Reinforcement Learning 1: Tabular Methods

Abdelrahman Khaled

Machine Learning Research Cluster
German University in Cairo

August 4, 2019

Outline

The Agent-Environment interface

The learner (who we call the agent) interacts with the surrounding (which we call the environment) once every timestep t with an action A_t. It then receives a reward R_t from the environment, and observes the new resulting state S_t.

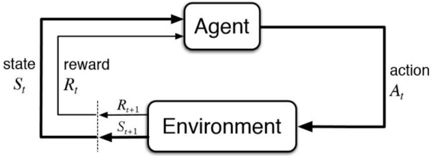

Figure: The agent-environment cycle. *Image source*

Going through the cycle multiple times nets a sequence that looks like: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$

MDPs

- MDPs can be modelled as graphs where nodes are states and edges are actions, and each edge has a weight which represents the reward.
- In a finite MDP, the sequence terminates at some timestep. We call a terminated run an episode.
- The return of an episode G is the total reward obtained in a single episode. G_t is the reward obtained from timestep t till the end of the episode. Maximizing G_t at all steps t is the same as maximizing G.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots$$

MDPs

- If we want the agent to continue the task rather than terminate, then $t \to \infty$, this means that (possibly) also $G \to \infty$. To solve this we introduce discounted returns.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $0 \leq \gamma \leq 1$ is the discount factor.

- We can also define the return recursively, resulting in:

Definition

$$G_t = R_{t+1} + \gamma G_{t+1}$$

MDPs

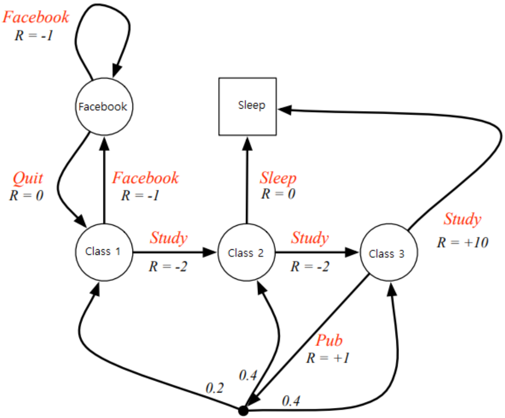

Figure: MDP from David Silver's UCL slides

MDPs

Definition

A policy π is a mapping from states to probabilities of selecting each possible action.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

Definition

The value function of a state s under policy π is the expected return when starting in s and following π afterwards.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

MDPs

Definition

The action-value function of taking an action a in a state s under policy π is the expected return when starting from state s and taking action a then following π afterwards.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$
$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

Note

The recursive forms of v and q are known as Bellman equations.

MDPs

Definition

A policy exists that is always equal to or better than other policies, that policy is denoted as π_*.

Using the above definition, then we can define the optimal value function v_* and optimal action-value function q_* as the expected return when following the optimal policy.

Note

An MDP can have more than one optimal policy, but all optimal policies are equivalent.

Dynamic Programming (DP)

A technique of solving problems by breaking it down into smaller sub-problems and solving them (without recomputing the solution of an already solved sub-problem), then combining the solutions of all the sub problems to obtain the solution for the main problem.

MDPs can be solved by dynamic programming.

- Being described by the bellman equation gives the value function and action-value function a recursive form, hence easily broken down into smaller sub-problems.

Policy Evaluation

First thing's first: How do we compute the value function v_π for some arbitrary policy π?

We can use the Bellman equation to define v_π as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

We can initialize v_0 to be anything, and iterate till we converge to v_π in the following manner:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Policy Evaluation

Input: policy π
Output: value function v_π
Static : num θ

```
forall State s do
    v[s] = 0;
end
eps = θ + 1;
while eps > θ do
    eps = 0;
    forall State s do
        val = v[s];
        v[s] = Σ_a π(a|s) Σ_{s',r} p(s', r|s, a)[r + γv[s']];
        eps = max(eps, |val − v[s]|);
    end
end
```

$$v[s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v[s']]$$

Policy Iteration

Now that we know how to evaluate a policy, we need to know how we can improve it!

To do that we need to understand that if we have two policies π' and π, then π' is better than π iff:

$$v_{\pi'}(s) \geq v_{\pi}(s), \forall s \in S \tag{1}$$

Knowing that, then we can always choose a better policy by acting greedily with respect to the known values of the current policy.

$$\pi' = greedy(v_{\pi}) \tag{2}$$

Policy Iteration

Output: policy π

forall *State s* **do**

  $\pi[s] = random\_action\_probability();$

end

$stable = false;$

while $!stable$ **do**

  $stable = true;$

  $v = policy\_evaluation(\pi);$

  forall *State s* **do**

    $act = \pi[s];$

    $\pi[s] = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v[s']];$

    if $act \neq \pi[s]$ **then**

      $stable = false;$

    end

  end

end

Principle of Optimality

An optimal policy can be describe as follows:

- Take an optimal first action at start state s that leads to state s'.
- Use an optimal policy starting from state s'.

Value iteration attempts to use this principle by assuming that $v(s')$ is optimal in order to update $v(s)$.

If these updates are done iteratively, then $v_*(s)$ will be reached eventually.

Value Iteration

Output: policy π
Static : num θ
$eps = \theta + 1;$
while $eps > \theta$ **do**
    $eps = 0;$
    forall *State s* **do**
        $val = v[s];$
        $v[s] = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v[s']];$
        $eps = max(eps, |val - v[s]|);$
    end
end
forall *State s* **do**
    $\pi[s] = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v[s']];$
end

Model Based Prediction & Control

- All the previous dynamic programming methods discussed are considered model-based methods, since they can only solve an MDP which has known parameters.
- We call an algorithm like policy evaluation a **prediction** algorithm since it can tell us how well a certain policy is doing by giving us the stat values.
- We call an algorithm like policy iteration a **control** algorithm since it tells what decisions the agent should make.
- All model-based methods are closer to search algorithms rather than learning algorithms.

Model-Free Control & Prediction

- Model-free methods solve MDPs through experience rather than through searching the environment.
- Model-free methods don't have to know everything about the MDP.
- A very important concept in model-free reinforcement learning is exploration vs exploitation.
  - **Exploration** is trying different strategies in the middle of learning in order to have a better chance of finding the optimal one.
  - **Exploitation** is using the learned information to create a better policy.

ϵ-Greedy Exploration

- An exploration method.
- Based on the current knowledge:
  - Choose the greedy action with probability $1 - \epsilon$.
  - Choose a random action with probability ϵ
- The epsilon can be changed as the agent learns how to traverse the environment better.

Monte Carlo Learning (MC)

- MC learning is a model-free method of learning that learns from a complete episode of experience.
- To learn using MC, an MDP needs to be finite (has to terminate at some point).
- MC uses the complete discounted return of an episode to calculate the value of a state, rather than the expected return.
- The idea is that as the number of episodes increases, the value of a state will get closer to the actual value.

Monte Carlo Prediction

For each state s:

- We can create a counter $n[s]$ that counts how many times s has been visited, and a variable $a[s]$ that contains the sum of all the returns obtained in state s.
- The value $v[s]$ is then estimated to be $v[s] = \frac{a[s]}{n[s]}$

Using this method we have to re-calculate all the means of every episode whenever we go through a new episode.

To tackle that, we can use incremental means!

Side Note: Incremental Mean

Let m_n be the mean of the first n terms of the sequence $a_1, a_2, \ldots$.

$$\begin{aligned} m_n &= \frac{1}{n}\sum_{i=1}^{n} a_i \\ &= \frac{(n-1)m_{n-1} + a_n}{n} \\ &= m_{n-1} + \frac{1}{n}(a_n - m_{n-1}) \end{aligned}$$

This result shows that the mean of the first n terms can be calculated using the mean of the first $n-1$ terms as well as the n^{th} term.

Monte Carlo Prediction

Input: policy π
Output: value function v_π
forall *State s* **do**

  $v[s] = 0;$
  $n[s] = 0;$

end
$S, G = episode();$
foreach *Timestep t* **do**

  $n[S[t]] = n[S[t]] + 1;$
  $v[S[t]] = v[S[t]] + \alpha(G[t] - v[S[t]]);$

end

Monte Carlo Control

It is sufficient for prediction to calculate just the value function v.

However for control in model-free methods, one needs to know the action-value function q.

This small addition will only change the above algorithm slightly.

Monte Carlo Control

Output: policy π

forall *State s, Action a* **do**

$\quad q[s, a] = 0;$

$\quad n[s, a] = 0;$

end

$S, A, G = episode();$

foreach *Timestep t* **do**

$\quad n[S[t], A[t]] = n[S[t], A[t]] + 1;$

$\quad q[S[t], A[t]] = q[S[t], A[t]] + \alpha(G[t] - q[S[t], A[t]]);$

$\quad \pi[S[t]] = \arg\max_a(q[S[t], a]);$

end

Temporal Difference Learning (TD)

- TD learning is a model-free method of learning that learns from incomplete episodes (every timestep).
- It starts with a guess of what the true value function is and updates its guess every time a state is visited.
- TD only uses the immediate reward plus the discounted expected reward from its previous guess.
- There are many TD algorithms. The simplest one is TD(0).

TD(0) Prediction

Input: policy π
Output: value function v_π
forall *State s* **do**
    $v[s] = 0;$
end
foreach *Episode e* **do**
    $s = reset\_environment();$
    while *sisnotterminal* **do**
        $a = use\_policy(\pi, s);$
        $r, s' = take\_action(a);$
        $v[s] = v[s] + \alpha(r + \gamma v[s'] - v[s]);$
        $s = s';$
    end
end

Q-Learning (TD(0) Control)

forall *State s, Action a* **do**

    $q[s, a] = 0;$

end

foreach *Episode e* **do**

    $s = reset\_environment();$

    while *sisnotterminal* **do**

        $a = use\_policy(q, s);$

        $r, s' = take\_action(a);$

        $q[s, a] = q[s, a] + \alpha(r + \gamma \max_{act} q[s', act] - q[s, a]);$

        $s = s';$

    end

end

References

- UCL. Advanced Topics 2015. (COMPM050/COMPGI13) *Reinforcement Learning*. David Silver. link
- Sutton and Barto. Reinforcement lerning. 2nd edition. Chapters 3-6.