Tufts

Class #22: Solving MDPs & Reinforcement Learning

Machine Learning (COMP 135): M. Allen, 20 Nov. 19

1

Review: The Bellman Equation

- Richard Bellman (1957), working in Control Theory, was able to show that the utility of any state s, given policy of action π, can be defined recursively in terms of the utility of any states we can get to from s by taking the action that π dictates:

$$U^{\pi}(s) \;=\; \sum_{s'} P(s,\pi(s),s')\;[R(s,\pi(s),s') + \gamma\, U^{\pi}(s')]$$

- Furthermore, he showed how to actually calculate this value using an iterative dynamic programming algorithm

2

Evaluating a Policy Iteratively

function POLICY-EVALUATION(mdp, π) **returns** a value function
  inputs: mdp, an MDP, and π, a policy to be evaluated
  local variables: Δ, maximal amount policy values change per iteration,
    Θ, a small positive constant

  $\forall s \in S:\; U(S) = 0$
  repeat while $\Delta \geq \Theta$
    $\Delta \leftarrow 0$
    $\forall s \in S:$
      $u \leftarrow U(s)$
      $U(s) \leftarrow \sum_{s'} P(s,\pi(s),s')[R(s,\pi(s),s') + \gamma\, U(s')]$
      $\Delta \leftarrow \max(\Delta,\, |U(s) - u|)$

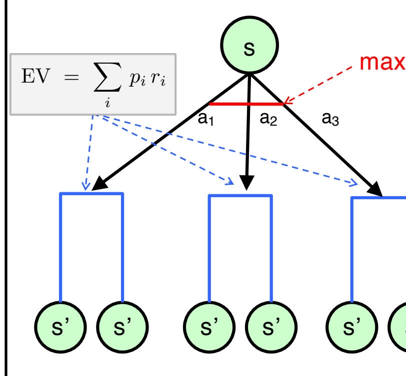
  return value function $U \approx U^{\pi}$

Note: if we set Θ to $\frac{\varepsilon\,(1-\gamma)}{\gamma}$ ***approximation error*** is at most ε

- Policy evaluation: given a policy, we calculate the expected value for every state if we follow the policy, iterating until values converge (quit changing very much)

3

Finding the Optimal Policy (π^*)



- Before, to calculate $U^{\pi}(s)$, we only looked at ***single set*** of actions: those given for each state by policy,

$$a_i = \pi(s_i)$$

- Now, we will consider ***all possible actions***, taking the one that is the best at each state before we sum over the various probabilities and rewards in the system

4

Bellman Equations

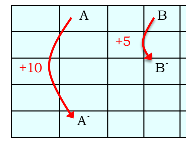
- We have seen that the utility of any state s in a given policy π can be calculated iteratively:

$$U^\pi(s) \;=\; \sum_{s'} P(s,\pi(s),s')\;[R(s,\pi(s),s') + \gamma\, U^\pi(s')]$$

- This same equation can be used to find the value of the ***best possible policy***, simply by calculating what we get if we always take the best action:

$$U^\star(s) = \max_\pi U^\pi(s)$$
$$= \max_a \sum_{s'} P(s,a,s')\;[R(s,a,s') + \gamma\, U^\star(s')]$$

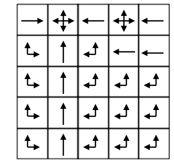
Solving for the Optimal Policy



Problem domain

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

U*



π*

- Before, we looked at the value of the ***purely random*** policy for this particular grid problem
- We can use the Bellman Equation to find the ***optimal policy***
  - Here we see the optimal value function, U*, and the associated optimal policy, π* (where in some cases, multiple actions are all equally good/bad)

Example from: Sutton & Barto, 1998

Policy Improvement

- Once we figure out the value for each state under our ***current*** policy, we can choose ***new actions***

$$U^\pi(s) = \sum_{s'} P(s,\pi(s),s')[R(s,\pi(s),s') + \gamma\, U(s')]$$

$$\pi'(s) = \arg\max_a \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma\, U(s')]$$

- Our choice is simple: just set our new policy in a ***greedy way***, choosing the best action available
  - This choice is based on the ***current set*** of values
  - Creates a new policy when we change some action
  - If the policy ***does change***, then we need to update our values again

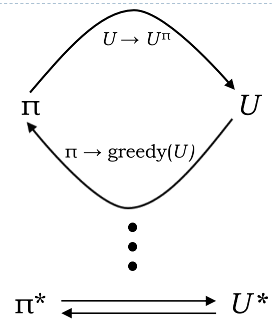
Improving Policies Iteratively

```
function POLICY-ITERATION(mdp) returns a policy
    inputs: mdp, an MDP
    local variables: U, a vector of utility values for states s ∈ S,
                     π, a policy to be updated

    ∀s ∈ S : U(S) = 0 and π(s) = a random action
    repeat while changed? = true
        U ← POLICY-EVALUATION(mdp, π)
        changed? ← false
        ∀s ∈ S :
            a ← π(s)
            π(s) ← arg max_{a∈A} Σ_{s'} P(s,a,s')[R(s,a,s') + γ U(s')]
            if : π(s) ≠ a, then : changed? ← true

    return π
```

- Again, a simple iterative algorithm:
  1. ***Evaluate*** the current policy.
  2. Set all actions to ***best ones*** found when evaluating.
  3. If the policy has changed, ***repeat***.
  4. When no action changes, ***end***.

Policy Iteration

- It can be shown that in time, this process will converge to a policy π* with value function U*, that is ***nearly*** optimal
- As with policy evaluation, we can put bounds on the amount of non-optimality (based on the value-update parameter Δ)



Learning the Value of a Policy

- The dynamic programming algorithm we have seen works fine if we ***already know everything*** about an MDP system, including:
  1. Probabilities of all state-action transitions
  2. Rewards we get in each case
- If we ***don't*** have this information, how can we figure out the value of a policy?
  - Turns out we can use a ***sampling method***
  - "Follow the policy, and see what happens"

Temporal Difference (TD) Updates

```
function TD-POLICY-EVALUATION(mdp, π) returns a value function
    inputs: mdp, an MDP, and π, a policy to be evaluated

    ∀s ∈ S, U(s) = 0
    repeat for each episode E:
        set start-state s ← s_0
        repeat for each time-step t of episode E, until s is terminal:
            take action π(s)
            observe: next state s′, one-step reward r
            U(s) ← U(s) + α[r + γU(s′) − U(s)]
            s ← s′

    return value function U ≈ U^π
```

- Agent in an MDP takes actions, sees new states and rewards
  - This information can be used to update the value function
  - This is slightly modified from text, to simplify it

Temporal Difference (TD) Updates


```
function TD-POLICY-EVALUATION(mdp, π) returns a value function
    inputs: mdp, an MDP, and π, a policy to be evaluated

    ∀s ∈ S, U(s) = 0
    repeat for each episode E:
        set start-state s ← s_0
        repeat for each time-step t of episode E, until s is terminal:
            take action π(s)
            observe: next state s′, one-step reward r
            U(s) ← U(s) + α[r + γU(s′) − U(s)]
            s ← s′

    return value function U ≈ U^π
```


- Now, we ***don't*** base value-update on a probability distribution
- Instead, based on the ***single state*** we actually see, over and over again, stopping whenever we hit a terminal condition, for some number of learning episodes

The Basic TD(0) Update

$$U(s) = U(s) + \alpha[r + \gamma U(s') - U(s)]$$

- When we make one-step update, we add one-step reward that we get, r, plus the difference between where we start, $U(s)$, and where we end up $U(s')$, discounted by the factor γ as usual
- If state where we end up s' is ***better*** after discounting, then the value of original state s goes ***up***
- If s' is ***worse***, the value of s goes ***down***

The Basic TD(0) Update

$$U(s) = U(s) + \alpha[r + \gamma U(s') - U(s)]$$

- We also weight the value-update amount by another constant α, (less than 1), called a step-size parameter

1. If this value shrinks to 0 over time, values stop changing
2. If we do this ***slowly***, the update will eventually ***converge to actual value*** of state if we follow the policy π
   - For example, if we update over episodes, $e = 1, 2, 3, \ldots$, we can set the parameter for each episode to be:

$$\alpha_e = \frac{1}{e}$$

Advantages and a Problem

- With TD updates, we only update the states we ***actually see*** given the policy we are following
  - Don't need to know MDP dynamics
  - May only have to update very few states, saving much time to get the values of those we ***actually reach*** under our policy
- However, this can be a source of difficulty: we may not be able to find a ***better*** policy, since we don't know values of states that we never happen to visit

Exploration and Exploitation

- If we use the Dynamic Programming method, we calculate the value of ***every state***
  - Easy to update policy (just be greedy)
  - This is exploitation: use best values seen to choose actions
- When we are learning, however, we sometimes don't know what certain states are like, because we've never actually seen them yet
  - Our current policy may never get us to things we really want
  - Thus, we must use exploration: try out things even if our current best policy ***doesn't think*** it's a good idea

Almost-Greedy Policies

- One simple way to add exploration is to use a policy that is ***mostly*** greedy, but ***not always***
- An "epsilon-greedy" (ε-greedy) policy sets some probability threshold, ε, and chooses actions by:
  1. Picking a random number $R \in [0, 1]$
  2. If $R \leq \varepsilon$, choosing the action ***at random***
  3. If $R > \varepsilon$, acting in a ***greedy*** fashion (as before)

Learning with ε-greedy policies

- We can add this idea to our sampling update method
- After we take an action, and see a state-transition from s to s', we do the same updates as before:

$$U(s) = U(s) + \alpha[r + \gamma U(s') - U(s)]$$

- When we choose actions, we do so in an ε-greedy way, ***sometimes*** following the policy based on ***learned values***, and ***sometimes trying random things***
- Over enough time, this can converge to true value function U^* of the optimal policy π^*

TD-Learning

```
function TD-LEARNING(mdp) returns a policy
    inputs: mdp, an MDP

    ∀s ∈ S, U(s) = 0
    repeat for each episode E:
        set start-state s ← s_0
        repeat for each time-step t of episode E, until s is terminal:
            choose action a, using ε-greedy policy based on U(s)
            observe next state s', one-step reward r
            U(s) ← U(s) + α[r + γU(s') − U(s)]
            s ← s'

    return policy π, set greedily for every state s ∈ S, based upon U(s)
```

- Algorithm is the same, but ***explores*** using sometimes-greedy and sometimes-probabilistic action-choices instead of fixed policy π
  - We reduce learning parameter α just as before to converge

Next Few Weeks

- **Topics**: Reinforcement Learning
- **HW 05**: due Wednesday, 20 November, 9:00 AM
- **Project 02**: due Monday, 25 November, 9:00 AM
- **Office Hours**: 237 Halligan, Tuesday, 11:00 AM – 1:00 PM
  - TA hours can be found on class website as well