Machine Learning in Finance
Section I3
Lecture 14

Michael G Sotiropoulos

NYU Tandon
Deutsche Bank

10-May-2019

Contents

Dynamic Programming: Review

Dynamic Programming (DP) is a collection of algorithms for finding optimal policies given a perfect model for the environment.

We start with the Bellman equation for the optimal state-value and action-value

$$v_\star(s) = \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\star(s')\right] \tag{1}$$

$$q_\star(s, a) = \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_\star(s', a')\right] \tag{2}$$

with states $s, s' \in \mathcal{S}$, actions $a, a' \in \mathcal{A}$, reward $r \in \mathbb{R}$, and discount $\gamma \leq 1$.
The state-value function is $v(s)$ and the action-value is $q(s, a)$.
The environment is described as an MDP via the probabilities $p(s', r|s, a)$.

Given state s, the future states s' are called the successor states.

- Solving the Bellman equations means finding update rules that continually improve the approximations $v(s)$ or $q(s, a)$ and converge to $v_\star(s)$ or $q_\star(s, a)$ in the limit of infinite iterations.

Dynamic Programming: Policy Evaluation and Improvement (I)

The first task is policy evaluation or prediction.

- Given policy π, i.e. the function $\pi(a|s)$, compute the value function $v_\pi(s)$

From the definition of the value function, we need to solve

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{3}$$

This is a linear system of equations with unknowns $v_\pi(s)$ .There are as many unknowns as states $s \in \mathcal{S}$. Brute force solution may be very inefficient.

Iterative policy evaluation is an algorithm that starts with an initial guess for the value function $v_0(s)$. Then, at each iteration k, it updates the estimate as

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{4}$$

The sequence $\{v_k\}$ converges to v_π as $k \to \infty$.

- A two-array implementation uses two arrays for storing $v_k(s)$ and $v_{k+1}(s)$
- A one-array implementation or a **sweep** uses one array for $v_{k+1}(s)$ and overwrites older values with newer ones.

Dynamic Programming: Policy Evaluation and Improvement (II)

Iteration stops when no state update exceeds a small threshold value θ.
The iterative policy algorithm with state sweep looks like this in pseudo-code

```
input: $\pi$
parameters: $\theta$
initialize: $V(s)$ for all $s \in \mathcal{S}$ except $V(\mathit{terminal}) = 0$
loop:
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ :
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
return $V(s)$
```

The same iterative algorithm can be applied to approximating the action-value.
We simply sweep over all possible pairs (s, a) and apply the update

$$Q(s,a) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma Q(s', a')\right] \tag{5}$$

Dynamic Programming: Policy Evaluation and Improvement (III)

The second task is to examine whether we can improve a given policy π.

The trick here is to follow policy π for all states except for state s, where we switch to action $a = \pi'(s)$. This gives us a new value function

$$q_\pi(s, a) = \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right] \tag{6}$$

If it turns out that $q_\pi(s, a) \geq v_\pi(s)$ then we expect that it is always better to choose action a once we are in state s
This is true because of the following **policy improvement theorem**

- If π and π' are deterministic policies and for all $s \in \mathcal{S}$

$$q_\pi\left(s, \pi'(s)\right) \geq v(s) \tag{7}$$

then π' is as good or better than π, i.e for all $s \in \mathcal{S}$

$$v_{\pi'}(s) \geq v_\pi(s) \tag{8}$$

Dynamic Programming: Policy Evaluation and Improvement (IV)

We have a procedure to check whether a policy improves if we change behavior for a single state s. Now we can get **greedy**.

For all states $s \in \mathcal{S}$ and all possible actions $a \in \mathcal{A}$ we select at state s the action a that has the highest $q_\pi(s, a)$. This generates a new policy π' as follows

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{9}$$

- The process of replacing a policy π with a better policy π' that is greedy with respect to π is called **policy improvement**.

If π' is as good but no better than π, then $\pi' = \pi$ and eq. (9) becomes the Bellman equation. We have found a solution!

Finally, we use sequence of alternating evaluation and improvement updates to obtain a monotonically improving policy and value function

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Figure: The policy evaluation-improvement sequence. From [SB]

Dynamic Programming: Policy Evaluation and Improvement (V)

The policy iteration/evaluation algorithm looks like this in pseudo-code

```
parameters: $\theta$
1. initialize: $V(s)$, $\pi(s)$ for all $s \in \mathcal{S}$
2. policy evaluation
    loop:
        $\Delta \leftarrow 0$
        for each $s \in \mathcal{S}$ :
         $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    until $\Delta < \theta$
3. policy improvement
    $stable \leftarrow true$
    for each $s \in \mathcal{S}$ :
        $a_{old} \leftarrow \pi(s)$
        $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
        if $a_{old} \neq \pi(s)$ then $stable \leftarrow false$
    if $stable$:
        return $V(s)$ ,$\pi(s)$
    else:
        go to 2
```

Dynamic Programming: Generalized Policy Iteration

In policy iteration, evaluation and improvement alternate in sequence. One affects the other through its output, but they do not "step on each other".

Generalized Policy Iteration (GPI) allows evaluation and improvement to communicate (or overwrite each other's results) while they both run. The final solution is returned once it becomes stable w.r.t. both evaluation and improvement.

- In GPI, evaluation and improvement both cooperate and compete
- Greedy search makes the value function inaccurate w.r.t. the new policy, but encourages exploration
- Making the value function consistent w.r.t. the current policy increases accuracy but decreases exploration

Most RL algorithms are some form of GPI.

Dynamic programming is the cleanest way of formulating RL problems. However it has two major drawbacks:

1. It requires full information about the environment
2. It is computationally inefficient (curse of dimensionality)

Below we examine ways to improve on DP algorithms.

Monte Carlo Methods: Prediction and Control (I)

Monte Carlo (MC) methods try to overcome the limitations of DP, by using simulated experience. Their advantages are

- No need to know the complete specification of the environment, i.e. the functional form of $p(s', r|s, a)$
- It is much easier to generate sample transitions from target distributions
- Learning occurs via sample averaging

In an MC method, we generate an episode and then for each state-action pair (s, a) that we saw during the episode, we compute the average reward.

Similarly to DP, here also we need to solve for policy evaluation (prediction), policy improvement (control) and combine the two in a GPI algorithm.

- Since we do not assume knowledge of the environment, we have to use the action-value function $q(s, a)$.

$$\pi_0 \xrightarrow{\mathrm{E}} q_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} q_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi_* \xrightarrow{\mathrm{E}} q_*$$

Figure: The policy evaluation-improvement sequence. From [SB]

Monte Carlo Methods: Prediction and Control (II)

With an initialization that explores states and actions (exploring starts) the MC algorithm looks like this in pseudo-code

```
parameters: $\theta$
initialize: $\pi(s) \in \mathcal{A}$, Q$(s,a) \in \mathbb{R}$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}$
            $Returns\,(s,a) \leftarrow$empty_list
loop for every episode:
    choose $S_0, A_0$ randomly so that all pairs $(S_0, A_0)$ are candidates
    generate episode  $S_0, A_0, R_1, S_1, A_{1,} \dots R_T$ following $\pi$
    $G \leftarrow 0$
    loop for step $t = T-1, T-2, \dots 0:$
        $G \leftarrow \gamma G + R_{t+1}$
        if $(S_t, A_t)$ not in sequence $S_0, A_0, R_1, S_1, A_{1,} \dots S_{t-1}, R_{t-1}$:
            append $G$ to $Returns\,(S_t, A_t)$
            $Q\,(S_t, A_t) \leftarrow average\,(Returns\,(S_t, A_t))$
            $\pi\,(S_t) \leftarrow \arg\max_a Q\,(S_t, A_t)$
```

Monte Carlo Methods: On/Off Policy (I)

- A method is called **on-policy** if it uses the same policy for prediction and control
- A method is called **off-policy** if it uses one policy for prediction and a different policy for control.
  The policy used for prediction is also called *behavior*. It is used for exploring state-action pairs.
  The policy used for control is also called *target*. This is the policy that is being improved.

The assumption of exploring starts is too restrictive.
A better method is on-policy MC with ϵ-greedy policy.
At each new episode, we do not randomly re-initialize, but follow a greedy policy to explore states.
This is similar to the multi-arm bandit problem of the previous lecture.

Monte Carlo Methods: On/Off Policy (II)

The on-policy ϵ-greedy MC algorithm looks like this in pseudo-code

```
parameter: small $\epsilon > 0$
initialize: $\pi \leftarrow$ some $\epsilon$-greedy policy
            $Q(s,a) \in \mathbb{R}$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
            $Returns(s,a) \leftarrow$ empty_list
loop for every episode:
    generate episode $S_0, A_0, R_1, S_1, A_1, \ldots R_T$ following $\pi$
    $G \leftarrow 0$
    loop for every step $t = T-1, T-2, \ldots 0$ :
        $G \leftarrow \gamma G + R_{t+1}$
        if $(S_t, A_t)$ not in sequence $S_0, A_0, R_1, S_1, A_1, \ldots S_{t-1}, R_{t-1}$:
            append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow average(Returns(S_t, A_t))$
            $A^\star \leftarrow \arg\max_a Q(S_t, A_t)$
            for all $a \in \mathcal{A}(S_t)$:
```

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon / |\mathcal{A}(S_t)| & \text{if} \quad a = A^\star \\ \epsilon / |\mathcal{A}(S_t)| & \text{if} \quad a \neq A^\star \end{cases}$$

In the last step the algorithm updates the policy using an ϵ-greedy method. The notation $|\mathcal{A}(S_t)|$ means the number of actions taken from state S_t.

Temporal-Difference Methods: Concept (I)

Temporal Learning (TD) is a pivotal method in Reinforcement Learning.

- TD combines the concepts of DP and MC
- Like DP, TD updates the current action-value estimates based on other concurrent action-value estimates, i.e. it **bootstraps**.
  Therefore, it does not wait until the end of an episode to do the updates.
- Like MC, TD learns from experience without requiring a model for the environment.

Let's compare the prediction methods in MC and TD within the simple context of updating the value function
In MC, we run to the end time T to obtain the payoff G_t, and then we update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right] \tag{10}$$

In TD, just after time t we update

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_t) - V(S_t) \right] \tag{11}$$

This is called TD(0) or one-step TD. Its generalization is n-step TD or TD(λ).

Temporal-Difference Methods: Sarsa (I)

Sarsa is on-policy TD control.
It continually estimates the action value q_π for the behavior policy π and at the same time it changes π towards a target that is greedy with respect to q_π.
The algorithm in pseudo-code looks like this

```
parameters: step size $\alpha \in (0,1]$, small $\epsilon > 0$
initialize: $Q(s,a) \in \mathbb{R}$ for all $s \in \mathcal{S}$,$a \in$
$\mathcal{A}$ with $Q(terminal, .) = 0$
loop for every episode:
    initialize $S$
    choose $A$ from $S$ using policy derived from $Q$ ($\epsilon$-greedy)
    loop for each step:
        take $A$ from $S$, observe $R$, $S'$
        choose $A'$ from $S'$ using policy derived from $Q$ ($\epsilon$-greedy)
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R + \gamma Q(S', A') - Q(S_t, A_t)]$
        $S \leftarrow S'; A \leftarrow A'$
    until terminal $S$
```

Sarsa converges almost surely to the optimal policy and action-value function in the limit when all state-action pairs are visited an infinite number of times.

Temporal-Difference Methods: Q-Learning (I)

Q-learning is off-policy TD control.
This algorithm was a major breakthrough in RL (Watkins 1989).

- The main idea in Q-learning is to approximate directly the optimal action-value $q_\star$ independently of the policy being followed during learning. This is achieved via the update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (12)$$

- The policy that we follow during learning (behavior) affects the results to the extent that it determines which state-action pairs are visited and therefore updated.
- If all the state-action pairs are visited and updated, Q-learning converges to the optimal target $q_\star$ independent of the behavior policy.

Temporal-Difference Methods: Q-Learning (II)

The Q-learning algorithm is structurally quite similar to Sarsa.
In pseudo-code looks like this

```
parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
initialize: $Q(s, a) \in \mathbb{R}$ for all $s \in \mathcal{S}$,$a \in \mathcal{A}$ with $Q(terminal, .) = 0$
loop for every episode:
    initialize $S$
    loop for each step:
        choose $A$ from $S$ using policy derived from $Q$ ($\epsilon$-greedy)
        take $A$ from $S$, observe $R$, $S'$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R + \gamma \max_a Q(S', a) - Q(S_t, A_t)]$
        $S \leftarrow S'$
    until terminal $S$
```

In TD(0) methods within one time step we update (bootstrap) and move to the next state. In TD(n) methods, the bootstrapping happens every n time steps.

Sarsa and Q-learning can be generalized to TD(n). In fact, this generates a whole spectrum of methods, with TD(0) on one end and MC on the other.

Further Reading

1. For a deeper discussion of the topics in this lecture read chapters 5 and 6 in [SB]
2. Become familiar with OpenAI Gym [GYM]. It provides ready-made components for simulating environments and coding RL algorithms.

References

[SB] Sutton R.S. and Barto A.G. "Reinforcement Learning An Introduction" 2nd edition. MIT, 2018

[GYM] Open AI Gym. A toolkit for Developing and Comparing RL Algorithms. gym.openai.com