Learning Project On RL

Mannem Charan AI21BTECH11019

Abstract

This report consists of my understanding on RL and different algorithms used to solve the RL problem and their implementation using Frozen-lake environment from OpenAI gym.

Contents

1 Introduction

Reinforcement Learning,is one way of machine learning where we try to design a *learner* which can do the task at hand after learning. To understand it, first we will compare RL with other ways of learning,

1) **Supervised Learning :** Here the model or learner *learns* how to do the task under some *supervision*.In other words, the model get to know the tags/*labels* of the samples, using which it will try to predict the labels of the unseen data.

Examples[1] : Linear Regression,SVM,etc

So the key things are,

- The model or *agent* gets immediate labels or *rewards*, i.e., there is no delay in rewards.

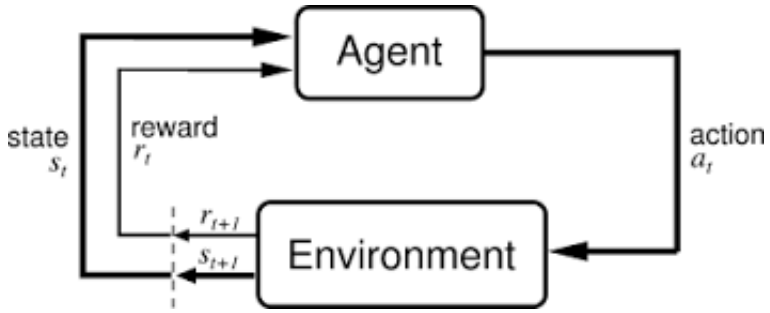


Fig. 1.1: Agent and Environment

- There is no exploration since agent learns from experience via training data.

2) **Un Supervised Learning :** Here the model learns how to do the task without any *supervision*, that means here in training data we will not provide the labels.

   Examples : K-means Clustering,DBSCAN,etc

   Here the key points are,
   - The model will not get any labels or rewards but it tries to learn the underlying pattern from the data(experience) given.
   - There is no exploration since the experience that model gains only depends on training data.

3) **Reinforcement Learning :** In Reinforcement Learning, we often use the term *agent* for the learner and the place where the agent *learns* is known as *environment*.Here agent learns about the environment by making decisions i.e., decisions impart what agent learns.

   So here the task is to make the agent learn the "good sequence of decisions" in the environment. And depending upon the decisions agent make, it will get reward and has some delayed consequences.So RL also takes future outcomes into consideration and tries to learn a good sequence of decisions which maximizes the total reward.

   The key points of RL which makes it differ from others are ,
   - The agent will get rewards as it keeps learning i.e., there is no fixed rewards it depends on what decisions the agent make.
   - There will be exploration since agent learns about the world by interacting/-exploring the world.

4) **Terminology used in RL :** The following figure 1.1 depicts how the agent interacts with environment.

   a) Action (*A*) : All possible moves that agent can make on environment.Agent interact with environment with *actions*.

b) State (S) : State is the current position of agent returned by environment.
c) Reward (R) : The immediate return the agent gets from the environment to evaluate the last action.
d) Policy (π) :As the agent gains experience by making decisions/actions,it tries to design a map between the past experiences to the decisons or *actions* to take.This is known as *policy* (π). The goal of an agent is to find best or *optimal*[2] policy for a given environment.
e) Value (V) : Contrast to reward, it is defined as the expected long-term return with discount (γ) [3]when the agent is in certain state s under a policy π. Mathematically we can write it as [4],

$$V^{\pi}(s) = E\left[R_{t+1} + \gamma V(S_{t+1}) | S_t = s\right] \tag{1.1}$$

Here t refers to the t^{th} timestep.

With this terms in hand, we will now see explicitly how agent interacts with environment using OpenAI gym library.

2 Environments using OpenAI Gym

Agent interact with the environment and learn from the experience.But often times it is hard to build the environment by hand when we begin learning the RL.The reward system should be designed carefully which helps the agent to understand what is the best way to make actions.
Same as Imagenet, which made itself a benchmark for testing different ML techniques performance on the same training - testing data, there is something known as Gym which is used to test different RL algorithms.

In my project, I used "Frozen-Lake Environment" a toy-text environment taken from Gym - a python library. Here is the code to be runned in terminal to get the gym library,

```
pip install gym==0.21.0
# New version has some bugs
```

Now after that, we can get our Frozen lake env with the following piece of code,

```
import gym
```

[2]To judge whether a policy is best or not we will evaluate the *value* for the policy.Optimal policy is the one with maximum value.

[3]The factor used for penalising the future rewards.It is used to bound the *value* of a state (mathematical application) but also have some importance in psychological sense.

[4]This equation is also known as Bellman Equation which is the building block of RL

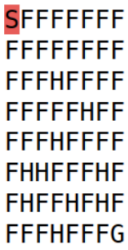


Fig. 2.2: Initial State of Frozen Lake Environment

```
env = gym.make("FrozenLake-v1",desc= None,map_name = "8x8",
    is_slippery = False)
env.reset() # Resets to current state.
env.render() # Used to show the current state of env.
#  S -> Start
#  F -> Frozen Region
#  H -> Hole
#  G -> Goal
# Game starts at S and stops at H or G.And target is to
    reach G .
```

The environment in the current state looks like 2.2, As the name itself self explanatory, the player will start at S and will move on Frozen regions F to reach goal G with minimum no.of steps without falling into the holes H.And the reward system looks like [5]

```
Frozen (F) -> -0.01
Hole (H) -> -0.2
Goal (G) -> 1.0
```

The *is_slippery* attribute is used to make the frozen regions slippier so setting it to True or False changes the env according to that.I took both versions of environment and used different RL algorithms to obtain the optimal path.

3 Frozen Lake

In this section, we will take default Frozen Lake environment without any slipperiness.To solve that, I used "**Q learning**" and "**SARSA**" algorithms to solve this RL problem.So we will see both of them one by one,

[5]It is different from the standard reward system mentioned in documentation, I changed that to visualise the performance of agents better

3.1 Solving using Q learning

Similar to the value, Q-value is defined as the expected long-term reward with discount at given state s and action a under some policy π.

$$Q^{\pi}(s,a) = E\left[R_t + \gamma Q^{\pi}(s',a') | S_t = s, A_t = a\right] \tag{3.1}$$

In Q-learning, we will learn the Q-value for each state-action pair in iterative manner with optimal policy being greedy.So we will construct a state-action table where each pair has a Q-value as shown below,

```
State_space = env.observation_space
Action_space = env.action_space
Q = np.zeros((State_space.n,Action_space.n)) # Initialising
    with zeros.
```

Then to interact with environment (To explore), we need a policy and I used ϵ - greedy algorithm to interact with the environment with ϵ being the probability to explore.You can see the code snippet below[6]

```
def beh_policy(env,state,Q_table,ϵ):
 # explores with epsilon prob
 if np.random.uniform() -ϵ < 1e-6:
      return env.action_space.sample()
 else :
 # exploits with 1 - epsilon prob
      return np.argmax(Q_table[state])
```

And as per Q learning we will update the Q value of each state-action pair (s,a) by following step,

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \eta\left[(r_t + \gamma max_a Q(s_{t+1},a)) - Q(s_t,a_t)\right] \tag{3.2}$$

With η being learning rate and γ is discount factor.
We can justify (3.2) as, we are updating the old Q value with the *difference* of new Q-value [7] and old Q-value under a learning rate η. As you can see it uses different policies to interact with env and to learn optimal policy.That's why Q learning algorithm also known as *Off policy*.
It is proven that, using Q learning algorithm with sufficient amount of training, one can find the *ideal* next state for each state possible for the agent in the environment.So with that confidence I trained my agent for a *total_eps* = 5000 using the

[6]The font used here is different in order to encode the unicode characters used here.
[7]Q value obtained using greedy policy in the new state

Q_learning function (You can see the in FrozenLake.ipynb).
Note : In the code, I decreased ϵ as the agent trains because,in start I gave preference to *explore* the environment,doing this helps agent to know about environment (To know where the holes/frozen regions are) and as it gained experience I allowed the agent to *exploit* it's experience.

I used the following values for the hyperparameters and changing each value has its own kind of effect in the learning process. Like if I decrease the ϵ or increase the ϵ_{min} it results in low performance of agent in training stage as it spends most of the time exploring.

```
η = 0.3 # Learning rate.

γ = 0.8 # Discount factor

ϵ = 0.9 # Prob to explore
ϵ_min = 0.1 # minimum epsilon as it decays.
π_Q,epi_list_Q,avg_time_step_Q,avg_rew_Q = Q_learning(env,η,γ,ϵ,ϵ_min,total_eps)
```

As you can see the *Q_learning* returns,

- $\pi_Q \rightarrow$ The optimal policy designed by Q learning.
- *epi_list_Q* $\rightarrow$ The list of episodes in which the agent reached the goal.
- *avg_time_step_Q* $\rightarrow$ The average time steps taken by the agent to reach the goal as it trains.
- *avg_rew_Q* $\rightarrow$ The average reward recieved by the agent as it trains.

Using these objects, we can see the performance of the agent in the training phase.

3.2 Performance of Agent using Q learning

1) **Success Rate :** Using *epi_list_Q* we can determine no. of times agent reached the goal.

```
print("Total no.of times agent reached the Goal in
   Training Phase {}".format(len(epi_list_Q)))
```

```
Output -> Total no.of times agent reached the Goal in
   Training Phase 3896
```

In general this number twiddled around 3600 to 4000 which gives as the percentage of success being around 72 to 80.

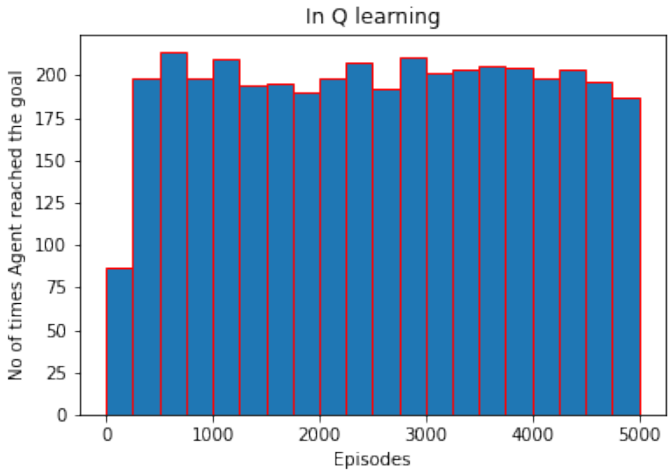


Fig. 3.3

And the distribution of no.of times agent reached the goal as agent trained can be seen in fig 3.3,

2) **Average time steps taken :** You can see the Fig 3.4, Initially agent took

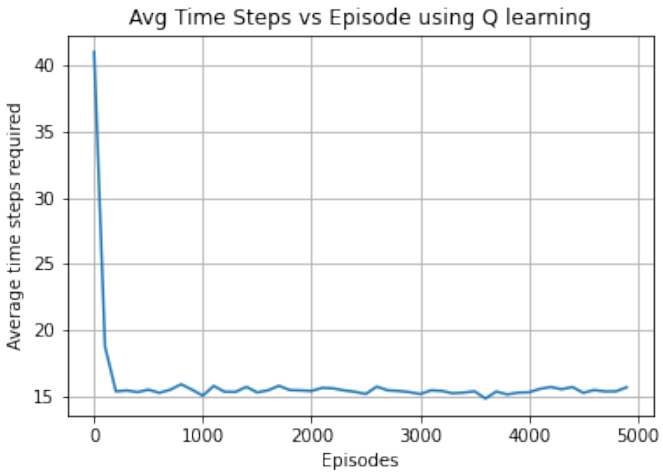


Fig. 3.4

more no.of time steps to reach the goal but after few hundred of episodes the average no.of time steps went down to around 15. And it became closer to 15

Fig. 3.5

as it trained.[8]

3) **Average reward gained :** You can see the fig 3.5
From the figure we can interpret that, initially the agent recieved negative rewards most of the times.But as the training goes on it recieved positive rewards resulting in positive average rewards.In the end,it got an average reward of little over 0.6 and less than 0.7[9].
4) **Testing the optimal policy designed by Q learning :** Since the env is not slippery, the actions are not stochastic.This implies that,optimal policy should be able to reach the goal with minimum no.of steps.I used *print_frames* function to see the animation of agent reaching the goal with its optimal policy.

```
from IPython.display import clear_output
from time import sleep

def print_frames(frames):
 for i, frame in enumerate(frames):
    clear_output(wait=True)
    print(frame['frame'])
    print(f"Timestep: {i + 1}")
```

[8]Eventhough the minimum no.of steps required to reach the goal is 14 since we are taking average of time steps for every $total_eps/50$ successfull episodes the value settled at 15.

[9]Eventhough the maximum possible reward that agent can gain is $13x - 0.01 + 1 = 0.87$ the max average is below 0.8 since we are taking average of rewards recieved by agent

```
    print(f"State: {frame['state']}")
    print(f"Action: {frame['action']}")
    print(f"Reward: {frame['reward']}")
  sleep(.2)
```

I did one test run using the policy π_Q because it is enough as actions are not stochastic.You can see the animation in the FrozenLake.ipynb file.
The agent successfully learned the optimal path by taking only 14 steps to reach the goal,also received a maximum reward of 0.87.

3.3 Solving using SARSA

State-Action-Reward-State-Action ($SARSA$) is an RL algorithm used to find the optimal policy for an agent traverse in the world. It is quite similar to **Q learning** other than the fact that SARSA is an *on-policy*. It means that, it uses same policy to interact with environment as well as to estimate the optimal policy. So in place of the (3.2) we will use the following update step,

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\left[(r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)\right] \tag{3.3}$$

where the s_t and a_t are state and action taken using the policy at t^{th} time step and s_{t+1} is the next new state,a_{t+1} is the next action to be taken using the same policy. Due to this sequential nature it is named as SARSA.
Due to this difference in update step, the **performance** will vary in both algorithms.I will discuss the difference later but I implemented the SARSA algo using SARSA function (You can view the code in FrozenLake.ipynb file).
And *SARSA* function returns the same objects as *Q_learning* function.I trained the agent using *SARSA* for 5000 episodes (same as in Q learning) and I took the same values for the $\eta, \gamma, \cdots$ as I took for *Q_learning*.

```
total_eps = 5000 # Total no. of episodes to be trained

η = 0.3 # Learning rate.

γ = 0.8 # Discount factor

ϵ = 0.9 # Prob to explore
ϵ_min = 0.1 # minimum epsilon as it decays.

π_sarsa,epi_list_sarsa,avg_time_step_sarsa,avg_rew_sarsa = SARSA(env,η,γ,ϵ,ϵ_min,total_eps)
```

Similar to what we did in Q learning, we will understand how SARSA performed in training phase.

3.4 Performance of agent using SARSA

1) **Success Rate :** Using *epi_list_sarsa* we can determine no. of times agent reached the goal.

```
print("Total no.of times agent reached the Goal in
   Training Phase {}".format(len(epi_list_sarsa)))
```

```
Output -> Total no.of times agent reached the Goal in
   Training Phase 4600
```

In general this number twiddled around 4550 to 4650 which gives as the percentage of success being around 91 to 93.Whereas in case of Q learning, this is around 72 to 80.
So clearly SARSA **outperformed** Q learning in terms of success rate in training.
And the distribution of no.of times agent reached the goal as agent trained can be seen in fig 3.6, Here also in each *bin*, the no .of episodes in which the agent

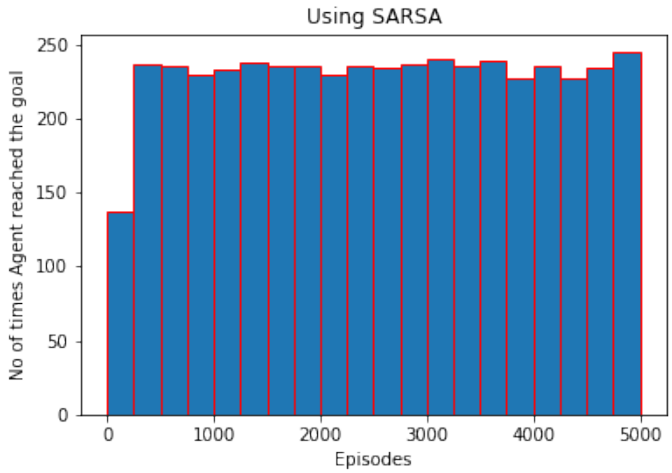


Fig. 3.6

reached the goal is higher compared to Q learning (compare with 3.3).

2) **Average time steps taken :** You can see the plot in Fig 3.7,
As you can see initially agent took more no.of time steps to reach the goal but after few hundred of episodes the average no.of time steps went down to around 15.And it became closer to 15 as it trained.
Compared to Q learning, even though the behaviour is same as it trains, initially SARSA took way more no.of steps to reach the goal compare to Q learning.

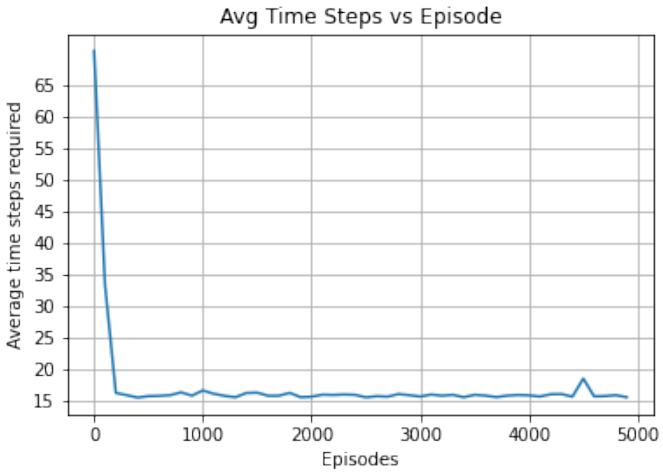


Fig. 3.7

3) **Average reward gained :** You can see the plot in fig 3.8

Fig. 3.8

From the figure we can interpret that, initially the agent recieved negative rewards most of the times.But as the training goes on it recieved more positive rewards resulting in positive average rewards.
In the end,it got an average reward close to 0.87,which is the maximum possible reward.So in comparison with Q learning, SARSA has more negative avg

rewards in early stages but as it trained it had **best** average reward compared to Q learning.

4) **Testing the optimal policy designed by SARSA :** Using the optimal policy π_{sarsa} generated by SARSA, I did one test run (You can see the animation in FrozenLake.ipynb). And the agent reached the goal in minimum no.of steps of 14. So even though the **paths** taken by Q learning and SARSA are different they both reached the goal in same number of steps.

3.5 Q learning vs SARSA using custom map

So I wanted to know more about how the Q learning and SARSA **chooses** their paths,for which I changed the initial state of agent by giving custom map.You can do that by manually giving the initial map as shown below,

```
# Custom Env
env_cus = ["SFFFFFFF",
           "FFHHHHHF",
           "FFHFHFHF",
           "FFFHFFHF",
           "FFHFHFHF",
           "FFHFFHHF",
           "FHFFFFHF",
           "FFFHFFFG"]
env2 = gym.make("FrozenLake-v1",desc = env_cus,is_slippery
   = False)
env2.reset()
env2.render()
```

The initial state looks like fig 3.9

Reason Behind using this custom env : Since Q learning and SARSA are differ in their update step.I made this map, to see how Q learning and SARSA "choosing" their paths.

It is a **fact** that Q learning learns its *optimal* path in less no.of steps. The reason behind this maybe because in its *greediness* of choosing maximum Q value in the new state. So exploiting,in a way,is in the nature of Q learning. This tend to make the learning fast (taking less no.of time steps) after sufficient exploration. Whereas SARSA learns its *optimal* path which is more *safer* to reach the goal. Since the action selection is in it's nature,the SARSA tries to choose a path for agent which is more safer to reach the goal. So compare to Q learning it kinda explores more,which makes it to choose a safer path eventhough its a longer path (sometimes).

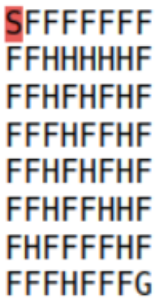


Fig. 3.9

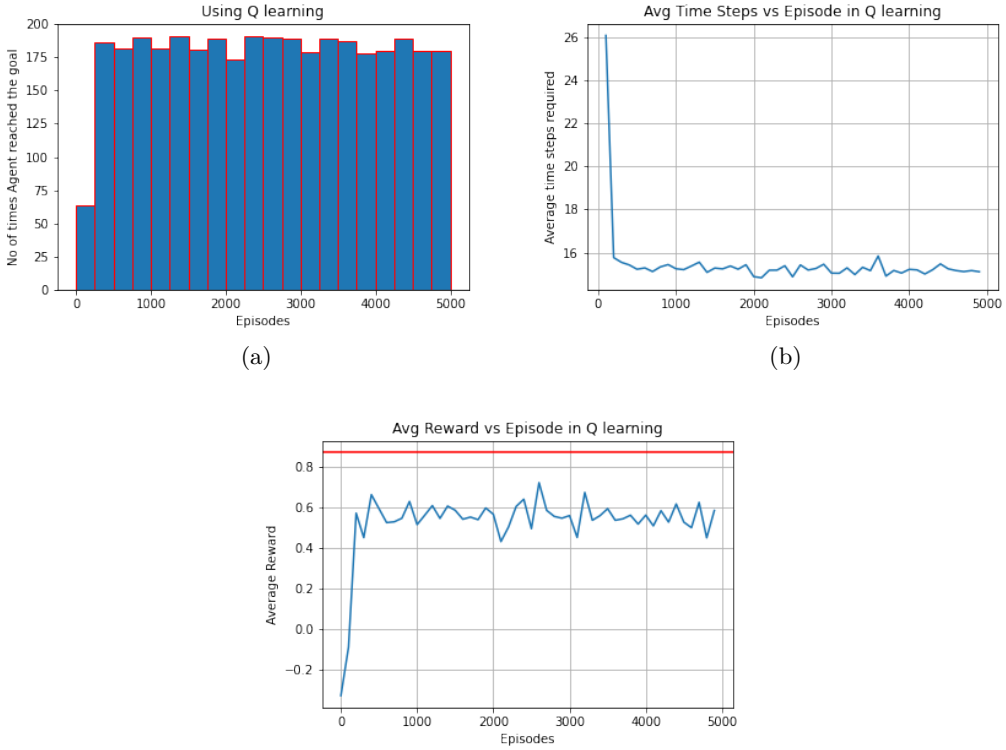
So in a way, Q learning tries to complete the task as soon as possible without thinking about safety of agent,whereas SARSA tries to choose a safe path to agent even though it costs more time to reach the goal.And depending on the application we should choose the appropriate algo.
So to *check* these facts I designed this 3.9 map which has

- A path of 14 steps by directly going 7 Right (→) steps and then by going 7 Down (↓) steps to reach Goal (G).As you can see this path is surrounded by Holes (H),so it can be considered as less safe path.
- A path of 16 steps by directly going 7 Down (↓) steps then 2 Right (→) steps, 1 Up (↑) step and going 2 Right (→) steps 1 Down (↓) step then going 2 Right (↓) steps. Compared to previous path it surrounded by less Holes (H), so it can be considered as safe path though needing more steps to reach the goal.

Observations after training:

1) I trained the agent using the previous *Q_learning* function for the same *total_eps* of 5000 using the same values of $\eta, \gamma, \cdots$. And the agent it trained reached the goal in just 14 steps as we intended it to do. The performance in training phase can be seen below [10],
2) Then I trained the agent using *SARSA* for 5000 episodes but I failed to see any results. The agent actually not moving from the Start (S),like it choosing Up (↑) or Left (←) as it's best actions.This behaviour can be reasoned as *agent is fearing to move from Start position* i.e., it is thinking that staying there itself is the best course of action.So to make the agent reach the goal, I can do two things,

[10]You can refer FrozenLake.ipynb file for the plots



(a) (b)

(c)

Fig. 3.10: Performance of agent trained with Q learning

- Increasing the no. of episodes to be trained which may help the agent to realise that there is a *safer* path.
- By changing the ϵ , ϵ_{min} values we can restrict the agent to exploit or explore more.

3) If I did the earlier way (Call it SARSA-1), the agent reached the goal in 16 steps everytime (PS - I actually thought it will never happen) after using a *total_eps* of >= 35000 [11].
Here I did not changed the values of ϵ or ϵ_{min} so that, we can see that for the same *degree of exploration* SARSA took around 35000 eps but Q learning took only 5000 eps to reach goal.You can see the plots for the perfomance of SARSA below,

[11]I tried different number of total eps but mostly the agent started reaching the goal using 35000 but then again I can't say this is the exact number

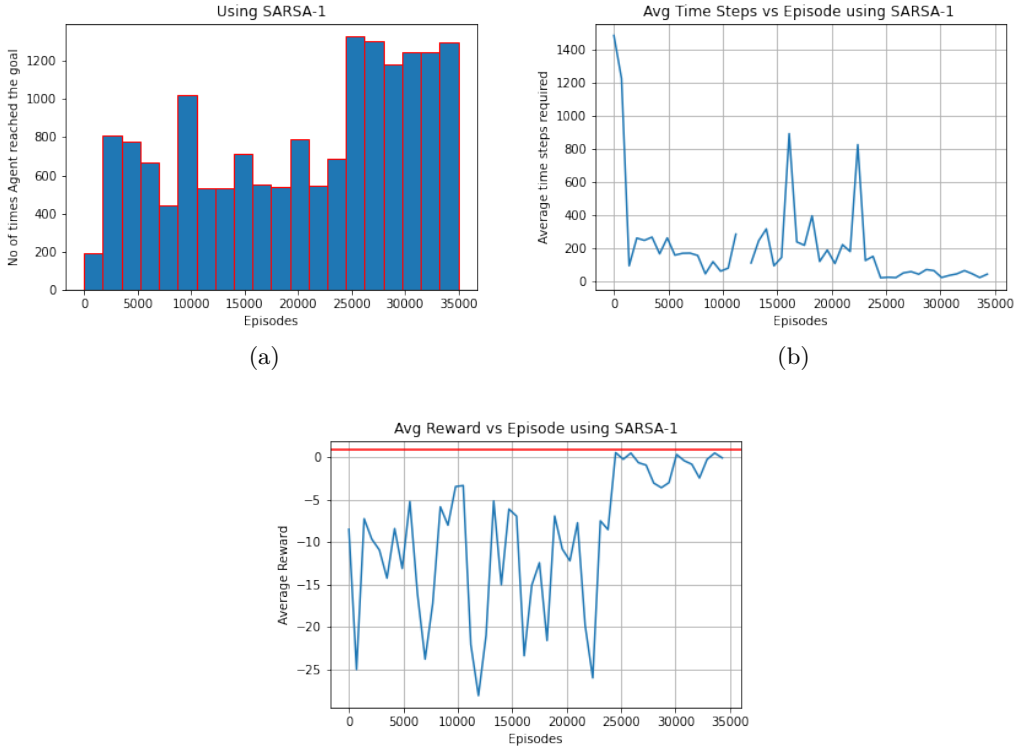
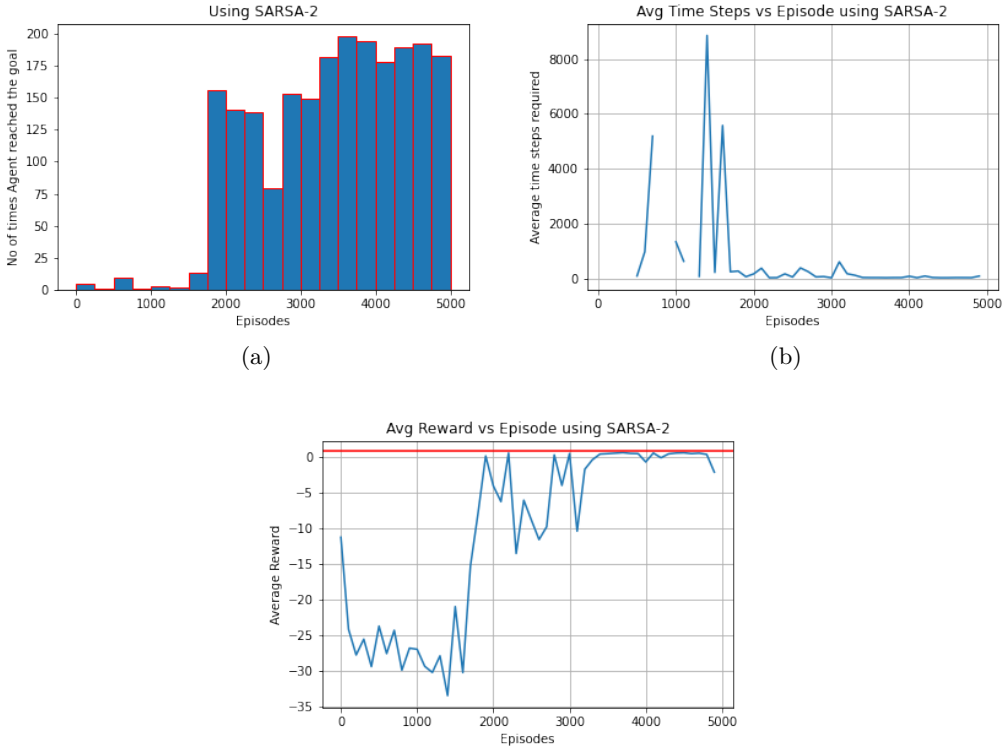


Fig. 3.11: Performance of agent trained with SARSA-1

4) If I did the later way (SARSA-2), then suprisingly if I make $\epsilon = \epsilon_{min} = 0.1$ the agent reached the goal in 16 steps. I never saw this coming.And the performance in the training phase is irregular,i.e., sometimes it has good success rate in training phase and sometimes may not.But most of the time the end policy obtained can be used to reach the goal in 16 steps.It is not a good way to learn the path since most of the time we are exploiting instead of exploring the environment. I can't think of any specific reason but it maybe because it does not know that there is a path of 14 steps.With more degree of exploitation, the agent actions are more *sensitive* to bad past experiences. So may be moving in Right (→) from Start position must have gave agent bad experience which made it never move in that direction.You can see the plots to understand the performance of agent below,
5) Nonetheless both algos followed the intended behaviour when *choosing* their optimal path.



(a) (b)

(c)

Fig. 3.12: Performance of agent trained with SARSA-2

Closing comments in Frozen Lake Env : I used Q learning and SARSA algorithm to devise the optimal policy for the Frozen Lake Environment.Then I understood the difference of algorithms using a custom environment which is designed to know which *optimal* path they took to reach the goal.

4 Frozen Lake with Slipperiness

In this section, I have taken the Frozen lake environment by setting *is_slippery* = ***True***. So according to official documentation,the agent will move in intended direction with probability of $\frac{1}{3}$ else will move in either perpendicular direction with equal probability of $\frac{1}{3}$ in both directions. That is ,

```
For example, if action is left and is_slippery is True,
    then:
- P(move left)=1/3
```

- P(move up)=1/3
- P(move down)=1/3

So since, the actions are stochastic, one cannot find a *deterministic* policy using which the agent can reach the goal everytime.So the *optimal* policy here will be the one which has good success rate while testing the agent.
To design the optimal policy I used the previously known Q learning and new algo known as *Value Iteration*.
Value Iteration : In value iteration, we will design the policy using the transition model of the environment.We will sweep through all the states and greedify them with respect to the current value function[12].After that we will design the policy by finding the best action to from the given which best maximises the value. You can see the below figure to understand better,

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
  $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

Figure 4.5: Value iteration.

Fig. 4.13

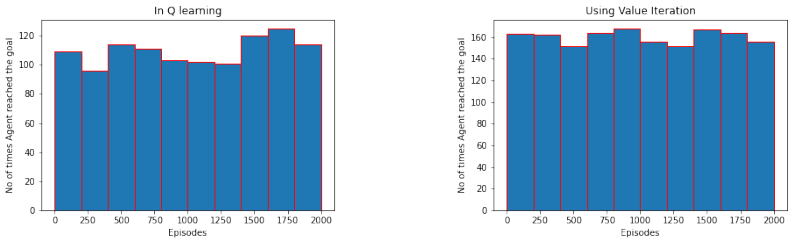
4.1 Perfomance of Q learning and Value Iteration

I trained one agent using Q learning for 5000 episodes and other agent using Value Iteration. Since actions are stocchastic, I tested both the agents for test_eps using *test_ agent* function [13]. You can see the difference in their performance with the plots below,

[12]Value function is a function between state and it's *value* but it generaly used as array with indexes being states

[13]You can refer the code

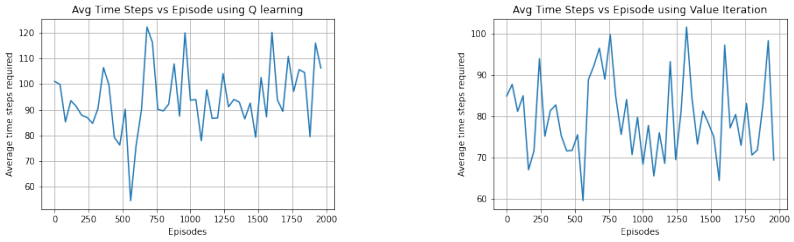
1) Agent reaching the goal : As you can see in Fig 4.14,agent trained with value iteration reached the goal more *frequently* compared to agent trained with Q learning.



(a) (b)

Fig. 4.14: Q learning vs Value Iteration in success rate.

2) Average time taken : As you can see in Fig 4.15, the agent trained with value iteration in general took lesser [14] no.of steps than Q learning.



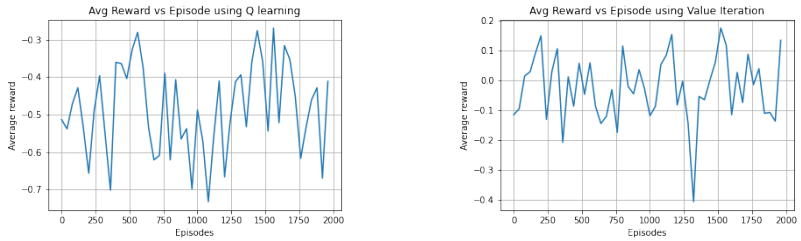
(a) (b)

Fig. 4.15: Q learning vs Value Iteration in average time taken

3) Average reward gained : As you can in Fig 4.16, most of the times agent trained by Q learning recieved negative rewards whereas in case of Value iteration the agent recieved more positive rewards.
So overall the agent trained by Value iteration gave better results than Q learning [15]

[14]Sometimes it happened that Q learning is on par with Value iteration but most of the times Value Iteration performed better than Q learning

[15]Changing the no.of episodes to be trained in case of Q learning might improve the performance but after verifying many times the performance of Value Iteration is always better or on par with Q learning



(a) (b)

Fig. 4.16: Q learning vs Value Iteration in average reward gained

5 Conclusion

- In this report,I used Frozen Lake Environment to train the agent using different RL algorithms to achieve the optimal policy.
- For the normal Frozen lake I used Q learning and SARSA algorithms to learn the optimal path. Both learned their own *optimal* path to reach the goal. So to understand the way the both algos choose their paths I used a custom map which helped to reveal that Q learning learns the path with minimum no.of time steps whereas SARSA learns *safer* path reach the goal.
- Then I took slippery version of Frozen Lake which motivated to use Value Iteration as it uses the transition model of the environment.I compared it with the old buddy Q learning by testing both algos for some no. of epsiodes. Then I observed that the agent trained with Value Iteration always performed better or on par with Q learning.