b)

**Passive Learning**

As the goal of the agent is to evaluate how good an optimal policy is, the agent needs to learn the expected utility *Uπ(s)* for each state *s*. This can be done in three ways.

1. ***Direct Utility Estimation***:

In this method, the agent executes a sequence of trials or runs (sequences of states-actions transitions that continue until the agent reaches the terminal state). Each trial gives a sample value and the agent estimates the utility based on the samples values. Can be calculated as running averages of sample values. The main drawback is that this method makes a wrong assumption that *state utilities are independent* while in reality they are [Markovian](https://en.wikipedia.org/wiki/Markov_property). Also, it is slow to converge.

Suppose we have a 4x3 grid as the environment in which the agent can move either Left, Right, Up or Down(set of available actions). An example of a run,

Total reward starting at *(1,1)* = 0.72

***2. Adaptive Dynamic Programming(ADP)***

ADP is a smarter method than Direct Utility Estimation as it runs trials to learn  the model of the environment by estimating the utility of a state as a sum of reward for being in that state and the expected discounted reward of being in the next state.

[https://latex.codecogs.com/gif.latex?U%5e%7b\pi%7d(s)&space;=&space;R(s)&space;+&space;\gamma&space;\sum_%7bs%27%7d&space;P(s%27&space;|&space;s,&space;\pi(s)&space;)&space;U%5e%7b\pi%7d(s%27)](https://www.codecogs.com/eqnedit.php?latex=U%5e%7b\pi%7d(s)&space;=&space;R(s)&space;+&space;\gamma&space;\sum_%7bs'%7d&space;P(s'&space;|&space;s,&space;\pi(s)&space;)&space;U%5e%7b\pi%7d(s'))

Where *R(s)* = reward for being in state *s*, *P(s'|s, π(s))* = transition model, *γ* = discount factor and *Uπ(s)* = utility of being in state *s'*.

It can be solved using value-iteration algorithm. The algorithm converges fast but can become quite costly to compute for large state spaces. ADP is a model based approach and requires the transition model of the environment. A model-free approach is Temporal Difference Learning.

1. ***Temporal Difference Learning (TD)***

TD learning does not require the agent to learn the transition model. The update occurs between successive states and agent only updates states that are directly affected.

[https://latex.codecogs.com/gif.latex?U%5e\pi(s)&space;\leftarrow&space;U%5e\pi(s)&space;+&space;\alpha(R(s)&space;+&space;\gamma(U%5e\pi(s%27)&space;-&space;U%5e\pi(s))](https://www.codecogs.com/eqnedit.php?latex=U%5e\pi(s)&space;\leftarrow&space;U%5e\pi(s)&space;+&space;\alpha(R(s)&space;+&space;\gamma(U%5e\pi(s')&space;-&space;U%5e\pi(s)))

Where *α* = learning rate which determines the convergence to true utilities.

While ADP adjusts the utility of *s* with all its successor states, TD learning adjusts it with that of a single successor state *s'*.  TD is slower in convergence but much simpler in terms of computation.

**Active Learning**

1. ***ADP with exploration function***

As the goal of an active agent is to learn an optimal policy, the agent needs to learn the expected utility of each state and update its policy.

Can be done using a passive ADP agent and then using value or policy iteration it can learn optimal actions. But this approach results into a greedy agent.

Hence, we use an approach that gives higher weights to unexplored actions and lower weights to actions with lower utilities.

[https://latex.codecogs.com/gif.latex?U_%7bi+1%7d(s)&space;\leftarrow&space;R(s)&space;+&space;\gamma\max_%7ba&space;\in&space;A%7d&space;f&space;\Big\(&space;\sum_%7bs%27%7dP(s%27|s,&space;a)U_i(s%27),&space;N(s,a)\Big\)](https://www.codecogs.com/eqnedit.php?latex=U_%7bi+1%7d(s)&space;\leftarrow&space;R(s)&space;+&space;\gamma\max_%7ba&space;\in&space;A%7d&space;f&space;\Big\(&space;\sum_%7bs'%7dP(s'|s,&space;a)U_i(s'),&space;N(s,a)\Big\))

Where *f(u,n)* is the exploration function that increases with expected value *u* and decreases with number of tries *n*

[https://latex.codecogs.com/gif.latex?f(u,n)&space;=&space;\left\%7b\begin%7bmatrix%7d&space;R%5e+,&space;if&space;n&space;%3C&space;N_e&space;\\&space;u,&space;otherwise&space;\end%7bmatrix%7d\right.](https://www.codecogs.com/eqnedit.php?latex=f(u,n)&space;=&space;\left\%7b\begin%7bmatrix%7d&space;R%5e+,&space;if&space;n&space;%3c&space;N_e&space;\\&space;u,&space;otherwise&space;\end%7bmatrix%7d\right.)

*R+* is an optimisic reward and *Ne* is the number of times we want an agent to be forced to pick an action in every state. The exploration function *converts a passive agent into an active one*.

1. ***Q-Learning***

Q-learning is a TD learning method which does not require the agent to learn the transitional model, instead learns Q-value functions *Q(s, a)* .

[https://latex.codecogs.com/gif.latex?U(s)&space;=&space;\max_a&space;Q(s,&space;a)](https://www.codecogs.com/eqnedit.php?latex=U(s)&space;=&space;\max_a&space;Q(s,&space;a))

Q-values can be updated using the following equation,

[https://latex.codecogs.com/gif.latex?Q(s,&space;a)&space;\leftarrow&space;Q(s,&space;a)&space;+&space;\alpha&space;\Big\(&space;R(s)&space;+\gamma&space;\max_%7ba%27%7d&space;Q(s%27,&space;a%27)&space;-&space;Q(s,&space;a)&space;\Big\)](https://www.codecogs.com/eqnedit.php?latex=Q(s,&space;a)&space;\leftarrow&space;Q(s,&space;a)&space;+&space;\alpha&space;\Big\(&space;R(s)&space;+\gamma&space;\max_%7ba'%7d&space;Q(s',&space;a')&space;-&space;Q(s,&space;a)&space;\Big\))  
Next action can be selected using the following policy,

[https://latex.codecogs.com/gif.latex?a_%7bnext%7d&space;=&space;arg&space;\max_%7ba%27%7d&space;f(Q(s%27,a%27),&space;N(s%27,&space;a%27))](https://www.codecogs.com/eqnedit.php?latex=a_%7bnext%7d&space;=&space;arg&space;\max_%7ba'%7d&space;f(Q(s',a'),&space;N(s',&space;a')))

Again this is simpler to compute but slower than ADP.

c)

<https://www.oreilly.com/library/view/statistics-for-machine/9781788295758/e8f0bbd1-b60a-47d2-9cd4-0d0661fc9212.xhtml>

The main problem with TD learning and DP is that their step updates are *biased* on the initial conditions of the learning parameters. The bootstrapping process typically updates a function or lookup Q(s,a) on a successor value Q(s',a') using whatever the current estimates are in the latter. Clearly at the very start of learning these estimates contain no information from any real rewards or state transitions.

If learning works as intended, then the bias will reduce asymptotically over multiple iterations. However, the bias can cause significant problems, especially for off-policy methods (e.g. Q Learning) and when using function approximators. That combination is so likely to fail to converge that it is called *the deadly triad* in [Sutton & Barto](http://incompleteideas.net/book/the-book.html).

Monte Carlo control methods do not suffer from this bias, as each update is made using a true sample of what Q(s,a) should be. However, Monte Carlo methods can suffer from high variance, which means more samples are required to achieve the same degree of learning compared to TD.

In practice, TD learning appears to learn more efficiently if the problems with *the deadly triad* can be overcome. Recent results using experience replay and staged "frozen" copies of estimators provide work-arounds that address problems - e.g. that is how DQN learner for Atari games was built.

There is also a middle ground between TD and Monte Carlo. It is possible to construct a generalised method that combines trajectories of different lengths - from single-step TD to complete episode runs in Monte Carlo - and combine them. The most common variant of this is TD(λλ) learning, where λλ is a parameter from 00 (effectively single-step TD learning) to 11 (effectively Monte Carlo learning, but with a nice feature that it can be used in continuous problems). Typically, a value between 00 and 11 makes the most efficient learning agent - although like many hyperparameters, the best value to use depends on the problem.

If you are using a value-based method (as opposed to a policy-based one), then TD learning is generally used more in practice, or a TD/MC combination method such as TD(λ) can be even better.

In terms of "practical advantage" for MC? Monte Carlo learning is conceptually simple, robust and easy to implement, albeit often slower than TD. I would generally not use it for a learning controller engine (unless in a hurry to implement something for a simple environment), but I would seriously consider it for policy evaluation in order to compare multiple agents for instance - that is due to it being an unbiased measure, which is important for testing.



