Analysis of Reinforcement Learning Algorithms for Continuous-State MDPs

Final Project - COMPSCI 687 - Reinforcement Learning

Kartik Choudhary[*]
University of Massachusetts Amherst

Rohan Acharya[*]
University of Massachusetts Amherst

December 12, 2022

Abstract

In this course we have primarily covered tabular reinforcement learning algorithms for finite MDPs where both the state space $\mathcal{S}$ and action space $\mathcal{A}$ are finite. Towards the end of the course we learned about value approximation for dealing with continuous state spaces, and policy gradient algorithm that works well when both the state and action space are continuous or extremely large. In this project we will examine three new algorithms - REINFORCE with Baseline, One-Step Actor-Critic for episodic tasks, and Episodic Semi-Gradient n-step SARSA which work for continuous state and action spaces, analyze the strengths and weaknesses of these algorithms, and compare their performance on environments from OpenAI Gym (Brockman et al., 2016).

1 Introduction

The general setting in a reinforcement learning problem is modeled as an agent interacting with an environment by taking actions based on a policy, and receiving a reward and transitioning to a new state after every interaction. This might go on indefinitely or terminate after some number of steps. The primary goal in reinforcement learning is to learn a policy that, when used to interact with the environment, will maximize the rewards the agent gets. Many reinforcement learning problems are modeled as Markov Decision Processes (MDPs), where the future is conditionally independent of the past given a sufficient aggregate of the present, which we call the state.

The Markov assumption simplifies the design of the solutions since we do not need to worry about the past states and actions if the current state is known. Algorithms that use the Markov assumption (e.g., n-step methods, Q-learning) can usually learn an optimal policy in a more efficient manner than those that do not (e.g. PI^2-CMA-ES (Stulp and Sigaud, 2012)).

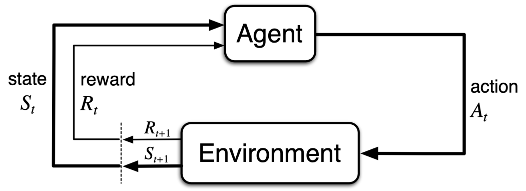

Figure 1: The agent-environment interaction in a Markov decision process. Source: Sutton and Barto (2018)

The tabular methods like Monte Carlo, SARSA, and Q-Learning that we have seen during the course, work well when the state and actions spaces are small. These methods work by learning the state-action value functions, i.e., Q-functions for all the state-action pairs and build a policy that is greedy with respect to that function, which, if the Q-function estimate is correct, produces an optimal deterministic policy

[*]Equal contribution

[1]. The biggest limitation with these methods is that they store the estimate of each state-action pair in a table, which is only updated when that state-action pair is visited during learning, so they don't scale well to larger state spaces, which limits their utility in problems with very large (or possibly infinite) state spaces.

The key strategy to dealing with very large or continuous state spaces is to use a parametrized function for state-action value function, which gives us a compact representation that can be learned more easily. Using such a parametrized function also means that updating the parameters based on experience from one state affects the estimates of many states, which makes learning more efficient, and can help the algorithm generalize better.

The algorithm that we will look at that uses a parametrized state-action value function is called the semi-gradient n-step SARSA algorithm, that is designed to be used in episodic settings.

Another way to learn a good policy in settings with large state spaces is to directly learn a parametrized policy along with a differentiable function that gives the "goodness" of any policy, and then update that policy using a gradient-based method.

The REINFORCE with baseline and one-step actor-critic algorithms are two such examples that we will look at in this study for episodic tasks.

We will use the Cart-Pole and Mountain Car environments from OpenAI Gym (Brockman et al., 2016) to compare these algorithms and identify the strengths and weaknesses of the different algorithms, as well as show what to look out for when using these methods when training on any problem.

2 Methodology

We have implemented all three algorithms in Python 3.10.6 using the standard NumPy and SciPy libraries to do the various computations and hold the weight vectors of different parametrized functions. In total we did six experiments - training agents using each of the three algorithms for both the MDPs. We used linear function approximation in all the experiments with the Fourier basis functions for representing the states and/or state-action pairs.

[1]provided that the state and action spaces are finite and the rewards are bounded

For each experiment, we started with a simple baseline - a fixed small step size parameter and order-1 Fourier basis functions. After that, we individually vary different parameters - such as step size schedules, exploration strategies, order of basis functions, etc., wherever applicable, and show the learning characteristics for the different configurations.

Finally, we show the results with configurations that we believe are the best amongst those that can be trained in a reasonable amount of time, and report any interesting observations that we made.

We have used the Fourier basis functions (see Section 3) for representing state-action pairs. Since the actions in both MDPs are integers, ($\{0, 1\}$ for cart-pole and $\{0, 1, 2\}$ for mountain car), and cosines do not change if shifted by integer multiples of 2π (i.e., $\cos(\pi a) = \cos\big(\pi(a + 2)\big)$), we mapped these actions to smaller numbers while learning as given below

$$0 \mapsto 0.1, \qquad 1 \mapsto 0.17, \qquad 2 \mapsto 0.23$$

The predicted actions by the algorithms were then remapped to the original integers before passing to the environment to get the reward and next state. This made the feature vectors $\phi(s, a_1)$ and $\phi(s, a_2)$ different when the difference between actions a_1 and a_2 was an even integer.

For the mountain car problem, both dimensions of the observed states were normalized to be between $[0, 1]$ which meant that feature values would not be dominated by one of the features and improve the learning speed for the algorithms.

3 Feature Construction

Linear models often do not work well if used with raw features, since they can represent only a very small family of features. To allow our models to represent more complex functions, we use more expressive feature representations, such as polynomial or Fourier features.

Both polynomial and Fourier features can represent any smooth function with arbitrary degree of precision with a high enough order, but we found the polynomial features did not work well and we used the Fourier basis functions to represent states and state-action pairs. If the state has m raw features - $[x_1, x_2, \ldots x_m]$, then the order-n Fourier features for the state would be all the possible combinations of

$$\phi(s) = \left[\cos\left(\pi \sum_{i=1}^{m} a_i x_i\right) \right]^\top \quad a_i \in \{0, 1, 2, \ldots, m\}$$

For example, the order-2 Fourier features for state with 2 raw features x_1, x_2, i.e., $n = 2$ and $m = 2$, would be

$$\begin{aligned}\phi(s) = [&1, \cos(\pi x_1), \cos(\pi x_2), \cos\big(\pi(x_1 + x_2)\big),\\ &\cos(2\pi x_1), \cos(2\pi x_2), \cos\big(\pi(2x_1 + x_2)\big),\\ &\cos\big(\pi(x_1 + 2x_2)\big), \cos\big(\pi(2x_1 + 2x_2)\big)]^\top\end{aligned}$$

We can see that even in such a small example the basis vector is large, in fact, the size of order-n Fourier feature vector for states of size m is $(n+1)^m$. This exponential scaling limits the order we can take before learning becomes too computationally expensive. Fortunately, lower order basis features are expressive enough for the problems we are solving in this project, further details are provided in Section 7

When using Fourier features, we also need to change the step size parameter for different dimensions. According to Konidaris et al. (2011), for the feature $\phi_i(s) = \cos\Big(\pi\big(c_1^{(i)} x_1 + \ldots c_m^{(i)} x_m\big)\Big)$ this was done according to the rule

$$\alpha_i = \frac{\alpha}{\sqrt{\left(c_1^{(i)}\right)^2 + \cdots + \left(c_m^{(i)}\right)^2}}$$

Where α is the base step size parameter, except when each $c_j^{(i)}$ is zero, in which case $\alpha_i = \alpha$. The details of tuning the schedule for α is given in Section 6. For more details on the basis functions used and different step sizes of dimensions, and other implementation details, see Appendix A.

4 Environments

All three algorithms were tested on two different environments from OpenAI Gym (Brockman et al., 2016), with continuous states and discrete action space. These are episodic MDPs with the discount factor $\gamma = 1$ for both of them.

4.1 Cart-Pole

This MDP is based on the cart-pole problem by Barto et al. (1983), as shown in Figure 3. A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pole is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

There are two possible actions, corresponding to applying force to the cart in the left or right direction. The state space is a 4-dimensional continuous space. The components of the 4 dimensions are the following

1. Position of the cart along the 1-D track: x
2. Velocity of the cart along the track: $\frac{dx}{dt}$
3. Pole angle: θ
4. Pole angular velocity: $\frac{d\theta}{dt}$

We want to apply force to the cart in a manner such that the pole it balanced upright. A reward of +1 is received for each time step until the episode terminates. The episode terminates when the x position of the cart goes outside the range $(-2.4, 2.4)$, when the pole angle goes outside the range $(-24°, 24°)$, or if the maximum episode length of 500 steps is reached.

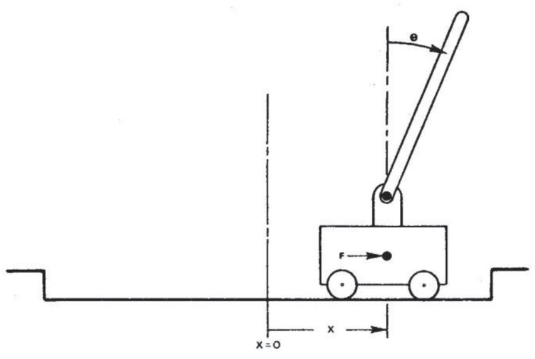

Figure 2: Diagram of the cart-pole system. Source: Barto et al. (1983)

The initial state is stochastic with each of the four parameters of the state sampled randomly and independently from the range

The velocity that is reduced or increased by the applied force depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it. The environment implements the dynamics based on the equations by Florian (2005). More details are given in Appendix B.

4.2 Mountain Car

This MDP consists of a car placed at the bottom of a sinusoidal valley, and the goal is to reach the flag at the top of the mountain, based on the description by Moore (1990). There are 3 possible actions - accelerate left, accelerate right, and do nothing. The states are continuous 2-D values, with the first dimension corresponding to the position of the car along the x-axis, and the second dimension corresponding to the velocity of the car.

The rewards are -1 for every time step until the episode ends. The goal of the agent is to try to get to the top of the mountain in as few steps as possible.

The car is not powerful enough to start from rest at the bottom of the valley and directly reach the goal, so it will need to gain momentum (perhaps by backing up first) before being able to reach the top. This is the same MDP that we used in multiple assignments during the course, except that now, the episodes have a maximum length of only 200 time steps before they are terminated, instead of 1000 steps that we had during the course.

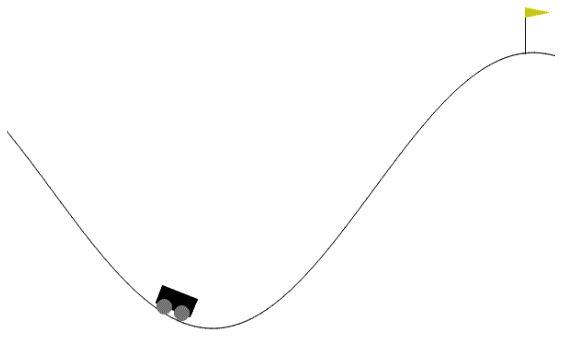
Figure 3: Diagram of the mountain car problem. Source: Brockman et al. (2016)

The stochastic initial state is $(X_0, 0)$ where X_0 is sampled uniformly from between $[-0.6, 0.4]$, while the initial velocity is always zero. The x position of the car is clipped to be in between $[-1.2, 0.5]$, while the velocity v is clipped between $[-0.07, 0.07]$. To speed up learning, during learning, we modified the source code to sample the initial state randomly from all the possible states. The episode ends when the car reaches the goal ($x \geq 0.5$) or when the maximum of 200 steps have been taken.

The dynamics of the system are deterministic given by the following equations.

$$v_{t+1} = v_t + (a-1)0.001 - \cos(3x_t)0.0025$$
$$x_{t+1} = x_t + v_{t+1}$$

Where a is the action

$$a = \begin{cases} 0 & \text{Accelerate left} \\ 1 & \text{Do nothing} \\ 2 & \text{Accelerate right} \end{cases}$$

If the x position of the car goes below -1.2 or above 0.5, then it is reset to -1.2 or 0.5 respectively, and velocity set to 0 to simulate inelastic collision.

5 Algorithms

5.1 Episodic semi-gradient n-step SARSA

The episodic semi-gradient n-step SARSA algorithm is an action-value function approximation algorithm, which is an extension of SARSA, but now instead of a tabular function, we learn a parametrized Q-function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \mapsto \mathbb{R}$, and our goal is to learn the optimal weights $\boldsymbol{w} \in \mathbb{R}^d$ that maximize the expected returns for our agent when following a policy that is greedy w.r.t. our learned function $\hat{q}$.

Since this is an n-step algorithm, we bootstrap after observing n rewards to complete our value estimate. The n-step return is given as

$$G_{t:t+n} = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \boldsymbol{w}_{t+n-1})$$

The update equation for the weights is then given as

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

This is called a semi-gradient method because for the update to be a true gradient update, we needed $G_{t:t+n}$ to be an unbiased estimator of the true state-action value function. Since we are bootstrapping, the term $G_{t:t+n}$ is dependent on our current weights, making this a semi-gradient method.

The pseudocode is given in Algorithm 1. We need to set two hyper-parameters for this - the step size α, and the parameter n.

Algorithm 1 Pseudocode for semi-gradient n-step SARSA algorithm in episodic setting (Modified from RL Book [Sutton and Barto, 2018])

```
1: Input: A differentiable action-value function parametrization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \mapsto \mathbb{R}$
2: Algorithm Parameter: Step size $\alpha > 0$
3: Algorithm Parameter: Small $\epsilon > 0$
4: Algorithm Parameter: Positive integer $n$
5:
6: Initialize value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\boldsymbol{w} = \mathbf{0}$)
7: All store and access operations ($S_t, A_t$, and $R_t$) can take their index mod $n + 1$
8:
9: for each episode do
10:     Initialize state $S_0 \neq$ terminal
11:     Select action $A_0 \sim \epsilon$-greedy wrt $\hat{q}(S_0, \cdot, \boldsymbol{w})$
12:     $T \leftarrow \infty$
13:
14:     for $t = 0, 1, 2, \ldots$ until $\tau = T - 1$ do
15:         if $t < T$ then
16:             Take action $A_t$
17:             Observe next reward $R_{t+1}$ and next state $S_{t+1}$
18:             if $S_{t+1}$ is terminal then
19:                 $T \leftarrow t + 1$
20:             else
21:                 Select action $A_{t+1} \sim \epsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \boldsymbol{w})$
22:             end if
23:         end if
24:         $\tau \leftarrow t - n + 1$   ▷ $\tau$ it the time whose estimate is being updated
25:         if $\tau \geq 0$ then
26:             $G \leftarrow \sum_{i=r+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
27:             if $\tau + n < T$ then
28:                 $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \boldsymbol{w})$
29:             end if
30:             $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G - \hat{q}(S_\tau, A_\tau, \boldsymbol{w})] \times \nabla\hat{q}(S_\tau, A_\tau, \boldsymbol{w})$
31:         end if
32:     end for
33: end for
```

5.2 REINFORCE with baseline

The REINFORCE algorithm is a policy-gradient method, which means that we learn a parametrized policy function π with weights $\boldsymbol{\theta}$ such that for any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, $\pi(a|s;\boldsymbol{\theta})$ gives the probability of taking action a while in state s.

The policy gradient theorem tells us that the gradient of the performance J w.r.t. the policy weights $\boldsymbol{\theta}$ when in state S_t, taking action A_t and getting return G_t is proportional to the following term

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \Big[G_t \frac{\nabla \pi(A_t|S_t;\boldsymbol{\theta})}{\pi(A_t|S_t;\boldsymbol{\theta})} \Big]$$

Algorithm 2 Pseudocode for REINFORCE with baseline algorithm in episodic setting (Modified from RL Book [Sutton and Barto (2018)])

```
1: Input: A differentiable policy parametrization $\pi(a|s,\boldsymbol{\theta})$
2: Input: A differentiable state-value function parametrization $\hat{v}(s,\boldsymbol{w})$
3: Algorithm Parameter: Step size $\alpha^{\boldsymbol{\theta}} > 0$
4: Algorithm Parameter: Step size $\alpha^{\boldsymbol{w}} > 0$
5:
6: Initialize value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\boldsymbol{w} = \mathbf{0}$)
7: Initialize policy paramters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
8:
9: for each episode do
10:     Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
11:
12:     for each step of the episode $0, 1, \ldots, T-1$ do
13:         $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
14:         $\delta \leftarrow G - \hat{v}(S_t,\boldsymbol{w})$
15:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t,\boldsymbol{w})$
16:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
17:     end for
18: end for
```

The REINFORCE algorithm uses the sample return from an episode as an unbiased estimate for the expected return, and is therefore a Monte Carlo method. Since it is a Monte Carlo method, it generally has a high variance which can make learning slow. To reduce the variance, and speed up learning, a baseline can be estimated that is correlated with the return and does not have a very high variance of its own. In our case, we will use the value estimate $\hat{v}$ as the baseline, which is itself a parametrized function with weights $\boldsymbol{w}$ that are also learned learned along with the policy function weights.

Thus, the update equation for the policy function weights $\boldsymbol{\theta}$ when in state S_t, taking action A_t, and observing return G_t are given as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \Big(G_t - \hat{v}(S_t;\boldsymbol{w}) \Big) \frac{\nabla \pi(A_t|S_t;\boldsymbol{\theta}_t)}{\pi(A_t|S_t;\boldsymbol{\theta}_t)}$$

Where α is the step size. The pseudocode is given in Algorithm 2 where we need to set two hyper-parameters: $\alpha^{\boldsymbol{\theta}}$ - the step size for updating $\boldsymbol{\theta}$, and $\alpha^{\boldsymbol{w}}$ - the step size for updating $\boldsymbol{w}$.

5.3 One-step actor-critic

One-step actor-critic is another policy-gradient method that can be though of as the TD-learning analogue to the REINFORCE algorithm. This means that instead of waiting for the episode to end to get the true return, we bootstrap with our current estimate of the value function to get the estimated return.

This is a one-step method, which means we bootstrap after observing the immediate reward, so our update rule becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \Big(R_{t+1} + \gamma \hat{v}(S_{t+1};\boldsymbol{w}) - \hat{v}(S_t,\boldsymbol{w}) \Big) \times \frac{\nabla \pi(A_t|S_t;\boldsymbol{\theta}_t)}{\pi(A_t|S_t;\boldsymbol{\theta}_t)} \big)$$

We can see that the policy is deciding which action to take, making it the actor, while the value function evaluates how good the action was, making it the critic, thus giving the algorithm its name: actor-critic method. The pseudocode is given in Algorithm 3, and once again we need to set two hyper-parameters: $\alpha^{\boldsymbol{\theta}}$ - the step size for updating $\boldsymbol{\theta}$, and $\alpha^{\boldsymbol{w}}$ - the step size for updating $\boldsymbol{w}$.

Algorithm 3 Pseudocode for one-step actor-critic algorithm in episodic setting (Modified from RL Book [Sutton and Barto (2018)])

```
1: Input: A differentiable policy parametrization $\pi(a|s,\boldsymbol{\theta})$
2: Input: A differentiable state-value function parametrization $\hat{v}(s,\boldsymbol{w})$
3: Algorithm Parameter: Step size $\alpha^{\boldsymbol{\theta}} > 0$
4: Algorithm Parameter: Step size $\alpha^{\boldsymbol{w}} > 0$
5:
6: Initialize value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\boldsymbol{w} = \mathbf{0}$)
7: Initialize policy paramters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
8:
9: for each episode do
10:     Initialize $S$        ▷ (First state of episode)
11:     $I \leftarrow 1$
12:     while $S$ is not terminal do
13:         $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
14:         Take action $A$, observe $S', R$
15:         $\delta \leftarrow R + \gamma\hat{v}(S',\boldsymbol{w}) - \hat{v}(S,\boldsymbol{w})$
16:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}}\delta\nabla\hat{v}(S,\boldsymbol{w})$
17:         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla \ln \pi(A|S,\boldsymbol{\theta})$
18:         $I \leftarrow \gamma I$
19:         $S \leftarrow S'$
20:     end while
21: end for
```

6 Tuning Hyper-Parameters

For the mountain car problem, we observed that the agent only receives a positive reinforcement signal (compared to the lowest possible total reward) when the agent successfully reaches the goal, which might slow down learning since the agent does not get any useful feedback if the episode terminates without reaching the goal state, even if the agent got very close to the goal state. To counter this, we tried using a modified reward function as suggested by Antonov (2020), but it did not lead to any significant improvement in learning speed, so in the end we used the original rewards without modification.

For the REINFORCE with baseline algorithm with the mountain car problem, we also experimented with only updating the weights of different parametrized functions only in the episodes where the agent reached the goal state, and keeping the weights unchanged after episodes where the episode ended in failure to reach the goal after the maximum of 200 steps. The idea was that we should not be updating the weights based on reward signal that does not tell us if our actions were good or not. This did not help with the learning process, so we decided to update the weights after every episode, since our idea was not very principled anyway.

The graphs in the following subsections are plotted by taking a running average of the values for a window of size 100, in order to reduce noise and make it easier to compare different plots.

6.1 Episodic semi-gradient n-step SARSA

As mentioned in Section 5.1 we have to set two hyper-parameters for the algorithm: the step size α, and the parameter n. Larger values of α would mean we are making large changes in our weights based on the sampled rewards. This can make training unstable since we are using estimates of the expected returns, instead of the true expected returns. Keeping α too small, on the other hand, would slow down learning and the algorithm would take longer to converge.

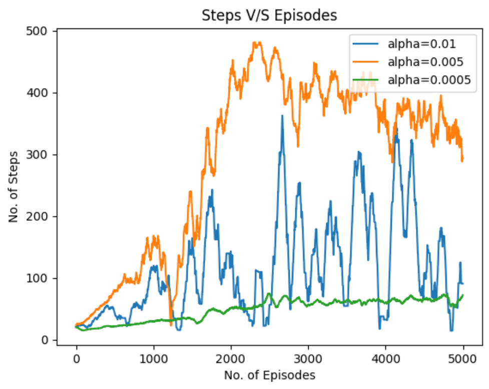

Figure 4: Learning curve on the cart-pole problem with the semi-gradient n-step SARSA algorithm for different values of α

As we can see in Figure 4, for the cart-pole problem, the value of $\alpha = 0.005$ gives the best results, where the learning is both fast and stable.

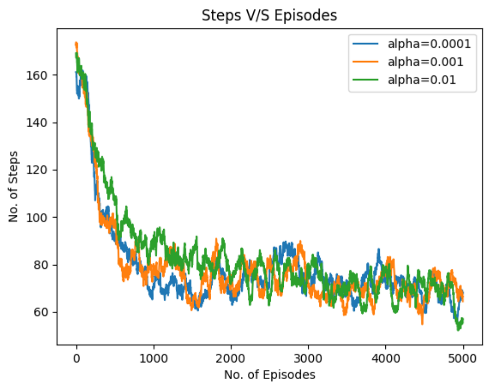

Figure 5: Learning curve on the mountain car problem with the semi-gradient n-step SARSA algorithm for different values of α

As we can see in Figure 5, for the mountain car problem, different values of α do not seem to have a big impact on the learning rate, but we observed that with higher learning rates the algorithm can sometimes diverge, and the performance might degrade in the middle of learning. Finally we decided on $\alpha = 0.001$ which reliably learns a good policy in multiple different runs.

Looking at the parameter n, larger values of n would mean our algorithm is more Monte Carlo-like, which means the estimate $G_{t:t+n}$ has lower bias but higher variance. Generally it is observed that some intermediate values of n perform the best.

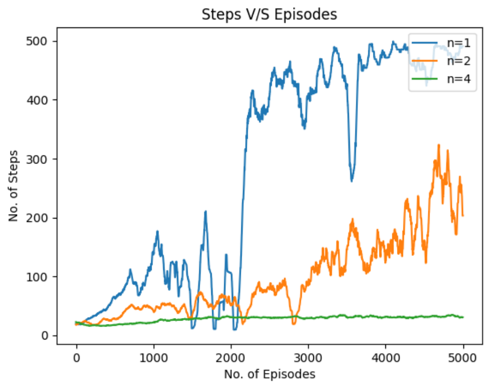

Figure 6: Learning curve on the cart-pole problem with the semi-gradient n-step SARSA algorithm for different values of n

As we can see in Figure 6, for the cart-pole problem, the best results come for the value of $n = 1$, which might indicate that having lower variance is more important in this case than having less bias in the estimate.

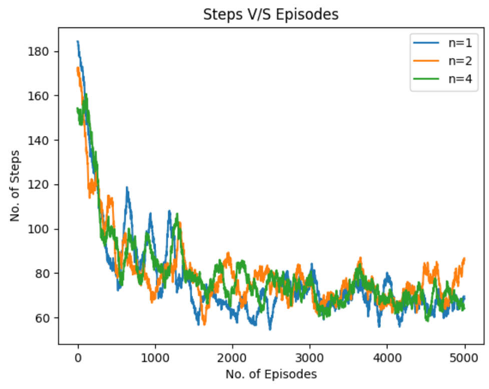

Figure 7: Learning curve on the mountain car problem with the semi-gradient n-step SARSA algorithm for different values of n

As we can see in Figure 7, for the mountain car problem, keeping $n = 2$ makes the learning more stable while still learning quickly.

6.2 REINFORCE with baseline

As mentioned in Section 5.2, we need to set two hyper-parameters: $\alpha^{\boldsymbol{\theta}}$ - the step size for updating $\boldsymbol{\theta}$, and $\alpha^{\boldsymbol{w}}$ - the step size for updating $\boldsymbol{w}$. Once again, keeping step size too high can introduce instabilities, while keeping them too low can slow down learning.

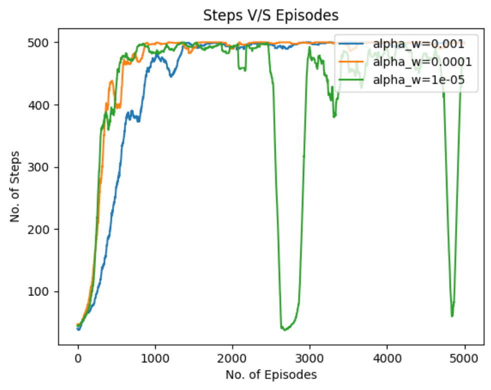

Figure 8: Learning curve on the cart-pole problem with the REINFORCE with baseline algorithm for different values of $\alpha^{\boldsymbol{w}}$

As we can see in Figure 8, for the cart-pole problem, keeping $\alpha^{\boldsymbol{w}} = 0.0001$ keeps the algorithm stable while also learning quickly.

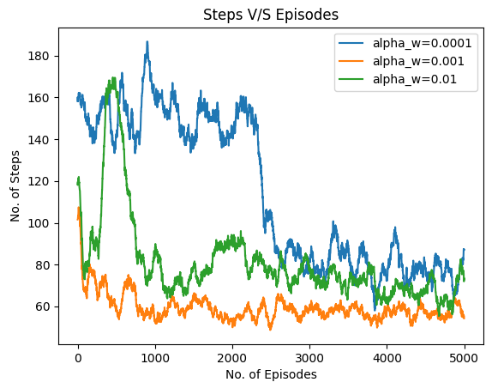

Figure 9: Learning curve on the mountain car problem with the REINFORCE with baseline algorithm for different values of $\alpha^{\boldsymbol{w}}$

As we can see in Figure 9, for the mountain car problem, keeping $\alpha^{\boldsymbol{w}} = 0.001$ allows us to learn the best policy without any large fluctuations in the number of steps required to reach the goal.

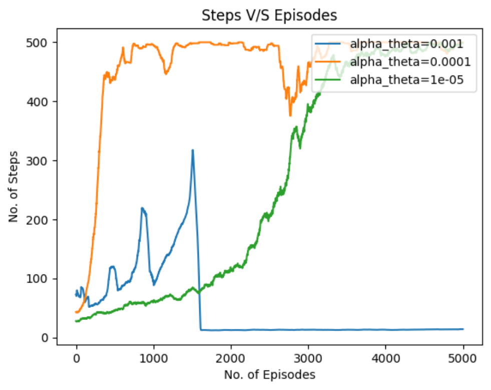

Figure 10: Learning curve on the cart-pole problem with the REINFORCE with baseline algorithm for different values of $\alpha^{\boldsymbol{\theta}}$

As we can see in Figure 10, for the cart-pole problem, keeping $\alpha^{\boldsymbol{\theta}} = 0.0001$ learns a near-optimal policy in the smallest number of episodes. Smaller values are too slow to learn, while the larger value of $\alpha^{\boldsymbol{\theta}} = 0.001$ diverges.

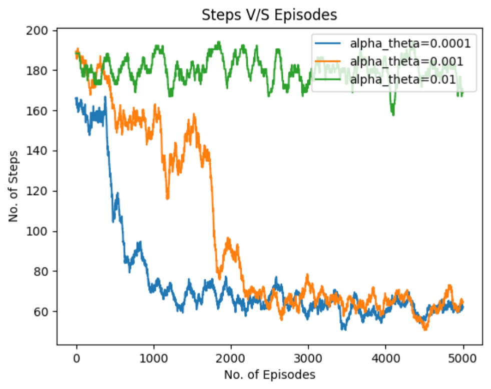

Figure 11: Learning curve on the mountain car problem with the REINFORCE with baseline algorithm for different values of $\alpha^{\boldsymbol{\theta}}$

As we can see in Figure 11, for the mountain car problem, keeping $\alpha^{\boldsymbol{\theta}} = 0.0001$ learns the best policy in the shortest amount of time. Keeping the step size too large with $\alpha^{\boldsymbol{\theta}} = 0.01$ leads to almost no learning at all.

6.3 One-step actor-critic

Similar to REINFORCE, here also we need to set two hyper-parameters: $\alpha^{\boldsymbol{\theta}}$ - the step size for updating $\boldsymbol{\theta}$, and $\alpha^{\boldsymbol{w}}$ - the step size for updating $\boldsymbol{w}$.

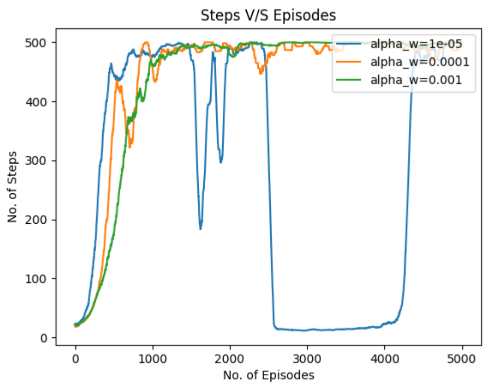

Figure 12: Learning curve on the cart-pole problem with the one-step actor-critic algorithm for different values of $\alpha^{\boldsymbol{w}}$

As we can see in Figure 12, for the cart-pole problem, keeping $\alpha^{\boldsymbol{w}} = 0.0001$ gives the best results with quick and stable convergence.

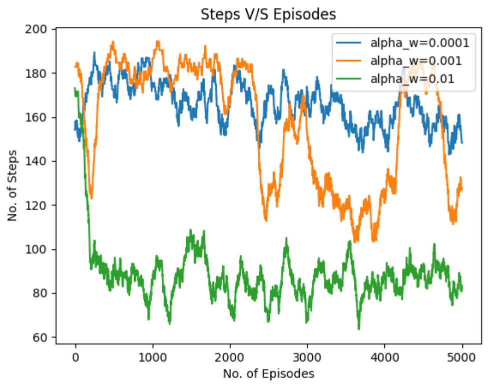

Figure 13: Learning curve on the mountain car problem with the one-step actor-critic algorithm for different values of $\alpha^{\boldsymbol{w}}$

As we can see in Figure 13, for the mountain car problem, keeping $\alpha^{\boldsymbol{w}} = 0.01$ allows us to learn the best policy without any large fluctuations in the number of steps required to reach the goal.

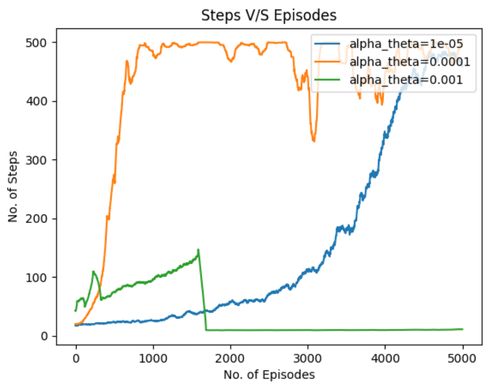

Figure 14: Learning curve on the cart-pole problem with the one-step actor-critic algorithm for different values of $\alpha^{\boldsymbol{\theta}}$

As we can see in Figure 14, for the cart-pole problem, keeping $\alpha^{\boldsymbol{\theta}} = 0.0001$ learns a near-optimal policy in the smallest number of episodes. Smaller values are too slow to learn, while the larger value of $\alpha^{\boldsymbol{\theta}} = 0.001$ diverges.

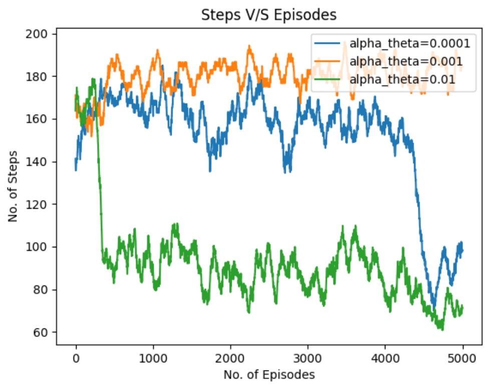

Figure 15: Learning curve on the mountain car problem with the one-step actor-critic algorithm for different values of $\alpha^{\boldsymbol{\theta}}$

As we can see in Figure 15, for the mountain car problem, keeping $\alpha^{\boldsymbol{\theta}} = 0.01$ learns the best policy in the shortest amount of time. Keeping the step size too small results in very slow learning.

6.4 Final values

After running an extensive hyper-parameter search, we arrived for the following values of different hyper-parameters.

1. Episodic semi-gradient n-step SARSA
   - cart-pole: $\alpha = 0.005, n = 1$
   - mountain car: $\alpha = 0.001, n = 2$
2. REINFORCE with baseline
   - cart-pole: $\alpha^{\boldsymbol{w}} = 0.0001, \alpha^{\boldsymbol{\theta}} = 0.0001$
   - mountain car: $\alpha^{\boldsymbol{w}} = 0.001, \alpha^{\boldsymbol{\theta}} = 0.0001$
3. One-step actor-critic
   - cart-pole: $\alpha^{\boldsymbol{w}} = 0.0001, \alpha^{\boldsymbol{\theta}} = 0.0001$
   - mountain car: $\alpha^{\boldsymbol{w}} = 0.01, \alpha^{\boldsymbol{\theta}} = 0.01$

7 Results

After tuning the hyper-parameters, we arrived at the best values for all the hyper-parameters to create the final learning curves. These curves are created by running each experiment 10 times independently with the same parameters, and the mean and standard deviations of the values are shown in the following graphs.

For each algorithm and for each MDP, we have created the graphs for total number of steps taken vs number of episodes completed, and number of steps taken to complete each episode.

For the n-step SARSA algorithm, we used the following schedule to decay the exploration parameter ϵ

$$\epsilon_t = \frac{0.95}{\frac{10t}{T} + 1} + 0.05$$

Where t is the episode number and T is the maximum number of episodes for which the algorithm will run.

7.1 Cart-pole

Since the goal is to maximize the amount of time where the pole is balanced, we want to see the total number of steps vs number of episodes completed graph to have a decreasing slope.

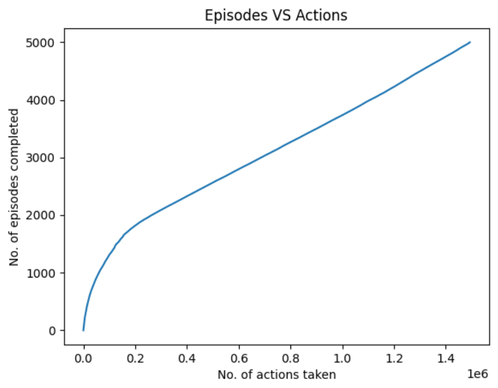

Figure 16: Learning curve on the cart-pole problem with the semi-gradient n-step SARSA algorithm for total number of actions vs number of episodes

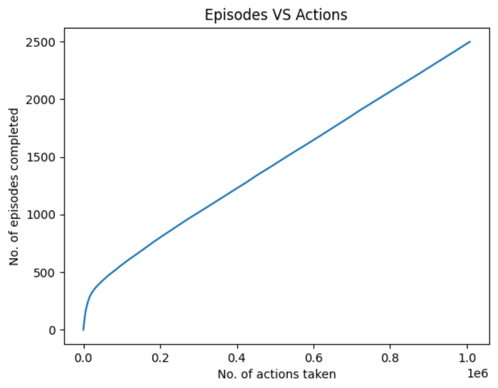

Figure 17: Learning curve on the cart-pole problem with the REINFORCE with baseline algorithm for total number of actions vs number of episodes

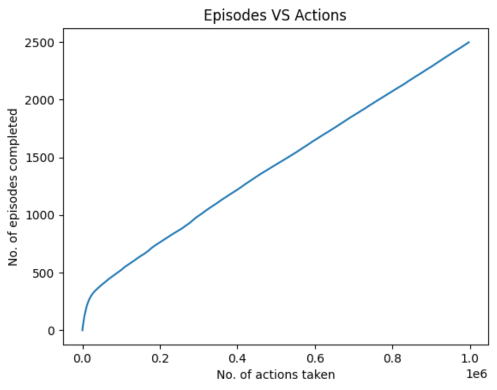

Figure 18: Learning curve on the cart-pole problem with the One-step actor-critic algorithm for total number of actions vs number of episodes

For the number of steps in each episode, we want the number of actions to increase as the agent learns to balance the pole for longer.

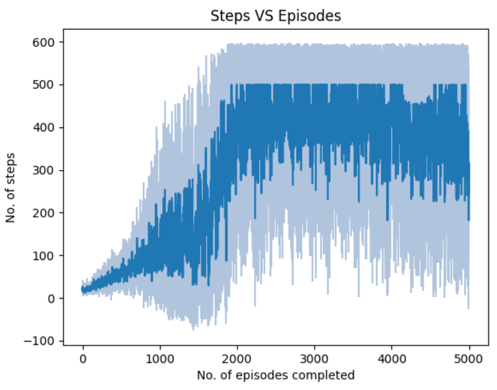

Figure 19: Learning curve on the cart-pole problem with the semi-gradient n-step SARSA algorithm for episode number vs number of steps required

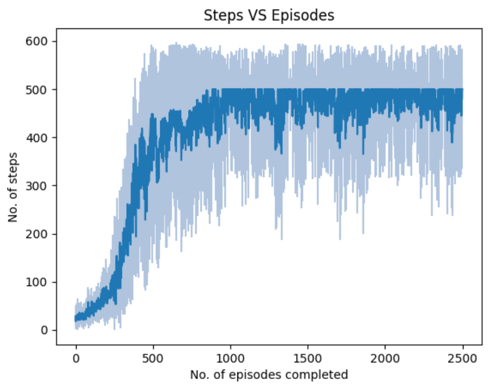

Figure 20: Learning curve on the cart-pole problem with the REINFORCE with baseline algorithm for episode number vs number of steps required

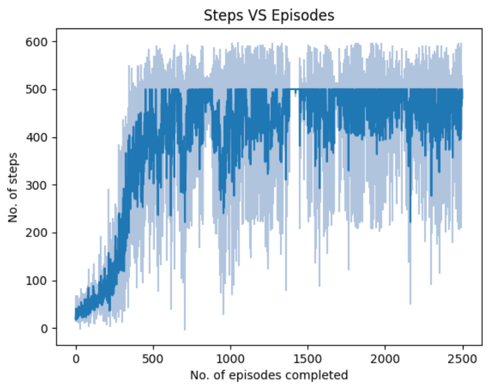

Figure 21: Learning curve on the cart-pole problem with the One-step actor-critic algorithm for episode number vs number of steps required

7.2 Mountain car

Since the goal is to minimize the amount of time required to reach the top, we want to see the total number of steps vs number of episodes completed graph to have an increasing slope.

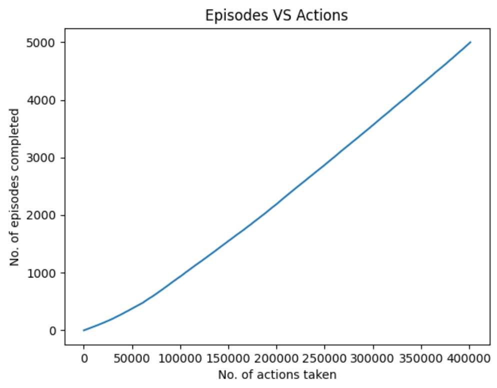

Figure 22: Learning curve on the mountain car problem with the semi-gradient n-step SARSA algorithm for total number of actions vs number of episodes

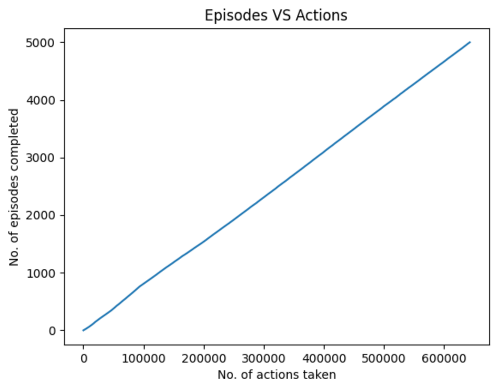

Figure 24: Learning curve on the mountain car problem with the One-step actor-critic algorithm for total number of actions vs number of episodes

For the number of steps in each episode, we want the number of actions to decrease as the agent learns to reach the summit faster.

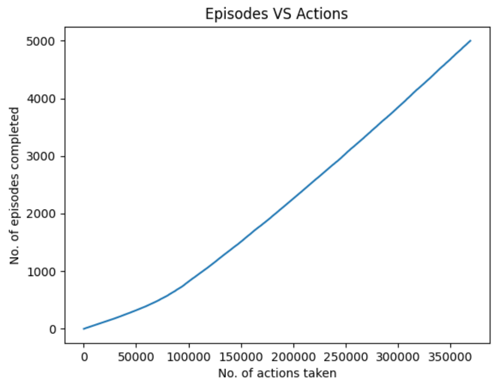

Figure 23: Learning curve on the mountain car problem with the REINFORCE with baseline algorithm for total number of actions vs number of episodes

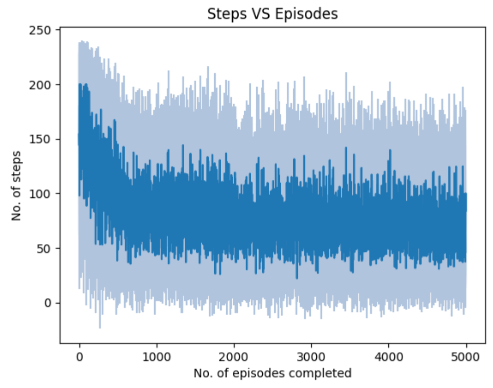

Figure 25: Learning curve on the mountain car problem with the semi-gradient n-step SARSA algorithm for episode number vs number of steps required

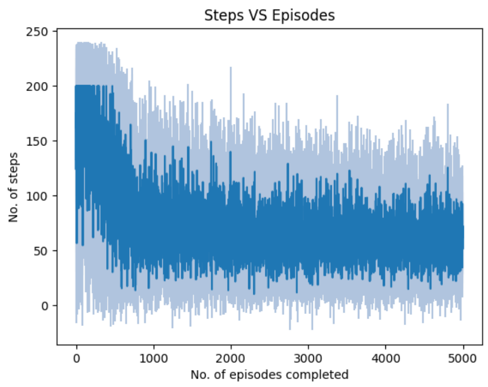

Figure 26: Learning curve on the mountain car problem with the REINFORCE with baseline algorithm for episode number vs number of steps required

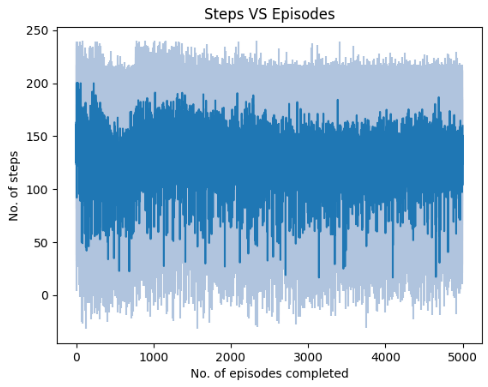

Figure 27: Learning curve on the mountain car problem with the One-step actor-critic algorithm for episode number vs number of steps required

8 Contributions

The first four sections of the report as well as the appendices were written by Kartik. In addition, he implemented the semi-gradient n-step SARSA and REINFORCE with baseline algorithms, and did their write-ups in the report.

Rohan implemented the one-step actor-critic algorithm and did the write-up for the same in the report, along with creating the learning plots and writing the abstract for the report.

Both students contributed equally in tuning the hyper-parameters and all sections of the report went through multiple iterations based on discussions and input by both students.

9 Future Work

For our implementation of all three algorithms for both the MDPs that we have considered, we have used Fourier Basis features of the order 2. As the size of the basis function grows rapidly with the order, we could not conduct experiments with higher order Fourier features due to time and computational constraints. For future work, a higher order for these features could be used to observe a likely increase in performance for the algorithms.

For tuning the hyper-parameters for each of the algorithm, we have individually varied the different hyper-parameters instead of an exhaustive grid-search, which can find better sets of hyper-parameters, but is computationally very expensive to run. For future work, a better and more systematic method of tuning the hyper-parameters, such as using the grid-search, could be followed to find better hyper-parameter values.

In all the experiments we have implemented linear models with Fourier basis functions. This was a design decision that we have collectively agreed upon. While these models work well on simple MDPs such as those studied in this report, a more complex implementation of the model, such as using a neural network, could be used as future work which could give an increase in the performance for the algorithms.

Acknowledgement

We would like to thank Prof. Bruno Castro da Silva for giving us the opportunity to study his course in Reinforcement Learning (COMPSCI 687) and work on this project. It has been an immense learning opportunity for us to study and implement various reinforcement learning algorithms and we will greatly benefit from taking this class in our academic as well as professional endeavors. We are also thankful to the TAs, Aline Weber and Dhawal Gupta who have been readily available for clarifying all our

doubts and helping us solve any problems that we had throughout the course of this semester.

References

Barto, Andrew G. et al. (1983). "Neuronlike adaptive elements that can solve difficult learning control problems." In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5, pp. 834–846. DOI: `10.1109/TSMC.1983.6313077`.

Moore, Andrew (1990). *Efficient Memory-based Learning for Robot Control.* Tech. rep. Pittsburgh, PA: Carnegie Mellon University.

Florian, Razvan V. (2005). "Correct equations for the dynamics of the cart-pole system." In.

Konidaris, George et al. (2011). "Value Function Approximation in Reinforcement Learning Using the Fourier Basis." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 25.1, pp. 380–385. DOI: `10.1609/aaai.v25i1.7903`. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/7903`.

Stulp, Freek and Olivier Sigaud (2012). "Path Integral Policy Improvement with Covariance Matrix Adaptation." In: DOI: `10.48550/ARXIV.1206.4621`. URL: `https://arxiv.org/abs/1206.4621`.

Brockman, Greg et al. (2016). *OpenAI Gym.* eprint: `arXiv:1606.01540`. URL: `https://www.gymlibrary.dev/`.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book. ISBN: 0262039249.

Antonov, Aleksandr (2020). *MountainCar-DQN.* `https://github.com/rndmBot/MountainCar-DQN.git`.

A Fourier Basis Functions

The Fourier basis features are sums of sine and cosine basis functions, but under some assumptions (see Section 9.5.2, RL Book [Sutton and Barto, 2018]), we can only use cosines. For the state $\boldsymbol{s} = (s_1, s_2, \dots, s_m)^\top$, the i^{th} Fourier feature is written as

$$x_i(s) = \cos(\pi s^\top c^{(i)})$$

Where

$$c^{(i)} = (c_1^{(i)}, \dots, c_m^{(i)})^\top$$

To speed up computation, these values were cached and the inner products were vectorized.

B Cartpole Dynamics

The equations given by Barto et al. (1983) have two mistakes - one with using the term $\mu_c \text{sgn}(\dot{x}_t)$ instead of $\mu_c N_c \text{sgn}(N_c \dot{x}_t)$, and the other being taking the value of g (acceleration due to gravity) as negative instead of positive.

The track in the problem we are dealing with in this report is frictionless so which simplifies the problem and the final differential equations describing the dynamics are given by

$$\ddot{\theta} = \frac{g \sin\theta + \cos\theta\left(\frac{-F - m_p l \dot{\theta}^2 \sin\theta}{m_c + m_p}\right)}{l\left(\frac{4}{3} - \frac{m_p \cos^2\theta}{m_c + m_p}\right)}$$

$$\ddot{x} = \frac{F + m_p l(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta)}{m_c + m_p}$$

Where,

$$\begin{aligned} g &= 9.8 m/s^2, \text{ acceleration due to gravity} \\ m_c &= 1.0 kg, \text{ mass of the cart} \\ m_p &= 0.1 kg, \text{ mass of the pole} \\ l &= 0.5 m, \text{ half-pole length} \\ F_t &= \pm 10.0 N, \text{ force applied to the cart's} \\ &\quad \text{center of mass at time } t \end{aligned}$$

See Florian (2005) for more details.