11 Reinforcement learning

In reinforcement learing, the objective is to learn a policy that can control an agent in an environment in such a way that the agent achieves a high reward in the long run. Reinforcement learning is a very general framework that can be used to solve many different tasks.

11.0.1 Markov decision process

Typically, a reinforcement learning problem is formulated as a Markov decision process in the following way: We assume that the agent interacts with the environment in a sequence of steps. At each step, the agent must decide upon an action a to perform. Based on the action, the environment changes its state s, and after each step the agent recieves some reward r.

Environment The agent observes the environment in form of a *state* s. The environment can either be fully observed, in which case the agent has access to all information about the environment, or it can be partially observed such that the agent can only access a smaller part of the total information. The dynamic behavior of the environment is governed by a set of rules that controls what happens to the environment state when the agent takes a particular action. These rules can be deterministic or stochastic: In the deterministic case, the rules define what the next state s' of the environment will be if the current state is s and the agent takes action a, which can be formulated as a function $s' = f(s, a)$. In the stochastic case, the rules define a probability distribution over possible next states, $p(s'|s, a)$.

Action The agent interacts with the environment by taking actions a. The set of possible actions can be discrete (such as go north, east, south, or west in a maze) or continuous (such as apply a voltage between 0V and 5V to a motor in a robot arm). Actions can also be multivariate, for example when controlling a complicated robot with many joints. The set of possible actions can depend on the state of the environment: For example, if we have reached a dead end in a maze, we cannot continue to walk forward.

Reward After taking an action, the agents recieves a reward r. The reward is a single number, and the goal of the agent is usually to optimize the total expected cumulative reward. In the general setting, the reward can depend on the current state, the chosen action, and the next state, and it can either be defined as a deterministic function $r = r(s, a, s')$ or in the stochastic setting as a probability distribution $p(r|s, a, s')$. When the environment is deterministic (the next state can be determined uniqely from the current state and action), the reward is typically written as a function of the current state and action, $r(s, a)$.

In settings where the reinforcement learning problem can potentially continue indefinitely, the expected reward is often weighted (discounted) such that the agent will focus on achieving high reward sooner rather than later. The discounting is also used to ensure that the expected reward does not diverge; if an agent is allowed to continue collecting reward forever, the expected cumulative reward of any reasonable policy is infinite and can thus not be used as an optimality criterion.

11.1 Deterministic value iteration

One way to approach the reinforcement learning problem is to estimate the *value* of each state denoted by $v(s)$. We refer to this as the *value function*. In the following, we consider a deterministic environment, $s' = f(s, a)$, with a finite number of states, and a deterministic reward associated with each state-action pair, $r(s, a)$. In this setting we have the following definition of the value function.

Definition 11.1 Value function (deterministic setting)

The value function is defined recursively as

$$v(s) = \max_a \left(r(s,a) + \gamma \cdot v(s') \right) .$$

Let us understand what this equation means: We are in state s and we can now take some action a. Depending on which action we take, we end up in a new state $s' = f(s,a)$. The value of our current state s is equal to: $r(s,a)$ (the reward we get for taking the action a from our current state s) plus $\gamma \cdot v(s')$ (the value of the next state s' discounted by γ) for the action a that leads to the highest value.

The discount factor γ is a number between 0 and 1: A typical value might be 0.9. The discount makes us give higher priority to immediate rewards, rather than rewards further out in the future. Furthermore, the discounting ensures that the value of the cumulative reward is never infinite.

Once we have estimated the value function, the optimal policy is to take the action that leads to the highest discounted future reward. Mathematically, we write this as

$$a^* = \arg\max_a \left(r(s,a) + \gamma \cdot v(s') \right) ,$$

where as usual $s' = f(s,a)$ is the next state and s is the current state. To find the optimal action, we need to know which next state each action will take us to.

But how can we in practice compute the value function? Since the value $v(s)$ depends on the value of the next state $v(s')$, we have a recursively defined optimality criterion: A set of equations that must be satisfied for all states. One way to estimate the value function is to start by initializing the the value of all states (perhaps randomly or all zeros) and then loop through all states and update their value according to the definition over and over again, until the value function converges. This approach is called *value iteration*.

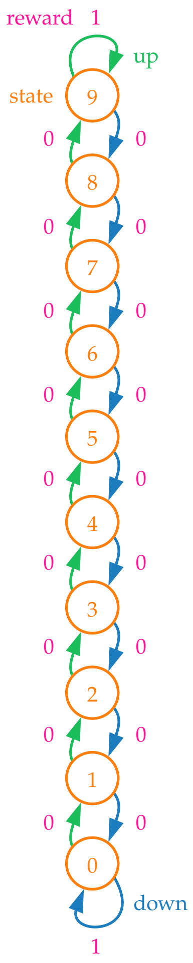

Figure 11.1: An agent can move *up* or *down* through ten states 0–9. Taking the action *up* in state 9 returns the agent to state 9, and similar, taking action *down* in state 0 returns the agent to state 0. The reward is 1 for transitioning into state 0 or 9, and zero for all other state-action pairs.

Example 11.1 Up and down the ladder

Consider the reinforcement learning problem illustrated in Fig. 11.1. We have 10 states named 0–9, and in each state we have two possible actions, *up* and *down*. The reward associated with all state-action pairs is zero except for the actions that self-transition in the top and bottom states (0 and 9) where the reward is one. The goal is thus to reach one of these two states as quickly as possible and stay there.

Since the only non-zero rewards are associated with state 0 and 9, the optimal policy is obviously to move *down* in states 0–4 and *up* in states 5–9. This will most quickly get the agent to one of the rewarding states and stay there forever.

Now, what is the *value* of the different states? This depends on the value we choose for the discount factor. Here, let us use $\gamma = 0.9$, and let us first consider state 0. Following the optimal policy (*down*) will keep us in state 0 forever, so we get reward 1 at each time step. The value of state 0, $v(0)$, can thus be computed as

$$\begin{aligned} v(0) &= 1 + \gamma v(0) && (11.1) \\ &= 1 + \gamma + \gamma^2 + \cdots && (11.2) \\ &= \frac{1}{1-\gamma} = 10 && (11.3) \end{aligned}$$

We can continue in a similar manner to compute the value of all other states. But note, that this approach required us to guess the optimal policy.

Instead, let us use *value iteration* to compute the value of all states. We can do this using a small computer program where we initialize the value of each state to zero and then repeatedly loop over all states and update their value.

```python
# Initial values
v = [0,0,0,0,0,0,0,0,0,0]
# Discount
gamma = 0.9
# Actions
actions = ['up', 'down']

# Next state function
def f(s, a):
    if a=='up':
        return min(s+1, 9)
    if a=='down':
        return max(s-1, 0)
# Reward function
def r(s, a):
    if (s==0 and a=='down') or (s==9 and a=='up'):
        return 1
    else:
        return 0

# 1000 value iterations
for t in range(1000):
    # Loop over all states
    for s in range(10):
        # Update value
        v[s] = max([r(s,a) + gamma*v[f(s,a)] for a in
    actions])
```

After running this code, the state values converge to:

s	0	1	2	3	4	5	6	7	8	9
$v(s)$	10	9	8.1	7.29	6.561	6.561	7.29	8.1	9	10

11.2 Deterministic q-learning

Q-learning is a *model free* reinforcement learning methods. In contrast to value iteration, q-learning does not require that we have knowledge about the state transition mechanism, but learns directly by interacting with the environment.

The "q" in the name stands for *quality*, and it is named so because the central aspect of the method is to estimate a quality function $q(s, a)$. Analogous to the value function discussed previously, the quality function is also called the *state-action value* function, and it measures the value of taking action a in state s. Similar to value iteration, the quality is measured in terms of discounted cumulative reward.

In the setting where state transitions and rewards are deterministic, we have the following recursive definition of the quality function.

Definition 11.2 Quality function (deterministic setting)

The quality function is defined recursively as

$$q(s, a) = r(s, a) + \gamma \max_{a'} \left(q(s', a') \right) . \tag{11.4}$$

Similar to value iteration, we can use this formula to iteratively update the quality function until it converges. However, there is an important difference: In the value iteration algorithm, we updated the value of a state based on the reward and the highest valued next state we could reach. This required us to know exactly which states we could reach from each state. In other words, we needed a model of the environment. Here, in q-learning, we update the quality function based on the reward and the quality of the next state. Thus, we do not need to know which states we can possibly reach from the current state: All we need to do is to take an action in the envirionment, which will bring us to some new state, and we can then update the quality function based on this interaction.

11.2.1 *Epsilon-greedy action selection*

With q-learning, we can thus update the quality function by interacting with the environment and learning about the state transitions as we go along. However, in order for the algorithm to converge, we must interact with the environment in such a way that we are guaranteed to visit all states and try all actions. Obviously, if there are states or actions that are never explored, we would have no way to learn about them. In value iteration, this issue was solved by simply looping over all states to do the updates.

In q-learning, the typical approach is to use the *epsilon greedy* strategy. The epsilon greedy strategy consists two options:

1. With probability $1 - \epsilon$ we take the best action according to the current estimate of the quality function,

$$a^* = \arg\max_a q(s, a).$$

2. With probability ϵ we take a random action.

The constant *epsilon* definies a trade-off between *exploration* and *exploitation*. In the one extreme $\epsilon = 1$, the algorithm takes completely random actions and thus explores the envirionment through a random walk. In the other extreme $\epsilon = 0$ the algorithm exploits its current knowledge by taking actions that are optimal under the current (possibly flawed) estimate of the quality function. In practice, a selecting a good value for *epsilon* can be highly problem dependent, but a typical reasonable value could be around $\epsilon = 0.1$ such that 10% of the actions are random.

Problems

1. Show that the value of state 5 in the "up and down the ladder" problem is $v(5) = 6.561$. Use the optimal strategy (always go *up*) with $\gamma = 0.9$ and compute

$$v(5) = 0 + \gamma v(6) = 0 + \gamma \left((0 + \gamma v(7) \right) = \ldots$$

Solutions

1. We start by expanding $v(5)$ until we have it expressed in terms of the final state $v(9)$

$$\begin{aligned} v(5) &= 0 + \gamma v(6) = \gamma v(6) \\ &= \gamma\left(0 + \gamma v(7)\right) = \gamma^2 v(7) \\ &= \gamma^2\left(0 + \gamma v(8)\right) = \gamma^3 v(8) \\ &= \gamma^3\left(0 + \gamma v(9)\right) = \gamma^4 v(9). \end{aligned}$$

We can then evaluate the value of the final state as

$$v(9) = 1 + \gamma v(9) \Leftrightarrow v(9) = \frac{1}{1-\gamma} = 10.$$

Inserting this, we arrive at

$$v(5) = \gamma^4 v(9) = 0.9^4 \cdot 10 = 6.561$$