Q Learning Algorithm

May 22, 2024

1 Q-Learning Algorithm

The algorithm iteself is not so much hard to implement, but the essence (both mathamatcially derived & intuition) took 20 years of development. Understanding Q-Leaning is crucial because this is the key to essentially most of teh **search side** of reiforcement learning, which is again also heavily used even framing reinforcement learning problem (specifically policy) as an optimization problem. ***reinforcement learning is both Search & Optimization***. **Notice that both search and optimization is already an abstraction on the real problem, one frame it in a infinite tree's scope, one frame it in a landscape's scope**.

1. Formal proofs fcome from mathamatical formulation of the complex space and intuitive ideas come from the coed itself.
2. When interpreting RL, first you need to see what the goal is, then you need to think about what is it mathamatically doing and how does it root back to bellman update.

Mathematical speaking:

Remember that all ***Q-learning idea*** is rooted back to ***TD with V(s) given pi***, which is all the way rooted back to ***Bellman expected/max update***. We know that bellman max update or expected update can be deem as walking on the space of R^n where the vector $\vec{V}$ stores one possbile reality of the MDP, we know that it is a contraction mapping and the max update converges to a single reality, the singularity, that have all the best state. Now moving to TD update, it can also be deemed as walking on the R^n space but just incrementally updating each entries by one instance learning, it is still a contractile mapping, same can be applied to Q-learning with max operator, maximization garanteed at convergence.

$$
\begin{aligned}
V(s_0) &= R(s_0) + \gamma \sum_{s'} P(s'|s_0, \pi(s_0))V(s') \\
V(s_1) &= R(s_1) + \gamma \sum_{s'} P(s'|s_1, \pi(s_1))V(s') \\
&\vdots \\
V(s_n) &= R(s_n) + \gamma \sum_{s'} P(s'|s_n, \pi(s_n))V(s')
\end{aligned}
$$

Non-mathematical derivation wise:

Remember back in Bellman update, minimum requirement max next one step, max all step. In RL, you have to visit the same state-action multiple times, to spend a bit more time, to learn from it.

- Bellman Max Update: always choose action that maximize the current state value when looking at the recurrence all next state expectation, thus maximizing.
  - **Bellman Expected Update**:

$$V^{\pi}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s,a)V(s')$$

  - **Bellman Max Update**:

$$V(s) \leftarrow R(s) + \gamma(max_{a \in A(s)} \sum_{s'} P(s'|s,a)V(s'))$$

- Temporal Difference Evaluation: modifying bellman update with a sampling and incrementally learning flavor, update without perfect knowledge
  - V^{new} = current V^{old} + (new one instance using V^{old} - current V^{old})
  - **Incremental Update**:

$$\mu_k \leftarrow \mu_{k-1} + \alpha_k(x_k - \mu_{k-1})$$

  - **TD Evaluation**:

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha(R(s) + \gamma V^{\pi}(s') - V^{\pi}(s))$$

- Q-Learning: the goal is to have a Q lookup table that tells you at a given state, what is the best globally, then we can use epslon greedy policy. It is a one instance learning looking at the one instance of next (state-action) pair from Q^{old}. Since we want to learn about the best/optimal path, we don't want to update towards a wrong understanding, especially when this (state-action) pair actually does have high rewards. To avoid the problem of agent just sampling a bad rollout from the environment and treat it as the understanding of the state, we always takes the max operator among all Q in memory to find the best reward of a given (state-action) pair.

  1. Rooted in MDP's mathamatical structure, current understanding can be update by a new instance (current reward + current look up table's next value after this action transition), this constitutes a new instance understanding and then compare with all older understanding of the current state to find the max of them al. Remember this unfolds recurrently.

     ```
     reward + self.DISCOUNT * max_next_q - self.Q_values[state][action]
     ```

  2. A natural question would be that, during initialization, the recurrence sample system may not have enough sample yet, so the next_state's value, which is then recurrent to another state and so on, may not have value to it yet. The **global state** is istantiated to be random, the value knows is just all random, the **conceptual global vector in vector space** is randomly initiated. As learning goes on, because of the learning rate, the initial starting point doesn't really matter and correctness converges

  3. This alo illustrates that RL agent need to really visit one state multiple times to actually learn and understand the state and the bigger global state. There will be more of a "memory" in knowing what is more optimal path to take.

– **Q's Perspective on Bellman Update**:

$$V(s) = max_a Q(s, a)$$

$$Q(s, a) \leftarrow \sum_{s'} P(s'|s, a)(R(s) + \gamma max_{a'} Q(s', a'))$$

– **Q-Learning**:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a))$$

[ ]:
```
def Q_run(self, num_simulation, tester=False, epsilon=0.4):
    '''running Q learning'''

    # Perform num_simulation rounds of simulations in each cycle of the overall␣
 ↪game loop
    for simulation in range(num_simulation):

        # Do not modify the following three lines
        if tester:
            self.tester_print(simulation, num_simulation, "Q")

        # reset the game & go to current state
        self.simulator.reset()
        state = self.simulator.state

        while not self.simulator.game_over():
            # choose an action based on the epsilon-greedy policy
            action = self.pick_action(state, epsilon)

            # get new random variable observation, one new instance from this␣
 ↪state and action pair
            next_state, reward = self.make_one_transition(action)

            # If next_state is None, it means the game is over
            if next_state is None:
                max_next_q = 0  # no future rewards if the game is over
            else:
                max_next_q = max(self.Q_values[next_state]) # global max in␣
 ↪memory, different from SARSA, look at all Q-values and select

            # Q-Learning update rule, new state discovered, initialize it with␣
 ↪nothing and value with nothing
            if state not in self.N_Q:
                self.N_Q[state] = [0, 0] # initialize it with nothing
                self.Q_values[state] = [0.0, 0.0] # initialize value with␣
 ↪nothing
```

```
            self.N_Q[state][action] += 1 # this state-action pair visited once␣
 ↪more

            #
            alpha = self.alpha(self.N_Q[state][action])
            self.Q_values[state][action] += alpha * (reward + self.DISCOUNT *␣
 ↪max_next_q - self.Q_values[state][action])

            # move to the next state & recursively explore the tree
            state = next_state if next_state is not None else state


def pick_action(self, s, epsilon):
    '''epsolon greedy algorithm'''
    if random.random() < epsilon: # Explore: choose a random action
        return random.choice([HIT, STAND])
    else: # Exploit: choose the best action based on current Q-values
        if self.Q_values[s][HIT] > self.Q_values[s][STAND]:
            return HIT
        else:
            return STAND
```

Start with good guesses explores help?