Reinforcement Learning

Lecture 4: Dynamic programming

Robert Lieck

Durham University

Lecture overview

Lecture covers Chapter 4 in Sutton & Barto [1] and adaptations from David Silver [2]

1. **Introduction**
   - definition
   - examples
   - planning in an MDP
2. **Policy evaluation**
   - definition
   - synchronous algorithm
3. **Policy iteration**
   - policy improvement
   - definition
   - modified policy iteration
4. **Value iteration**
   - definition
   - summary and extensions

Definition: Dynamic programming

Dynamic programming is an optimisation method for sequential problems. DP algorithms are able to solve complex 'planning' problems.

Given a complete MDP, dynamic programming can find an optimal policy. This is achieved with two principles:

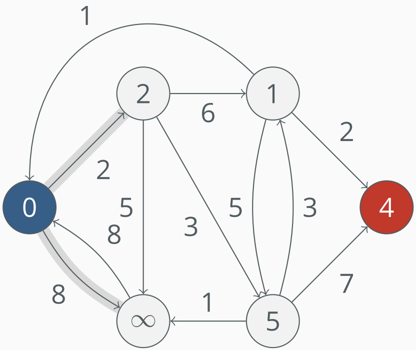
1. Breaking down the problem into subproblems
2. Caching and reusing optimal solutions to subproblems to find the overall optimal solution

Planning: what's the optimal policy?

Famous examples

- Dijkstra's algorithm
- Backpropagation
- Doing basic math

...so it's really just recursion and common sense!



Dynamic programming for planning MDPs

In reinforcement learning, we want to use dynamic programming to solve MDPs. So given an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π:

First, we want to find the value function v_π for that policy:

- This is done by **policy evaluation** (the prediction problem)

Then, when we're able to evaluate the policy, we want find the best policy v_* (the control problem). This is done with two strategies:

1. **Policy iteration**
2. **Value iteration**

Follow along in Colab:

Policy evaluation definition

Definition: Policy evaluation

We want to evaluate a given policy π.
We'll achieve this with the Bellman **expectation** equation, $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_\pi$

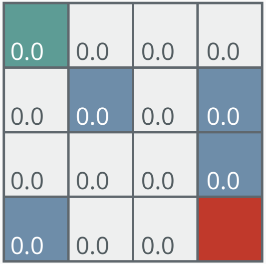
	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

Example: frozen lake environment

Policy evaluation synchronous algorithm

Algorithm: policy evaluation

```
def policy_evaluation(env, policy, γ, theta):
  V = np.zeros(env.num_states)
  while True:
    delta = 0
    for s in range(env.num_states):
      Vs = 0
      for a, a_prob in enumerate(policy[s]):
        for prob, s', reward, done in env.P[s][a]:
          Vs += a_prob * prob * (reward + γ * V[s'])
      delta = max(delta, abs(V[s]-Vs))
      V[s] = Vs
    if delta < theta:
      break
  return V
```



Policy evaluation synchronous algorithm

Recap: Bellman expectation equation

$$
\begin{aligned}
v_\pi(s) &= \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a) \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)
\end{aligned}
$$

Algorithm: policy evaluation

```
(iteration=1, γ=1)
  for s in range(env.num_states):
    Vs = 0
    for a, a_prob in enumerate(policy[s]):
      for prob, s', reward, done in env.P[s][a]:
        Vs += a_prob * prob * (reward + γ * V[s'])
→V[s] = Vs
```

iteration 1, $\pi =$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.25	

Policy evaluation synchronous algorithm

Recap: Bellman expectation equation

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a)$$

$$= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

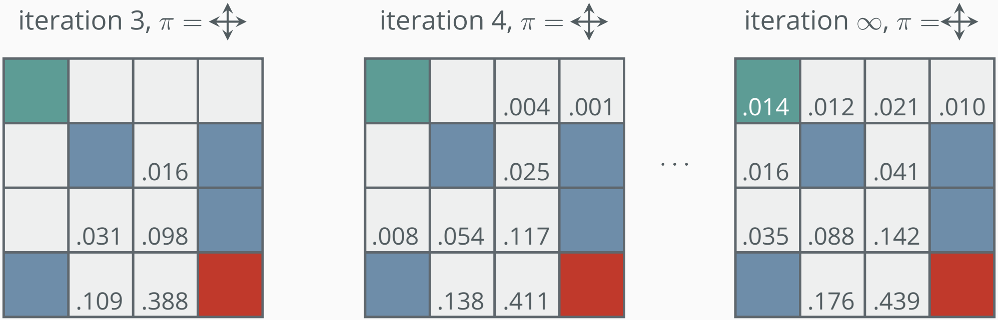
Algorithm: policy evaluation

```
(iteration=2, γ=1)
  for s in range(env.num_states):
    Vs = 0
    for a, a_prob in enumerate(policy[s]):
      for prob, s', reward, done in env.P[s][a]:
        Vs += a_prob * prob * (reward + γ * V[s'])
→V[s] = Vs
```

iteration 2, $\pi =$

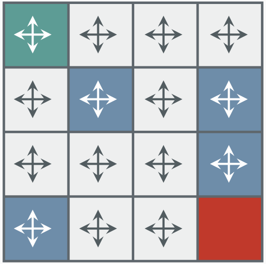
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.06	0.0
0.0	0.06	0.34	

Policy evaluation synchronous algorithm



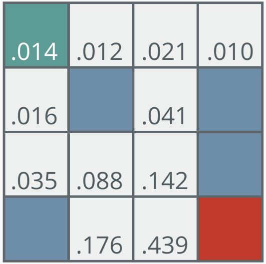
Policy iteration greedy policy improvement

random policy



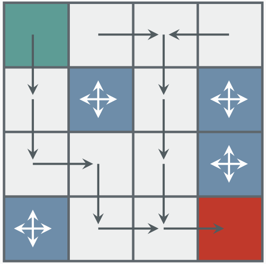
$\rightarrow$

iteration ∞, $\pi =$



$\max_a$

improved policy



Policy iteration definition

Definition: Policy iteration

Given a policy π (e.g. starting with a random policy), **iteratively** evaluate:

$$v_\pi(s) = \mathbb{E}[R_{t+1}, +\gamma R_{t+2} + \ldots \mid S_t = s]$$
$$\pi' = \text{greedy}(v_\pi)$$

This always converges to the optimal policy π^*. That is, if the improvements stop:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

then the Bellman equation has been satisfied $v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$ therefore $v_\pi = v_*(s)$ for all $s \in \mathcal{S}$

Example: learning a better policy

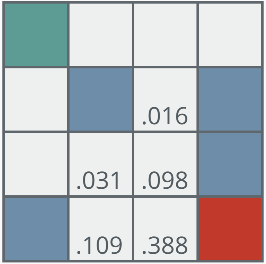
Policy iteration modified policy iteration

Algorithm: modified policy iteration

What if we don't do iterative policy evaluation to ∞?
What if we just do a crude, e.g. $k = 3$ small amount of iteration?
Does it still converge?

- Yes! It still converges to the optimal policy
- except in the case $k = 1$ which is equivilent to value iteration

iteration 3, $\pi =$



Value iteration definition

Bellman optimality equation

If we recap the definition of the optimal value function according to the Bellman optimality equation:

$$
\begin{aligned}
v_*(s) &= \max_a q_*(s,a) \\
&= \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')
\end{aligned}
$$

We can also iteratively apply the update with the one-step look-ahead to learn $v_*(s)$

Algorithm: value iteration

```
def value_iteration(env, γ, theta):
  V = np.zeros(env.nS)
  while True:
    delta = 0
    for s in range(env.nS):
      v_s = V[s]
      q_s = np.zeros(env.nA)
      for a in range(env.nA):
        for prob, s', reward, done in env.P[s][a]:
          q_s[a] += prob * (reward + γ * V[s'])
      V[s] = max(q_s)
      delta = max(delta, abs(V[s] - v_s))
    if delta < theta: break
  policy = greedily_from(env, V, gamma)
  return policy, V
```

Take Away Points

Summary

In summary, dynamic programming:

- solves the planning problem, but not the full reinforcement learning problem
- requires a complete model of the environment
- policy evaluation solves the prediction problem
- there's a spectrum between policy iteration and value iteration
- these solve the control problem

Extensions:

- Asynchronous DP (read section 4.5 of Sutton & Barto [1])
- Play with the interactive demo by Andrej Karpathy

References I

[1] Richard S Sutton and Andrew G Barto. Reinforcement learning: An introduction (second edition). Available online . MIT press, 2018.

[2] David Silver. Reinforcement Learning lectures. https://www.davidsilver.uk/teaching/. 2015.