Reinforcement learning

Temporal Difference Learning

- Model is defined by the reward and next state probability distributions, 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. We first discuss how this exploration is done and later see model-free learning algorithms for deterministic and nondeterministic cases. Though we are not going to assume full knowledge of the environment model, we will however require that it be stationary.
- 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

- To explore, one possibility is to use ǫ-greedy search where with probability ǫ, we choose one action uniformly randomly among all possible actions, namely, explore, and with probability 1 − ǫ, 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 ǫ value and gradually decrease it. We need to make sure that our policy is soft, that is, the probability of choosing any action a ∈ A in state s ∈ S is greater than 0.
- We can choose probabilistically, using the softmax function to convert values to probabilities

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

- 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 an as

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

- When T is large, all probabilities are equal and we have exploration. 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

- the simpler deterministic case, where at any state-action pair, there is a single reward and the next state possible.

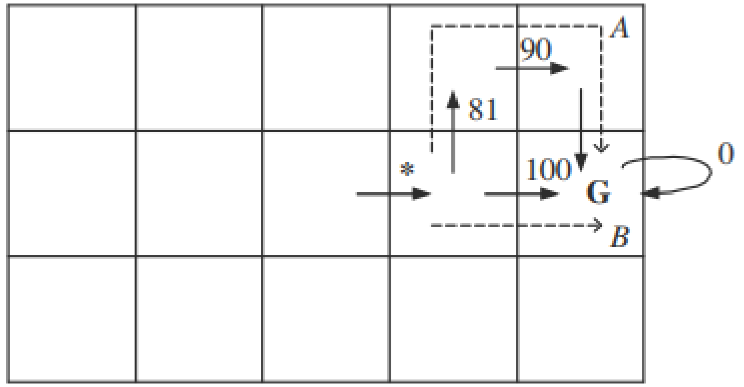
$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$$

- and we simply use this as an assignment to update Q(st, at). When in state st, we choose action by one of the stochastic strategies we saw earlier, which returns a reward rt+1 and takes us to state st+1. We then update the value of the previous action as

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

- where the hat denotes that the value is an estimate. ˆQ(st+1, at+1) is a later value and has a higher chance of being correct. We discount this by γ, add the immediate reward (if any), and take this as the new estimate for the previous ˆQ(st, at).

- 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.



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(rt+1|st, at) from which rewards are sampled, and there is a probability distribution for the next state P(st+1|st, at).
- 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

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

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

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

```
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
```

- 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.
- 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 pseudocode.

```
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
```

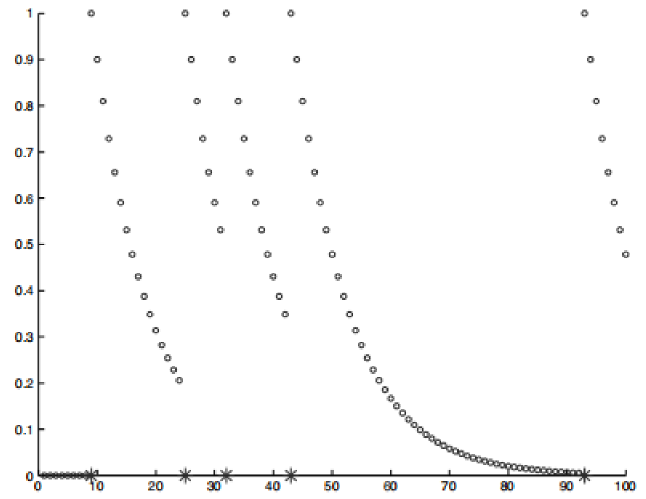
- Sarsa converges with probability 1 to the optimal policy and state action values if a GLIE policy is employed to choose actions. A GLIE (greedy in the limit with infinite exploration) policy is where (1) all state-action pairs are visited an infinite number of times, and (2) the policy converges in the limit to the greedy policy (which can be arranged, e.g., with ǫ-greedy policies by setting ǫ = 1/t).
- 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:

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

- This again is the delta rule where rt+1 + γV (st+1) is the better, later prediction and V (st) 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 in state s, its eligibility is set to 1; the eligibilities of all other state-action pairs are multiplied by γλ. 0 ≤ λ ≤ 1 is the trace decay parameter.

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

- If a state-action pair has never been visited, its eligibility remains 0; if it has been, as time passes and other state-actions are visited, its eligibility decays depending on the value of γ and λ.

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

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

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

- This updates all eligible state-action pairs, where the update depends on how far they have occurred in the past. The value of λ defines the temporal credit: if λ = 0, only a one-step update is done.

```
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
```

- As λ gets closer to 1, more of the previous steps are considered. When $\lambda = 1$, all previous steps are updated and the credit given to them falls only by γ per step.
- 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.