

Playing Blackjack using Model-free Reinforcement Learning in Google Colab!



Pranav Mahajan [Follow](#)

Jan 14 · 11 min read

A comparative study of algorithms like Monte-Carlo Control and Temporal-Difference Control used to solve games like Blackjack.



I felt compelled to write this article because I noticed not many articles explained Monte Carlo methods in detail whereas just jumped straight to Deep Q-learning applications.

In this article you'll get to learn about

1. The motivation behind model-free algorithms in RL (Reinforcement Learning)
2. Inner workings of those algorithms while applying them to solve Blackjack as an example in a Colab notebook in browser itself (no need to install anything)!

Before we start I'd like to let you know that this article assumes a basic knowledge of very basic concepts of Reinforcement Learning, if you don't it's alright here's a quick recap :

- In a usual RL setup, agent takes an action in an environment and gets in return observations and rewards from the environment.

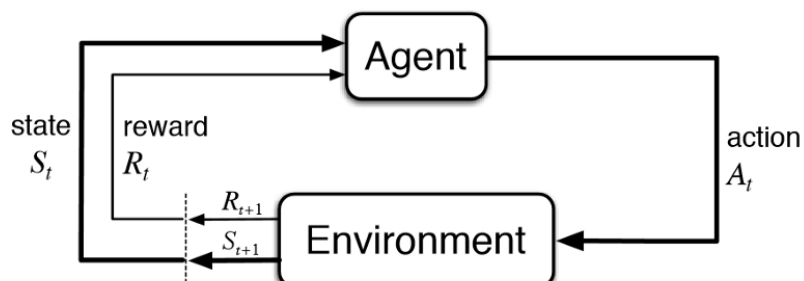


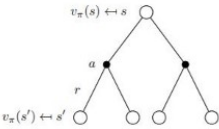
Illustration (<https://i.stack.imgur.com/eoeSq.png>)

Reinforcement is the strengthening of a pattern of behavior as a result of an animal receiving a stimulus in an appropriate temporal relationship with another stimulus or a response. —Pavlov's Monograph (1927)

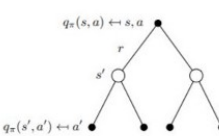
- These tasks that agent performs to get better at can be either episodic or continuing, here Blackjack is an episodic game, i.e. it ends with you either winning or losing.
- Agents look forward to maximizing their cumulative 'expected' rewards or also known as 'expected return'. Here we give lesser importance to rewards we might get way into the future as compared to some reward we might get in the immediate step. i.e. $G_t = R_t + 1 + \gamma R_{t+2} + \dots$
- We assume our environment to have Markov Property i.e. Future state or rewards are independent of the past state given the present state i.e. $P(S_{t+1} | S_t) = P(S_{t+1} | S_1, S_2, S_3 \dots S_t)$.

Policy for an agent can be thought of as a strategy the agent uses, it usually maps from perceived states of environment to actions to be taken when in those states.

- We define state-value pairs $V(s)$ corresponding to a policy π : as the expected return an agent will get if it were to start in that state and follow the policy π . Remember $V(s)$ always corresponds to some policy π .
- We also define action-value function $Q(s,a)$ as the value of taking action 'a' in state 's' under policy π .
- $V(s) = E[G_t | S_t = s]$ and $Q(s,a) = E[G_t | S_t = s, A_t = a]$ which can also written in the way shown in the picture below. That form will more useful in computing $V(s)$ and $Q(s,a)$



$$v_{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{\pi}(s') \right)$$



$$q_{\pi}(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_{\pi}(s',a')$$

$\mathcal{P}_{ss'}$ is a property of the environment, also referred to as $P(s', r|s, a)$ in the book by Sutton and Barto.

Various Model-based methods like Dynamic Programming use the Bellman Equation (a recursive relation between $V(S_t)$ and $V(S_{t+1})$) to iteratively find optimal Value functions and Q functions.

And that's the end of recap!

What do we mean by Model-free methods? and why use them?

To use model-based methods we need to have complete knowledge of the environment i.e. we need to know $P_{ss'}$ (please refer to above picture): the transition probability we'll end up in state $S_{t+1}=s'$ if the agent is in state $S_t=s$ and takes action $A_t=a$. For example, if a bot chooses to move forward, it might move sideways in case of slippery floor underneath it. In games like Blackjack our action space is limited like we can either choose to "hit" or "stick" but we can end up in any of the many possible states of which you know none of the probabilities! In Blackjack state is determined by your sum, the dealers sum and whether you have a usable ace or not as follows:

```
env = gym.make('Blackjack-v0')
print(env.observation_space)
print(env.action_space)
```

States: (32*10*2 array)

- Players current sum: [0,31] i.e. 32 states
- Dealer's face up card: [1,10] i.e. 10 states
- Whether the player has a usable ace or not: [0] or [1] i.e. 2 states

Actions:

- Either stick or hit: [0] or [1] i.e 0 for stick , 1 for hit

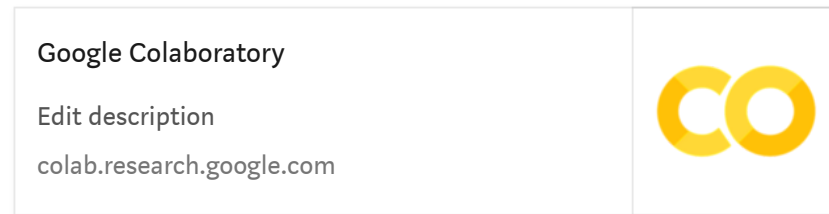
What to do when we don't have model of the environment? You take samples by interacting with the again and again and estimate such information from them. Model-free are basically trial and error approaches which require no explicit knowledge of environment or transition probabilities between any two states.

Thus we see that model-free systems cannot even think bout how their environments will change in response to a certain action. This way they have reasonable advantage over more complex methods where the real bottleneck is the difficulty of constructing a sufficiently accurate environment model. (For example, we can't possibly start listing out probabilities of next card the dealer will pull out in each state of Blackjack.)

(Side note) Perhaps the first to succinctly express the essence of trial-and-error learning as a principle of learning was Edward Thorndike in his "Law of Effect" in 1911 but the idea of trial-and-error learning goes as far as the 1850s.

With understood the motivation behind model-free methods, let's see a few algorithms!

>> Now onward we'll dive into code while explaining the algorithms.
it'll be helpful if you open the Colab notebook in another tab!



Monte-Carlo Prediction Algorithm:

In order to construct better policies, we need to first be able to evaluate any policy. If an agent follows a policy for many episodes, using Monte-Carlo Prediction, we can construct the Q-table (i.e. “estimate” the action-value function) from the results from these episodes.

So we can start with a stochastic policy like “stick” with 80% probability if sum is greater than 18 as we don’t want to exceed 21. Else if sum is less than 18, we’ll “hit” with 80% probability. Following code generates episodes using the following policy and then later we’ll evaluate this policy:

(Note here an ‘episode’ i.e. the quantity that is returned is a list of (state, action, reward) tuples corresponding to every action taken in the episode)

```
1 def generate_episode_from_limit_stochastic(bj_env):
2     episode = []
3     state = bj_env.reset() #here, bj_env is the env instance
4     while True:
5         probs = [0.8, 0.2] if state[0] > 18 else [0.2, 0.8]
6         action = np.random.choice(np.arange(2), p=probs)
7         next_state, reward, done, info = bj_env.step(action)
8         episode.append((state, action, reward))
9         state = next_state
```

Now, we want to get the Q-function given a policy and it needs to learn the value functions directly from episodes of experience. Note that in Monte Carlo approaches we are getting the reward at the end of an episode where..

Episode = S1 A1 R1, S2 A2 R2, S3 A3 R3..... ST(Sequence of steps till the termination state)

We’ll learn value functions from sample returns with the MDP, recollect from the recap that:

$Q(s,a) = E[G_t \mid S_t = s, A_t = a]$ and $G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$ for a policy π .

What is the sample return? Say we played for 10 episodes using a policy and we got rewards 2,6,5,7 when we visited the same state 'S' for 4 out of 10 times then the sample return would be $(2+6+5+7)/4 = 20/4 = 5 \sim V(s)$. Thus sample return is the average of returns(rewards) from episodes. In what order did we end up visiting the state doesn't really matter here, estimate for each value is calculated independently!

This way we can build either a V-table or a Q-table, in order to create a Q-table we'll need to keep track of reward we got for each we visited a (state,action) pair and also keep a track of how many times we visited the state say an N-table.

Depending on which returns are chosen while estimating our Q-values

- First visit MC: **In an episode**, we average returns only for **first** time (s,a) is visited. It's statistically an unbiased approach.
- Every visit MC: **In an episode**, we average returns only for **every** time (s,a) is visited. It's statistically an biased approach.

For example: In an episode, S1 A1 R1, S2 A2 R2, S3 A3 R3, S1 A1 R4 →End. Then first visit MC will consider rewards till R3 in calculating the return while every visit MC will consider all rewards till the end of episode.

Here, in Blackjack it doesn't affect much whether we use first visit or every visit MC. Here's the algorithm for first-visit MC prediction

```

Input: policy  $\pi$ , positive integer num_episodes
Output: value function  $Q$  ( $\approx q_\pi$  if num_episodes is large enough)
Initialize  $N(s, a) = 0$  for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
Initialize returns_sum( $s, a$ ) = 0 for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
for  $i \leftarrow 1$  to num_episodes do
    Generate an episode  $S_0, A_0, R_1, \dots, S_T$  using  $\pi$ 
    for  $t \leftarrow 0$  to  $T-1$  do
        if  $(S_t, A_t)$  is a first visit (with return  $G_t$ ) then
             $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$ 
            returns_sum( $S_t, A_t$ )  $\leftarrow$  returns_sum( $S_t, A_t$ ) +  $G_t$ 
        end
    end
     $Q(s, a) \leftarrow$  returns_sum( $s, a$ )/ $N(s, a)$  for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
return  $Q$ 

```

MC pred, pseudocode

But we'll implement every-visit MC prediction as shown below:

```

1  def mc_prediction_q(env, num_episodes, generate_episode, ga
2      # initialize empty dictionaries of arrays
3      returns_sum = defaultdict(lambda: np.zeros(env.action_s
4      N = defaultdict(lambda: np.zeros(env.action_space.n))
5      Q = defaultdict(lambda: np.zeros(env.action_space.n))
6      # loop over episodes
7      for i_episode in range(1, num_episodes+1):
8          # Let us monitor our progress :)
9          if i_episode % 1000 == 0:
10             print("\rEpisode {}/{}.".format(i_episode, num_
11
12
13         # Generating an episode using our 80-20 policy we d
14         episode = generate_episode(env)
15         # obtain the states, actions, and rewards
16         states, actions, rewards = zip(*episode)
17         '''
18         This discounts array is the amount by which we wann
19         discounts = [1,gamma, gamma^2, gamma^3.....]
20         then we compute the total return Gt= Rt+1 *1 + Rt+2

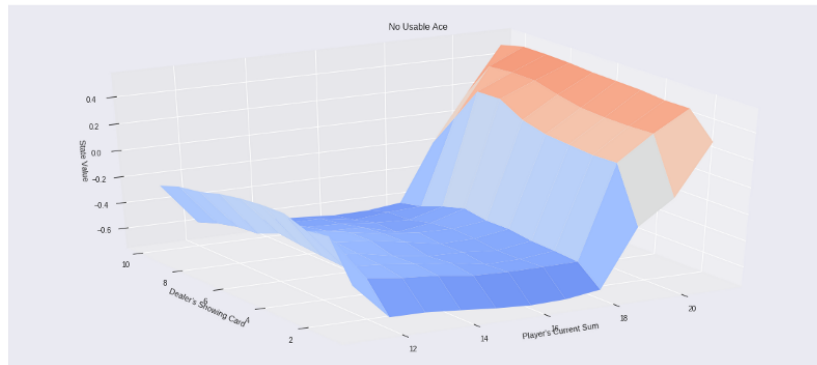
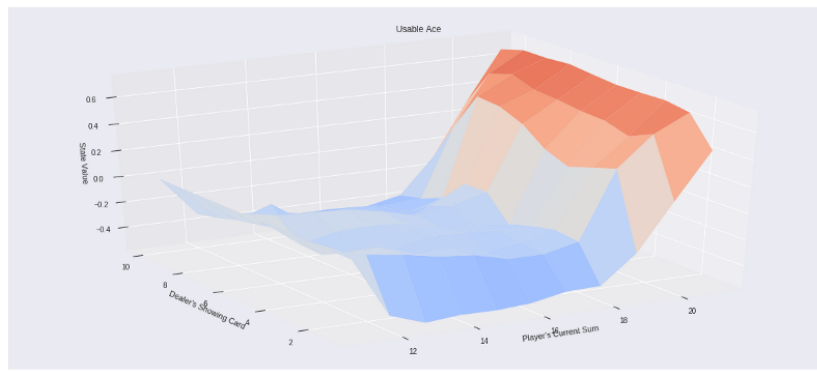
```

We first initialize a Q-table and N-table to keep a tack of our visits to every [state][action] pair.

Then in the generate episode function, we are using the 80–20 stochastic policy as we discussed above.

```
sum(rewards[i:]*discounts[:-(1+i)]) #is used to calculate Gt
```

This will estimate the Q-table for any policy used to generate the episodes! Once we have Q-values, getting V-values is fairly easy as $V(s) = Q(s, \pi(s))$. Let's plot the state values $V(s)$!

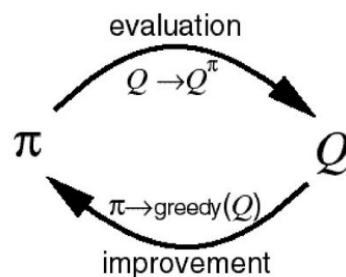


Plotted $V(s)$ for each of $32 \times 10 \times 2$ states, each $V(s)$ has value between $[-1,1]$ as the reward we get is $+1, 0, -1$ for win, draw and loss

So now we know how to estimate the action-value function for a policy, how do we improve on it? Using the ...

Monte-Carlo Control Algorithm

So here's the plan in a nutshell. We start with a stochastic policy and compute the Q-table using MC prediction. So we now have the knowledge of which actions in which states are better than other i.e. they have greater Q-values. So we can improve upon our existing policy by just greedily choosing the best action at each state as per our knowledge i.e. Q-table and then recompute the Q-table and chose next policy greedily and so on! Sounds good?



- ❑ **MC policy iteration:** Policy evaluation using MC methods followed by policy improvement
- ❑ **Policy improvement step:** greedify with respect to value (or action-value) function

From slides based on RL book by Sutton and Barto

- But we have a problem, what if for a state 'St' there were 2 actions 'stick' and 'hit' what if agent chooses 'hit' early on in the many

games the agent will play and wins whereas 'stick' should've been the better action, it never got to explore the action as the algorithm went on greedily choosing it multiple times. Thus to solve this exploration-exploitation dilemma, we'll use ϵ -greedy policy i.e. we'll explore, take a random action with probability ' ϵ ' (epsilon) rather than greedily exploiting the learnt Q-values. Naturally we'd like to keep the ϵ values ~ 1 at the start and reduce it till close to 0 by the end of the learning (total no. of episodes).

- Incremental mean: Remember how we took average of all the returns to estimate Q-values in MC Prediction? But now unlike MC Pred, in MC Control our policy is undergoing change every cycle! We can write the same equation in terms of previous Q-values in this way: Trust me it's not out of the blue you can arrive at the same equation yourself if you see that $N(S_t, A_t) * Q(S_t, A_t)$ is the G_t until this step and thus $G_t - Q(S_t, A_t)$ is incremental change.

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + (1/N(S_t, A_t)) * (G_t - Q(S_t, A_t))$$

- Constant alpha: Now as the $N(S_t, A_t)$ increases i.e. we visit the same state-action pair many times in our interaction, the incremental change term decreases, which means our latter experiences will affect lesser and lesser than starting ones. To solve this we can replace $(1/N)$ term by a constant α , which is hyperparameter, for us to choose.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha * (G_t - Q(S_t, A_t))$$

Having learnt these crucial practical changes to idea of just sampling return, here's the algorithm for first-visit MC control!

```

Input: positive integer num_episodes, small positive fraction  $\alpha$ , GLIE  $\{\epsilon_t\}$ 
Output: policy  $\pi$  ( $\approx \pi_*$  if num_episodes is large enough)
Initialize  $Q$  arbitrarily (e.g.,  $Q(s, a) = 0$  for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}(s)$ )
for  $i \leftarrow 1$  to num_episodes do
     $\epsilon \leftarrow \epsilon_i$ 
     $\pi \leftarrow \epsilon\text{-greedy}(Q)$ 
    Generate an episode  $S_0, A_0, R_1, \dots, S_T$  using  $\pi$ 
    for  $t \leftarrow 0$  to  $T - 1$  do
        if  $(S_t, A_t)$  is a first visit (with return  $G_t$ ) then
             $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$ 
        end
    end
return  $\pi$ 

```

MC control, pseudocode

We'll implement every-visit MC control, because it's just slightly easier to code


```

1  def get_probs(Q_s, epsilon, nA): #nA is no. of actions in t
2      # obtains the action probabilities corresponding to eps
3      policy_s = np.ones(nA) * epsilon / nA
4      best_a = np.argmax(Q_s)
5      policy_s[best_a] = 1 - epsilon + (epsilon / nA)
6      return policy_s
7
8  '''
9  Now we will use this get_probs func in generating the episo
10 Note that we are no longer using the stochastic policy we s
11 '''
12 def generate_episode_from_Q(env, Q, epsilon, nA):
13     # generates an episode from following the epsilon-greed
14     episode = []
15     state = env.reset()
16     while True:
17         action = np.random.choice(np.arange(nA), p=get_prob
18                                   if state in Q else env.
19         next_state, reward, done, info = env.step(action)
20         episode.append((state, action, reward))
21         state = next_state
22         if done:
23             break
24     return episode
25
26 '''
27 '''

```

We're just using 3 functions to make the code look cleaner. To generate episode just like we did for MC prediction, we need a policy. But note that we are not feeding in a stochastic policy, but instead our policy is epsilon-greedy wrt our previous policy. get_probs function gives us the action probabilities for the same i.e. $\pi(a|s)$

And the update_Q function just updates Q-values using incremental mean and constant alpha. Finally we call all these functions in the MC control and ta-da!

```

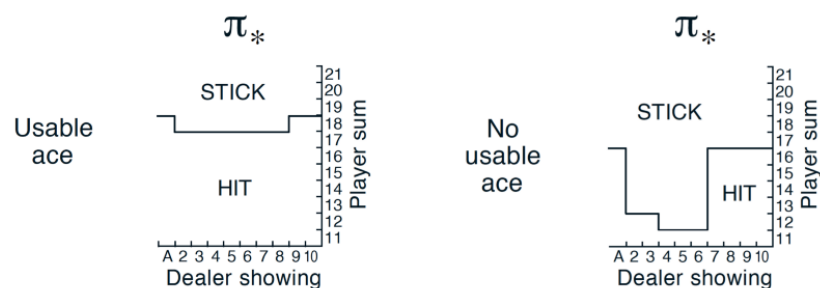
1  def mc_control(env, num_episodes, alpha, gamma=1.0, eps_sta
2      nA = env.action_space.n
3      # initialize empty dictionary of arrays
4      Q = defaultdict(lambda: np.zeros(nA))
5      epsilon = eps_start
6      # loop over episodes
7      for i_episode in range(1, num_episodes+1):
8          # monitor progress
9          if i_episode % 1000 == 0:
10             print("\rEpisode {} / {}".format(i_episode, num_
11                 sys.stdout.flush()
12             # set the value of epsilon
13             epsilon = max(epsilon*eps_decay, eps_min)
14             # generate an episode by following epsilon-greedy p

```

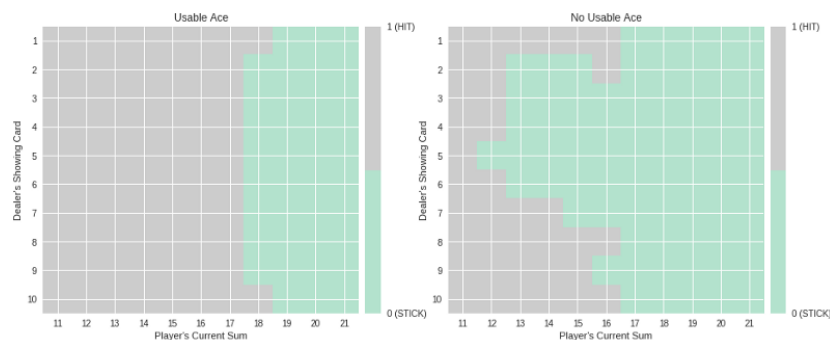
MC Control

Feel free to explore the notebook comments and explanations for further clarification!

Thus finally we have an algorithm that learns to play Blackjack, well a slightly simplified version of Blackjack at least. Let's compare the policy learnt compared to the optimal policy mentioned in RL book by Sutton and Barto.



RL, Sutton Barto Fig 5.2



Our learnt policy (rotated by 90 deg)

```

(21, 8, True)
Game has ended! Your Reward: 1.0
You won :)

(9, 10, False)
(19, 10, False)
Game has ended! Your Reward: 1.0
You won :)

(14, 9, False)
Game has ended! Your Reward: -1
You lost :(

(10, 6, False)
(14, 6, False)
Game has ended! Your Reward: 1.0
You won :)

```

Play using our learnt policy!

Wolla! There you go, we have an AI that wins most of the times when it plays Blackjack! But there's more...

Temporal Difference methods

Now Blackjack isn't the best environment to learn advantages of TD methods as Blackjack is an episodic game and Monte Carlo methods assume episodic environments. In MC control, at the end of each episode, we update the Q-table and update our policy. Thus we have no way of finding out which was the wrong move that led to the loss but it won't matter in a short game like Blackjack. If it were a longer game like chess, it would make more sense to use **TD control methods because they boot strap , meaning it will not wait until the end of the episode to update the expected future reward estimation(V) , it will only wait until the next time step to update the value estimates.**

(Side note) TD methods are distinctive in being driven by the difference between temporally successive estimates of the **same quantity**. More over the origins of temporal-difference learning are in part in animal psychology, in particular, in the notion of secondary reinforcers. Secondary reinforcer is a stimulus that has been paired with a primary reinforcer (simplistic reward from environment itself) and as a result the secondary reinforcer has come to take similar properties.

For example, in MC control:

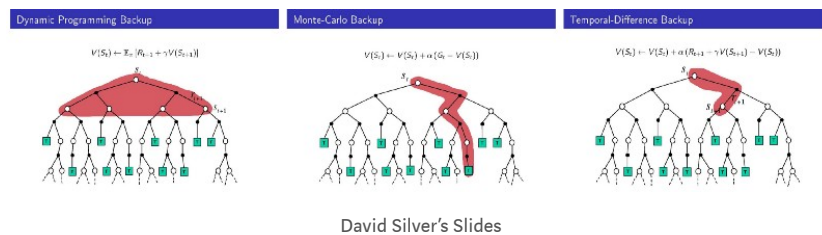
$$V(s) = E [G_t \mid St = s] \text{ and } G_t = R_t + 1 + \gamma R_{t+2} + \dots$$

But the in TD control:

$$\begin{aligned}
v(s) &= \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]
\end{aligned}$$

Just like in Dynamic Programming, TD uses Bellman Equations to update at each step.

Following picture can help explain the difference between DP, MC and TD approaches.



Thus we can think of the incremental mean in a different way as if G_t was the Target or our expectation of the return the agent would have got, but instead got the return $Q(S_t, A_t)$ so it would make sense to push Q -values towards G_t by $\alpha * (G_t - Q(S_t, A_t))$.

Similarly in case of TD methods, instantaneous TD target is $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ and thus the TD error will be $(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$

Depending on different TD targets and slightly different implementations the 3 TD control methods are:

- SARSA or SARSA(0)

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

SARSA update equation

Which when implemented in python looks like this:

```

1  def update_Q_sarsa(alpha, gamma, Q, state, action, reward, n
2      """Returns updated Q-value for the most recent experienc
3      current = Q[state][action] # estimate in Q-table (for c
4      # get value of state, action pair at next time step
5      Qsa_next = Q[next_state][next_action] if next_state is n
6      target = reward + (gamma * Qsa_next) # con

```

- SARSAMAX or Q-learning

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

SARSAMAX update equation

Which when implemented in python looks like this:

```
1 def update_Q_sarsamax(alpha, gamma, Q, state, action, reward):
2     """Returns updated Q-value for the most recent experience"""
3     current = Q[state][action] # estimate in Q-table (for current state-action pair)
4     Qsa_next = np.max(Q[next_state]) if next_state is not None else 0
5     target = reward + (gamma * Qsa_next) # compute target
6     Q[state][action] = current + alpha * (target - current) # update Q-value
```

- Expected SARSA

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

Expected SARSA update

Which when implemented in python looks like this:

```
1 def update_Q_expsarsa(alpha, gamma, nA, eps, Q, state, action):
2     """Returns updated Q-value for the most recent experience"""
3     current = Q[state][action] # estimate in Q-table
4     policy_s = np.ones(nA) * eps / nA # current policy (for state s)
5     policy_s[np.argmax(Q[next_state])] = 1 - eps + (eps / nA) # update policy
6     Qsa_next = np.dot(Q[next_state], policy_s) # get expected next state action value
7     target = reward + (gamma * Qsa_next) # compute target
```

NOTE that Q-table in TD control methods is updated every time-step every episode as compared to MC control where it was updated at the end of every episode.

I know I haven't explained TD methods in as much depth as the MC methods instead analyzed in a comparative fashion, but for those who are interested all of the 3 methods are implemented in the notebook. You are welcome to explore the whole notebook for and play with functions for a better understanding!

That's all folks! Hope you enjoyed!

References:

1. Reinforcement Learning: An Introduction (Book by Andrew Barto and Richard S. Sutton)
2. David Silver's slides (UCL course on RL)

