ECE 493 Assignment 3 Report
Akshay Patel | Student ID: 20732500

Unless otherwise noted, all figures and algorithm explanations are from the second edition Reinforcement Learning textbook used for this course, written by Richard S. Sutton and Andrew G. Barto.

Domain Description

Maze World is the deterministic environment that was used for this assignment. States, actions, and dynamics are defined as follows. States are defined as the agent being in a specific cell of the maze, and they are represented by four coordinates. Tasks can have walls that the agent cannot pass through, pits that are terminal states where the agent receives a large negative reward, and all tasks have a goal state for which the agent receives a positive reward. The action space in this environment consists of {up, down, right, left} represented by integers {0, 1, 2, 3} respectively. The dynamics of this environment are that given a state *s*, and an action *a*, taking action *a* from a state *s* will result in the agent being in a state *s'*. However, if the action takes the agent into a wall or takes the agent out of the grid, then the state will remain the same. If the agent reaches the goal or runs into a pit, then the game ends as those are terminal states. The reward function is that reaching the goal gives you a reward of +1 and ends the episode, running into a pit yields a penalty of -10 and ends the episode, running into a wall will give you a penalty of -0.3, and apart from these, the agent will receive a -0.1 reward for reaching any other cell in the grid.

Maze World is deterministic, meaning that if the agent takes a specific action from a specific state, it will always go to the same next state as well as receive the same reward value. The objective of this environment given this reward function is to reach the goal state as quickly as possible to maximize the reward.

Algorithm Explanations

Expected Sarsa

Expected Sarsa is a learning algorithm that is similar to Q-Learning except for the fact that instead of taking the maximum over the next state-action pairs, Expected Sarsa uses the expected value which is calculated by taking into account how probable each action is given the current policy. Expected Sarsa has the same pseudocode as Q-Learning, except for the update rule which is changed, and given below (from the RL textbook page 133).

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

Given the next state, Expected Sarsa moves *deterministically* in the same direction as Sarsa moves *in expectation*, which is why this algorithm is called Expected Sarsa. Expected Sarsa is clearly more complex and uses more computational resources than Sarsa given the difference in update rules. Expected Sarsa can also be described as being more complex and more computationally expensive than Q-Learning as it requires more floating-point operations in the expectation calculation. However, in comparison to Sarsa, this extra computational expense results in the elimination of variance due to the random selection of the next action. Given the same amount of learning time, Expected Sarsa should perform slightly better than Sarsa.

What's even more interesting about Expected Sarsa is that it shows a significant advantage over Sarsa across a wide range of values for alpha, the step-size parameter or learning rate. Expected Sarsa can **safely** set alpha = 1 without suffering from any degradation of asymptotic performance. Sarsa cannot perform very well without the learning rate (alpha) having a small value.

Expected Sarsa can be thought of as subsuming and generalizing Q-Learning while improving over Sarsa with the reduction in variance, albeit at the additional computational cost. Expected Sarsa may perform much better than both of Sarsa and Q-Learning. Also, Expected Sarsa can be both on-policy or off-policy.

In my implementation, I use essentially the same code as my Q-Learning algorithm except that the learn function is changed to use the update outlined above using expectation of the value from the next state-action pairs. The expectation probabilistic distribution I used is ((1-epsilon)/num_max_actions)+(epsilon/num_actions) if a given action returns the maximum possible value for a given state, or (epsilon/num_actions) for a non-max action. I also use an epsilon-greedy selection mechanism, which over time gradually decreases as I decay epsilon by 1% every episode. By doing so, I can encourage exploration in the beginning episodes, and encourage exploitation as the agent has more experience in the environment. However, in the plots below, I mistakenly did not call the epsilon decay function after every episode, so we don't see the benefits of this mechanism there.

The hyperparameters I used for Expected Sarsa are as follows: learning rate = 1 as Expected Sarsa performs well across a wide range of learning rates and a large learning rate helped with faster convergence; reward decay = 0.9 so as to account for future reward; epsilon greedy = 0.1 with epsilon decay factor of 1% each episode. In my experiments, a very small learning rate resulted in slower convergence in a limited number of episodes (2000), which is why I increased it.

<u>Double Q-Learning</u>

Double Q-Learning is a double learning variant of Q-Learning that tries to avoid maximization bias (see the RL book for an explanation of maximization bias). Double Q-Learning divides the time steps in two, updating one of two Q-tables depending on a random selection between the two. The two approximate value functions are treated symmetrically, and the behaviour policy can use both action-value estimates from both Q-tables. An epsilon-greedy policy for Double Q-Learning can be based on the sum or average of the two action-value estimates. In my implementation, I use the sum of the two action-value estimates from the two Q-tables.

Compared to Q-Learning, Double Q-Learning can eliminate the harm caused by maximization bias leading to better learning.

Double Q-Learning clearly requires more computational space as it requires the storing of two Q-tables and the summing of all corresponding Q-table values during the choice of an action. In an environment with a large state-space and a large action-space, the necessity to store two entire Q-tables can prove to be much more computationally expensive than Q-Learning or Sarsa. In terms of computational expense, it requires less operations than Expected Sarsa.

The hyperparameters I used for Double Q-Learning after various experiments are as follows: learning rate = 0.05 as a slightly larger learning rate lead to faster convergence; reward decay of 0.95 so as not to discount the future rewards as much since Q-Learning can sometimes take the more risky path compared to methods like Sarsa (e.g. Cliff Walk scenario); and epsilon greedy = 0.1 with epsilon decay factor of 1% each episode. In my experiments, a very small learning rate resulted in slower convergence in a limited number of episodes (2000), which is why I increased it to 0.05.

Complete pseudocode for the Double Q-Learning algorithm is presented below, upon which my implementation is based. I use an epsilon-greedy selection mechanism, which over time gradually decreases as I decay epsilon by 1% every episode. By doing so, I can encourage exploration in the beginning episodes, and encourage exploitation as the agent has more experience in the environment. However, in the plots below, I mistakenly did not call the epsilon decay function after every episode, so we don't see the benefits of this mechanism there.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(\mathit{terminal}, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \operatorname{argmax}_a Q_1(S', a)\big) - Q_1(S, A)\Big)$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \operatorname{argmax}_a Q_2(S', a)\big) - Q_2(S, A)\Big)$$

    $S \leftarrow S'$
  until S is terminal

Sarsa(Lambda) – Eligibility Traces

Eligibility traces are one of the basic mechanisms of reinforcement learning that almost any TD method, including Sarsa or Q-Learning, can be combined with to produce a more general algorithm that might learn more efficiently. They unify and generalize TD and Monte Carlo methods. When TD methods are combined with eligibility traces, they produce a set of methods spanning a spectrum with Monte Carlo methods at one end, when lambda = 1, and one-step TD methods at the other end, when lambda = 0. In the middle are intermediate methods that can often be better than either extreme method when lambda is 0 or 1. Eligibility traces also offer an elegant, algorithmic mechanism with significant computational advantages through a short-term memory vector, the eligibility trace. This is better than n-step methods in terms of computational expense because eligibility traces only require a single trace vector rather than a store of the last n feature vectors. The trace decay factor, lambda, which is in [0,1] interval, determines the rate at which the trace falls. See the RL book for a more thorough and in-depth explanation of the different kinds of eligibility traces that exist.

The hyperparameters I used for Sarsa(Lambda) are as follows: learning rate = 0.01 to make small improvements towards optimal policy; reward decay = 0.9 to take into account future rewards; epsilon greedy = 0.1 with epsilon decay factor of 1%; and lambda = 0.9 to fade the eligibility trace slowly as it gets farther away from the goal state.

In my implementation, I decided to implement Sarsa(Lambda). The temporal-difference method for action values, known as Sarsa(Lambda) approximates a forward view of the next state-action pairs. Complete pseudocode for Sarsa(Lambda) is given below from the ECE 493 Lecture videos on YouTube.

Sarsa(λ) Algorithm

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s,a

Repeat (for each episode) :

  Initialize s,a

  Repeat (for each step of episode) :

    Take action a, observe r,s'

    Choose a' from s' using policy derived from Q (e.g. $?$ - greedy)

    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$

    $e(s,a) \leftarrow e(s,a) + 1$

    For all s,a : s.t. $e(s,a) > 0$

      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$

      $e(s,a) \leftarrow \gamma\lambda e(s,a)$

    $s \leftarrow s'; a \leftarrow a'$

  Until s is terminal

My implementation is based on the pseudocode given above. It is clear that this eligibility trace with Sarsa requires a similar space complexity as Double Q-Learning due to the use of the eligibility trace. However, it requires more floating point operations as the entire Q-table is modified and the entire eligibility trace is modified for all state-action pairs. Given a large state space and a large action-space, this can be much more computationally expensive in terms of operations compared to Double Q-Learning or Expected Sarsa. I also use an epsilon-greedy action selection mechanism, which over time gradually decreases as I decay epsilon by 1% every episode. By doing so, I can encourage exploration in the beginning episodes, and encourage exploitation as the agent has more experience in the environment. However, in the plots below, I mistakenly did not call the epsilon decay function after every episode, so we don't see the benefits of this mechanism there.

REINFORCE: Monte-Carlo Policy Gradient Control

I used this article (https://medium.com/@ts1829/policy-gradient-reinforcement-learning-in-pytorch-df1383ea0baf) to better understand and explain Policy Gradient in PyTorch below.

Policy Gradient methods learn a parameterized policy that can select actions without consulting a value function, which is quite different from the algorithms we have used so far which are action-value methods that learn the values of actions and then select actions based on their estimated action values. Note that a value function could still be used to learn the policy parameter, but this is not required for action selection. The policy parameter is learned based on the gradient of some performance measure with respect to the policy parameter.

A policy gradient basically attempts to train an agent without explicit mapping of values to each state-action pair in an environment by taking small steps and updating the policy based on the reward associated with that step.

The following algorithm is known as REINFORCE, for which a derivation and complete explanation is given in the RL book in Chapter 13.

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

In my implementation, which is based on the above pseudocode and the examples given in the PyTorch repository on GitHub, I use a simple feed forward neural network with one hidden layer

that has 128 neurons along with a dropout of 0.6 and an Adam optimizer. The dropout is used to regularize the NN to reduce overfitting and reduce the computational expense of the NN by randomly dropping out nodes during training. More information regarding dropout methods can be found here: https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/ . I use a one-hot encoding for the current state that the agent is in inside the Maze as an input into the NN.

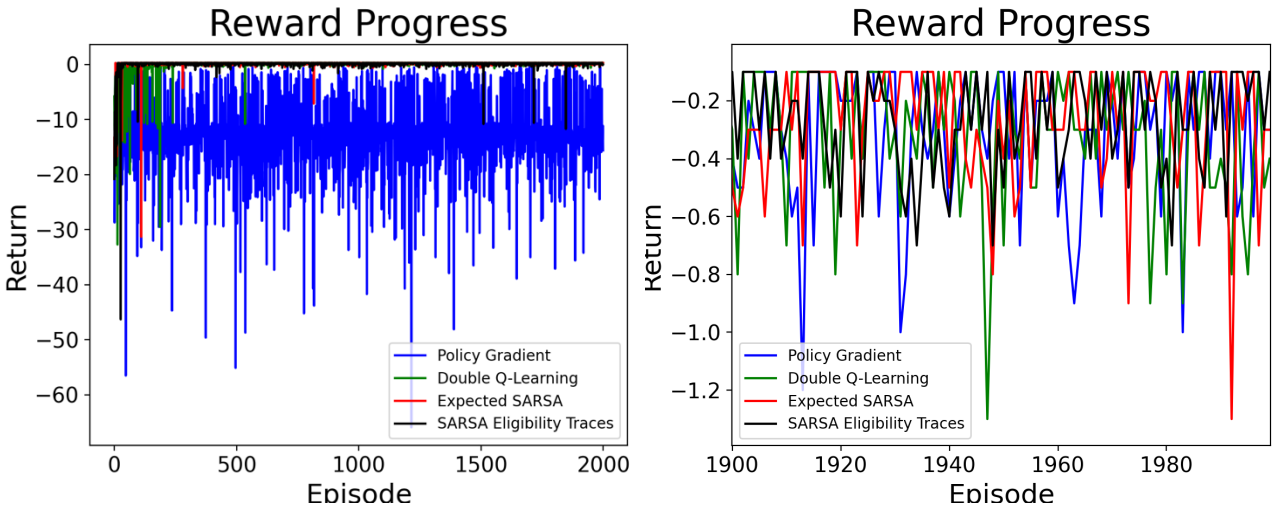
I choose an action based on the policy probability distribution using the PyTorch package. The stochastic policy returns a probability for each possible action in our action space as an array of length 4, since we have 4 actions in Maze World. Using this vector of probabilities, we choose an action, save the log probability, and then return our action. The policy is updated after each episode or after 10000 steps when the agent is not able to complete the episode. The 10000 is used as the maximum allowed steps in a single episode to prevent endless episodes where the agent is looping through the same states. The rewards multiplied by the log probabilities are fed to the Adam optimizer and the weights of the NN are updating using stochastic gradient ascent. This should increase the likelihood of actions that resulted in our agent receiving a larger reward.

In terms of computational resources, this method uses much more space (from the neural network architecture) and requires many more operations to fully train the neural network weights in order to correctly learn the policy.

For my hyperparameters, I use the same NN architecture for all tasks, along with a reward decay of 0.999 so as to minimize the discounting of future rewards as much as possible. For the learning rate that is passed into Adam, I used the following learning rates after experimenting and determining what worked well in the time I had to experiment: 5e-5 for Task 1, 1e-2 for Task 2, and 1e-3 for Task 3.

Results and Analysis

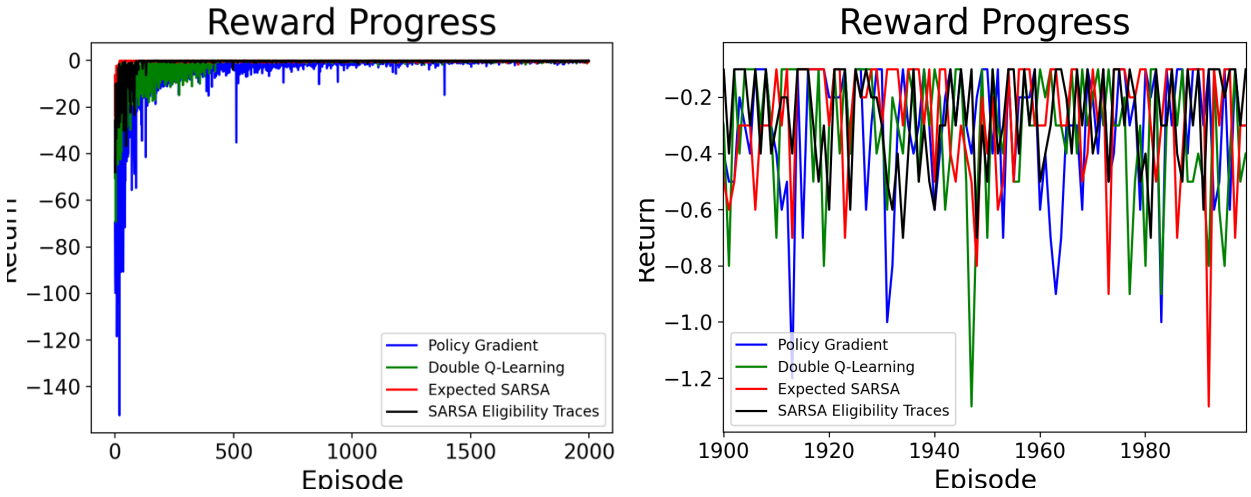
Task 1



As we can see in the plots above, Double Q-Learning, Expected Sarsa, and Sarsa with Eligibility Traces (or Sarsa(Lambda)) perform quite well in Task 1, with all of them converging over the last 100 episodes. All 4 of the algorithms are able to obtain the maximum reward of +0.3 from the environment in this task, but Policy Gradient does not seem to converge very well after 2000 episodes as can be seen in the plot on the left. We can see that Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) still have some episodes in which they do not obtain the maximum value at the end of training, but this is most likely due to the epsilon-greedy selection. These spikes would likely have been minimized with epsilon decay as the algorithms would have chosen actions based on their developed policies and choosing actions based on epsilon much less. Between the three algorithms that do converge (everything except Policy Gradient), Sarsa(Lambda) seems to converge the fastest. Compared to normal Q-Learning and Sarsa, all three of Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) converge faster and have less variance, albeit at some more computational costs. This can be seen by comparing the plots above to the plots from Assignment 2 Report. In terms of the variance over the last 100 episodes, Policy Gradient has the highest variance of approximately 28, while Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) have variances over the last 100 episodes of 0.02, 0.01, and 0.3 respectively which could have been even lower with epsilon decay being used in the experiments.

The Policy Gradient algorithm did not converge as policy gradients, especially REINFORCE without baseline, suffer from high variance and low convergence. The stochastic policy may take different actions in different episodes, and one wrong action can completely alter the episode result. Perhaps what happened is that the policy got stuck in a local maximum and decided to optimize for running into the pit as fast as possible from some states to maximize reward by minimizing negative rewards accumulated through long episodes, but in other states it decided to go for the goal state when it knew that was the optimal path. What may have helped Policy Gradient converge better in this Task 1 is possibly a lower learning rate so it doesn't decide to run to the pits to maximize reward, possibly using REINFORCE with Baseline to minimize variance, and to train for a larger number of episodes. Entropy-based or curiosity-based exploration could also be added to this existing algorithm to encourage more exploration instead of going into the terminal pits.

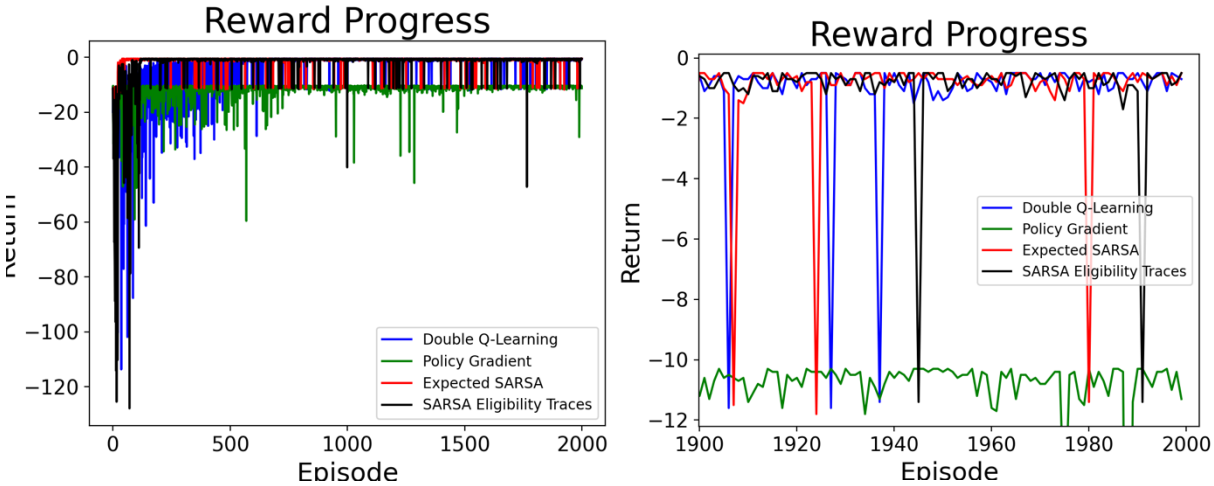
Task 2



As we can see in the plots above, all 4 of the algorithms perform quite well in Task 2, with all of them converging in the later episodes. All 4 algorithms converge in order to obtain the maximum reward of -0.1 from the environment. However, Double Q-Learning and Policy Gradient take longer to converge compared to Expected Sarsa and Sarsa(Lambda). We can see that Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) still have some episodes in which they do not obtain the maximum value at the end of training, but this is most likely due to the epsilon-greedy selection. These spikes would likely have been minimized with epsilon decay as the algorithms would have chosen actions based on their developed policies and choosing actions based on epsilon much less. Compared to normal Q-Learning and Sarsa, all three of Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) converge faster and have less variance, albeit at some more computational costs. This can be seen by comparing the plots above to the plots from Assignment 2 Report.

In terms of the variance over the last 100 episodes, Sarsa(Lambda) has the highest variance of approximately 0.3, while Double Q-Learning, Expected Sarsa, and Policy Gradient have variances over the last 100 episodes of 0.06, 0.04, and 0.6 respectively which could have been even lower for Double Q-Learning and Expected Sarsa with epsilon decay being used in the experiments.

Task 3



As we can see in the plots above, Double Q-Learning, Sarsa(Lambda), and Expected Sarsa perform quite well in Task 3, with all of them converging over the last 100 episodes in order to obtain the maximum reward off -0.50 from this task. Policy Gradient does converge and performs well but not as well as the other algorithms and the maximum reward obtained by it is -8.8 in Task 3. Between the three algorithms that do converge and obtain the maximum reward from this Task, Double Q-Learning takes the longest time to converge. We can see that Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) still have some episodes in which they do not obtain the maximum value at the end of training, but this is most likely due to the epsilon-greedy selection. These spikes would likely have been minimized with epsilon decay as the algorithms would have chosen actions based on their developed policies and choosing actions based on epsilon much less.

Compared to normal Q-Learning and Sarsa, all three of Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) converge faster and have less variance, albeit at some more computational costs. This can be seen by comparing the plots above to the plots from Assignment 2 Report.

In terms of the variance over the last 100 episodes, Policy Gradient has the highest variance of 3.7, whereas the Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) have variances of 3.4, 3.5, and 2.3 respectively.

What may have caused Policy Gradient to not converge to the optimal policy is the algorithm getting stuck in a local maximum while doing gradient ascent in order to find the optimal policy. Perhaps what could have helped Policy Gradient converge better would be encouraging more exploration. Entropy-based or curiosity-based exploration could be added to this existing algorithm to encourage more exploration in order to reach the goal state instead of going into the terminal pits.

Overall Analysis

Out of the 4 new algorithms outlined in this report, Policy Gradient is the most difficult to implement, as well as the one that requires the most hyperparameter tuning for each Task in order to remotely do well at the task. Moreover, Policy Gradient has very long episodes that if not capped with a maximum number of steps, can lead to very long training. The other 3 algorithms are much easier to implement and converge much faster with lower variance and at the cost of less computational resources, and so Policy Gradient (REINFORCE without Baseline) is not the best option for this environment.

As seen through the three tasks above, Double Q-Learning, Expected Sarsa, and Sarsa(Lambda) all perform significantly better than normal Q-Learning and Sarsa, albeit at the cost of computational resources. These three algorithms are slightly more complex to implement than normal Q-Learning and Sarsa, but not much more difficult, and so they are preferred over Q-Learning and Sarsa.

Out of the three algorithms that remain, Double Q-Learning does not perform the best in terms of convergence and learning, nor is it the least computationally expensive. Expected Sarsa and Sarsa(Lambda) perform much better. Between Expected Sarsa and Sarsa(Lambda), it seems that both perform equally as well, but Expected Sarsa does not require the extra computational space and numerous operations that Sarsa(Lambda) requires, and so Expected Sarsa is the recommended algorithm for this environment.

Something I would have liked to measure but was not able to is the amount of CPU Usage per algorithm and the amount of time in seconds that each algorithm took to complete the 2000 episodes per task. This information could have been very insightful in comparing the algorithms' performance from a quantitative compute-based perspective.