**Monte-Carlo Model Free Predictor:**

Hyper-Parameters: discount factor, number of episodes, epsilon (for epsilon greedy policy)

Inputs: environment, initial sample policy (another behavioural policy if using an off-policy method) and hyper parameters

Outputs: policy, Q(s,a) (if needed)

Environment should consist all parameters of the world you wish to simulate. For example, for a grid world, the environment should specify the list of actions one can take in a particular state, free spaces, obstacle locations, dynamic object behaviour if any, goal locations, transition function and rewards functions. However, for a Monte-Carlo Model Free Predictor when given a (state, action) pair, the environment is allowed to return only the next state, reward for reaching that state and if next state is some sort of terminal / goal state or not, but no transition probabilities. In short, the environment and agent are isolated in every way except for this simple request-response relationship. Monte-Carlo methods in general have the characteristic of sampling from given probability distribution and averaging over these sampled values to estimate the expected values. However, for obtaining good estimates of these expected values one has to sample a lot of values. In the reinforcement learning paradigm, one has to generate a large amount of episodes to come to a good estimate of the expected value for each state or even (state, action) pair. Therefore, it faces scalability issues as the number of states or actions increases.

Monte-Carlo methods have two variants: on-policy and off-policy. The basic difference between these two is that, the on-policy variant uses one policy to generate samples, do evaluation and updation, whereas the off-policy variant employs two different policies: one for exploring the state space and generate samples, known as the behavioural policy and one for the evaluation and updation, known as the target policy. The behavioural policy is mostly an epsilon-soft policy or some policy that encourages exploration of the state and action space, while the target policy is deterministic. It has been proven that the on-policy variant converges faster than the off-policy variant.

**TD Learning:**

Hyper-Parameters: discount factor, number of episodes, epsilon (for epsilon greedy policy), learning rate

Inputs: environment, sample Q values (another behavioural policy if using an off-policy method) and hyper parameters

Outputs: policy, Q(s,a) (if needed)

TD or Temporal Difference Learning consists of a set of algorithms that deals with evaluating and updating the state or action values after each steps within an episode. TD methods combines the advantages of both dynamic programming (DP) methods and Monte-Carlo methods by employing bootstrapping (estimate values based on other estimates) and also an incremental update with each step. So unlike Monte-Carlo methods, TD methods do not wait till the end of the episode to update values and unlike dynamic programming TD methods do not require the model of the environment and next-state probability distributions. Two main algorithms under TD are: SARSA and Q-Learning. The major difference between these two is that SARSA is an on-policy method while Q-Learning is an off-policy method.

In SARSA, a random state *S* and set of Q values are initialized based on which an action *A* is chosen which in turn results in a reward *R* and a next state *S'.* Since, the Q value is being learnt we require the next action as well to evaluate and update the current state-action value and by consequence, the policy. So the next action *A'* is also chosen based on the current Q values. This sequence of state, action, reward, next state and next action is where the name SARSA comes from.The reason SARSA is a on-policy method is because, it is the initialized set of Q values on which the policy is based on, that is continually used to generate each step in an episode and being updated after each step.

In Q-Learning, a random state *S,* a set of Q values and an exploratory policy are initialized. The exploratory policy is used to choose an action *A*, which in turn results in a next state *S'.* However, the next action is chosen as the action that maximizes the Q value for the state *S'.* This Q value is used to update the current state's Q value. In the next step, again the exploratory policy is used to generate an action while the Q values are referred to evaluate and update the current state's Q value. So in short, it uses an exploratory policy to explore the state and action space while it learns optimal Q values on which the target policy is based on, thereby making it an off-policy method.

A slightly altered variant is the Expected SARSA algorithm, where instead of choosing the Q value of next state and action pair according to the current policy, it computes the weighted sum of all Q values of the next state weighted by the action probabilities P (a | s). In situations where the randomness comes from only the policy i.e action probabilities Expected SARSA is proven to have an empirical advantage over SARSA and Q Learning. One reason is that in such situations, the learning rate can be directly set to 1.0 without the algorithm suffering from any degradation of asymptotic performance. Hence, the Expected SARSA will converge much faster than SARSA or Q Learning. This algorithm also has the off-policy variant where a separate policy is used as a behavioural policy to choose actions and the weighted sum of Q values of the target policy is used for updation.

Maximization bias is a phenomenon that causes the algorithm to learn the Q values of sub-optimal actions or favour sub-optimal actions more often than is ideal. This is mainly a side-effect of using maximization either implicitly in the form of epsilon-greedy action selection or explicitly as in the off-policy case where we choose directly the value of the action that maximizes the Q value. But the reason it exists is because we use the same set of Q values to choose our actions and update our values. One solution that is proposed is, Double Q Learning, where two sets of Q values are learnt by dividing the set of steps taken into two sets and using them to learn the two sets of Q values. So the basic idea behind this algorithm is to use one set of Q values to choose the maximizing action and use the other Q function to return the Q value for earlier computed maximizing action. Therefore, only one estimate is updated on each step based on the other set of Q values thereby learning to choose action in a completely unbiased fashion. Although it doubles the memory requirements, it does not affect the computational requirements in each step.