COMPSCI 687: Reinforcement Learning Fall 2023

Final Project Report

Group: Tejas Sivan, Saluru Durga Sandeep

1 Introduction

This report briefly summarizes how we applied **REINFORCE with Baseline** and **Semi-gradient n-step SARSA** on two established domains, **Cartpole** and **Acrobot**.

2 REINFORCE with Baseline

The REINFORCE with Baseline algorithm is an extension of the basic REINFORCE algorithm, which belongs to the family of Policy Gradient methods. The improvement this algorithm holds over standard REINFORCE is the use of a *baseline* to mitigate variance in gradient estimation, thereby enhancing learning stability. In the following section, we will further explore the algorithm by fine-tuning various parameters that impact the learning curves.

2.1 Algorithm Overview

- **REINFORCE:** The REINFORCE algorithm is a policy-gradient-based RL algorithm that is Monte-Carlo in nature (episodic); it uses the complete returns from time t. This family of RL algorithms learns a parameterized policy that picks actions without necessarily using any value function. Parameterized policies are often simpler to learn than parameterizing action-values, and action probabilities change smoothly, guaranteeing stronger convergence. In this algorithm, the parameterized policy is adjusted based on the policy gradient weighted against the return.

$$\theta \leftarrow \theta + \alpha_\theta \cdot G_t \cdot \gamma^t \cdot \nabla_\theta \ln(\pi(A_t|S_t, \theta)) \tag{1}$$

- **Baseline:** This algorithm utilizes a *baseline*, that can reduce the variance of the gradient estimates (a problem that commonly plagues regular REINFORCE). This baseline can practically be any function that does not vary with the action; for this project, we have taken the state-value estimate.

- **Weight Update Equation:** This equation computes the gradient of the logarithm of the policy with respect to the parameters and scales it by the advantage, which is the difference between the total obtained return (cumulative discounted return) and baseline. This scaled gradient is then used to update the policy parameters in the direction that maximizes the expected return, favoring actions that are expected to yield higher returns compared to the baseline.

$$w \leftarrow w + \alpha_w \cdot \delta \cdot \nabla_w \hat{v}(S_t, w)$$
$$\theta \leftarrow \theta + \alpha_\theta \cdot \delta \cdot \gamma^t \cdot \nabla_\theta \ln(\pi(A_t|S_t, \theta))$$

where $\delta = G_t - \hat{v}(S_t, w)$

2.2 Pseudocode

Algorithm 1: `runEpisode` to generate an episode for a given policy.
Input:
1) Game environment - Acrobot, CartPole
2) Policy Network - PNet
3) Value Network - VNet

$S \leftarrow S_0 \sim d_0$; `// Sample Initial State - In gym, env.reset()`
$Returns \leftarrow 0$
$Trajectory \leftarrow []$; `// Empty List`

for $t = 0, 1, 2, \ldots$ **do**
  $probabs \leftarrow PNet(S)$; `// Compute Action Probabilities`
  $baseline \leftarrow VNet(S)$; `// Compute state-value`
  $A \sim probabs$; `// sample action based on probabilities`
  Take action A and observe next state S', reward R
  $S \leftarrow S'$
  $Returns \leftarrow Returns + R$
  $Trajectory.add((log(probabs[A]), R, baseline))$; `// Store the tuple`
  if S' *is terminal* **then**
    End the loop
Return Trajectory;

Algorithm 2: `REINFORCE with Baseline` for estimating optimal policy.
Input:
1) Game environment - Acrobot, CartPole
2) A differentiable policy parameterization - Eg, Policy Network using Deep Neural Network (DNN) - PNet
3) A differentiable state-value parameterization - Eg, Value Network using DNN - VNet
4) Learning rates - $\alpha_\theta > 0$ and $\alpha_w > 0$
5) Initialize PNet - θ and VNet - w parameters - Eg, Random, Kaiming He
6) If using DNN, Choose an Optimizer for PNet and VNet - Eg, Adam with α_θ and α_w
7) Learning Rate Decay - SchedulerLR
8) Number of Episodes - numEpisodes

for $episode = 1, 2, \ldots numEpisodes$ **do**
  $Trajectory \leftarrow runEpisode(PNet(\theta), VNet(w))$; `// Generate an epsiode`
  $T = len(Trajectory)$
  for $t = 0, 1, 2, \ldots T-1$ **do**
    $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} \cdot R_k$; `//` $R_k \leftarrow Trajectory[k][1]$
    $baseline \leftarrow Trajectory[t][2]$
    $\delta_t = G_t - baseline$
    $w \leftarrow w + \alpha_w \cdot \delta_t \cdot \nabla_w \hat{v}(S_t, w)$; `//` $\hat{v}(S_t, w)$ `= baseline - Backprop of VNet`
    $\theta \leftarrow \theta + \alpha_\theta \cdot \delta_t \cdot \gamma^t \cdot \nabla_\theta \ln(\pi(A_t|S_t, \theta))$; `//` $\ln(\pi(A_t|S_t, \theta))$ `= Trajectory[t][0] - Backprop`
    SchedulerLR.step()
Return PNet ; `// Optimal Policy`

2.3 Advantages:

- This algorithm is well-suited to deal with continuous state values without requiring discretization
- Parameterized policies are often simpler to learn than parameterizing action values, and with policy gradient methods, in general, action probabilities change smoothly
- Including a baseline helps reduce the variance in gradient estimation significantly. This reduction stabilizes learning, leading to more reliable and consistent updates to the policy

3 Semi-gradient n-step SARSA

Now, we will discuss about semi-gradient n-step SARSA. We will understand the terms *semi-gradient*, *n-step*, and SARSA by the end of this section.

3.1 Algorithm Overview

- **SARSA:** SARSA (State-Action-Reward-State-Action) is an on-policy temporal difference learning algorithm, i.e., the agent updates its own policy by interacting with the environment according to its current policy. Note that SARSA does involve policy evaluation (through the Q-values it learns), but its primary objective is to learn the optimal policy by estimating action values and taking actions based on these value estimates (maximum of Q-value or epsilon greedy during the update). Q-values can be approximated using a function, for instance, deep neural networks.

- **Semi-gradient:** Semi-gradient descent refers to using the gradients of the Q-values during back-propagation (in neural networks) to update the function. We call it semi-gradient because it's not actually the true gradient of the loss function. Referring to the equation 2, even though we did not consider the full gradient. this update will still work. It's easy to compute gradients for linear functions. Complex function approximations like neural networks will be challenging to write from scratch. We will use torch autograd module to calculate the gradient, which will be used automatically during backpropagation.

$$w_{t+1} = w_t + \alpha \cdot \Big[R_{t+1} + \gamma \cdot \hat{Q}(S_{t+1}, A_{t+1}, w_t) - \hat{Q}(S_t, A_t, w_t)\Big] \cdot \nabla \hat{Q}(S_t, A_t, w_t) \tag{2}$$

- **n-step:** When we update Q-values by considering the current action, the next state, the reward, and the next action. This is also known as one-step SARSA. n-step SARSA extends one-step SARSA by considering multiple steps into the future (n steps) to update Q-values. Simple intuition is that when we look at the n-steps, we can better estimate the Q-value when compared to only one step. We will discuss in later sections how the value of n larger than one resulted in a better learning curve.

$$\begin{aligned} G_{t:t+n} &= R_{t+1} + \gamma \cdot R_{t+2} + ... + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot \hat{Q}(S_{t+n}, A_{t+n}, w_{t+n-1}) && \text{if t+n} < \text{T} \\ &= G_t && \text{if t+n} \geq \text{T} \end{aligned}$$

Weight update equation

$$w_{t+n} = w_{t+n-1} + \alpha \cdot \Big[G_{t:t+n} - \hat{Q}(S_t, A_t, w_{t+n-1})\Big] \cdot \nabla \hat{Q}(S_t, A_t, w_{t+n-1})$$

The above equations and notations are referred from **"Reinforcement Learning: An Introduction" by Andrew Barto and Richard S. Sutton**

3.2 Pseudocode

Algorithm 3: `Semi-Gradient n-step SARSA` for estimating q_*.
Input:
1) Game environment - Acrobot, CartPole
2) A differentiable action-value function parameterization - QNet using DNN
3) Learning rate $\alpha > 0$ and exploration $\epsilon > 0$
4) Initialize QNet weights w - Eg. Random, Kaiming He
5) If using DNN, Choose an Optimizer for QNet Eg, Adam with α
6) Learning Rate Decay - SchedulerLR
7) Number of Episodes - numEpisodes

$states \leftarrow EmptyList$
$actions \leftarrow EmptyList$
$rewards \leftarrow [0]$
for $episode = 1, 2, \ldots numEpisodes$ **do**
  Initialize $S_0 \neq Terminal$; `// env.reset() in gym API`
  states.add(S_0) ; `// Store` S_0
  $A_0 \leftarrow$ Select action based on ϵ-greedy using QNet(S_0, w)
  actions.add(A_0) ; `// Store` A_0
  $T \leftarrow \infty$
  for $t = 0, 1, 2, \ldots$ **do**
    if $t < T$ **then**
      Take action A_t;
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1};
      states.add(S_{t+1});
      rewards.add(R_{t+1});
      if S_{t+1} *is terminal* **then**
        $T = t + 1$;
      else
        $A_{t+1} \leftarrow$ Select action based on ϵ-greedy using QNet(S_{t+1}, w);
        actions.add(A_{t+1});
    $\tau \leftarrow t - n + 1$;
    if $\tau \geq 0$ **then**
      $G = \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} \cdot$ rewards[i];
      if $\tau + n < T$ **then**
        $G = G + \gamma^n \cdot$ QNet(states[$\tau + n$], w)[actions[$\tau + n$]];
      $w \leftarrow w + \alpha \cdot [G - \hat{q}(S_\tau, A_\tau, w)] \cdot \nabla \hat{q}(S_\tau, A_\tau, w)$; `//` $\hat{q}(S_\tau, A_\tau, w) =$ `QNet(states[`τ`], w)[actions[`τ`]]`
  $\epsilon \leftarrow \epsilon * 0.99$; `// Exploration decay`
  if $\tau = T - 1$ **then**
    Terminate Loop; Reached the end of the episode
Return Trajectory;

4 Cart-Pole

4.1 Domain Description

A pole is attached to a cart; we aim to balance this pole by applying force to move the cart left or right.

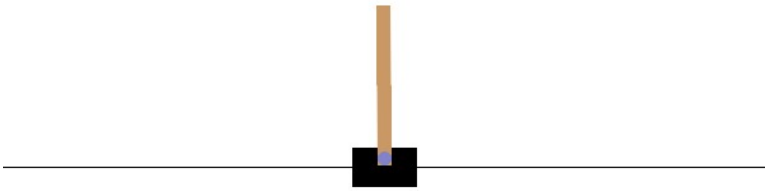

Figure 1: Cart-Pole

- **State representation:** [x, v, w, w_v] where x is the cart position, v is the cart velocity, w is the pole angle, w_v is the pole angular velocity
- **Actions:** 0 - pushing the cart to the left, 1 - pushing the cart to the right
- **Rewards** +1 for every time step and a maximum reward of 500 (the episode will be terminated at t=500)
- **Initial State Distribution:** Initial values in the state, i.e., x, v, w, w_v, are assigned a uniformly random value in (-0.05, 0.05)
- **Terminal State:** Episode will be terminated if one of the following conditions is satisfied: 1. Pole Angle is greater than $\pm 12°$, 2. center of the cart reaches the edge of the display, i.e., $\pm$ 2.4, 3. Episode length is greater than 500

For our project, we have utilized OpenAI's Gym interface to model the Cartpole-v1 environment.

4.2 REINFORCE with Baseline Hyper-parameter Tuning

This section delves into how we have improved REINFORCE with Baseline step-by-step. Through extensive tuning, we've refined our grasp of the algorithm. Here, we'll showcase multiple learning curves, each influenced by altering a hyper-parameter, accompanied by our observations on these changes. Each hyper-parameter setting ran for five iterations to plot the learning curves. We have reached the best model through step-by-step changes to the model, which will be discussed in the following sub-sections

4.2.1 Initial Parameters of the Baseline Model

Firstly, we have started with the following set of model parameters.

- Initialization - Random
- Policy Learning Rate - 0.001
- Value Learning Rate - 0.001
- Number of Hidden Layers - 1
- Hidden Layer size - 16
- Learning Rate Decay - No
- Episodes - 500 (during tuning, we have run 500 episodes for faster fine-tuning)

Following is the learning curve, which is the average reward across five iterations at each episode

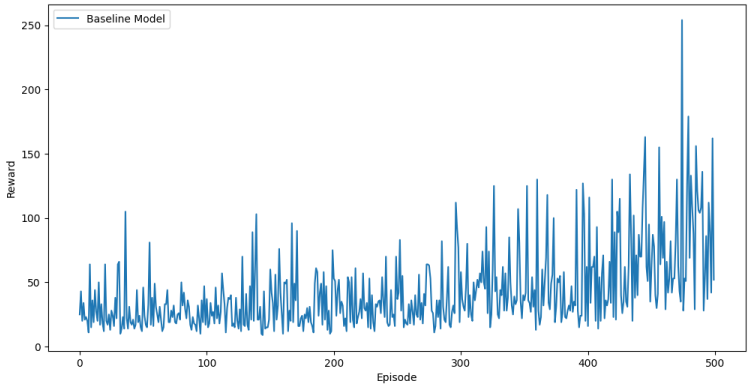

Figure 2: Baseline Model Performance - REINFORCE with Baseline

The above figure 2 shows that the baseline model is not reaching optimal returns, i.e., 500. There could be various reasons for this behavior, such as insufficient model complexity, the learning rate being too low, initialization, and so on. We have chosen to increase the model complexity by increasing the hidden layer size from 16 to 32 and 64

4.2.2 Hidden Layer Size

After increasing the hidden layer size from 16 to 32 and 64, it can be clearly seen (refer to figure 3) that the model has improved significantly. Instead of increasing the hidden layer size, we have increased the number of hidden layers for additional model complexity.

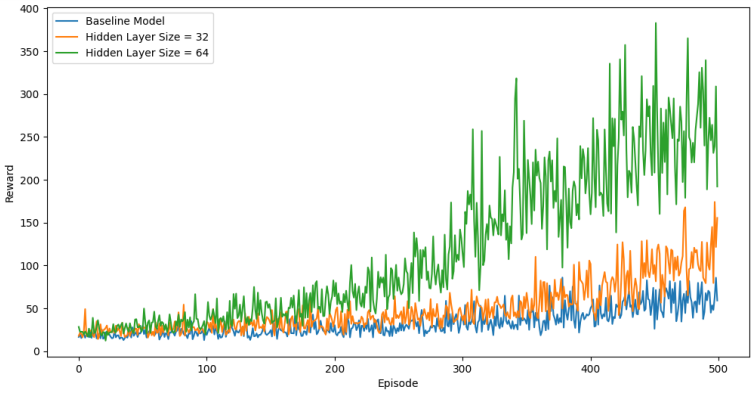

Figure 3: Increasing Hidden Layer size - REINFORCE with Baseline

4.2.3 Increasing Hidden Layers

It can be seen that increasing hidden layers significantly improved performance. Refer figure 4.

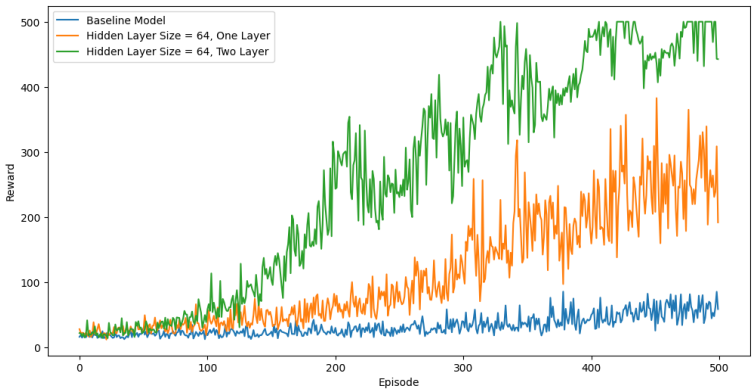

Figure 4: Increasing Hidden Layers - REINFORCE with Baseline

Now, there are other ways to improve the learning curve, such as faster convergence to optimal solution stability of the curve (i.e., smoother curve). We have decided to tune the learning rates of Policy Network and Value Network (generally, value network has a higher learning rate than policy network)

4.2.4 Learning Rate Tuning

After tuning different learning rates for policy and value networks, we found that policy lr = 0.003 and value lr = 0.01 have a better learning curve than other learning rate combinations. The following figure 5 shows how this helps us in faster convergence.

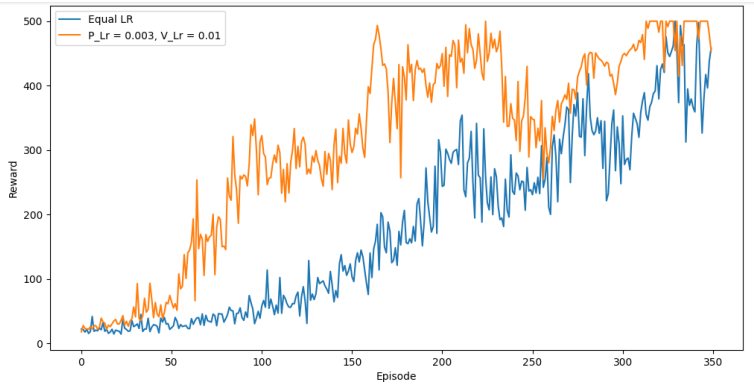

Figure 5: Learning Rate Tuning - REINFORCE with Baseline

Generally, Kaiming He initialization helps in consistent learning curves across different iterations. We have experimented using Kaiming He and Random initialization.

4.2.5 Initialization weights

Surprisingly, the learning curve of He initialization is similar to that of random initialization. Refer to figure 6.

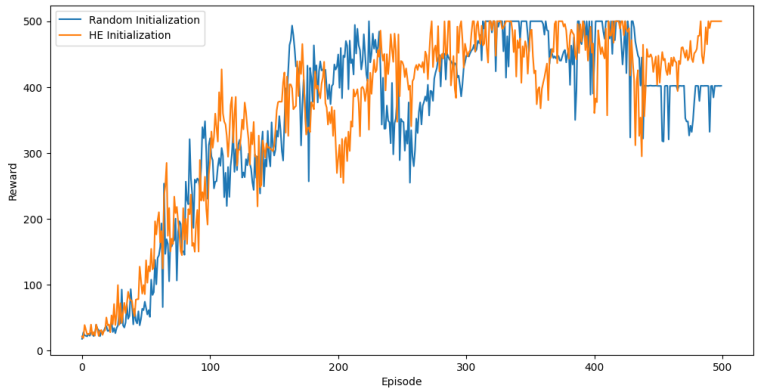

Figure 6: He vs Random Initialization - REINFORCE with Baseline

4.2.6 Learning Rate Decay

Generally, a higher learning rate at the beginning of the training process helps the algorithm converge faster by taking larger steps toward the optimum. However, as training progresses, reducing the learning rate can help the model converge accurately, preventing overshooting and oscillations around the optimal. This can also be observed in the cartpole domain using REINFORCE with baseline. Refer 7. When we did not decay the learning rate, the model could not converge at the optimum solution.

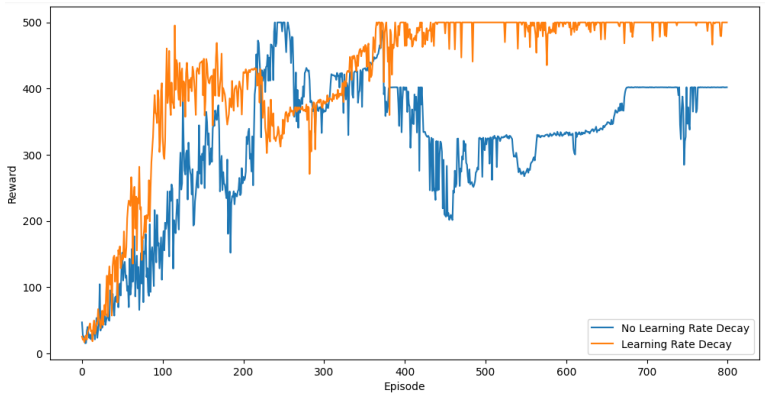

Figure 7: Learning Rate Decay - REINFORCE with Baseline

4.3 Final Model and Its results

After extensive fine-tuning, we found the best hyper-parameters such that my model can reach the optimum faster and more consistently across different iterations, i.e., stability is higher.

Following are the final parameters

- Initialization - He
- Policy Learning Rate - 0.003
- Value Learning Rate - 0.01
- Number of Hidden Layers - 2
- Hidden Layer size - 64
- Learning Rate Decay - Yes, Scheduler with stepsize 25
- Episodes - 1000

Following is the final learning curve for cart-pole using REINFORCE with Baseline (average across ten iterations)

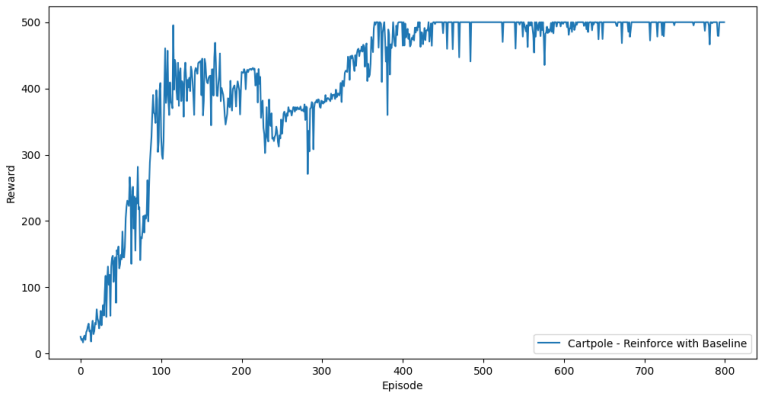

Figure 8: Cartpole Best Model - REINFORCE with Baseline

4.4 Semi-gradient n-step SARSA Hyper-parameter tuning

This section delves into hyper-parameter tuning, a crucial aspect of our exploration. Through extensive tuning, we've refined our grasp of the algorithm. Here, we'll showcase multiple learning curves, each influenced by altering a hyper-parameter, accompanied by our observations on these changes. Each hyper-parameter setting ran for five iterations to plot the learning curves.

4.4.1 Learning Rate

It is very important to get the right learning rate for our deep learning model. If the learning rate is too high, then the training curve will be noisy and have a high variance at the optimal value (it will not stabilize even after we run for many episodes). In the second case, when the learning rate is too low, the training curve reaches the optimal value after many episodes (very slow convergence rate). The following figure 9 illustrates this.

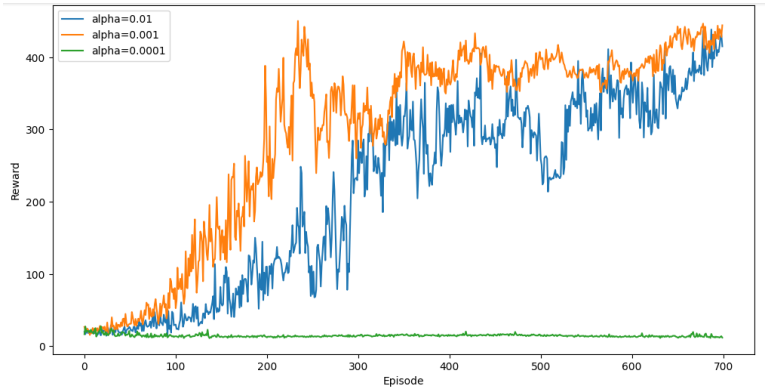

Figure 9: Learning Rate - SARSA n-step

The following figure 10 shows how setting very low alpha results in no learning in limited episodes. Keeping the other parameters the same and changing the alpha value to 0.001 will result in reaching the optimal solution in much fewer iterations. (Orange curve in figure 9)

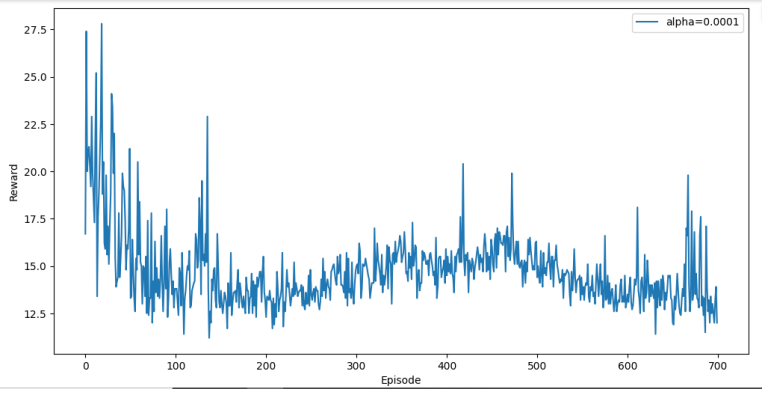

Figure 10: Learning Rate = 0.0001- SARSA n-step

As shown in figure 9, When the learning curve rate is high (alpha = 0.01 - blue curve), the learning curve is noisy compared to alpha = 0.001. In our best model, we have selected alpha between 0.0001 and 0.001.

4.4.2 Random vs He Initialization

In our MDP, initialization played a crucial role. We have compared Kaiming He initialization vs Random initialization. Unlike random initialization methods that choose weights randomly, He initialization specifically scales the initial weights based on the number of input units to a layer. This is particularly effective because it adapts to the network's architecture, promoting better convergence than purely random initialization methods. As you can see from figure 11, Kaiming He has a better learning curve on average and reaches reward 500 consistently, unlike random initialization. In random initialization, only some iterations are converging at optimal by the end of 700 episodes. Other iterations are not able to reach the optimal. There is a lot of variance across iterations in the case of random initialization. There is less variance across iterations in the case of Kaiming He initialization.

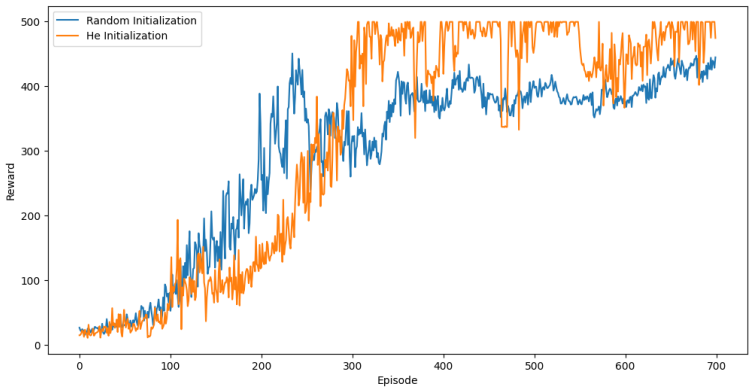

Figure 11: He initialization - n-step

4.4.3 Number of Hidden Layers

Number of Hidden layers - decides the overall complexity of the model. More layers imply a complex model. It's very important to choose the model complexity appropriately. If our model complexity is low (fewer hidden layers), then the model will not be able to learn much. If our model complexity is high (more hidden layers), it might take too long to run one iteration. We want a balance between reaching an optimal solution and a reduction in run-time to reach that optimal solution. This can be clearly seen from the figure 12 as you can see, between two and three hidden layers, not much difference in their learning curves. Keeping run-time as an important factor, we will stick to 2 hidden layers in our final model.

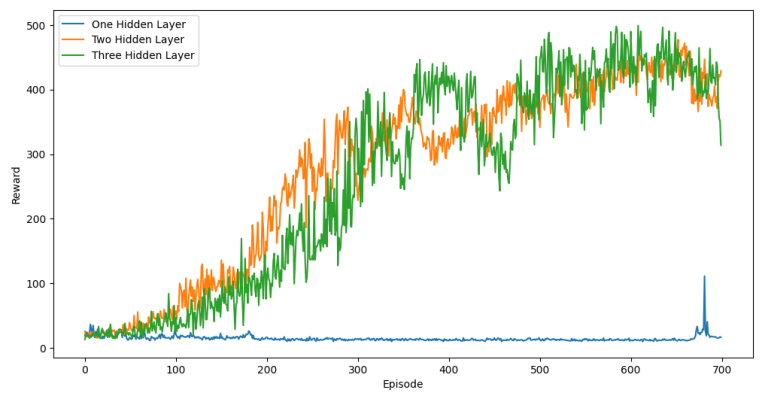

Figure 12: Number of Hidden Layers - SARSA

In the case of one hidden layer, the model may not be complex enough to capture the complex relationships between states, actions, and rewards. figure 13. We might have to run for more episodes to reach the optimal solution (implies more run-time)

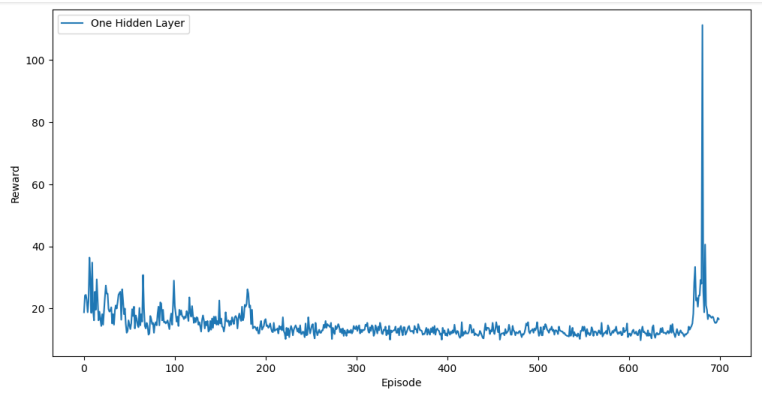

Figure 13: Number of Hidden Layers = 1 - SARSA

4.4.4 Hidden Layer size

Like the number of hidden layers, hidden layer size decides the overall complexity of the model. A higher number of nodes in each hidden layer implies a complex model. It's very important to choose the model complexity appropriately. If our model complexity is low (fewer nodes at each hidden layer), then the model will not be able to learn much (orange curve in 14). If our model complexity is high (a higher number of nodes at each hidden layer), running one iteration might take too long. From figure 14, as you can see, having nodes around 128 gives a much better learning curve compared to 64 and 32. We have experimented with 256 nodes at each hidden layer, and the run-time increased, but the performance is very similar to 128 nodes. Therefore, going ahead with 128 nodes in the hidden layers.

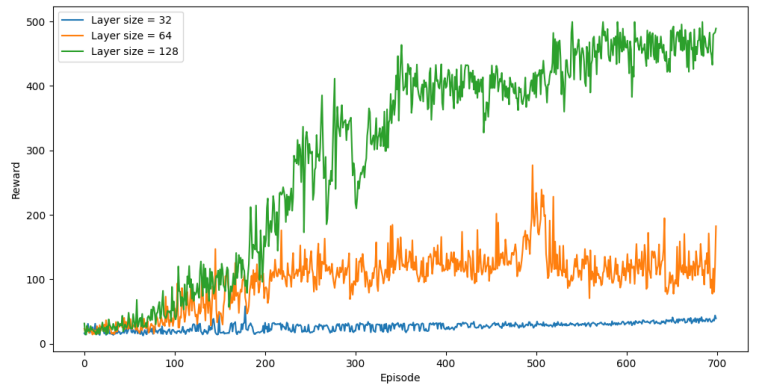

Figure 14: Hidden Layer Size - SARSA

4.4.5 Impact of n

Generally, increasing n results in a better learning curve (smoother curve). The reason is that larger n allows more future rewards to be considered in the update. This often results in more reliable estimates of the Q-values by considering a longer sequence of actions and rewards. Very high n is also not ideal because it can lead to an increase in memory requirements to keep track of longer sequences of experiences, which can increase computational costs. In the following figure 15, we can see that having n = 10 has a better learning curve than n = 1 and n = 5

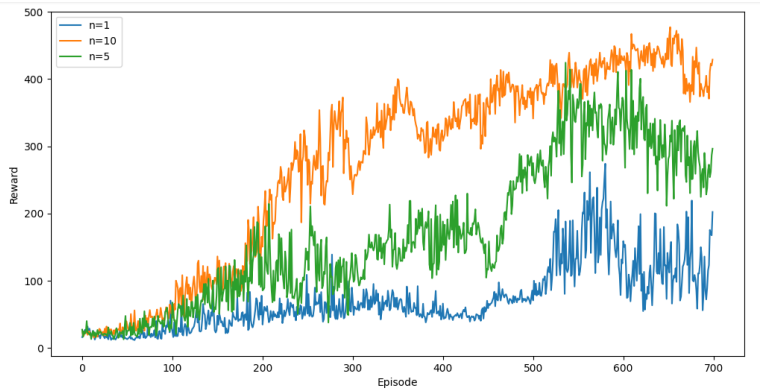

Figure 15: Values of n - SARSA

4.4.6 Learning rate decay

It is evident from the figure 16, that we need to decay our learning rate as training progresses. The reason could be that a higher learning rate initially helps the algorithm converge faster by taking larger steps toward the optimum returns. However, as training progresses, reducing the learning rate can help the model converge accurately, preventing overshooting and oscillations around the optimal solution. It is also observed how not decaying has higher oscillations even after reaching optimum results. For more stability, we used learning rate decay using scheduler LR in PyTorch

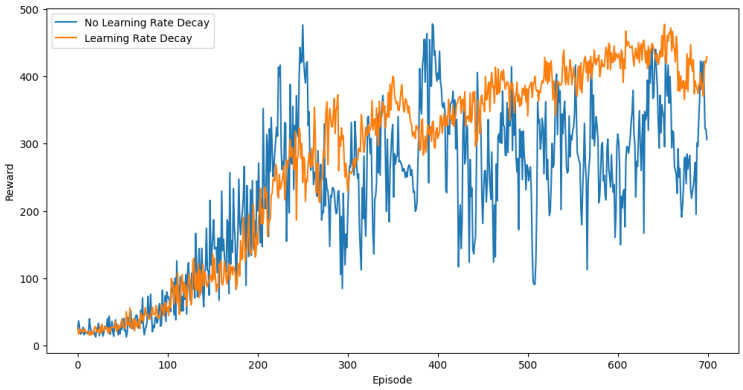

Figure 16: Learning Rate Decay - SARSA

4.4.7 Exploration Decay - Linear vs Exponential

It is also important how we are exploring the domain as the training progresses. We generally would like the agent to explore more in the starting phase of training. As training progresses, we want to decrease exploration and exploit the information collected till now. This is a classical exploration and exploitation trade-off. In this cart-pole domain, exponential decay resulted in faster convergence in optimal solution and was more stable than linear exploration decay. Refer to figure 17

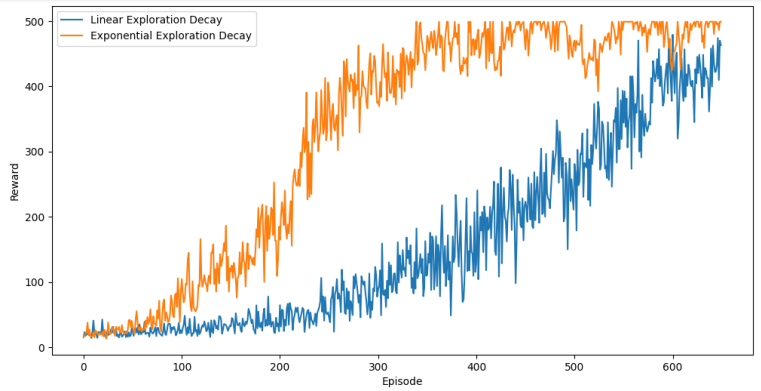

Figure 17: Exploration Decay - SARSA

Please note that tuning all these parameters gave us an idea of which is the ideal range to find the potential hyper-parameter combinations. The final model is based on further tuning of a few parameters in the model. For example, manually decaying LR at particular episodes instead of using scheduler step size.

4.5 Semi-gradient n-step SARSA Final results

After extensive fine-tuning, we found the best hyper-parameters such that my model can reach the optimum faster and more consistently across different iterations, i.e., stability is higher.

Following are the final parameters

- Initialization - He
- n - 10
- Policy Learning Rate - 0.0005
- Number of Hidden Layers - 2
- Hidden Layer size - 128
- Learning Rate Decay - Yes, Scheduler with stepsize 100
- Episodes - 1000
- Exploration Decay - Exponential

Following is the final learning curve for cart-pole using semi-gradient n-step SARSA (average across ten iterations)

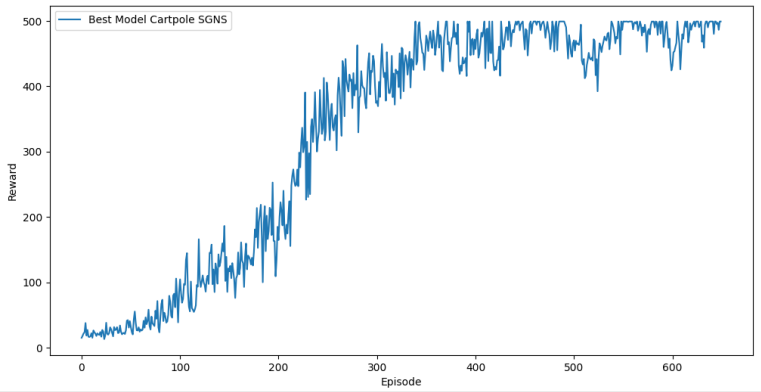

Figure 18: Cartpole Best Model - Semi-gradient n-step SARSA

5 Acrobot - v1

5.1 Domain Description

The Acrobot problem is a classic Reinforcement Learning problem, designed to test algorithms for learning complex motor control tasks. It is represented as a two-link pendulum system connected by a joint, where the aim is to swing the system of links until the bottom end reaches a certain height. A pictorial representation can be found in fig 19

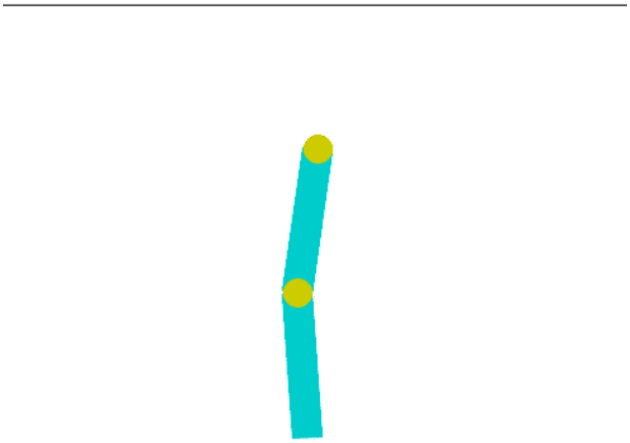

Figure 19: Acrobot

The system comprises two links joined linearly by a joint that can be moved. The top link is attached to a fixed point, while the bottom link has a free end. The objective of this problem is to apply torque on the controlled joint to swing the free end to a certain specified height (represented by a black horizontal line in fig 19).

- **State representation:** The state of the system is represented by 6 variables:

No.	Variable	Min Value	Max Value
1	Cosine of θ_1	-1	1
2	Sine of θ_1	-1	1
3	Cosine of θ_2	-1	1
4	Sine of θ_2	-1	1
5	Angular velocity of θ_1	-4π	4π
6	Angular velocity of θ_2	-9π	9π

Table 1: State Representation of Acrobot

  where θ_1 is the angle of the first joint (0 degrees represents the first link pointing straight down) and θ_2 is relative to the angle of the first link (0 degrees means both angles are the same).

- **Actions:** There are three discrete actions at play here: **0** - apply (-1) torque to the joint, **1** - apply (0) torque to the joint, **2** - apply (1) torque to the joint
- **Rewards** The aim is for the free end to reach the target height as quickly as possible, so a constant negative reward of -1 is applied at each step. Reaching the goal state provides a reward of 0 (the episode will be terminated at t=500).
- **Initial State Distribution:** θ_1, θ_2, and their associated angular velocities are assigned a uniformly random value in the range [-0.1, 0.1]
- **Terminal State:** The episode terminates if the goal state $cos(\theta_1) - cos(\theta_2 + \theta_1) > 1.0$ is reached; else it terminates after an episode length of 500.

For our project, we have utilized OpenAI's Gym interface to model the Acrobot environment.

5.2 REINFORCE with Baseline Hyper-parameter Tuning

In this section, we explore the different hyperparameters we tuned to find an optimal set of hyperparameters. Each of the runs of these hyper-parameter configurations had five iterations to observe results and compare.

5.2.1 Learning Rate

For all learning algorithms, the learning rate is a crucial hyperparameter that determines the quality of the convergence. This is particularly pertinent in REINFORCE with Baseline as we have **two** learning rates to set here - the Policy Learning Rate (PLR) that affects the learning rate for the parameterized policy, and the Value Learning Rate (VLR) that affects the baseline. From our experiments, we could see that when PLR is larger than VLR, we do not get convergence - this is possibly due to the policy updates dominating the learning process, or due to the large policy updates overshooting. We obtain the best result when our VLR is larger than our PLR, as shown in fig 20

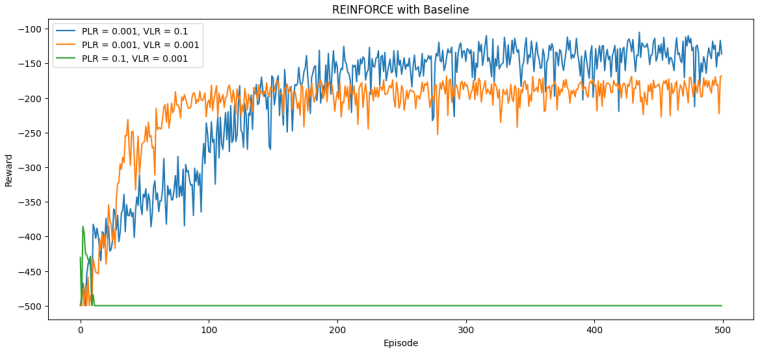

Figure 20: Learning Rate - comparison

5.2.2 Hidden Layer Size

The dimension of the hidden layers in a model indicates the number of neurons contained within each hidden layer. Opting for a larger hidden layer size proves advantageous for tackling more intricate problems, albeit at the cost of increased computational demands; whereas for simpler problems, simpler models converge faster and are less prone to overfit. We found that a hidden layer size of 64 failed to converge to a strong reward, whereas larger hidden layer sizes (128 and 256) did. Out of the two, we have chosen 256 as it displayed less variance and is therefore a better fit.

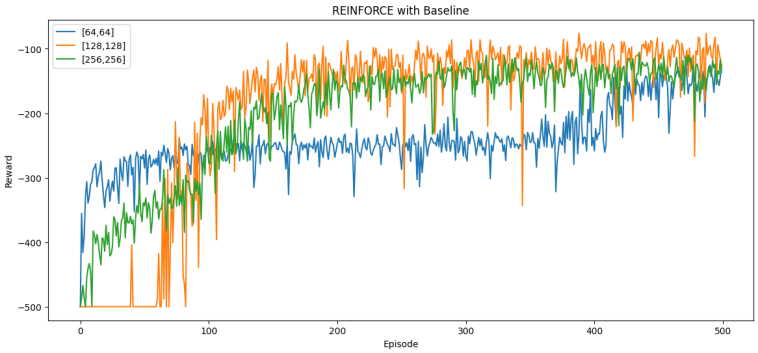

Figure 21: Layer Size - comparison

5.2.3 Number of Hidden Layers

The complexity of a model is influenced by the quantity of hidden layers it incorporates. In addressing intricate problems, there is a preference for more complex models with a higher number of hidden layers. However, the trade-off involves the computational expense associated with large, complex models, coupled with their susceptibility to overfitting. Through observation, we noted that both single-layer and double-layer networks attained convergence to the optimal value.

However, the single-layer network demonstrated a faster convergence rate, showcasing the advantage of having a simpler model.

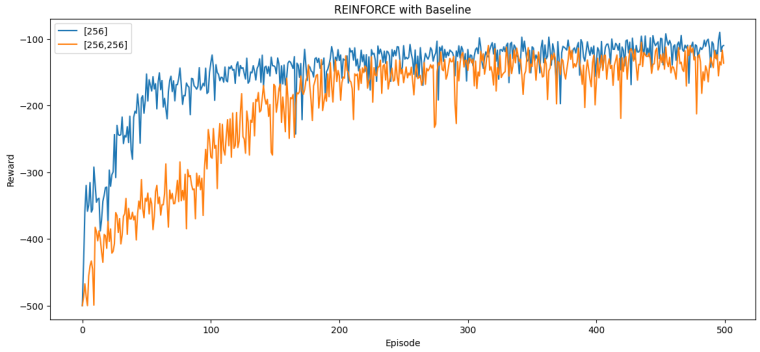

Figure 22: Number of Hidden Layers - comparison

5.2.4 Activation Function

Activation functions are used in neural networks to achieve non-linearity in the output of a neuron, thereby enabling the network to model more complicated relationships. The Rectified Linear Unit (ReLU) family of activation functions, which replace all negative values in the input with zero, is a commonly used activation function for neural networks. For our experiments, we have compared regular ReLU and Leaky ReLU (that allows a miniscule portion of the negative input through). From Fig. 23, we can see that we obtained much better convergence from Leaky ReLU - this is likely because it can handle the "Dying ReLU" problem (where neurons can become inactive and the gradient flowing through it stays zero forever).

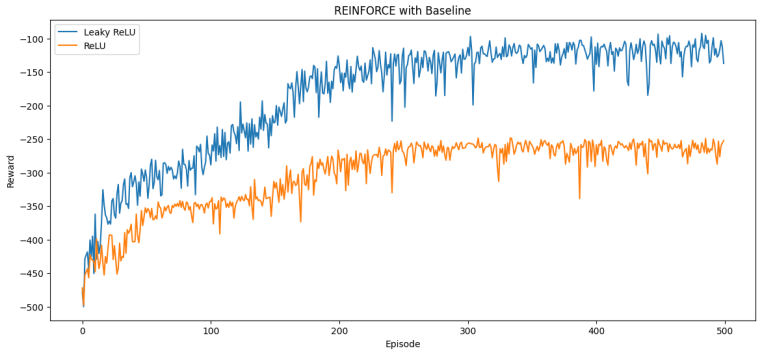

Figure 23: ReLU vs. Leaky ReLU - comparison

5.3 REINFORCE with Baseline Final Model and Its results

After extensive fine-tuning, we have isolated the best hyper-parameters, which are detailed below:

- Policy Learning Rate - 0.001
- Value Learning Rate - 0.1
- Number of Hidden Layers - 1
- Hidden Layer size - 256
- Episodes - 1000
- Activation Function - Leaky ReLU (negative slope - 0.1)

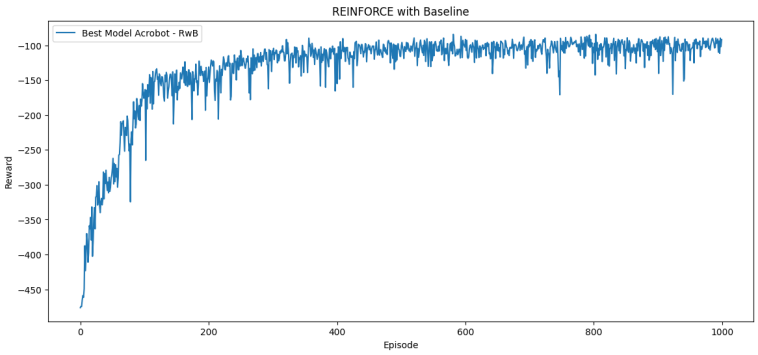

Figure 24: Acrobot Best Model - REINFORCE with Baseline

5.4 Semi-gradient n-step SARSA Hyper-parameter tuning

In this segment, we delve into the variety of hyperparameters that underwent tuning in our quest for an optimal set. Each set of hyperparameter configurations was executed through five iterations, allowing us to observe results and conduct comparisons.

5.4.1 Impact of n

The value of n affects how many steps in the future we consider while updating our Q-values. We observed that in general, increasing the value of n resulted in better convergence until a certain point. This follows logically, as the larger the value of n, the larger the sequence of experience we have to draw from. From our empirical tests, we found a value of $n = 5$ has a sufficiently good curve, while not being as computationally expensive as $n = 10$. A few comparisons are made in Fig. 25

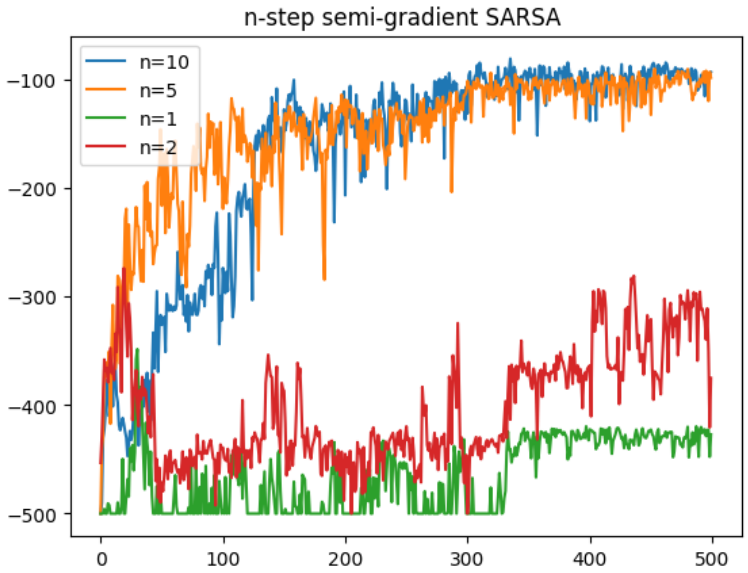

Figure 25: Values of n - comparison

5.4.2 Hidden Layer Size

The size of the hidden layers in a model refers to the number of neurons within each hidden layer. These are responsible for capturing and learning complex patterns and representations in the input data, and so the size of the individual hidden layers also affects the complexity of the model. A high hidden layer size is more beneficial for more complex problems, at the expense of being more computationally expensive. We found our optimal value at Layer Size = 32. A model with hidden layers of size 16 never quite reached convergence at the desired level; meanwhile, a model with hidden layers of size 64 did reach convergence at the desired reward level, however it was observed to be noisier, most likely due to the more complex model overfitting. So we have chosen 32 as our desired layer size.

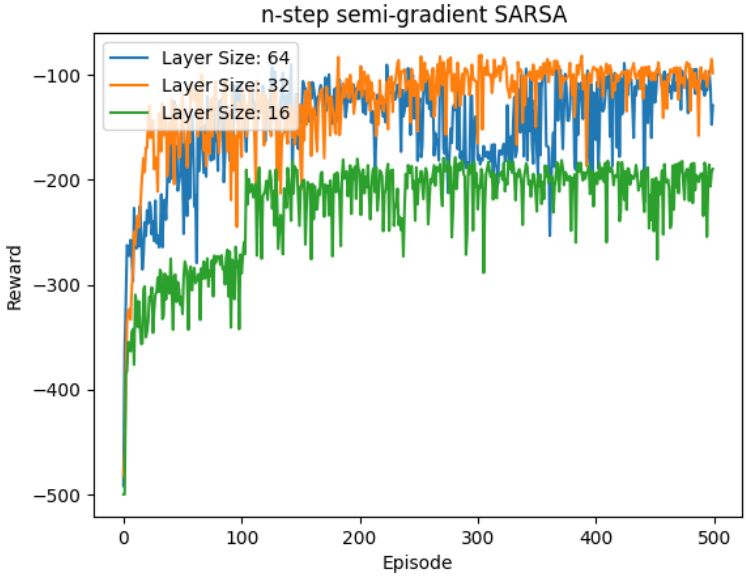

Figure 26: Layer Size - comparison

5.4.3 Number of Hidden Layers

The number of hidden layers determines the complexity of the model - more complex models featuring a large number of hidden layers are preferred, the more complicated a problem is. The drawback to large, complex models is that they are computationally expensive, and models that are too complicated are also prone to overfitting. We observed that while both single-layer and double-layer networks converged to the optimal value, the single-layer network converged faster and has the advantage of being a simpler model.

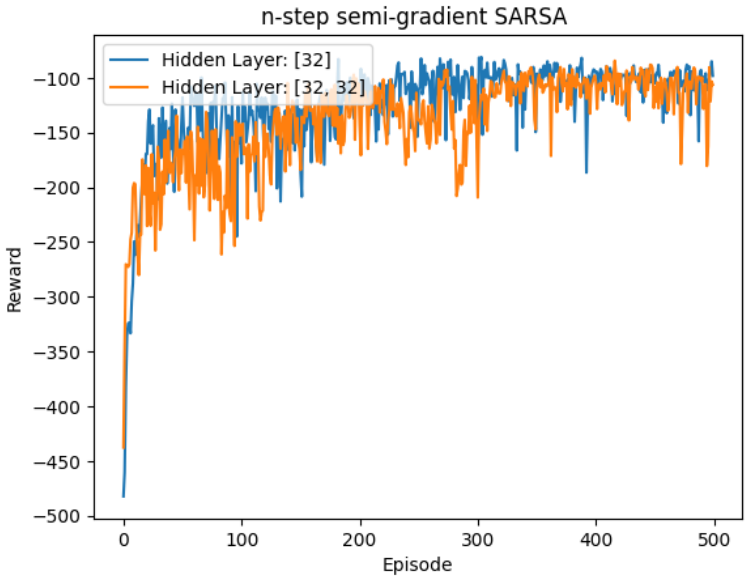

Figure 27: Number of Hidden Layers - comparison

5.4.4 Learning Rate

The Learning Rate is an important parameter to be tuned, as it affects the size of the updates the model makes. When the learning rate is set too high, the training curve tends to be erratic with high variance around the optimal values. Conversely, in scenarios where it is too low, the training curve takes an extensive amount of time to reach optimal values, resulting in a notably slow convergence rate. Thus, picking an appropriate learning rate is necessary. From our observations, we have seen that a learning rate of $\alpha = 0.1$ produces a sub-optimal convergence, likely due to the updates

by the model not being fine-grained. $\alpha = 0.01$ and $\alpha = 0.001$ achieve similar convergence, but we pick $\alpha = 0.001$ as it displays lower variance.

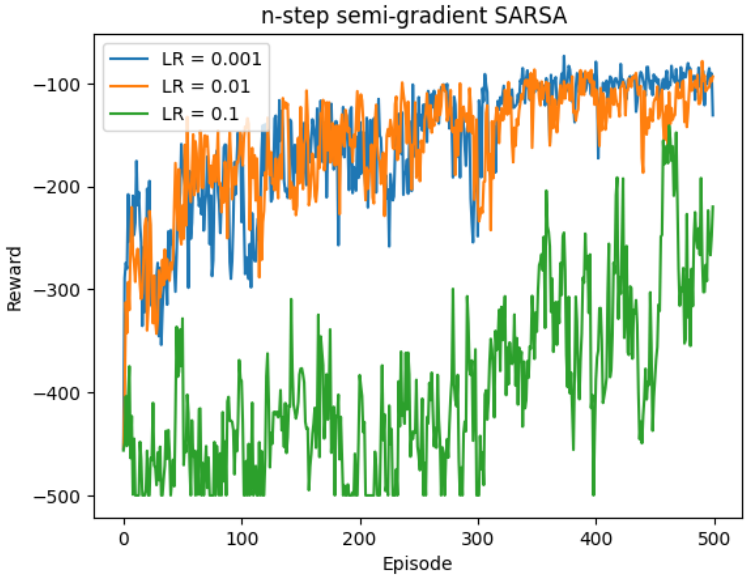

Figure 28: Learning Rate - comparison

5.4.5 Exploration Rate Decay

How we choose to decay ϵ, the Exploration Rate, is crucial: at the beginning, we wish for the agent to explore a lot, and as the training proceeds, the agent should exploit more than it explores. In our experiments, we observed that exponential decay led to a stronger and faster convergence over linear decay, so we have chosen to use exponential exploration decay.

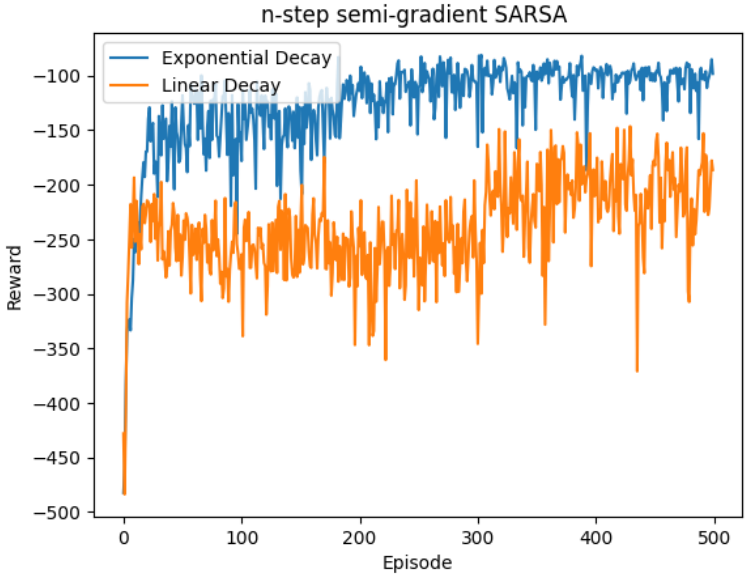

Figure 29: Exponential vs. Linear Exploration Decay - comparison

5.5 Semi-gradient n-step SARSA Final results

After extensive fine-tuning, we found the best hyper-parameters such that the model can reach the optimum faster and more consistently across different iterations, i.e., stability is higher.

Following are the final parameters

- Initialization - He

- n - 5
- Learning Rate - 0.001
- Number of Hidden Layers - 1
- Hidden Layer size - 32
- Learning Rate Decay - Yes, Manually reducing after 300 episodes and 500 episodes
- Episodes - 1000
- Exploration Decay - Exponential

Fig. 30 is the final learning curve for Acrobot using semi-gradient n-step SARSA (averaged across ten iterations)

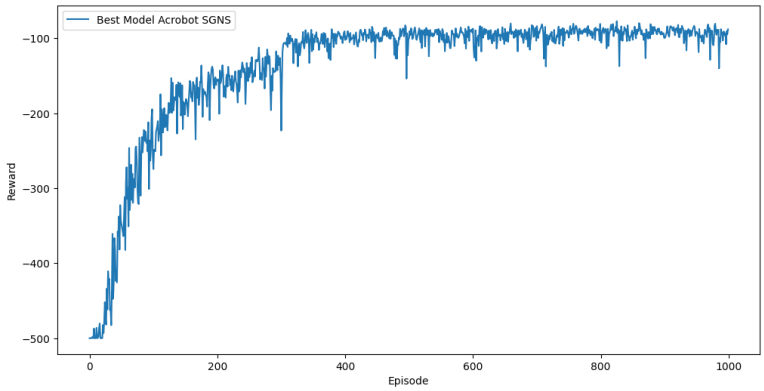

Figure 30: Acrobot Best Model - Semi-gradient n-step SARSA

6 Conclusion

- This project gave us an opportunity to look at some of the advanced algorithms in Reinforcement Learning. While implementing these algorithms, we had to make many design decisions, such as the size of the deep learning network, activation functions, and learning rates.
- Through extensive fine-tuning, we were able to find the hyper-parameters that helped the agent reach the optimal solution faster and with stable convergence. All these challenges helped us understand the algorithms in and out. We were able to reach optimal solutions for both Acrobot and Cart-pole using REINFORCE with baseline and Semi-gradient n-step SARSA
- Final Model GIFs are included in the zip files
- References: "Reinforcement Learning: An Introduction" by Andrew Barto and Richard S. Sutton
- Contributions: Tejas Sivan - REINFORCE with Baseline, Saluru Durga Sandeep - semi-gradient n-step SARSA (Implemented on both the MDPs)