Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA in python

Vaibhav Kumar
Mar 20, 2019 · 9 min read ★

TD, SARSA, Q-Learning & Expected SARSA along with their python implementation and comparison

> If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. — Andrew Barto and Richard S. Sutton

Pre-requisites

- Basics of Reinforcement learning
- Markov chains, Markov Decision Process (MDPs)
- Bellman equation
- Value, policy functions and iterations

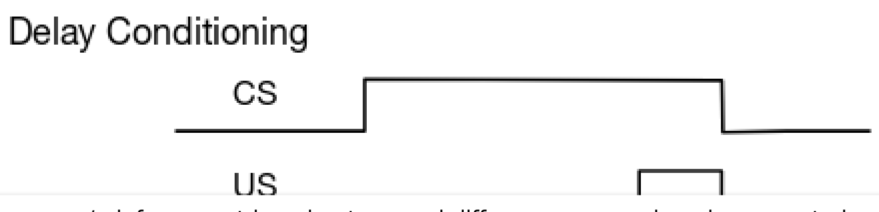
Some Psychology

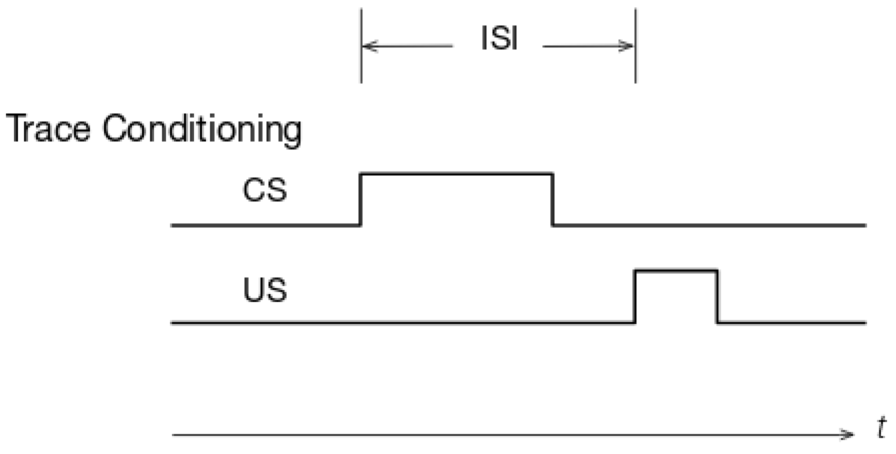
You may skip this section, it's optional and not a pre-requisite for the rest of the post.

I love studying artificial intelligence concepts while correlating them to psychology — Human behaviour and the brain. Reinforcement learning is no exception. Our topic of interest — Temporal difference was a term coined by Richard S. Sutton. This post is derived from his and Andrew Barto 's book — *An introduction to reinforcement learning* which can be found here. To understand the psychological aspects of temporal

Ivan Pavlov performed a series of experiments with dogs. A set of dogs were surgically modified so that their saliva could be measured. These dogs were presented with food (unconditioned stimulus — US) in response to which excretion of saliva was observed (unconditioned response — UR). This is stimulus-response pair is natural and thus conditioned. Now, another stimulus was added. Right before presenting the food a bell was rung. The sound of bell is a conditioned stimulus (CS). Because this CS was presented to the dog right before the US, after a while it was observed that the dog started salivating at the sound of the bell. This response was called the conditioned response (CR). Effectively, Pavolov was successful to make the dog salivate on the sound of bell. An amusing representation of this experiment was shown in the sitcom — The Office.

The time interval between the onset of CS and US is called Inter-Stimulus Interval (ISI) and is a very important characteristic of the Pavlovian conditioning. Based on ISI the whole experiment can be divided into types:





Source: Introduction to Reinforcement learning by Sutton and Barto — Chapter 14

As shown in the easily comprehensible diagram above, in Delay Conditioning the CS appears before the US as well as all the while along the US. It's like having the bell ring before and all along presenting the food. While in Trace Conditioning CS appears and is ceased before the occurrence of the US.

In the series of experiments, it was observed that a lower value of ISI showed a faster and more evident response (salivating of dog) while a longer ISI showed a weaker response. By this, we can conclude that to reinforce a stimulus-response pair the interval between the conditioned and unconditioned stimuli shall be less. This forms the basis of the Temporal Difference learning algorithm.

Model-dependent and model-free reinforcement learning

Model-dependent RL algorithms (namely value and policy iterations) work with the help of a transition table. A transition table can be thought of as a life hack book which has all the knowledge the agent needs to be successful in the world it exists in. Naturally, writing such a book is very tedious and impossible in most cases which is why model dependent learning algorithms have little practical use.

Temporal Difference is a model-free reinforcement learning algorithm. This means that the agent learns through actual experience rather than through a readily available all-knowing-hack-book (transition table). This enables us to introduce stochastic elements and large sequences of state-action pairs. The agent has no idea about the reward and transition systems. It does not know what will happen on taking an arbitrary action at

Temporal Difference Learning

Temporal Difference algorithms enable the agent to learn through every single action it takes. TD updates the knowledge of the agent on every timestep (action) rather than on every episode (reaching the goal or end state).

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\Big[\text{Target} - \text{OldEstimate}\Big]$$

The value ***Target-OldEstimate*** is called the target error. StepSize is usually denoted by **α** is also called the learning rate. Its value lies between 0 and 1.

The equation above helps us achieve ***Target*** by making updates at every timestep. Target is the utility of a state. Higher utility means a better state for the agent to transition into. For the sake of brevity of this post, I have assumed the readers know about the Bellman equation. According to it, the utility of a state is the expected value of the discounted reward as follows:

$$\text{Target} = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right]$$

In layman terms, we are letting an agent run free into a world. The agent has no knowledge of the state, the rewards and transitions. It interacts with the environment (make random or informed actions) and learns new estimates (values of state-action pairs) by updating it’s existing knowledge continuously after taking every action.

The discussion till now shall give rise to several questions such as — What is an environment? How will the agent interact with the environment? How will the agent choose actions i.e what action will the agent take in a particular state (policy)?

This is where SARSA and Q-Learning come in. These are the two control policies that will guide our agent in an environment and enable it to learn interesting things. But

Environment

An environment can be thought of as a mini-world where an agent can observe discrete states, take actions and observe rewards by taking those actions. Think of a video game as an environment and yourself as the agent. In the game Doom, you as an agent will observe the states (screen frames) and take actions (press keys like Forward, backward, jump, shoot etc) and observe rewards. Killing an enemy would yield you pleasure (utility) and a positive reward while moving ahead won't yield you much reward but you would still want to do that to get future rewards (find and then kill the enemy). Creating such environments can be tedious and hard (a team of 7 people worked for more than a year to develop Doom).

OpenAI gym comes to the rescue! gym is a python library that has several in-built environments on which you can test various reinforcement learning algorithms. It has established itself as an academic standard to share, analyze and compare results. Gym is very well documented and super easy to use. You must read the documents and familiarize yourself with it before proceeding further.

For novel applications of reinforcement learning, you will have to create your own environments. It's advised to always refer and write gym compatible environments and release them publicly so that everyone can use them. Reading the gym's source code will help you do that. It is tedious but fun!

SARSA

> SARSA is acronym for State-Action-Reward-State-Action

SARSA is an on-policy TD control method. A policy is a state-action pair tuple. In python, you can think of it as a dictionary with keys as the state and values as the action. Policy maps the action to be taken at each state. An on-policy control method chooses the action for each state during learning by following a certain policy (mostly the one it is evaluating itself, like in policy iteration). Our aim is to estimate ***Qπ(s, a)*** for the current policy π and all state-action ***(s-a)*** pairs. We do this using TD update rule applied at every timestep by letting the agent transition from one state-action pair to

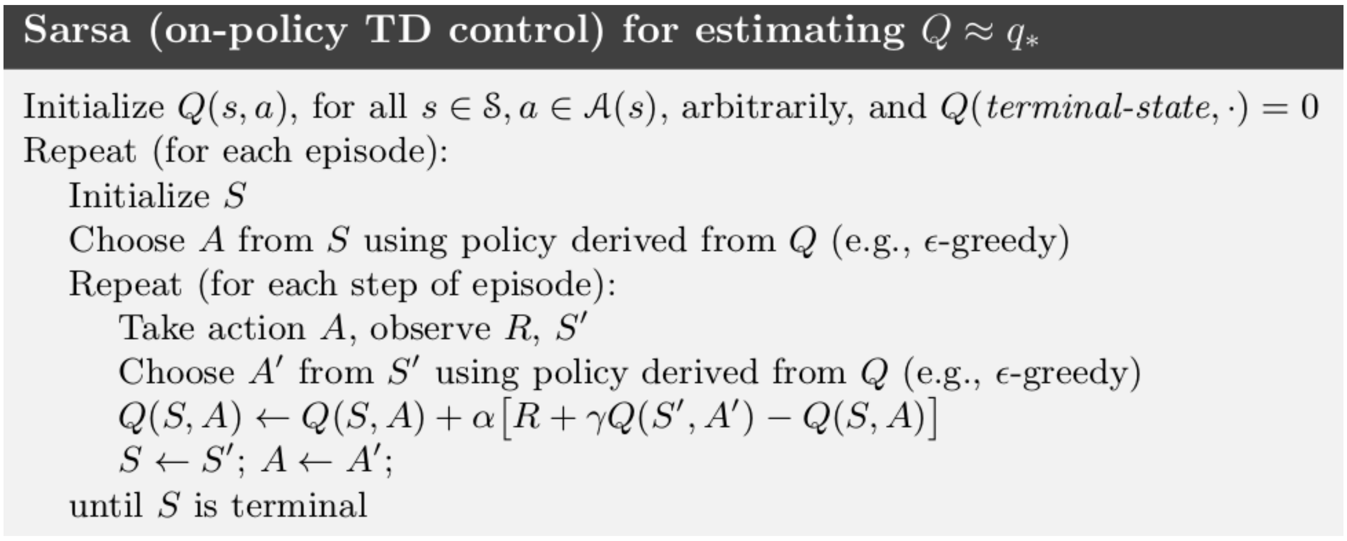
Q-value- You must be already familiar with the utility value of a state, Q-value is the same with the only difference of being defined over the state-action pair rather than just the state. It's a mapping between state-action pair and a real number denoting its utility. Q-learning and SARSA are both policy control methods which work on evaluating the optimal Q-value for all action-state pairs.

The update rule for SARSA is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

Source: Introduction to Reinforcement learning by Sutton and Barto — 6.7

If a state S is terminal (goal state or end state) then, ***Q(S, a) = 0*** ∀ ***a ∈A*** where ***A*** is the set of all possible actions



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

```
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Source: Introduction to Reinforcement learning by Sutton and Barto —Chapter 6

The action ***A'*** in the above algorithm is given by following the same policy (ε-greedy over the Q values) because SARSA is an on-policy method.

ε-greedy policy

Epsilon-greedy policy is this:

1. Generate a random number ***r ∈[0,1]***

3. Else choose a random action

It shall become more clear after reading the python code.

```python
def epsilon_greedy(Q, epsilon, n_actions, s, train=False):
    """
    @param Q Q values state x action -> value
    @param epsilon for exploration
    @param s number of states
    @param train if true then no random actions selected
    """
    if train or np.random.rand() < epsilon:
        action = np.argmax(Q[s, :])
    else:
        action = np.random.randint(0, n_actions)
    return action
```

epsilon_greedy.py hosted with ❤ by **GitHub** **view raw**

> ***Please Note***: *In the book, "RL by Sutton and Barto" a random action is taken if r>***ε.** *They have taken the value of* **ε***=0.1 while for me* **ε***=0.9. These two are basically the same thing.*

The value of ***ε determines the exploration-exploitation of the agent.***

If ***ε*** is large, the random number ***r*** will hardly ever be larger than ε and a random action will hardly ever be taken (less exploration, more exploitation)

If ***ε*** is small, the random number ***r*** will often be larger than ε which will cause the agent to choose more random actions. This stochastic characteristic will allow the agent to explore the environment even more.

As a rule of thumb, ***ε*** is usually chosen to be 0.9 but can be varied depending upon the type of environment. In some cases, ***ε*** is annealed over time to allow higher exploration followed by higher exploitation.

Here's a quick and simple python implementation of SARSA applied on the Taxi-v2 gym environment

```python
import gym
import numpy as np
```

```
SARSA on policy learning python implementation.
This is a python implementation of the SARSA algorithm in the Sutton and Barto's book
RL. It's called SARSA because - (state, action, reward, state, action). The only diffe
between SARSA and Qlearning is that SARSA takes the next action based on the current p
while qlearning takes the action with maximum utility of next state.
Using the simplest gym environment for brevity: https://gym.openai.com/envs/FrozenLake
"""


def init_q(s, a, type="ones"):
    """
    @param s the number of states
    @param a the number of actions
    @param type random, ones or zeros for the initialization
    """
    if type == "ones":
        return np.ones((s, a))
    elif type == "random":
        return np.random.random((s, a))
    elif type == "zeros":
        return np.zeros((s, a))


def epsilon_greedy(Q, epsilon, n_actions, s, train=False):
    """
    @param Q Q values state x action -> value
    @param epsilon for exploration
    @param s number of states
    @param train if true then no random actions selected
    """
    if train or np.random.rand() < epsilon:
        action = np.argmax(Q[s, :])
    else:
        action = np.random.randint(0, n_actions)
    return action

def sarsa(alpha, gamma, epsilon, episodes, max_steps, n_tests, render = False, test=Fa
    """
    @param alpha learning rate
    @param gamma decay factor
    @param epsilon for exploration
    @param max_steps for max step in each episode
    @param n_tests number of test episodes
    """
    env = gym.make('Taxi-v2')
    n_states, n_actions = env.observation_space.n, env.action_space.n
    for episode in range(episodes):
        print(f"Episode: {episode}")
        total_reward = 0
        s = env.reset()
        a = epsilon_greedy(Q, epsilon, n_actions, s)
        t = 0
        done = False
        while t < max_steps:
            if render:
                env.render()
            t += 1
            s_, reward, done, info = env.step(a)
            total_reward += reward
            a_ = epsilon_greedy(Q, epsilon, n_actions, s_)
            if done:
                Q[s, a] += alpha * ( reward  - Q[s, a] )
            else:
                Q[s, a] += alpha * ( reward + (gamma * Q[s_, a_] ) - Q[s, a] )
            s, a = s_, a_
            if done:
                if render:
                    print(f"This episode took {t} timesteps and reward {total_reward}'
                timestep_reward.append(total_reward)
                break
    if render:
        print(f"Here are the Q values:\n{Q}\nTesting now:")
    if test:
        test_agent(Q, env, n_tests, n_actions)
    return timestep_reward

def test_agent(Q, env, n_tests, n_actions, delay=0.1):
    for test in range(n_tests):
        print(f"Test #{test}")
        s = env.reset()
        done = False
        epsilon = 0
        total_reward = 0
        while True:
            time.sleep(delay)
            env.render()
            a = epsilon_greedy(Q, epsilon, n_actions, s, train=True)
            print(f"Chose action {a} for state {s}")
            s, reward, done, info = env.step(a)
            total_reward += reward
            if done:
```

```python


if __name__ =="__main__":
    alpha = 0.4
    gamma = 0.999
    epsilon = 0.9
    episodes = 3000
    max_steps = 2500
    n_tests = 20
    timestep_reward = sarsa(alpha, gamma, epsilon, episodes, max_steps, n_tests)
    print(timestep_reward)
```

Q-Learning

Q-Learning is an off-policy TD control policy. It's exactly like SARSA with the only difference being — it doesn't follow a policy to find the next action ***A'*** but rather chooses the action in a greedy fashion. Similar to SARSA its aim is to evaluate the Q values and its update rule is:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

Source: Introduction to Reinforcement learning by Sutton and Barto — 6.8

Observe: Unlike SARSA where an action ***A'*** was chosen by following a certain policy, here the action ***A'*** (***a*** in this case) is chosen in a greedy fashion by simply taking the max of ***Q*** over it.

Here's the Q-learning algorithm:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Here's the python implementation of Q-learning:

```
import gym
import numpy as np
import time

"""
Qlearning is an off policy learning python implementation.
This is a python implementation of the qlearning algorithm in the Sutton and
Barto's book on RL. It's called SARSA because - (state, action, reward, state,
action). The only difference between SARSA and Qlearning is that SARSA takes the
next action based on the current policy while qlearning takes the action with
maximum utility of next state.
Using the simplest gym environment for brevity: https://gym.openai.com/envs/FrozenLake
"""

def init_q(s, a, type="ones"):
    """
    @param s the number of states
    @param a the number of actions
    @param type random, ones or zeros for the initialization
    """
    if type == "ones":
        return np.ones((s, a))
    elif type == "random":
        return np.random.random((s, a))
    elif type == "zeros":
        return np.zeros((s, a))


def epsilon_greedy(Q, epsilon, n_actions, s, train=False):
    """
    @param Q Q values state x action -> value
    @param epsilon for exploration
    @param s number of states
    @param train if true then no random actions selected
    """
    if train or np.random.rand() < epsilon:
        action = np.argmax(Q[s, :])
    else:
        action = np.random.randint(0, n_actions)
    return action

def qlearning(alpha, gamma, epsilon, episodes, max_steps, n_tests, render = False, tes
    @param gamma decay factor
    @param epsilon for exploration
    @param max_steps for max step in each episode
    @param n_tests number of test episodes
    """
    env = gym.make('Taxi-v2')
    n_states, n_actions = env.observation_space.n, env.action_space.n
    Q = init_q(n_states, n_actions, type="ones")
    timestep_reward = []
    for episode in range(episodes):
        print(f"Episode: {episode}")
        s = env.reset()
        a = epsilon_greedy(Q, epsilon, n_actions, s)
        t = 0
        total_reward = 0
        done = False
        while t < max_steps:
            if render:
                env.render()
            t += 1
            s_, reward, done, info = env.step(a)
            total_reward += reward
            a_ = np.argmax(Q[s_, :])
            if done:
                Q[s, a] += alpha * ( reward  - Q[s, a] )
            else:
                Q[s, a] += alpha * ( reward + (gamma * Q[s_, a_]) - Q[s, a] )
            s, a = s_, a_
            if done:
                if render:
                    print(f"This episode took {t} timesteps and reward: {total_reward
                timestep_reward.append(total_reward)
                break
    if render:
        print(f"Here are the Q values:\n{Q}\nTesting now:")
    if test:
        test_agent(Q, env, n_tests, n_actions)
    return timestep_reward

def test_agent(Q, env, n_tests, n_actions, delay=1):
    for test in range(n_tests):
        print(f"Test #{test}")
        s = env.reset()
        done = False
        epsilon = 0
        while True:
            a = epsilon_greedy(Q, epsilon, n_actions, s, train=True)
            print(f"Chose action {a} for state {s}")
            s, reward, done, info = env.step(a)
            if done:
                if reward > 0:
                    print("Reached goal!")
                else:
                    print("Shit! dead x_x")
                time.sleep(3)
                break


if __name__ =="__main__":
    alpha = 0.4
    gamma = 0.999
    epsilon = 0.9
    episodes = 10000
    max_steps = 2500
    n_tests = 2
    timestep_reward = qlearning(alpha, gamma, epsilon, episodes, max_steps, n_tests,
    print(timestep_reward)
```

Expected SARSA

Expected SARSA, as the name suggest takes the expectation (mean) of Q values for every possible action in the current state. The target update rule shall make things more clear:

Source: Introduction to Reinforcement learning by Sutton and Barto —6.9

And here's the python implementation:

```
import gym
import numpy as np
import time

"""
```

```
RL: It's called SARSA because - (state, action, reward, state, action). The only diff
between SARSA and Qlearning is that SARSA takes the next action based on the current p
while qlearning takes the action with maximum utility of next state.
Using the simplest gym environment for brevity: https://gym.openai.com/envs/FrozenLake
"""


def init_q(s, a, type="ones"):
    """
    @param s the number of states
    @param a the number of actions
    @param type random, ones or zeros for the initialization
    """
    if type == "ones":
        return np.ones((s, a))
    elif type == "random":
        return np.random.random((s, a))
    elif type == "zeros":
        return np.zeros((s, a))


def epsilon_greedy(Q, epsilon, n_actions, s, train=False):
    """
    @param Q Q values state x action -> value
    @param epsilon for exploration
    @param s number of states
    @param train if true then no random actions selected
    """
    if train or np.random.rand() < epsilon:
        action = np.argmax(Q[s, :])
    else:
        action = np.random.randint(0, n_actions)
    return action

def expected_sarsa(alpha, gamma, epsilon, episodes, max_steps, n_tests, render = False
    """
    @param alpha learning rate
    @param gamma decay factor
    @param epsilon for exploration
    @param max_steps for max step in each episode
    @param n_tests number of test episodes
    """
    env = gym.make('Taxi-v2')
    n_states, n_actions = env.observation_space.n, env.action_space.n
    Q = init_q(n_states, n_actions, type="ones")
    timestep_reward = []
    for episode in range(episodes):
        s = env.reset()
        t = 0
        done = False
        while t < max_steps:
            if render:
                env.render()
            t += 1
            a = epsilon_greedy(Q, epsilon, n_actions, s)
            s_, reward, done, info = env.step(a)
            total_reward += reward
            if done:
                Q[s, a] += alpha * ( reward  - Q[s, a] )
            else:
                expected_value = np.mean(Q[s_,:])
                # print(Q[s,:], sum(Q[s,:]), len(Q[s,:]), expected_value)
                Q[s, a] += alpha * (reward + (gamma * expected_value) - Q[s, a])
            s = s_
            if done:
                if True:
                    print(f"This episode took {t} timesteps and reward {total_reward}'
                timestep_reward.append(total_reward)
                break
    if render:
        print(f"Here are the Q values:\n{Q}\nTesting now:")
    if test:
        test_agent(Q, env, n_tests, n_actions)
    return timestep_reward

def test_agent(Q, env, n_tests, n_actions, delay=0.1):
    for test in range(n_tests):
        print(f"Test #{test}")
        s = env.reset()
        done = False
        epsilon = 0
        total_reward = 0
        while True:
            time.sleep(delay)
            env.render()
            a = epsilon_greedy(Q, epsilon, n_actions, s, train=True)
            print(f"Chose action {a} for state {s}")
            s, reward, done, info = env.step(a)
            total_reward += reward
            if done:
                print(f"Episode reward: {total_reward}")
                time.sleep(1)
```

```
if __name__ =="__main__":
    alpha = 0.1
    gamma = 0.9
    epsilon = 0.9
    episodes = 1000
    max_steps = 2500
    n_tests = 20
    timestep_reward = expected_sarsa(alpha, gamma, epsilon,
                                     episodes, max_steps, n_tests,
                                     render=False, test=True
                                     )
    print(timestep_reward)
```

Comparison

I've used the following parameters to test the three algorithms in Taxi-v2 gym environment

- alpha = 0.4
- gamma = 0.999
- epsilon = 0.9
- episodes = 2000
- max_steps = 2500 (max number of time steps possible in a single episode)

Here are the plots showcasing the comparison between the above three policy control methods:

Convergence:

Clearly, by the following plots, Q learning (green) converges before both SARSA (orange) and expected SARSA (Blue)

SARSA, Q-learning & Expected SARSA — Convergence comparison

Performance:

For my implementation of the three algorithms, Q-learning seems to perform the best and Expected SARSA performs the worst.

Conclusion

Temporal Difference learning is the most important reinforcement learning concept. It's further derivatives like DQN and double DQN (I may discuss them later in another post) have achieved groundbreaking results renowned in the field of AI. Google's alpha go used DQN algorithm along with CNNs to defeat the go world champion. You are now equipped with the theoretical and practical knowledge of basic TD, go out and explore!

> In case I made some errors please mention them in the responses. Thanks for reading.

Machine Learning Reinforcement Learning Artificial Intelligence Temporal Difference Q Learning