      end if
    end for
  end for

5.1 Experimentation

For the Neural Network representation of action-value function we used a single hidden layer Neural Network. For neural network, input is the state and action (for 687-Gridworld it is a one-hot encode of row and column number + one-hot encode of 4 discrete actions, for Acrobot it is 6 neurons for 6 elements of state + one-hot encode of 3 discrete actions) and output is action-value. Based on our experimentation we found that using a simple stochastic gradient descent update was slow and not performing very well. We used an adaptive learning rate optimization scheme where we build momentum and velocity for each gradient and update the gradient. Particularly, we implemented the Adam optimizer (Kingma and Ba, 2014) based gradient update instead of stochastic gradient update. The optimizer was initialized with the given learning rate as the initial learning rate. We ran the algorithm for 1000 episodes as the return converged in 1000 episodes.

Hyper-parameters we tuned:

- learning rate $\alpha \in \{10^{-2}, 10^{-3}\}$
- the n in the n-step for the Sarsa $\in \{5, 10\}$
- in case of using ϵ greedy policy without decay we used $\epsilon \in \{0.01, 0.05\}$
- in case of using ϵ greedy policy with decay we set $\epsilon = 1$ and $\beta \in \{0.9, 0.95, 0.97\}$

We kept the hidden layer size of action-value function neural networks to be 10. While decaying the ϵ value we decayed it by a factor of β after every 10 episodes using exponential decay, i.e., after completing 10, 20, 30, ... episodes, $\epsilon = \epsilon * \beta$.

For hyper-parameter tuning, we did a grid search on the parameter space. We selected a parameter and did a complete run of the algorithm. If the results were good, we ran the algorithm with the hyper-parameter for 20 times to assess the hyper-parameter and generate the plots. We then found the hyper-parameter which performed best (converged to the highest return) on average of the 20 runs of the algorithm.

5.2 Results for 687-Gridworld

5.2.1 Fixed ϵ

The hyper-parameters which gave the best performance (highest average return) in the case when we did not decay the ϵ value were: $\alpha = 10^{-3}$,

n value of n-step $= 5$, hidden layer size of neural network $= 10$ and ϵ value $= 0.05$. The mean return at the end of 1000 episode is 3.4.

As we can see in Figure 9, as the agent completes more and more episodes the return of the agent is increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 10, as the agent takes more and more actions, it starts to learn and it takes fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken.

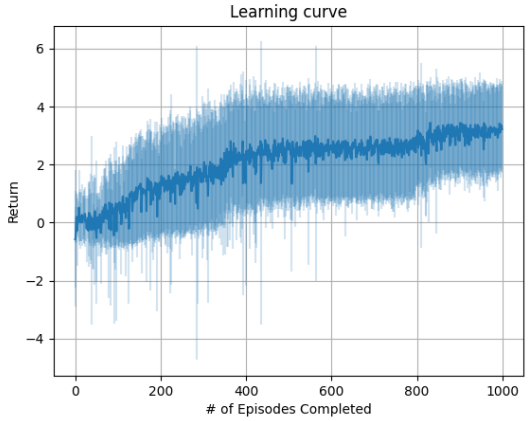

Figure 9: Episodes completed vs. Return for n-step Sarsa without ϵ decay on 687-Gridworld

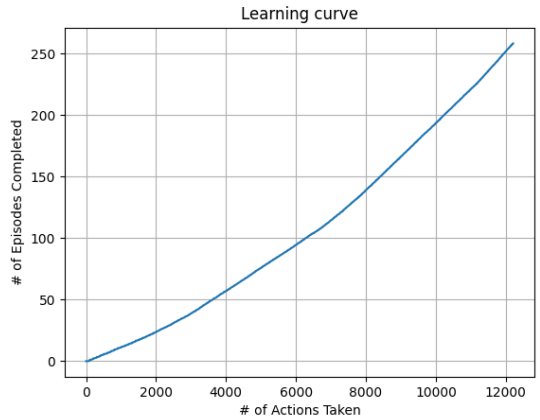

Figure 10: Actions taken vs. Episodes completed for n-step Sarsa without ϵ decay on 687-Gridworld

5.2.2 Decaying ϵ

The hyper-parameters which gave the best performance (highest average return) in the case when we decayed the ϵ value every 10 iterations were: $\alpha = 10^{-3}$, n value of n-step $= 5$, hidden layer size of neural network $= 10$ and ϵ value $= 1$, β value $= 0.9$. The mean return at the end of 1000 episode is 3.7.

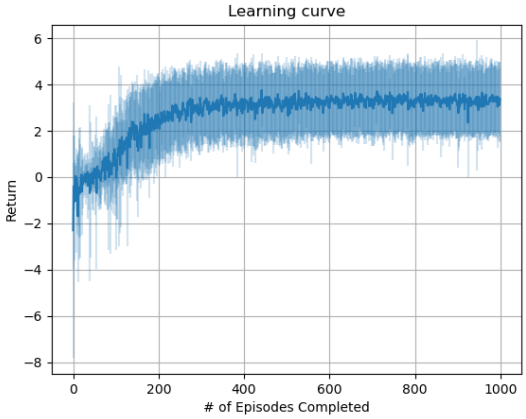

Figure 11: Episodes completed vs. Return for n-step Sarsa with ϵ decay on 687-Gridworld

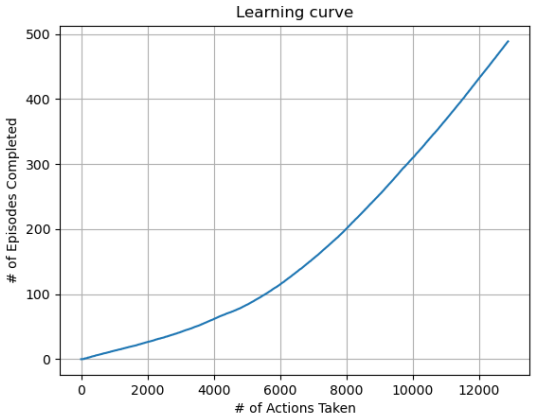

Figure 12: Actions taken vs. Episodes completed for n-step Sarsa with ϵ decay on 687-Gridworld

As we can see in Figure 11, as the agent completes more and more episodes the return of the agent increases. In this environment return is proportional to the number of steps the agent will take to reach the goal. When we compare this to the Figure 9 we can see that when we decay the ϵ value by the β factor every 10 iterations the standard deviation is reduced to quite a lot of extent and the average return of 4 is reached much faster. Also, the agent performs poorly in the initial few episodes, but then it quickly picks up the optimal policy. Thus, high exploration in initial few episodes proved beneficial to the agent.

As we can see in Figure 12, as the agent takes more and more actions, it starts to learn and

it takes fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken. When we compare this to the Figure 10 we can see that learning is low in initial episodes but picks up faster when agent switches from explore to exploit mode.

5.3 Results for Acrobot

5.3.1 Fixed ϵ

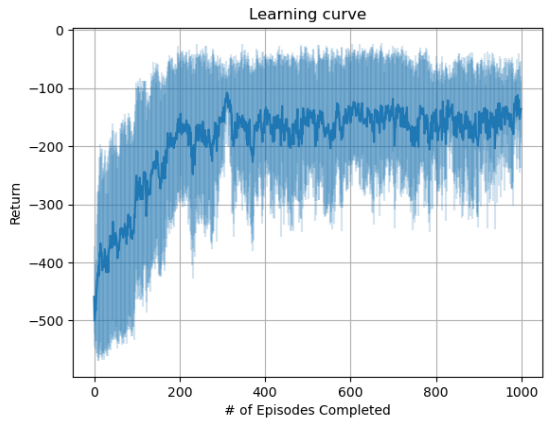

Figure 13: Episodes completed vs. Return for n-step Sarsa without ϵ decay on Acrobot

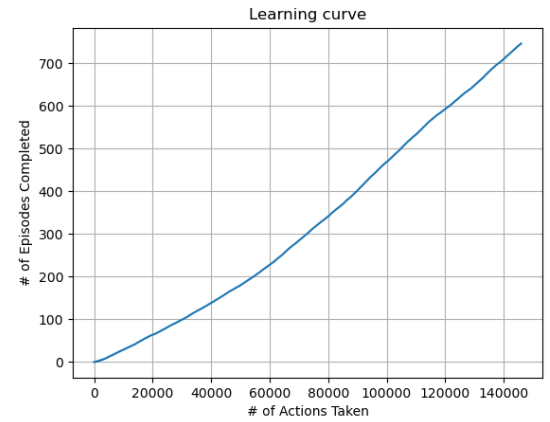

Figure 14: Actions taken vs. Episodes completed for n-step Sarsa without ϵ decay on Acrobot

The hyper-parameters which gave the best performance (highest average return) in the case when we did not decay the ϵ value were: $\alpha = 10^{-2}$, n value of n-step $= 10$, hidden layer size of neural network $= 10$ and ϵ value $= 0.05$. The mean return at the end of 1000 episodes is -133.70

As we can see in Figure 13, as the agent completes more and more episodes the return of the agent starts increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 14, as the agent takes more and more actions, it continues to learn and it starts taking fewer and fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken.

5.3.2 Decaying ϵ

The hyper-parameters which gave the best performance (highest average return) in the case when we decayed the ϵ value every 10 iterations were: $\alpha = 10^{-2}$, n value of n-step $= 5$, hidden layer size of neural network $= 10$ and ϵ value $= 1$, β value $= 0.97$. The mean return at the end of 1000 episodes is -140.62

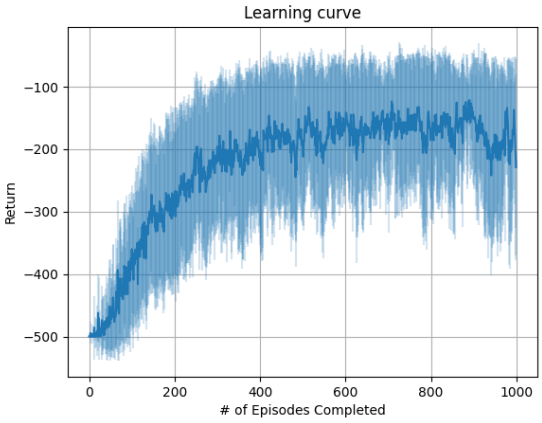

Figure 15: Episodes completed vs. Return for n-step Sarsa with ϵ decay on Acrobot

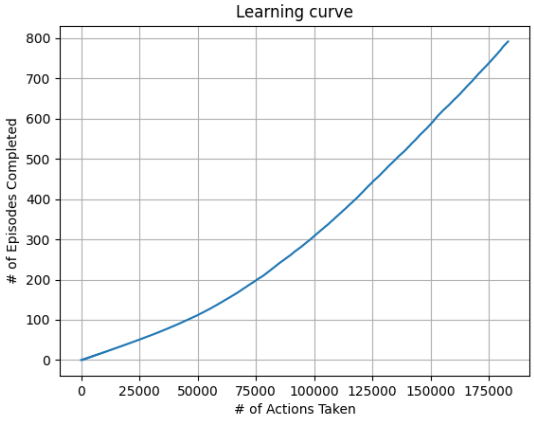

Figure 16: Actions taken vs. Episodes completed for n-step Sarsa with ϵ decay on Acrobot

As we can see in Figure 15, as the agent com-

pletes more and more episodes the return of the agent starts increasing. Thus, the agent is learning. In this environment return is proportional to the number of steps the agent will take to reach the goal. As we can see in Figure 16, as the agent takes more and more actions, it continues to learn and it starts taking fewer actions to complete the episodes. Thus, we observe an increasing slope with number of actions taken.

Decaying ϵ didn't significantly benefit learning or made it faster in this environment. We observe that initially the agent is performing poorly when it is in exploration phase and discovering what different actions has to offer and then when it switches to exploitation phase it starts to learn faster.

In the Figure 13 and Figure 15 we can see that the standard deviation is pretty high. We observed that some runs of the algorithm the return were consistently poor (it was learning, i.e., returns were increasing, but the returns were low), while in some runs the returns were very good and it reached the maximum value. As mentioned above, this may be because the neural network might get stuck in a local optima. In average, across runs, we did see a consistent learning behavior. When compared to the Actor Critic and REINFORCE with Baseline algorithms we can say that the n-step Sarsa algorithm had comparatively lesser standard deviation w.r.t. both these algorithms.

References

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.