Assignment 03 Mohammad

Implementation

> 💡 Implement Expected SARSA

Expected Sarsa learning is implemented in `learning_algorithms/RL_brain_expected_sarsa.py`.

> 💡 Implement Double Q-Learning

Double Q-Learning is implemented in `learning_algorithms/RL_brain_double_q_learning.py`.

> 💡 Implement Eligibility Traces

Eligibility Traces is implemented in `learning_algorithms/RL_brain_eligibility_trace_sarsa.py`.

The implementation as seen by the title of the file is the `SARSA` variant.

> 💡 Implement Policy Gradient (REINFORCE or REINFORCE with baseline) - Any function approximation - linear or Neural Networks can be used.

Gradient Reinforce is implemented in `monte_carlo/MC_policy_gradient.py`.

💡 Describing each algorithm you used, define the states, actions, dynamics. Define the mathematical formulation of your algorithm, show the Bellman updates you use. Clearly specify the different hyper-parameters values (learning rate, discount factor, epsilon etc) you used for the algorithms and justify the reasons for these choices.

Overview of all Hyper-Parameters Used

Throughout this report, various hyper-parameters will be discussed. Below is their description, range and symbols.

The Hyper-Parameters used are

- γ : Discount Rate
- α : Learning Rate / StepSize
- ϵ : Epsilon (greedy)
- λ : Decay Rate

Discount rate γ controls how much to discount future rewards and is in the range $[0, 1]$.

Learning rate α determines the speed of "learning" and is in the range of $[0, 1]$. This is sometimes known as step-size.

Epsilon ϵ is a small value and is $0 < \epsilon << 1$. It is typically used in the action selection process when choosing between best actions vs all actions.

Decay rate λ is in reference to an eligibility vector and determines the decay of said vector and is in the range $[0, 1]$.

Dynamics

All algorithms are performed on grid-world with the same dynamics function.

$$p(s\prime, r|s, a) = Pr\{S_t = s\prime, R_t = r|S_{t-1} = s, A_t = a\}$$

The environment is deterministic, therefore a valid transition will have a probability of 1, and everything else will have a probability of 0.

Expected Sarsa

Description

Expected Sarsa estimates the state-action function similar to Q-Learning, but uses an expected value for the state-value function.

Algorithm 1 Expected Sarsa

Initialize $Q(s, a)$ arbitrarily for all s,a
loop {over episodes}
  Initialize s
  repeat {for each step in the episode}
    choose a from s using policy π derived from Q
    take action a, observe r and s'
    $V_{s'} = \sum_a \pi(s', a) \cdot Q(s', a)$
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma V_{s'} - Q(s, a) \right]$
    $s \leftarrow s'$
  until s is terminal
end loop

Expected Sarsa Pseudo Code

States

Expected sarsa stores an estimation of the state-action-value function, also known as the Q-table.

Actions

The next action is chosen via the ϵ - greedy method, with an $\epsilon = 0.1$.

Mathematical Formulation

Expected Sarsa has an update rule similar to Q-learning, except it uses an expected value.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a|s_{t+1})Q(S_{t+1}) - Q(S_t, A_t)]$$

The name "Expected" comes from the term,

$$\sum_a \pi(a|s_{t+1})Q(S_{t+1}),$$

which is the expected value of Q for state S_{t+1}.

The policy for a state S_t is defined as:

- $Pr(best\_action) = 1 - \epsilon$
- $Pr(non\_best\_actipn) = \epsilon$

where $best\_action$ refers to the $argmax_a Q(S_t, a)$.

Hyper Parameters

The hyper-parameters used and their values are shown below.

The Hyper-Parameters used are

- γ : Discount Rate = 0.9
- α : Learning Rate = 0.01
- ϵ : Epsilon (greedy) = 0.1

A small learning rate was used to avoid large updates to the Q table.

A small ϵ was used to provide greater weightings to choose greedy actions and bias Q values, via the expectation, to the greedy state-action values.

A large discount rate γ was used to give the agent a larger horizon for rewards.

Double Q Learning

Description

Double Q-Learning estimates two versions of the Q-table independently to avoid maximisation bias [Reinforcement Learning Second Edition].

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 $Q(terminal, \cdot) = 0$

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:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \text{argmax}_a\, Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \text{argmax}_a\, Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

Double Q Learning Pseudo Code

States

Double Q-Learning stores two Q-tables, $Q_1(S, A)$ and $Q_2(S, A)$, each updated with a probability of 0.5 during the "learn" phase of each iteration.

Actions

The next action is chosen via the ϵ - greedy method, with an $\epsilon = 0.1$, except the average of Q_1 and Q_2 are used opposed to a single Q table.

Mathematical Formulation

Updates are made the same way as Q-learning, however one of Q_1, Q_2 are updated each iteration with probability 0.5.

Hyper Parameters

The Hyper-Parameters used are:

- γ : Discount Rate = 0.9
- α : Learning Rate = 0.01
- ϵ : Epsilon (greedy) = 0.1

A small learning rate was used to avoid large updates to the Q table.

A small ϵ was used to provide greater weightings to choose greedy actions.

A large discount rate γ was used to give the agent a larger horizon for rewards.

Eligibility Traces (Sarsa(λ))

Description

Eligibility traces use the concept of backward view, which tells the agent to look at past rewards to view how it got there vs looking forward to expected rewards. This backward view is represented as the eligibility vector.

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 :

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

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

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

  Until s is terminal

Eligibility Trace Sarsa Lambda

States

Sarsa(λ) stores a Q table and an eligibility vector e.

Actions

The next action is chosen via the ϵ - greedy method, with an $\epsilon = 0.1$.

Mathematical Formulation

The Q table has the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha + (r + \gamma Q(s\prime, a\prime) - Q(s,a))$$

Hyper Parameters

The Hyper-Parameters used are:

- γ : Discount Rate = 0.9
- α : Learning Rate = 0.01
- ϵ : Epsilon (greedy) = 0.1
- λ : Decay Rate = 0.9

A small learning rate was used to avoid large updates to the Q table.

A small ϵ was used to provide greater weightings to choose greedy actions.

A large discount rate γ was used to give the agent a larger (forward) horizon for rewards.

A large lambda rate λ was used to give smaller decay for the state-action pairs in the eligibility vector.

Policy Gradient

Description

The following pseudo code was used to implement policy gradient reinforce.

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)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Policy Gradient Reinforce Pseudo Code

States

For policy gradient, the one-hot encoding of state-actions was used. Therefore the features were represented as an array of size 400. At any given state only one element is 1, with the rest being 0's. The 400 array elements correspond to state-action pairs.

Actions

Actions are chosen based on the soft-max function. The probability distribution of each action is described by the policy

$$\pi(A_t|S_t) = \frac{e^{\theta \cdot x(S_t, A_t)}}{\sum_a e^{\theta \cdot x(S_t, a)}}$$

Mathematical Formulation

$$\nabla ln \pi(A_t|S_t, \theta) = \theta - \sum_b \pi(b|S_t) * \theta$$

Where $x(S_t, A_t)$ are the features given a state $S = S_t$ and action $A = A_t$.

Hyper Parameters

The Hyper-Parameters used are:

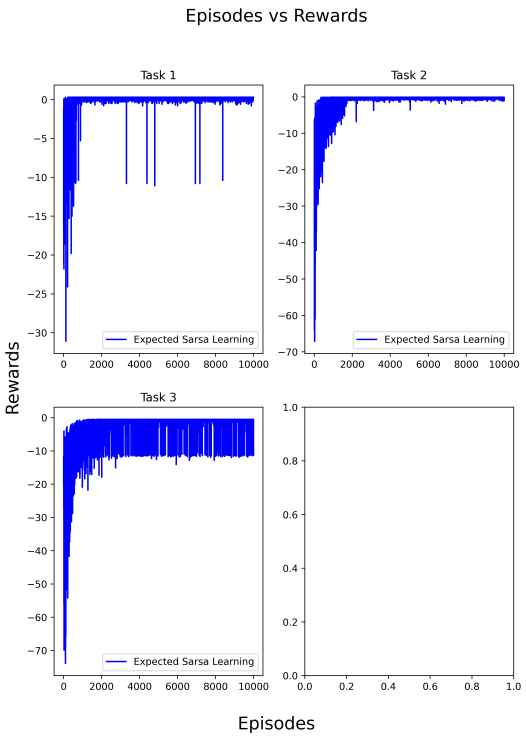
- γ : Discount Rate = 0.99
- α : Learning Rate = 0.015
- λ : Decay Rate = 0.9

Quantitative Analysis

Some quantitative analysis of the results, a default plot for comparing all algorithms is given. You can do more plots than this.

Expected Sarsa

Expected sarsa was able to reach convergence with tasks 1 and 2. However, the variance for task 3 is very high, indicating that it is not as stable.



The rate of convergence is quite fast, converging to acceptable values in less 3000 iterations for tasks 1 and 2. Task 3 may need more episodes for better convergence.

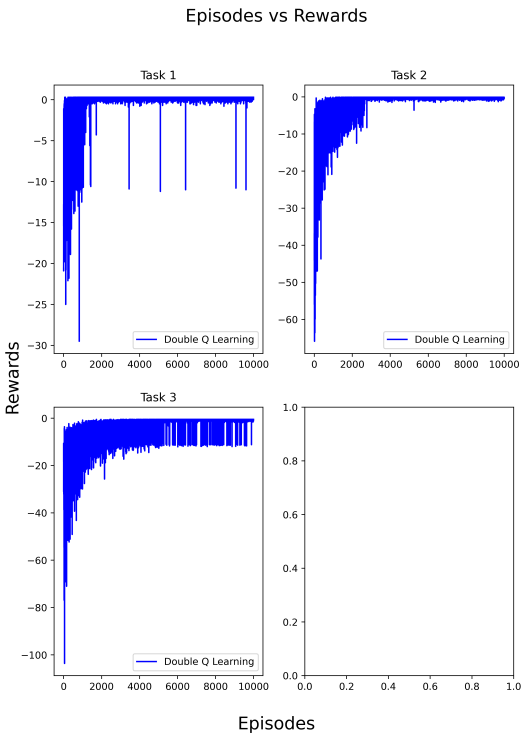
Expected Sarsa had the most issues with task 3, as it keeps bouncing between the goal and the first pit (or a pit near the goal).

Expected Sarsa starts off with higher rewards than the other algorithms and the range of rewards in less than 1000 episodes.

```
Task 1, Expected Sarsa Learning :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.015515999999999997
Task 2, Expected Sarsa Learning :
  max reward = -0.09999999999999987
  medLast100=-0.30000000000000004
  varLast100=0.0557390000000001
Task 3, Expected Sarsa Learning :
  max reward = -0.5000000000000002
  medLast100=-0.7000000000000004
  varLast100=5.360730999999999
```

Double Q Learning

Similar double Q-learning was able to reach convergence with tasks 1 and 2. Looking at the variance it can be seen that in the last episode double Q-learning has a small variance, even though the graph shows a thick band



between -10 and -0.5. This indicates that it still reaches the goal more often than not (at least at the end of the last episode.).

The rate of convergence is quite fast, converging to acceptable values in less 3000 iterations for tasks 1 and 2. Task 3 may need more episodes for better convergence.

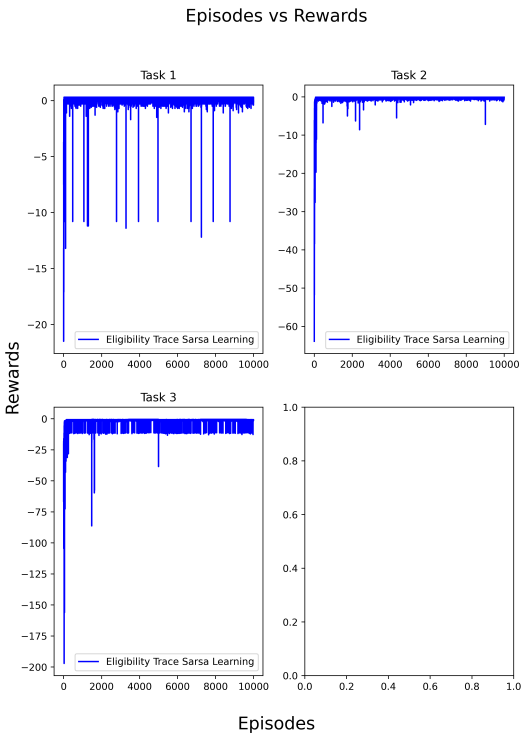
Double Q had the most issues with task 3, as it keeps bouncing between the goal and the first pit (or a pit near the goal).

Double Q starts off with very low rewards, but quickly reduces the range of rewards in less then 1000 episodes.

```
Task 0, Double Q Learning :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.012770999999999998
Task 1, Double Q Learning :
  max reward = -0.09999999999999987
  medLast100=-0.1499999999999999
  varLast100=0.033036000000000044
Task 2, Double Q Learning :
  max reward = -0.50000000000000002
  medLast100=-0.70000000000000004
  varLast100=0.09044400000000012
```

Eligibility Trace (Sarsa (λ))

Eligibility trace Sarsa (λ) reaches convergence for tasks 1 and 2 with the ideal values and small variance. In task 3 there is some significant variance but it reaches the ideal reward.



The speed of convergence was very abrupt at the beginning of the algorithm for all 3 tasks.

Sarsa (λ) had the most trouble with task 3. The wide horizontal line seen on the graph is the algorithm struggling with rewards between -10 and -0.5. This indicates it was either falling into the nearest pit or reaching the goal. This contributes to the high variance.

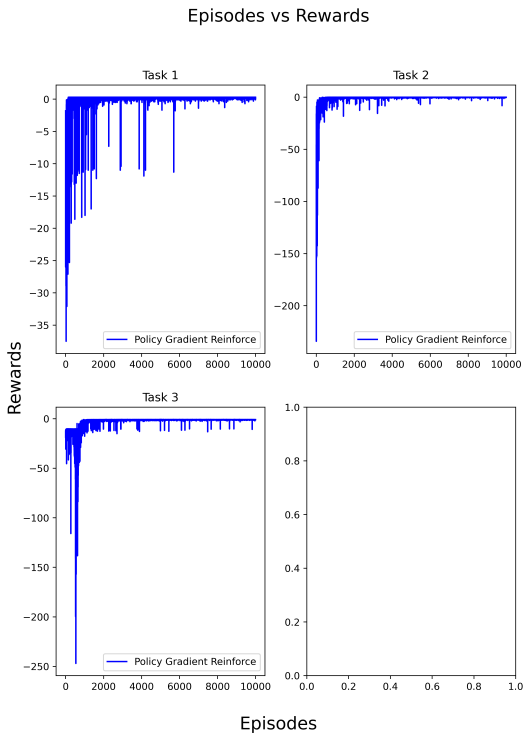
Sarsa(λ) starts off with very low rewards, but quickly reduces the range of rewards in less then 1000 episodes.

```
Task 1, Eligibility Trace Sarsa Learning :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.029223999999999997
Task 2, Eligibility Trace Sarsa Learning :
  max reward = -0.09999999999999987
  medLast100=-0.19999999999999996
  varLast100=0.056275000000000096
Task 3, Eligibility Trace Sarsa Learning :
  max reward = -0.5000000000000002
  medLast100=-0.9000000000000006
  varLast100=4.513074999999999
```

Policy Gradient

Policy gradient reached convergence for all 3 tasks with minimal variance.

Policy gradient reinforce seems to converge to values < 10 in between 1000 to 2000 episodes.



Policy Gradient Reinforce

Policy gradient had the most trouble with tasks 1 and 3 as it initially attempts to converge to the nearest pit (as seen by clumps of values close to -10). Once the algorithm was able to explore past these pits, the rewards began to rise.

Policy gradient has a large range of negative values for rewards. This is because it is a monte-carlo algorithm, so it cannot learn within the same episode. In early episodes, if it does not reach a pit, it wanders with no clear direction, thus the huge negative rewards.

```
Task 0, Policy Gradient Reinforce :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.006216
Task 1, Policy Gradient Reinforce :
  max reward = -0.09999999999999987
  medLast100=-0.09999999999999987
  varLast100=0.012675000000000023
Task 2, Policy Gradient Reinforce :
  max reward = -0.7000000000000004
  medLast100=-0.9000000000000006
  varLast100=0.03592400000000003
```

Expected Sarsa vs Sarsa(λ) vs Double Q Learning vs Policy Gradient

Policy Gradient provided the best results when compared to Sarsa, Sarsa(λ) and Double Q Learning. It has the best variance in the last 100 iterations of the last episode and has the most stable convergence for **all** tasks. Most algorithms

converge ideally for tasks 1 and 2 but fail in task 3. Policy gradient with enough episodes converges for task 3 as well. A large number of episodes were used to test all algorithms accurately.

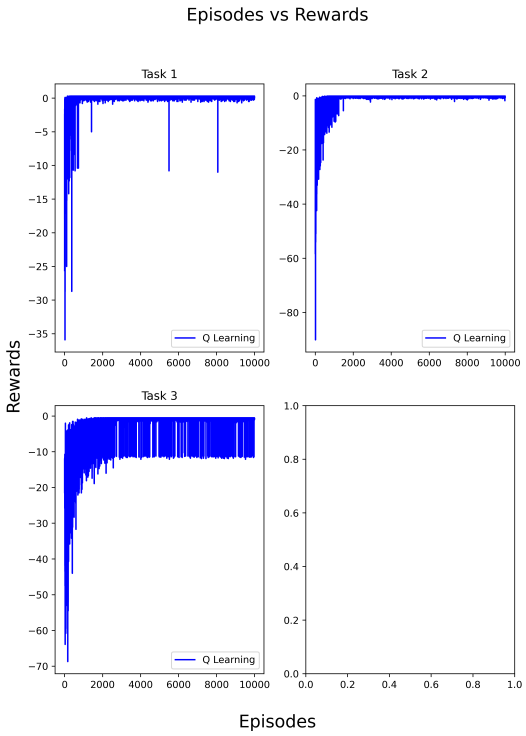
Of the other algorithms, Double Q learning did better then Sarsa(λ) and Expected Sarsa as it had a better variance for its last 100 iterations and had a similar performance for all other metrics.

Expected Sarsa starts off with higher rewards when compared to Sarsa(λ).

Comparison to Q Learning & Sarsa Learning

Compare the results of these algorithms to the others that you had implemented in Assignment 2. Give reasons for the observed comparative performances.

Sarsa Learning



Q Learning

```
Task 0, Sarsa Learning :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.036891000000000014
Task 1, Sarsa Learning :
  max reward = -0.09999999999999987
  medLast100=-0.30000000000000004
  varLast100=0.05205100000000009
Task 2, Sarsa Learning :
  max reward = -0.5000000000000002
  medLast100=-0.7000000000000004
  varLast100=4.439658999999999
```

```
Task 0, Q Learning :
  max reward = 0.30000000000000004
  medLast100=0.30000000000000004
  varLast100=0.021699999999999994
Task 1, Q Learning :
  max reward = -0.09999999999999987
  medLast100=-0.30000000000000004
  varLast100=0.07138400000000013
Task 2, Q Learning :
  max reward = -0.5000000000000002
  medLast100=-0.7000000000000004
  varLast100=3.3708360000000006
```

Compared to plain Sarsa and Q learning, double Q learning and Expected Sarsa has no immediately obvious differences. However, of all these algorithms, in the last 100 iterations of the last episode, double Q learning had significantly better variance while still reaching acceptable values. Sarsa and Q learning share a property with Expected Sarsa in that they start off with higher initial values. All of these algorithms have a similar rate of convergence and reduce the range of reward values quickly.

Compared to policy gradient, policy gradient performed much better than Sarsa and Q learning. Policy gradient has the smallest variance as well as a similar convergence rate as the other two algorithms.

Qualitative Analysis

💡 Some qualitative analysis of you observations where one algorithm works well in each case, what you noticed along the way, explain the differences in performance related to the algorithms.

Expected Sarsa

Expected Sarsa relied on a good use for the policy. Initially, a policy was used where each action had a probability of $\frac{Q(S_t,A_t)}{\sum_a Q(S_t,a)}$, however, this did not lead to good results (therefore a different method was used, as described in the appropriate section above). Expected Sarsa did not offer any benefits over regular Sarsa.

Eligibility Trace (Sarsa(λ))

Eligibility trace for Sarsa(λ) performed similarly to Expected Sarsa and regular Sarsa. Eligibility trace did not offer any immediate benefit compared to regular Sarsa in these tests.

Double Q Learning

Double Q learning was the easiest algorithm to implement, and provides enough benefits to use this opposed to Q learning even though it uses double the memory (via the second Q table).

Policy Gradient

Policy gradient was the hardest to implement of the 4 algorithms. Two approaches where used to implement policy gradient, neural nets and linear. The neural nets approach never reached convergence therefore linear had to be used. A variety of methods where used for the hyper parameter selection, including a dynamic approach that changed values at the end of each episode (to promote exploration early and exploitation in the end). This approach marginally improved task 3 as it allowed the agent to explore outside the pits near the start location, but was a detriment for tasks 1 and 2 as it explored too much in those cases, therefore it was not used.

Additionally, without the use of an ϵ, this algorithm provides better long term performance as there are less large spikes caused by random bad actions being taken.

Additionally, this implementation of policy gradient uses the same amount of memory as all the other algorithms. This is due to the fact that the size of the θ is the same size as the Q table for other algorithms.