Reinforcement Learning Project

Anshumaan Chauhan
achauhan@umass.edu

Eric Anderson
edanderson@umass.edu

1. Algorithm Description

In most of the Reinforcement Learning (RL) problems, the environment is non-stationary, that means as we are converging towards the optimal policy, the true state and action values are changing simultaneously. In these cases it makes more sense to have high weights on the recent reward, rather than on the current estimates.

$$v_{new} = v_{old} + \alpha * [Target - v_{old}] \tag{1}$$

This can be achieved by making the value of α constant in the Moving Average Update Equation 1.

General Policy Iteration (GPI) is an alternating sequence of executing the Policy Evaluation and Policy Iteration algorithm in order to converge to optimal policy. There are different ways of implementing Policy Evaluation and Improvement steps - by making slight variations in each component, which leads to a large set of RL algorithms. In general, the task of learning an optimal policy is known as Control Problem.

Policy Iteration algorithm is guaranteed to converge if we visit each state (or state-action) infinitely often. However, when we execute a step in Policy Improvement, we usually get a greedy policy with respect to the action value (q). This makes it impossible to visit the non-greedy actions. Exploring Starts - starts an episode with random state-action pairs, is not used with the real-world problems as it introduces unlikely assumptions.

On-Policy and Off-Policy methods are two methods that guarantee to take all actions.

1. **On-Policy**: In these methods, we select an action in an ϵ-greedy fashion, that is, with probability (1-ϵ) we choose the best possible action, and with probability ($\frac{\epsilon}{|A|}$) we choose random action from the action space $|A|$.
2. **Off-Policy**: Utilizes two policies, one is used to sample experiences (known as *behavior policy*) and one is updated in a greedy fashion w.r.t the current value function (known as *target policy*). Importance Sampling is the algorithm used to learn the optimal target policy using the samples generated by the behavior policy.

We can divide the RL methods into two parts on the basis of MDP formulation. If the states and actions in the MDP are discrete then we use Tabular methods, else we use Approximate Solution Methods.

Tabular methods are mostly used by the model-free methods (algorithms that do not require the knowledge of transition dynamics and the reward function) - such as Monte Carlo Methods (MC) and Temporal Difference Learning (TD). We can say that TD comprises of the entire spectrum of methods, where on one end lies MC methods (updates after generating a complete episode) and on the other end is TD(0), also known as One Step TD (updates the value function after each step of the episode).

When the number of states and actions are infinite, it is impossible to use Tabular methods. A more practical implementation includes parameterizing the value and action functions (most likely using Neural Networks) - these are called Approximate Solution Methods. Here the neural networks are function of states - that is given a state the model predicts the action or state function of it. The weights (parameters) w of the model are usually updated using the standard gradient descent update rule 2.

$$w_{t+1} = w_t - \frac{1}{2} * \alpha * \nabla[U_t - \hat{v}(S_t, w_t)]^2 \tag{2}$$

In most of the cases, the target value U_t has part of it using the Neural Network, so it is actually $U_t(w)$. If we ignore the gradient update coming from U_t, then those set of algorithms are known as Semi-Gradient algorithms. Convergence is still guaranteed in some cases, for example when a linear value function approximator is used.

1.1. Episodic Semi Gradient n-step SARSA

Episodic Semi Gradient SARSA is a linear value function approximation methods, and one of their drawbacks is that they cannot take into account the interactions between the features, so overcome this issue usually a new state feature representation is used which includes these inter-dependencies. There are several techniques to generate these featurizations such as follows:

- Polynomials
- Radial Basis Function
- Coarse Coding

- Fourier Basis
- Tile Coding

In our project, we have used Tile Coding as a featurization technique for the Episodic Semi Gradient SARSA method. More information about the transformation is present in Section 2

1.1.1 Episodic Semi Gradient n-step SARSA

As discussed earlier, semi-gradient methods converge under certain cases only, one of which is when a linear value function approximator is used. Linear Value Function Approximation is just one single matrix multiplication of the weight matrix with the state representation.

$$\hat{v}(s, w) = w^T x(s) = \sum_{i}^{d} w_i x_i(s)$$

where x(s) is the new state representation consisting of the inter-dependencies between the different features of the state s. The gradient of this function with respect to the model parameters w, is simply the state representation x(s). The update equation for Episodic Semi Gradient n-step SARSA is as follows:

$$w_{t+1} = w_t + \alpha * [U_t - \hat{v}(S_t, w_t)] * x(S_t)$$

SARSA is a TD-Learning based Policy Evaluation method, which uses bootstrapping (estimates of following states for improving estimate of current state) - saves time, as we do not have to wait for an entire episode to end before making an update unlike Monte Carlo methods. SARSA is guaranteed to converge surely if:

- Finite $\mathcal{S}$ and $\mathcal{A}$, bounded rewards and $\gamma \in [0, 1)$ - All the MDPs (Mountain Car, Acrobot and Cart Pole) are having finite state space due to the Tile Coding state featurization.
- Step size α (or Learning Rate) decay appropriately - We are decreasing the value of learning rate by a factor of 0.95 after each epoch.
- All state-action pairs are visited infinitely often - We used ϵ-greedy policy to ensure that each state-action is visited oftenly.
- The policy must converge in limit to a greedy policy. Value of epsilon is decayed exponentially (eventually in limit becomes close to 0), so that SARSA converges to optimal action value in limit.

Algorithm 3 presents the pseudocode for the Episodic Semi Gradient n-step SARSA.

1.2. REINFORCE with Baseline

As discussed earlier, Neural Networks can be utilized as the parameterized model for Policy and Value/Action functions. A Policy Network is a network that parameterizes the mapping between states and action probabilities (acts as policy). The output of the model can be for a discrete set of actions - a probability value for each action, or a continuous spectrum - predicts the mean and variance of the action distribution (and later action is sampled from this distribution).

We can refactor any Reinforcement Learning problem as a Supervised Learning problem if we know beforehand which action is best for any given state. However, in real world applications it is impossible to know best actions for each given state. Another way is to use the rewards we get to judge how good the action was, and when scaled with the log probabilities of the action, we get our objective function. We usually want to maximize this product value, however because we are using it as a loss , therefore we want to minimize the negative product. This is how vanilla Policy Gradients (a.k.a REINFORCE) algorithm works - does not require value/action functions for the updating the policy.

One variation of Policy Gradient algorithm is Monte Carlo Policy Gradients which is episodic in nature, and uses the returns from the episode in order to estimate the gradients of network. Although they are more stable and guarantee convergence, they are slow in nature as they have to wait for an episode to finish before making any update. A faster version is inspired from TD approaches, which makes updates after each step in the episode.

REINFORCE with Baseline is a Monte Carlo approach based Policy Gradient method, which uses two networks - Policy Network and Critic Network. Critic network is a network used to predict the value function of a state and acts as a baseline. We can safely assume baseline is a value subtracted from the estimated return to reduce down the high variance observed in vanilla Policy Gradient.

The flow of this algorithm is as follows:

1. Agent observes the state it currently is in, and uses the Policy Network to get the action probabilities, which are used to sample an action.
2. Agent executes that action and observes reward and next state
3. Steps 1 and 2 are executed till we reach the terminal state. Actions, States and Rewards at each step are stored.
4. For each time step, Critic Method is utilized to get the value functions of the current state. We get the error by calculating the difference between estimated return (G) and value function of the current state.
5. TD Error along with log probability of the chosen action are used to update the models (Equations are presented in the Pseudocode).

Being from the family of Monte Carlo Methods it shares the drawback of relatively high variance and slow convergence. However it is more stable in comparison

to the vanilla Policy Gradient method which uses the incremental returns directly to update the model.

Algorithm 2 presents the pseudocode for the REINFORCE with Baseline.

1.3. Actor Critic Algorithm

Actor Critic can be thought of as the Temporal Difference variation of the REINFORCE with Baseline method we discussed earlier (with some additional modifications). It uses 2 networks known as Actor and Critic which are used to parameterize the Policy and the Value Function respectively. The goal of the Actor is to learn a policy that will maximize the expected returns and of the Critic is to correctly learn the close estimates of the true value function.

In this algorithm Temporal Difference Error is used to train the models. Critic predicts the value functions, so it use the TD Error in terms of Mean Squared Error Loss (MSE Loss) as its optimizing function. Whereas Actor uses the negative weighted TD Error, where the weights are log probability of the action taken, as the Loss function.

Some of the limitations of this method are High Variance and Slow Convergence. Although the variance is relatively less as compared to REINFORCE with Baseline, but is high in comparison to the model-based methods. Due to the model-free nature of this algorithm, it suffers from slow convergence, as it relies on the sampling for update. Lastly a common drawback of all the Neural Network based strategies is that they are highly sensitive to the hyperparameters.

Algorithm 1 presents the pseudocode for the Actor Critic

2. Feature Transformation

In the book there are many possible options available for the state featurization (named earlier), however we chose Tile Coding because it has proven to work well as showcased in Section 10.1.

2.1. Tile Coding

We have implemented Tile Coding from scratch which is used to discretize a continuous state environment. It uses a combination of a Hash Table and a Tiling functionality for this task.

Basically Hash Table uses a dictionary which has (Tiling number, State, Action) as the key and a number (count) as the corresponding value. We used Python's in-built hash function to get a new value (within the range of length of dictionary) in case the length of dictionary exceeds the number of tiles.

In the Tiling functionality we first quantize the continuous state input, and then we check which tile within each tiling is encapsulating this quantized state using the Hash Table. A vector of length T is then returned where each discrete value is within the range of number of tiles in each tilings.

Number of tiles for each MDP was decided using $t = T^{D+A}$ formula, where T is the number of tilings, D is the number of dimensions in the state space, and A is the total number of allowed actions.

3. Formulation of Markov Decision Processes (MDPs)

We have used gym library for implementing different MDPs. For simplicity we have used three out of the five available Classic Control environments - Acrobot, Cart Pole and Mountain Car. Classic Control environments have stochastic initial state (starting point) and are considered easier as compared to others when solved using a policy. In this section we will briefly go over the dynamics and state representation of the MDPs.

3.1. Cart Pole

Cart Pole is a problem where a pole is attached to a cart (this joint is not allowed to move), and we have to apply force (compulsory) to push the cart (on a friction-less surface) left or right at each time step, while preventing the pole from falling.

State of the environment is composed of 4 elements: Cart Position x, Cart Velocity v, Pole Angle ω and Pole Velocity $\dot{\omega}$ (Angular Velocity). State Space $\mathcal{S}$ (also known as Observation Space) has a range for each of the features present in the state representation. Cart Position, Cart Velocity, Pole Angle, and Pole Velocity are confined between $[-4.8, 4.8], [-\infty, \infty], [-24°, 24°], [-\infty, \infty]$ respectively.

Action Space $\mathcal{A}$ is composed of only 2 actions $\in$ [Push Cart Left, Push Cart Right].

Reward Function $\mathcal{R}$ gives us reward of +1 for each step taken (only when the pole has not fallen).

Let's explain the transition dynamics of the environment, for each action taken a force of magnitude 10 is applied on the cart. If the action was left, a force of -10 is applied, else a fore of 10 is applied. The transition equations used for each step taken is as follows:

$$x_{t+1} = x_t + \tau * v_t \tag{3}$$
$$v_{t+1} = v_t + \tau * d \tag{4}$$
$$\omega_{t+1} = \omega_t + \tau * \dot{\omega}_t \tag{5}$$
$$\dot{\omega}_{t+1} = \dot{\omega}_t + \tau * c \tag{6}$$

where the constants and intermediate values are as follows:

$$g = 9.8 \quad \text{(gravity)}$$

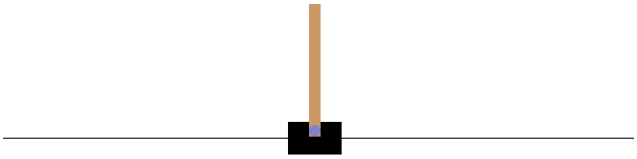

Figure 1. Cart Pole Rendered MDP

$$m_c = 1.0 \quad \text{(cart's mass)}$$
$$m_p = 0.1 \quad \text{(pole's mass)}$$
$$m_t = m_c + m_p \quad \text{(total mass)}$$
$$l = 0.5 \quad \text{(pole's length)}$$
$$\tau = 0.02 \quad \text{(agent executes an action every 0.02 seconds)}$$
$$b = \frac{F + m_p * l * \dot{\omega}_t^2 * \sin \omega_t}{m_t}$$
$$c = \frac{g * \sin \omega_t - \cos \omega_t * b}{l * (\frac{4}{3} - \frac{m_p * \cos {\omega_t}^2}{m_t})}$$
$$d = b - \frac{m_p * l * c * \cos \omega_t}{m_t}$$

Termination conditions: If ω<-12° or ω>12°, then the state is terminal because the pole fell. If the cart position is x <-2.4 or x >2.4, the state is also terminal since the center of the cart reached the edge of the allowed range for the cart position. Episode also terminates if the agent has exceeded more than 500 steps (Reward Threshold is 500).

3.2. Acrobot

Acrobot is an environment where we have a system of 2 joints as represented in Figure 2. The first joint is fixed, whereas the second one is to free. Our aim is to start from some initial state and apply torque on this lower chain in a way that it reaches the minimum required height.

State of the environment is composed of 6 elements: Cosine of theta1, Sine of theta1, Cosine of theta2, Sine of theta2, Angular velocity of theta1 and Angular velocity of theta2. theta1 is the angle of the first joint, whereas theta2 is relative to the angle of first link. State Space $\mathcal{S}$ (also known as Observation Space) has a range for each of the features present in the state representation. The starting state parameters of the environment are uniformly sampled from the range [-0.1,0.1]. The ranges for the cosine and sine of theta1 and theta2 are [-1,1], whereas the range for angular velocity of theta1 and theta2 are $[-4\pi, 4\pi]$ and $[-9\pi, 9\pi]$ respectively.

Figure 2. Acrobot Rendered MDP

Action Space $\mathcal{A}$ is composed of only 3 discrete actions $\in$ [Apply Torque of -1, Apply Torque of 0, Apply Torque of +1].

Reward Function $\mathcal{R}$ gives us reward of -1 for each step taken. So the goal is to reach the minimum height required in as few as possible steps. When we reach the termination state then we get a reward of 0.

The next state is calculated using a transition function that is modeled by a ODE Solver using 4-th order Runge-Kutta methods.

Termination conditions: If the free end reached the minimum desire height - this is measured $-\cos(theta1) - \cos(theta2 + theta1) > 1.0$. The Episode is also terminated if number of executed steps are more than 500. (Reward Threshold is -100).

3.3. Mountain Car

Mountain Car is a deterministic MDP where a car is place in between a valley generated using a sine curve as shown in Figure 3. The car is allowed to move in either direction, and has a goal of reaching the terminal state in as few as possible steps similar to Acrobot.

State of the environment is composed of 2 elements: Position (x) and Velocity (v) of the car. State Space $\mathcal{S}$ (also known as Observation Space) has a range for each of the features present in the state representation. The starting state parameter - position of the car is uniformly sampled

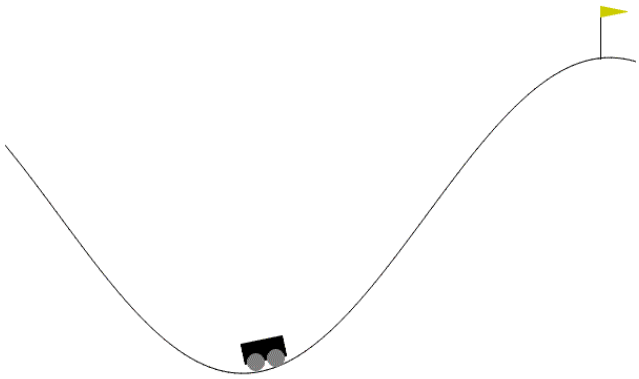

Figure 3. Mountain Car Rendered MDP

from the range [-0.6,0.4], whereas the value of velocity is always 0. The allowed ranges for both position and velocity of the car is $[-\infty, \infty]$.

Action Space $\mathcal{A}$ is composed of only 3 discrete actions $\in$ [Accelerate Left, Do not accelerate, Accelerate Right].

Reward Function $\mathcal{R}$ gives us reward of -1 for each step taken. So the goal is to reach the flag position in as few as possible steps. When we reach the termination state then we get a reward of 0.

The environment transition dynamics are given using the following equations:

$$v_{t+1} = v_t + (\text{action} - 1) * F - cos(3x_t) * g \quad (7)$$

$$x_{t+1} = x_t + v_{t+1} \quad (8)$$

where the constants are:

$$g = 0.0025 \quad \text{(Gravity)}$$

$$F = 0.001 \quad \text{(Force)}$$

The position and velocity values are clipped in the range of $[-1.2, 0.6]$ and $[-0.07, 0.07]$ respectively.

Termination conditions: If the position of car is 0.5 or more, or when the episode has taken more than 200 actions.

4. Hyperparameter Tuning

Reinforcement Learning algorithms are known to be very sensitive to hyperparameter values, especially ones which uses Neural Networks. We performed tuning search on the following list of hyperparameters in a coarse to fine random search fashion:

- Network Architectures - While exploring different neural network architectures we manipulated two factors to observe their impact on the performance of the algorithm - namely, network depth and width. Depth is related to concatenating more layers to the model, whereas the width is related to increasing/decreasing the number of neurons in the layer. During depth adjustments, the width parameters were kept constant and layers were added/removed. We tested with n-layered neural networks in depth adjustment where n $\in [1, 2, 3]$. For each of these architectures, different widths were explored, encompassing smaller networks with [8,10,16,20,64] neurons in each layer to larger networks with [128,256] neurons. These experiments allowed us to observe the impact of both depth and width on the learning of the algorithm.
- Optimizers - We evaluated two set of optimization techniques using Stochastic Gradient descent and Adam optimizer. In case of Stochastic Gradient Descent we conducted search for hyperparameters Learning rate and Momentum. Various values of Momentum ranging in the range of [0,1] were tested (details about learning rate tuning is mentioned in the next point). On the other hand, for Adam optimizer we only tested with the learning rate.
- Learning Rates (α) - Learning rate has a huge significance in the training of the algorithm as it controls the pace at which we update the weights of our networks in response to the loss function. Knowing that different models work for different MDPs and their diverse functioning, we have tested sufficient amount of values for this hyperparameter. For each of the network architecture we defined, we tested different learning rate values within the range [1e-8, 0.5] in order to find the optimal policy (and also identify which learning rates are not effective).
- Gamma (γ)- Gamma plays a major role in the expected returns we get from an algorithm as it governs how much importance we are giving to the future rewards. We systematically checked various values of gamma within the range of [0, 0.99], that is, from no importance given to the future rewards to a high importance given.

For all possible combinations of these hyperparameters, we generated a reward versus number of iterations graph, to check whether the algorithm is learning efficiently or not. Next section goes over the results we got after testing these hyperparameters on the MDPs.

5. Experimental Results

To determine the best optimizer to use across all problems, we used Actor-Critic in the Cart Pole environment to gauge the performance of Stochatic Gradient Descent and the Adam optimizer. We found Adam performed faster in the experiments described below for Actor-Critic and so we elected to use this optimizer in every environment for every neural network.

5.1. Actor Critic

Our Actor Critic algorithm utilized models of similar architecture for both the Actor and the Critic. Both consisted of two layers of size 128 neurons with a ReLU activation func-

tion. The Actor then applies the softmax function to convert actions to probabilities/confidence scores. This same model architecture was found to be best for all of the environments we tried.

5.1.1 Cart Pole

Every experiment listed was performed over an average of 5 runs of the algorithm across 400-500 episodes.

For model architectures, we maintained a two-layer approach but modified the number of neurons in each layer. Layer sizes of 10 neurons per layer were able to approach reward values peaking around 140 sum of reward around episode 300. Layer sizes of 20 neurons performed much better, achieving peak performance of around 250 return within 350 episodes. Further increasing the layer sizes to 128 resulted in rewards peaking around 475 at episode 150 and 500 around episodes 260 - 280, as shown in Figure 4 (our best performing Actor-Critic model on Cart Pole).

When adjusting learning rates, we performed coarse-to-fine search over values ranging from 5e-1 to 1e-8, reducing the order by roughly 10 each time. Our best-performing model uses a learning rate of 0.005.

Another hyperparameter search was performed over γ. Here, we made smaller adjustments with values ranging from .7 to .99. Lower values of γ would struggle to converge. We found best results with $\gamma = 0.99$.

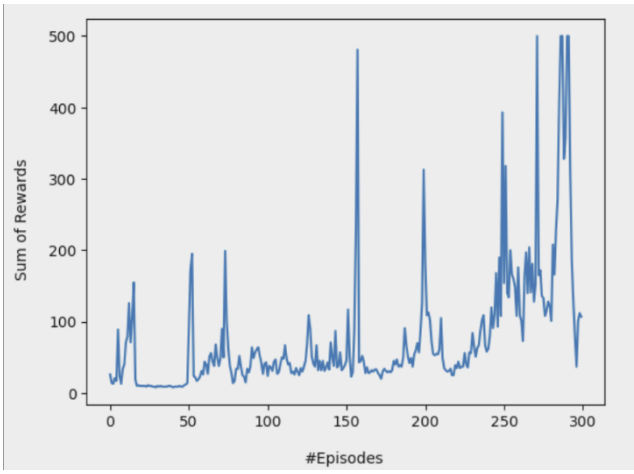

Figure 4. Best Performing Cart Pole with Learning Rate 0.005, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer

5.1.2 Mountain Car

The same coarse to fine approach was used in the Mountain Car environment as in Cart Pole. However, we were unable to reach an optimal solution within the Mountain Car environment or even find a model that learned any sort of reasonable policy as shown in Figure 5. We believe mountain car is simply a hard environment for this (and all of the other) algorithms.

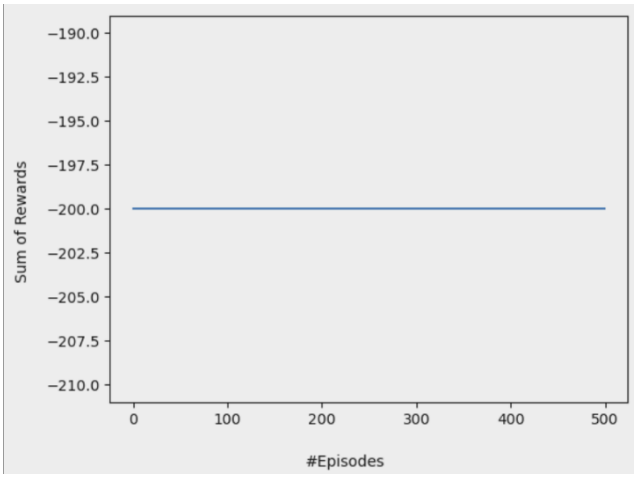

Figure 5. Typical Mountain Car Performance with Learning Rate 0.01, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer. All sets of hyperparameters had the same results

5.1.3 Acrobot

Similar to the Cart Pole environment, we initially experimented with layer sizes of 10 and 20 resulting in poor performance. The algorithm failed to converge to -100 reward until we increased the layer sizes to 128.

When adjusting learning rates, we performed coarse-to-fine search over values ranging from 5e-1 to 1e-8, reducing the order by roughly 10 each time. Low learning rates of around 1e-5, for example, would struggle to improve performance above -500 for a couple hundred episodes. Letting this algorithm run for another 300 episodes showed highly unstable performance and was unable to consistently pass -300 sum of rewards. Our best-performing model uses a learning rate of 0.001 as shown in Figure 6. This model was able to converge to a good solution, reaching a reward of -100 within roughly 120 episodes. Another hyperparameter search was performed over γ. Here, we made smaller adjustments with values ranging from .7 to .99. Lower values of γ would struggle to converge. We found best results with $\gamma = 0.99$.

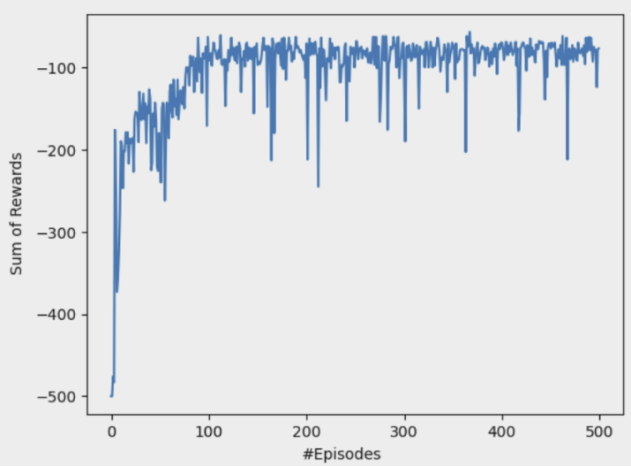

Figure 6. Best Performing Acrobot with Learning Rate 0.001, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer

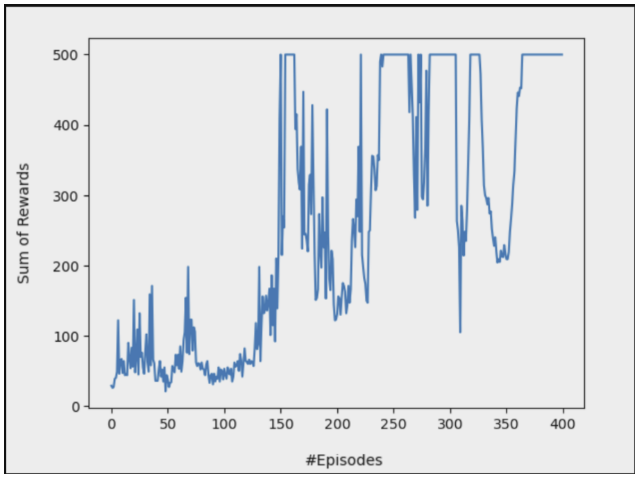

Figure 7. Best Performing Cart Pole with Learning Rate 0.01, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer

5.2. REINFORCE with Baseline

Our REINFORCE with Baseline algorithm utilized models of similar architecture for both the Policy Network and the Value Estimate. Both consisted of two layers of size 128 neurons with a ReLU activation function. The Policy Network then applies the softmax function to convert actions to probabilities/confidence scores. This same model architecture was found to be best for all of the environments we tried.

5.2.1 Cart Pole

We tried a smaller network at first for this algorithm on Cart Pole, using two layers of size 20 neurons at first. Performance was mediocre and failed to converge. We then switched to layer sizes of 128 as in Actor-Critic and saw increases in performance.

When adjusting learning rates, we performed coarse-to-fine search over values ranging from 1e-1 to 1e-6, reducing the order by 10 each time. Lower learning rates of around 0.001 would converge slowly, reaching a sum of reward of 400 around 250 episodes and would later converge to 500 around 300-360 episodes. Our best-performing model uses a learning rate of 0.01.

Another hyperparameter search was performed over γ. Here, we made smaller adjustments with values ranging from .7 to .99. Lower values of γ would struggle to converge. We found best results with $\gamma = 0.99$, same as we did with Actor Critic. On the Cart Pole Environment, our best model achieved an average return of 500 within 150 episodes as shown in Figure 7.

5.2.2 Mountain Car

Mountain car failed to converge and even failed to learn in any way across all hyperparameter settings we tried. We used the same search method and the same range of values as the prior environment. Performance is displayed in Figure 8. We believe mountain car is simply a hard environment for this (and all of the other) algorithms.

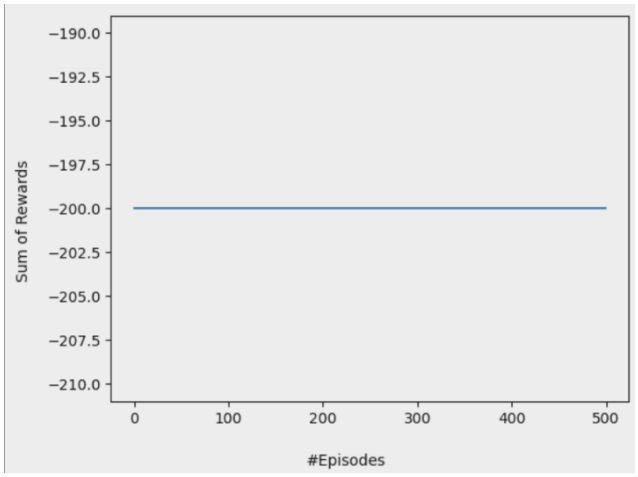

Figure 8. Typical Mountain Car Performance with Learning Rate 0.01, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer. All sets of hyperparameters had the same results

5.2.3 Acrobot

When adjusting learning rates, we performed coarse-to-fine search over values ranging from 5e-1 to 1e-8, reducing the order by 10 each time. Our best-performing model uses a learning rate of 0.01. REINFORCE with Baseline achieved

similar convergence speed as that of Actor Critic by reaching the reward threshold of -100 in around 95 episodes as shown in Figure 9.

We tried a smaller network at first for this algorithm, using two layers of size 20 neurons at first. Performance was mediocre and failed to converge. We then switched to layer sizes of 128 as in Actor-Critic and saw increases in performance.

Another hyperparameter search was performed over γ. Here, we made smaller adjustments with values ranging from .7 to .99. Lower values of γ would struggle to converge. We found best results with $\gamma = 0.99$.

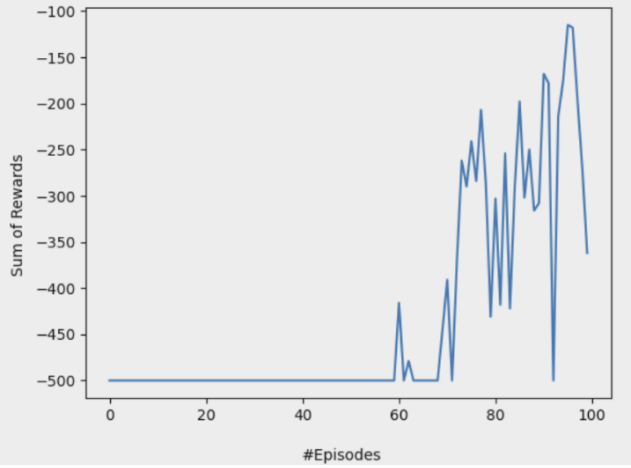

Figure 9. Acrobot Performance with Learning Rate 0.01, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer.

5.3. Episodic Semi Gradient n-step SARSA

5.3.1 Cart Pole

For the Cart Pole Environment with this algorithm, performance deviated quite a bit from that of Actor-Critic and REINFORCE with Baseline. We experimented with network sizes of 10, 16, and 128 as before but none of these networks provided satisfactory results, unable to break 15-20 sum of rewards within 500 episodes. We tried another network size of 64 and saw our best performance, though still poor overall, with a platue around 25 sum of rewards. There are peaks in performance to roughly 50 sum of rewards around episode 220, but this is likely due to randomness.

Given that this algorithm requires a tile coding, we tried values between 5 to 100 and saw best performance with 100 tiles. We figure lower values are not able to properly capture the space.

Another hyperparameter search was performed over γ. Here, we made smaller adjustments with values ranging from .7 to .99. Any γ value besides .99 would look like noise oscillating around an average sum of rewards of 10 across all 500 episodes. We found best results with $\gamma = 0.99$.

Another adjustment made was to the value of ϵ for the $\epsilon \sim$ greedy selection of actions. We tried values of 0.01 to 0.3 and found the best-performing value to be 0.1. We tried decaying epsilon over time using an exponential decaying function - a recommendation of the textbook - but did not find any performance improvement in this environment. The decay factor for ϵ is $\frac{e^{0.1 * \text{episode}}}{3}$.

As recommended in class the learning rate was scheduled as well. We did this by multiplying the learning rate by 0.95 after each step. In this environment, there was no significant improvement in performance. Given more time we could have tried multiplying by other factors. Our best learning rate was set to 0.001.

Our best performing model is shown in Figure 10 with the following hyperparameters: number of tilings = 100, initial epsilon = 0.1, number of steps = 5, $\gamma = 0.99$

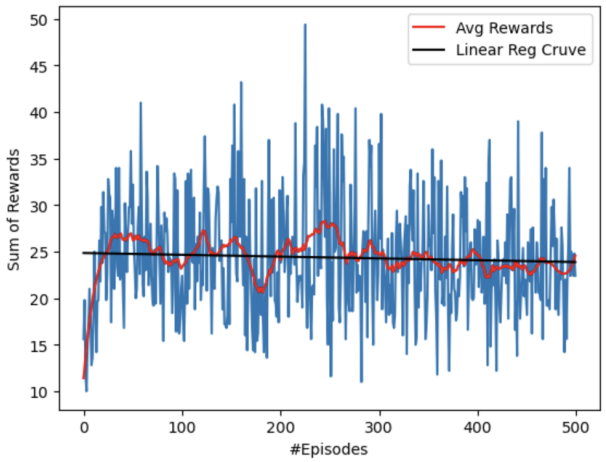

Figure 10. Best Performing Cart Pole with Learning Rate 0.001, $\gamma = .99$, on a two layer Neural Net with 128 Neurons Per layer, number of tilings = 100, and $\epsilon = 0.1$

5.3.2 Mountain Car

The same adjustments to hyperparameters as described in Cart Pole were used on this environment but all were completely unable to learn to improve performance as shown by Figure 11. We believe mountain car is simply a hard environment for this (and all of the other) algorithms.

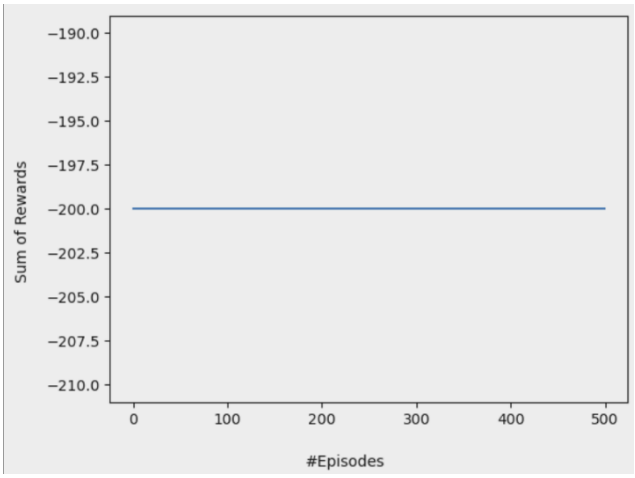

Figure 11. Typical Mountain Car Performance with Learning Rate 0.01, γ = .99, on a two layer Neural Net with 128 Neurons Per layer, number of tilings = 100, and ϵ = 0.1. All sets of hyperparameters had the same results

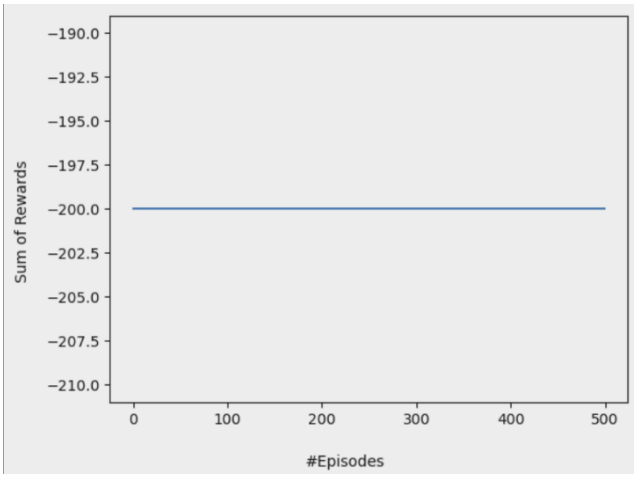

Figure 12. Best Performing Acrobot with Learning Rate 1e-5, γ = .99, on a two layer Neural Net with 128 Neurons Per layer, number of steps = 7, number of tilings = 8, and ϵ = 0.1, without learning rate and ϵ scheduling

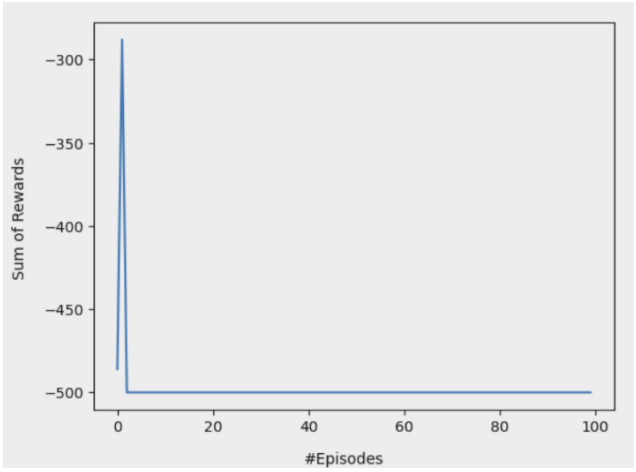

Figure 13. Best Performing Acrobot with Learning Rate 1e-5, γ = .99, on a two layer Neural Net with 128 Neurons Per layer, number of steps = 7, number of tilings = 8, and ϵ = 0.1, with learning rate and ϵ scheduling

5.3.3 Acrobot

We performed the same hyperparameter tuning for Acrobot as we did in Cart Pole for this algorithm. In this algorithm we did see benefit from introducing scheduling to both the learning rate and ϵ for action selection. Overall, this algorithm did not perform well on Acrobot. Figure 12 shows how the variant of the algorithm without any scheduling performed; no learning or performance improvement. Whereas Figure 13, out best performing model, shows how including scheduling resulted in a brief spike in performance to around -290 before flatlining. In our best performing model, lower tiling size of 8 proved to increase performance where higher values flatlined at -500.

Algorithm 1 Actor-Critic

class Actor:
  function __init__(self, state_space, action_space):
    Initialize layers
  function forward(self, state):
    Apply ReLU and Softmax
class Critic:
  function __init__(self, state_size):
    Initialize layers
  function forward(self, state):
    Apply ReLU
function critic_loss(δ):
  return 0.5 * δ^2
function actor_loss(action_prob, reward, I, delta):
  return I * -action_prob * delta
function run_actor_critic(gamma, Learning Rate as LR):
  Initialize variables, models, and Adam optimizer and Learning rate as LR
  for run **in** 1, 2, ... :
    for episode **in** 1, 2, 3, .. 500 :
      Initialize episode, variables
      while episode has not terminated:
        Calculate action probabilities using Actor
        Create distribution from action probabilities then sample and take action
        Estimate state values using Critic
        Calculate $\delta \leftarrow R + \gamma\hat{v}(S', w) - \hat{v}(S, w)$
        Calculate w_loss, theta_loss
        Update Actor with backpropagation $\theta \leftarrow \theta + \alpha^{\theta} I\delta\nabla \ln \pi(A|S, \theta)$
        Update Critic with backpropagation $w \leftarrow w + \alpha^{w}\delta\nabla\hat{v}(S, w)$
        Update state
    Update running average

Algorithm 2 REINFORCE with Baseline

class policy_model:
  def __init__(self, state_space, action_space):
    Initialize neural network layers
  def forward(self, state):
    Apply ReLU and Softmax
class value_model:
  def __init__(self, state_size):
    Initialize neural network layers
  def forward(self, state):
    Apply ReLU
function policy_loss(A, γ, t, δ):
  $\gamma^t * -\ln(A) * \delta$
function value_loss($\hat{v}$, G, S):
  $.5 * (G - \hat{v}(S, w))^2$
function generate_episode(policy):
  Initialize environment
  while episode not ended:
    Compute action probabilities
    Select and execute action
    Append state, reward, action probabilities
  return states, actions, rewards, episode length
function reinforce_with_baseline(gamma, M, Learning Rate):
  Initialize policy_model and value_estimator
  Define policy_optimizer and value_optimizer with Adam and Learning Rate
  For episode in 1, 2, ... 500 :
    Generate states, actions, rewards, T using generate_episodes()
    Append sum of rewards to episode_rewards
    For t in range(T)
      Calculate return G for time step t ($G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$)
      Compute ($\delta \leftarrow G - \hat{v}(S_t, w)$)
      Compute policy loss
      Backpropagate and update policy_model ($\theta \leftarrow \theta + \alpha^{\theta} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \theta)$)
      Compute value loss
      Backpropagate and update value_model ($w \leftarrow w + \alpha^{w} \delta \nabla \hat{v}(S_t, w)$)

Algorithm 3 Episodic N-Step SARSA

class hashTable:
  def __init__(self, size):
    Initialize size and dictionary
  def size_dict(self):
    return Size of the dictionary
  def getTileIndex(self, coords):
    Check and return tile index
function get_tile_index(hash_table, num_tilings, state, actions):
  Quantize state for discretization
  Initialize Tiles list
  for each tiling:
    Calculate tile coordinates
    Get tile index using hash table
  return tensor of Tiles
class ActionNetwork:
  def __init__(self, num_tilings):
    Initialize layers
  def forward(self, state):
    Apply ReLU
function policy(action_network, env, hash_table, num_tilings, state, epsilon):
  Calculate action values and choose action
function policy_loss(G, hash_table, num_tilings, action_network, states, actions, tau):
  return Squared error loss
function n_step_sarsa(env, hash_table, num_tilings, epsilon, n, gamma, epochs):
  Initialize Tile Coding Hash Table, $\hat{q}(S, w)$ as ActionNetwork model, Adam Optimizer with exponentially decaying learning rate with a multiplicative factor of .95

loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  Represent S_0 with Tile Coding
  Select and store an action $A_0 \sim \epsilon$-greedy w.r.t $\hat{q}(S_0, \cdot, w)$
  $T \leftarrow \infty$
  Decay ϵ by $\frac{e^{0.1 * \text{episode}}}{3}$
  loop for $t = 0, 1, 2, \ldots$:
    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}
      Represent S_{t+1} with Tile Coding
      if S_{t+1} is terminal **then**
        $T \leftarrow t + 1$
      else
        Select and store $A_{t+1} \sim \epsilon$-greedy w.r.t $\hat{q}(S_{t+1}, \cdot, w)$
      end if
    end if
    if $\tau \geq 0$ **then**
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
      if $\tau + n < T$ **then**

        Calculate Policy Loss ($G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, w)$)
      end if

      Perform Backpropagation and update of ActionNetwork
      ($w \leftarrow w + \alpha \left[G - \hat{q}(S_\tau, A_\tau, w)\right] \nabla \hat{q}(S_\tau, A_\tau, w)$)
    end if
  end loop
end loop until $\tau = T - 1$