UNIT-5

Reinforcement Learning: Single State Case: K-Armed Bandit, Elements of Reinforcement learning, Model based Learning, Temporal Difference learning, Generalizing from examples. [TB-1]

In reinforcement learning, the learner is a decision-making agent that takes actions in an environment and receives reward (or penalty) for its actions in trying to solve a problem. After a set of trial-anderror runs, it should learn the best policy, which is the sequence of actions that maximize the total reward**.** Reinforcement learning (RL) is a general framework where agents learn to perform actions in an environment so as to maximize a reward. The two main components are the environment, which represents the problem to be solved, and the agent, which represents the learning algorithm.

Reinforcement Learning is a feedback-based Machine learning technique in which an agent learns to behave in an environment by performing the actions and seeing the results of actions. **For each good action, the agent gets positive feedback, and** for **each bad action, the agent gets negative feedback or penalty.**

In Reinforcement Learning, the agent learns automatically using feedbacks without any labeled data, unlike supervised learning. Since there is no labeled data, so the agent is bound to learn by its experience only. RL solves a specific type of problem where decision making is sequential, and the goal is long-term, such as **game-playing, robotics**, etc.

The agent interacts with the environment and explores it by itself. The primary goal of an agent in reinforcement learning is to improve the performance by getting the maximum positive rewards.

Types of Reinforcement learning

There are mainly two types of reinforcement learning, which are:

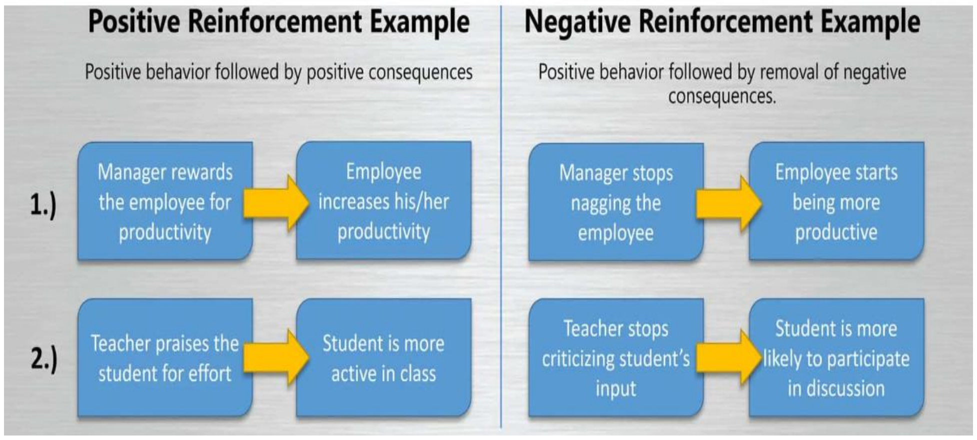
- **Positive Reinforcement**
- **Negative Reinforcement**

Positive Reinforcement:

The positive reinforcement learning means adding something to increase the tendency that expected behavior would occur again. It impacts positively on the behavior of the agent and increases the strength of the behavior. This type of reinforcement can sustain the changes for a long time, but too much positive reinforcement may lead to an overload of states that can reduce the consequences.

Negative Reinforcement:

The negative reinforcement learning is opposite to the positive reinforcement as it increases the tendency that the specific behavior will occur again by avoiding the negative condition. It can be more effective than the positive reinforcement depending on situation and behavior, but it provides reinforcement only to meet minimum behavior.


Positive Reinforcement Example
Positive behavior followed by positive consequences
1.)
Manager rewards the employee for productivity
Employee increases his/her productivity
2.)
Teacher praises the student for effort
Student is more active in class
Negative Reinforcement Example
Positive behavior followed by removal of negative consequences.
Manager stops nagging the employee
Employee starts being more productive
Teacher stops criticizing student's input
Student is more likely to participate in discussion

INTRODUCTION: -

Let us say we want to build a machine that learns to play chess. In this case we cannot use a supervised learner for two reasons. First, it is very costly to have a teacher that will take us through many games and indicate us the best move for each position. Second, in many cases, there is no such thing as the best move; the goodness of a move depends on the moves that follow. A single move does not count; a sequence of moves is good if after playing them we win the game. The only feedback is at the end of the game when we win or lose the game.

Another example is a robot that is placed in a maze. The robot can move in one of the four compass directions and should make a sequence of movements to reach the exit. As long as the robot is in the maze, there is no feedback and the robot tries many moves until it reaches the exit and only then does it get a reward. In this case there is no opponent, but we can have a preference for shorter trajectories, implying that in this case we play against time.

These two applications have a number of points in common: there is a decision maker, called the agent that is placed in an environment (see figure 18.1). In chess, the game-player is the decision maker and the environment is the board; in the second case, the maze is the environment of the robot. At any time, the environment is in a certain state that is one of a set of possible states—for example, the state of the board, the position of the robot in the maze. The decision maker has a set of actions possible: legal movement of pieces on the chess board, movement of the robot in possible directions without hitting the walls, and so forth. Once an action is chosen and taken, the state changes. The solution to the task requires a sequence of actions, and we get feedback, in the form of a reward rarely, generally only when the complete sequence is carried out. The reward defines the problem and is necessary if we want a learning agent. The learning agent learns the best sequence of actions to solve a problem where "best" is quantified as the sequence of actions that has the maximum cumulative reward. Such is the setting of reinforcement learning.

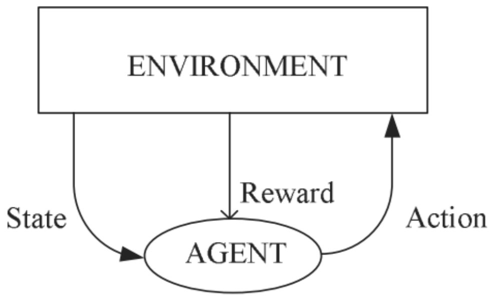


Figure 18.1 The agent interacts with an environment. At any state of the environment, the agent takes an action that changes the state and returns a reward.

Reinforcement learning is different from the learning methods we discussed before in a number of respects. It is called "learning with a critic," as opposed to learning with a teacher which we have in supervised learning. A critic differs from a teacher in that it does not tell us what to do but only how well we have been doing in the past; the critic never informs in advance. The feedback from the critic is scarce and when it comes, it comes late. This leads to the credit assignment problem. After taking several actions and getting the reward, we would like to assess the individual actions we did in the past and find the moves that led us to win the reward so that we can record and recall them later on. As we see shortly, what a reinforcement learning program does is that it learns to generate an internal value for the intermediate states or actions in terms of how good they are in leading us to the goal and getting us to the real reward. Once such an internal reward mechanism is learned, the agent can just take the local actions to maximize it.

SINGLE STATE CASE: K-ARMED BANDIT

We start with a simple example. The K-armed bandit is a hypothetical slot machine with K levers. The action is to choose and pull one of the levers, and we win a certain amount of money that is the reward associated with the lever (action). The task is to decide which lever to pull to maximize the reward. This is a classification problem where we choose one of K. If this were supervised learning, then the teacher would tell us the correct class, namely, the lever leading to maximum earning. In this case of reinforcement learning, we can only try different levers and keep track of the best. This is a simplified reinforcement learning problem because there is only one state, or one slot machine, and we need only decide on the action. Another reason why this is simplified is that we immediately get a reward after a single action; the reward is not delayed, so we immediately see the value of our action.

Let us say Q(a) is the value of action a. Initially, Q(a) = 0 for all a. When we try action a, we get reward $r_a \geq 0$. If rewards are deterministic, we always get the same r_a for any pull of a and in such a case, we can just set $Q(a) = r_a$. If we want to exploit, once we find an action a such that Q(a) > 0, we can keep choosing it and get r_a at each pull. However, it is quite possible that there is another lever with a higher reward, so we need to explore.

We can choose different actions and store Q(a) for all a. Whenever we want to exploit, we can choose the action with the maximum value, that is,

$$(18.1) \qquad \text{choose } a^* \text{ if } Q(a^*) = \max_a Q(a)$$

If rewards are not deterministic but stochastic, we get a different reward each time we choose the same action. The amount of the reward is defined by the probability distribution p(r|a). In such a case, we define $Q_t(a)$ as the estimate of the value of action a at time t. It is an average of all rewards received when action a was chosen before time t. An online update can be defined as

$$(18.2) \qquad Q_{t+1}(a) \leftarrow Q_t(a) + \eta[r_{t+1}(a) - Q_t(a)]$$

where $r_{t+1}(a)$ is the reward received after taking action a at time $(t+1)^{st}$ time.

Note that equation 18.2 is the delta rule: η is the learning factor, r_{t+1} is the desired output, and Q_t (a) is the current prediction. $Q_{t+1}(a)$ is the expected value of action a at time t + 1 and converges to the mean of p(r|a) as t increases.

The full reinforcement learning problem generalizes this simple case in a number of ways. First, we have several states. This corresponds to having several slot machines with different reward probabilities, $p(r|s_i, a_j)$, and we need to learn $Q(s_i, a_j)$, which is the value of taking action a_j when in state s_i. Second, the actions affect not only the reward but also the next state, and we move from one state to another. Third, the rewards are delayed and we need to be able to estimate immediate values from delayed rewards.

ELEMENTS OF REINFORCEMENT LEARNING: -

There are four main elements of Reinforcement Learning, which are given below:

- Policy
- Reward Signal
- Value Function
- Model of the environment

1. **Policy: -**
   A policy can be defined as a way how an agent behaves at a given time. It maps the perceived states of the environment to the actions taken on those states. A policy is the core element of the RL as it alone can define the behavior of the agent. In some cases, it may be a simple function or a lookup table, whereas, for other cases, it may involve general computation as a search process. It could be deterministic or a stochastic policy:
   For deterministic policy: a = π(s)

For stochastic policy: π(a | s) = P[At =a | St = s]

2. **Reward Signal**: -
   The goal of reinforcement learning is defined by the reward signal. At each state, the environment sends an immediate signal to the learning agent, and this signal is known as a reward signal. These rewards are given according to the good and bad actions taken by the agent. The agent's main objective is to maximize the total number of rewards for good actions. The reward signal can change the policy, such as if an action selected by the agent leads to low reward, then the policy may change to select other actions in the future.

3. **Value Function**: The value function gives information about how well the situation and action are and how much reward an agent can expect. A reward indicates the immediate signal for each good and bad action, whereas a value function specifies the good state and action for the future. The value function depends on the reward as, without reward, there could be no value. The goal of estimating values is to achieve more rewards.

4. **Model**: The last element of reinforcement learning is the model, which mimics the behavior of the environment. With the help of the model, one can make inferences about how the environment will behave. Such as, if a state and an action are given, then a model can predict the next state and reward. The model is used for planning, which means it provides a way to take a course of action by considering all future situations before actually experiencing those situations. The approaches for solving the RL problems with the help of the model are termed as the model-based approach. Comparatively, an approach without using a model is called a model-free approach.

The learning decision maker is called the agent. The agent interacts with the environment that includes everything outside the agent. The agent has sensors to decide on its state in the environment and takes an action that modifies its state. When the agent takes an action, the environment provides a reward. Time is discrete as t = 0, 1, 2,..., and $s_t \in S$ denotes the state of the agent at time t where S is the set of all possible states. $a_t \in \mathcal{A}(s_t)$ denotes the action that the agent takes at time t where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t. When the agent in state s_t takes the action at, the clock ticks, reward $r_{t+1} \in \mathcal{R}$ is received, and the agent moves to the next state, s_{t+1}. The problem is modeled using a *Markov decision process (MDP)*. The reward and next state are sampled from their respective probability distributions, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. Note that what we have is a Markov system where the state and reward in the next time step depend only on the current state and action. In some applications, reward and next state are deterministic, and for a certain state and action taken, there is one possible reward value and next state.

Markov Decision Process or MDP, is used to **formalize the reinforcement learning problems**. If the environment is completely observable, then its dynamic can be modeled as a **Markov Process**. In MDP, the agent constantly interacts with the environment and performs actions; at each action, the environment responds and generates a new state.

MDP contains a tuple of four elements (S, A, P_a, R_a):

A set of finite States S

- A set of finite Actions A
- Rewards received after transitioning from state S to state S', due to action a.
- Probability P_a.

MDP uses **Markov property**, and to better understand the MDP,

Markov Property:
It says that "If the agent is present in the current state S1, performs an action a1 and move to the state s2, then the state transition from s1 to s2 only depends on the current state and future action and states do not depend on past actions, rewards, or states."

OR

in other words, as per Markov Property, the current state transition does not depend on any past action or state. Hence, MDP is an RL problem that satisfies the Markov property. Such as in a **Chess game, the players only focus on the current state and do not need to remember past actions or states**.

Depending on the application, a certain state may be designated as the initial state and in some applications, there is also an absorbing terminal (goal) state where the search ends; all actions in this terminal state transition to itself with probability 1 and without any reward. The sequence of actions from the start to the terminal state is an ***episode***, or a ***trial***.

The policy, π, defines the agent's behavior and is a mapping from the states of the environment to actions: $\pi: S \rightarrow A$. The policy defines the action to be taken in any state s_t: $a_t = \pi(s_t)$. The value of a policy π, $V^\pi(s_t)$, is the expected cumulative reward that will be received while the agent follows the policy, starting from state s_t.

In the finite-horizon or episodic model, the agent tries to maximize the expected reward for the next T steps:

$$(18.3) \qquad V^\pi(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right]$$

Certain tasks are continuing, and there is no prior fixed limit to the episode. In the infinite-horizon model, there is no sequence limit, but future rewards are discounted:

$$(18.4) \qquad V^\pi(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right]$$

where $0 \le \gamma < 1$ is the discount rate to keep the return finite. If $\gamma = 0$, then only the immediate reward counts. As γ approaches 1, rewards further in the future count more, and we say that the agent becomes more farsighted. γ is less than 1 because there generally is a time limit to the sequence of actions needed to solve the task. The agent may be a robot that runs on a battery. We prefer rewards sooner rather than later because we are not certain how long we will survive.
For each policy π, there is a $V^\pi(s_t)$, and we want to find the optimal policy π^* such that

$$(18.5) \qquad V^*(s_t) = \max_\pi V^\pi(s_t), \forall s_t$$

In some applications, for example, in control, instead of working with the values of states, $V(s_t)$, we prefer to work with the values of state-action pairs, $Q(s_t, a_t)$. $V(s_t)$ denotes how good it is for the agent to be in state s_t, whereas $Q(s_t, a_t)$ denotes how good it is to perform action a_t when in state s_t. We define $Q^*(s_t, a_t)$ as the value, that is, the expected cumulative reward, of action a_t taken in state s_t and then obeying the optimal policy afterward. The value of a state is equal to the value of the best possible action:

$$\begin{aligned} V^*(s_t) &= \max_{a_t} Q^*(s_t, a_t) \\ &= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma V^*(s_{t+1})\right] \end{aligned}$$

$$(18.6) \qquad V^*(s_t) = \max_{a_t} \left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1}) \right)$$

Bellman's equation: -

To each possible next state s_{t+1}, we move with probability $P(s_{t+1}|s_t, a_t)$, and continuing from there using the optimal policy, the expected cumulative reward is $V^*(s_{t+1})$. We sum over all such possible next states, and we discount it because it is one time step later. Adding our immediate expected reward, we get the total expected cumulative reward for action at. We then choose the best of possible actions. Equation 18.6 is known as *Bellman's equation*. Similarly, we can also write

$$(18.7) \qquad Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

Once we have $Q^*(s_t, a_t)$ values, we can then define our policy π as taking the action a_t^*, which has the highest value among all $Q^*(s_t, a_t)$:

$$(18.8) \qquad \pi^*(s_t) : \text{Choose } a_t^* \text{ where } Q^*(s_t, a_t^*) = \max_{a_t} Q^*(s_t, a_t)$$

This means that if we have the $Q^*(s_t, a_t)$ values, then by using a greedy search at each local step we get the optimal sequence of steps that maximizes the cumulative reward.

MODEL-BASED LEARNING: -

We start with model-based learning where we completely know the environment model parameters, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. In such a case, we do not need any exploration and can directly solve for the optimal value function and policy using dynamic programming. The optimal value function is unique and is the solution to the simultaneous equations given in equation 18.6. Once we have the optimal value function, the optimal policy is to choose the action that maximizes the value in the next state:

$$(18.9) \qquad \pi^*(s_t) = \arg\max_{a_t} \left(E[r_{t+1}|s_t, a_t] + \gamma \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) V^*(s_t + 1) \right)$$

1. **Value Iteration: -**
   To find the optimal policy, we can use the optimal value function, and there is an iterative algorithm called value iteration that has been shown to converge to the correct V^* values. Its pseudo code is given in figure 18.2

```
Initialize $V(s)$ to arbitrary values
Repeat
    For all $s \in \mathcal{S}$
        For all $a \in \mathcal{A}$
            $Q(s,a) \leftarrow E[r|s,a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)V(s')$
        $V(s) \leftarrow \max_a Q(s,a)$
Until $V(s)$ converge
```

Figure 18.2 Value iteration algorithm for model-based learning.

We say that the values converged if the maximum value difference between two iterations is less than a certain threshold δ:

$$\max_{s \in \mathcal{S}} |V^{(l+1)}(s) - V^{(l)}(s)| < \delta$$

where l is the iteration counter. Because we care only about the actions with the maximum value, it is possible that the policy converges to the optimal one even before the values converge to their optimal values. Each iteration is $O(|S|^2|A|)$, but frequently there is only a small number $k < |S|$ of next possible states, so complexity decreases to $O(k|S||A|)$.

2. **Policy Iteration: -**
In policy iteration, we store and update the policy rather than doing this indirectly over the values. The pseudo code is given in figure 18.3.

```
Initialize a policy $\pi'$ arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using $\pi$ by
        solving the linear equations
            $V^{\pi}(s) = E[r|s,\pi(s)] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,\pi(s))V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s,a] + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a)V^{\pi}(s'))$
Until $\pi = \pi'$
```

Figure 18.3 Policy iteration algorithm for model-based learning.

The idea is to start with a policy and improve it repeatedly until there is no change. The value function can be calculated by solving for the linear equations. We then check whether we can improve the policy by taking these into account. This step is guaranteed to improve the policy, and when no improvement is possible, the policy is guaranteed to be optimal. Each iteration of this algorithm takes $O(|A||S|^2 + |S|^3)$ time that is more than that of value iteration, but policy iteration needs fewer iterations than value iteration.

TEMPORAL DIFFERENCE LEARNING: -

Model is defined by the reward and next state probability distributions, and as we saw in section 18.4, when we know these, we can solve for the optimal policy using dynamic programming. However, these methods are costly, and we seldom have such perfect knowledge of the environment.

The more interesting and realistic application of reinforcement learning is when we do not have the model. This requires exploration of the environment to query the model. When we explore and get to see the value of the next state and reward, we use this information to update the value of the current state. These algorithms are called temporal difference algorithms because what we do is look at the difference between our current estimate of the value of a state (or a state-action pair) and the discounted value of the next state and the reward received.

1. Exploration Strategies
2. Deterministic Rewards and Actions
3. Nondeterministic Rewards and Actions
4. Eligibility Traces

1. **Exploration Strategies: -**

   To explore, one possibility is to use ε -greedy search where with probability ε, we choose one action uniformly randomly among all possible actions, namely, explore, and with probability 1 − ε, we choose the best action, namely, exploit. We do not want to continue exploring indefinitely but start exploiting once we do enough exploration; for this, we start with a high ε value and gradually decrease it. We need to make sure that our policy is soft, that is, the probability of choosing any action $a \in \mathcal{A}$ in state $s \in S$ is greater than 0.

   We can choose probabilistically, using the softmax function to convert values to probabilities

$$(18.10) \qquad P(a|s) = \frac{\exp Q(s,a)}{\sum_{b\in\mathcal{A}} \exp Q(s,b)}$$

   and then sample according to these probabilities. To gradually move from exploration to exploitation, we can use a “temperature” variable T and define the probability of choosing action a as

$$(18.11) \qquad P(a|s) = \frac{\exp[Q(s,a)/T]}{\sum_{b\in\mathcal{A}} \exp[Q(s,b)/T]}$$

   When T is large, all probabilities are equal and we have exploration. When T is small, better actions are favored. So the strategy is to start with a large T and decrease it gradually, a procedure named annealing, which in this case moves from exploration to exploitation smoothly in time.

2. **Deterministic Rewards and Actions: -**

   In model-free learning, we first discuss the simpler deterministic case, where at any state-action pair, there is a single reward and next state possible. In this case, equation 18.7

$$(18.7) \qquad Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$$

   reduces to

$$(18.12) \qquad Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

   and we simply use this as an assignment to update $Q(s_t, a_t)$. When in state s_t, we choose action at by one of the stochastic strategies we saw earlier, which returns a reward r_{t+1} and takes us to state s_{t+1}. We then update the value of previous action as

$$(18.13) \qquad \hat{Q}(s_t, a_t) \leftarrow r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$$

where the hat denotes that the value is an estimate. $\hat{Q}(s_{t+1}, a_{t+1})$ is a later value and has a higher chance of being correct. We discount this by γ and add the immediate reward (if any) and take this as the new estimate for the previous $\hat{Q}(s_t, a_t)$. This is called a *backup* because it can be viewed as taking the estimated value of an action in the next time step and "backing it up" to revise the estimate for the value of a current action.

Initially all $\hat{Q}(s_t, a_t)$ are 0, and they are updated in time as a result of trial episodes. Let us say we have a sequence of moves and at each move, we use equation 18.13 to update the estimate of the Q value of the previous state-action pair using the Q value of the current state-action pair. In the intermediate states, all rewards and therefore values are 0, so no update is done. When we get to the goal state, we get the reward r and then we can update the Q value of the previous state-action pair as γr. As for the preceding state-action pair, its immediate reward is 0 and the contribution from the next state-action pair is discounted by γ because it is one step later. Then in another episode, if we reach this state, we can update the one preceding that as $\gamma^2 r$, and so on. This way, after many episodes, this information is backed up to earlier state-action pairs. Q values increase until they reach their optimal values as we find paths with higher cumulative reward, for example, shorter paths, but they never decrease

3. **Nondeterministic Rewards and Actions: -**

If the rewards and the result of actions are not deterministic, then we have a probability distribution for the reward $p(r_{t+1}|s_t, a_t)$ from which rewards are sampled, and there is a probability distribution for the next state $P(s_{t+1}|s_t, a_t)$. These help us model the uncertainty in the system that may be due to forces we cannot control in the environment: for instance, our opponent in chess, the dice in backgammon, or our lack of knowledge of the system. For example, we may have an imperfect robot which sometimes fails to go in the intended direction and deviates, or advances shorter or longer than expected.

In such a case, we have

$$(18.14) \qquad Q(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

Q learning algorithm (off – policy method): -

We cannot do a direct assignment in this case because for the same state and action, we may receive different rewards or move to different next states. What we do is keep a running average. This is known as the Q learning algorithm:

$$(18.15) \qquad \hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

We think of $r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$ values as a sample of instances for each (s_t, a_t) pair and we would like $\hat{Q}(s_t, a_t)$ to converge to its mean. As usual η is gradually decreased in time for convergence, and it has been shown that this algorithm converges to the optimal Q^* values

The pseudo code of the Q learning algorithm is given in figure 18.5.

```
Initialize all $Q(s,a)$ arbitrarily
For all episodes
    Initalize $s$
    Repeat
        Choose $a$ using policy derived from $Q$, e.g., $\epsilon$-greedy
        Take action $a$, observe $r$ and $s'$
        Update $Q(s,a)$:
            $Q(s,a) \leftarrow Q(s,a) + \eta(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
        $s \leftarrow s'$
    Until $s$ is terminal state
```

Figure 18.5 Q learning, which is an off-policy temporal difference algorithm.

We can also think of equation 18.15 as reducing the difference between the current Q value and the backed-up estimate, from one time step later. Such algorithms are called temporal difference (TD) algorithms.

On-policy method(Sarsa algorithm): -

SARSA stands for State Action Reward State action, which is an on-policy temporal difference learning method. The on-policy control method selects the action for each state while learning using a specific policy. The goal of SARSA is to calculate the Q π (s, a) for the selected current policy π and all pairs of (s-a). The main difference between Q-learning and SARSA algorithms is that unlike Q-learning, the maximum reward for the next state is not required for updating the Q-value in the table.

In SARSA, new action and reward are selected using the same policy, which has determined the original action. The SARSA is named because it uses the quintuple Q(s, a, r, s', a'). Where,
s: original state
a: Original action
r: reward observed while following the states
s' and a': New state, action pair.

This is an off-policy method as the value of the best next action is used without using the policy. In an on-policy method, the policy is used to determine also the next action. The on-policy version of Q learning is the Sarsa algorithm whose pseudo code is given in figure 18.6.

```
Initialize all $Q(s,a)$ arbitrarily
For all episodes
    Initalize $s$
    Choose $a$ using policy derived from $Q$, e.g., $\epsilon$-greedy
    Repeat
        Take action $a$, observe $r$ and $s'$
        Choose $a'$ using policy derived from $Q$, e.g., $\epsilon$-greedy
        Update $Q(s,a)$:
            $Q(s,a) \leftarrow Q(s,a) + \eta(r + \gamma Q(s',a') - Q(s,a))$
        $s \leftarrow s'$, $a \leftarrow a'$
    Until $s$ is terminal state
```

Figure 18.6 Sarsa algorithm, which is an on-policy version of Q learning.

We see that instead of looking for all possible next actions a' and choosing the best, the on-policy Sarsa uses the policy derived from Q values to choose one next action a' and uses its Q value to calculate the temporal difference. On-policy methods estimate the value of a policy while using it to take actions. In off-policy methods, these are separated, and the policy used to generate behavior, called the ***behavior policy***, may in fact be different from the policy that is evaluated and improved, called the ***estimation policy***.

The same idea of temporal difference can also be used to learn V (s) values, instead of Q(s, a). TD learning uses the following update rule to update a state value:

$$(18.16) \qquad V(s_t) \leftarrow V(s_t) + \eta[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

This again is the delta rule where $r_{t+1}+\gamma V(s_{t+1})$ is the better, later prediction and $V(s_t)$ is the current estimate. Their difference is the temporal difference, and the update is done to decrease this difference. The update factor η is gradually decreased, and TD is guaranteed to converge to the optimal value function $V^*(s)$.

4. **Eligibility Traces: -**

The previous algorithms are one-step—that is, the temporal difference is used to update only the previous value (of the state or state-action pair). An eligibility trace is a record of the occurrence of past visits that enables us to implement temporal credit assignment, allowing us to update the values of previously occurring visits as well. We discuss how this is done with Sarsa to learn Q values; adapting this to learn V values is straightforward.

To store the eligibility trace, we require an additional memory variable associated with each state-action pair, e(s, a), initialized to 0. When the state-action pair (s, a) is visited, namely, when we take action a in state s, its eligibility is set to 1; the eligibilities of all other state-action pairs are multiplied by γλ. $0 \le \lambda \le 1$ is the trace decay parameter.

$$(18.17) \qquad e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t, \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases}$$

We remember that in Sarsa, the temporal error at time t is

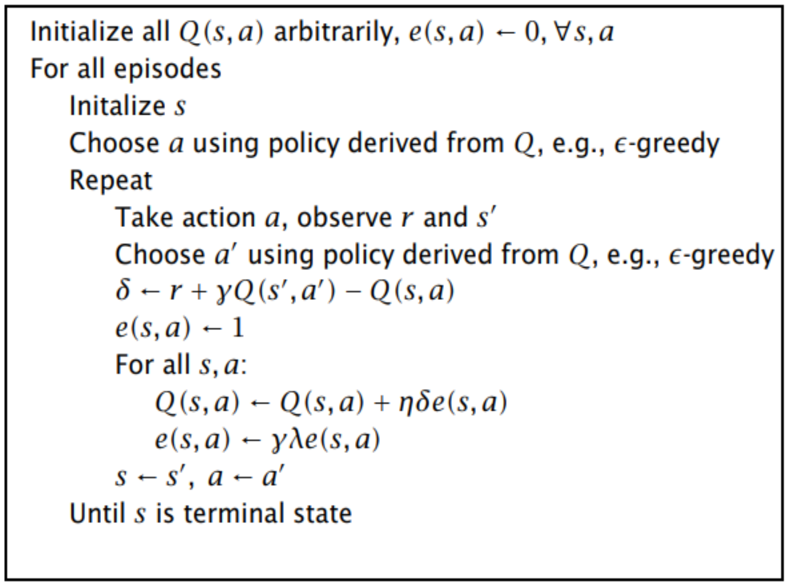
$$(18.18) \qquad \delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

In Sarsa with an eligibility trace, named Sarsa(λ), all state-action pairs are updated as

$$(18.19) \qquad Q(s,a) \leftarrow Q(s,a) + \eta\delta_t e_t(s,a), \ \forall s,a$$

Sarsa(λ) algorithm: -

In online updating, all eligible values are updated immediately after each step; in offline updating, the updates are accumulated and a single update is done at the end of the episode. Online updating takes more time but converges faster. The pseudo code for Sarsa(λ) is given in figure 18.8. Q(λ) and TD(λ) algorithms can similarly be derived



```
Initialize all $Q(s,a)$ arbitrarily, $e(s,a) \leftarrow 0, \forall s,a$
For all episodes
    Initalize $s$
    Choose $a$ using policy derived from $Q$, e.g., $\epsilon$-greedy
    Repeat
        Take action $a$, observe $r$ and $s'$
        Choose $a'$ using policy derived from $Q$, e.g., $\epsilon$-greedy
        $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
        $e(s,a) \leftarrow 1$
        For all $s,a$:
            $Q(s,a) \leftarrow Q(s,a) + \eta\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        $s \leftarrow s'$, $a \leftarrow a'$
    Until $s$ is terminal state
```

Figure 18.8 Sarsa(λ) algorithm.

GENERALIZATION: -

Until now, we assumed that the Q(s, a) values (or V (s), if we are estimating values of states) are stored in a lookup table, and the algorithms we considered earlier are called tabular algorithms. There are a number of problems with this approach:

1. When the number of states and the number of actions is large, the size of the table may become quite large.
2. States and actions may be continuous, for example, turning the steering wheel by a certain angle, and to use a table, they should be discretized which may cause error.
3. When the search space is large, too many episodes may be needed to fill in all the entries of the table with acceptable accuracy.

Instead of storing the Q values as they are, we can consider this a regression problem. This is a supervised learning problem where we define a regressor Q(s, a|**θ**), taking s and a as inputs and parameterized by a vector of parameters, **θ**, to learn Q values. For example, this can be an artificial neural network with s and a as its inputs, one output, and **θ** its connection weights.

To be able to train the regressor, we need a training set. In the case of Sarsa(0), we saw before that we would like $Q(s_t, a_t)$ to get close to $r_{t+1}+\gamma Q(s_{t+1},a_{t+1})$. So, we can form a set of training samples where the input is the state-action pair (s_t, a_t) and the required output is $r_{t+1}+\gamma Q(s_{t+1},a_{t+1})$. We can write the squared error as

$$(18.20) \qquad E^t(\boldsymbol{\theta}) = [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]^2$$

Training sets can similarly be defined for Q(0) and TD(0), where in the latter case we learn V (s), and the required output is $r_{t+1} - \gamma V(s_{t+1})$. Once such a set is ready, we can use any supervised learning algorithm for learning the training set.

If we are using a gradient-descent method, as in training neural networks, the parameter vector is updated as

$$(18.21) \qquad \Delta\boldsymbol{\theta} = \eta[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]\nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t)$$

This is a one-step update. In the case of Sarsa(λ), the eligibility trace is also taken into account:

$$(18.22) \qquad \Delta\boldsymbol{\theta} = \eta\delta_t \boldsymbol{e}_t$$

where the temporal difference error is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

and the vector of eligibilities of parameters are updated as

$$(18.23) \qquad \boldsymbol{e}_t = \gamma\lambda \boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t)$$

with e_0 all zeros. In the case of a tabular algorithm, the eligibilities are stored for the state-action pairs because they are the parameters (stored as a table). In the case of an estimator, eligibility is associated with the parameters of the estimator. We also note that this is very similar to the momentum method for stabilizing back propagation. The difference is that in the case of momentum previous weight changes are remembered, whereas here previous gradient vectors are remembered. Depending on the model used for $Q(s_t, a_t)$, for example, a neural network, we plug its gradient vector in equation 18.23.

UNIT WISE IMPORTANT QUESTIONS: -

1. What is Reinforcement Learning? Explain K-Armed Bandit.
2. Discuss on Elements of Reinforcement learning
3. Identify an example of a reinforcement learning application thatcan be modeled by a POMDP. Define the states, actions, observations, and reward.
4. Explain Value iteration algorithm for model-based learning
5. Explain Policy iteration algorithm for model-based learning
6. Identify different Algorithms in Temporal Difference Learning
7. Explain Deterministic Rewards and Actions
8. Explain Non Deterministic Rewards and Actions
9. Explain Q learning off-policy temporal difference algorithm.
10. Explain Sarsa algorithm an on-policy version of Q learning
11. Outline the concept of `Generalizing from examples.