Tufts

Class #25:
Reinforcement Learning

Machine Learning (COMP 135): M. Allen, 22 Apr. 20

1

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"

2

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

  $\forall s \in S,\ U(s) = 0$
  repeat for each episode E:
    set start-state $s \leftarrow s_0$
    repeat for each time-step t of episode E, until s is terminal:
      take action $\pi(s)$
      observe: next state s', one-step reward r
      $U(s) \leftarrow U(s) + \alpha[r + \gamma\, U(s') - U(s)]$
      $s \leftarrow s'$

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

- Agent in an MDP takes actions, sees new states and rewards
- 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

3

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*** than original state s after discounting, then the value of s goes ***up***
- If s' is ***worse than*** s, the value of s goes ***down***

4

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 + γ U(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

Randomness and Weighting in Learning

- Our algorithm uses two parameters, α and ε (plus the usual discount factor γ), to control its overall behavior
- Each can be adapted over time to control algorithm

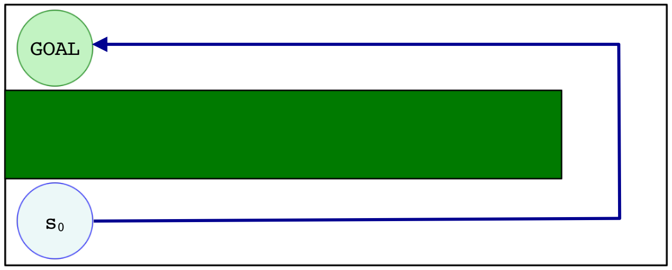
1. ε: the amount of randomness in the policy
   - When we don't know much, set it to a ***high value***, so that we start off with lots of random exploration
   - We ***reduce*** this value over time until ε = 0, and we are being purely greedy, and just exploiting what he have learned
2. α: the weight on each learning-update step
   - Reduce this over time, as well: when $\alpha = 0$, U-values don't change anymore, and we can converge on final policy values

Randomness and Weighting in Learning

- The control parameters α and ε give us simple ways to control complex learning behavior
- We ***don't always*** want to reduce each over time
- In a purely ***stationary environment***, where system dynamics don't ever change, and all probabilities stay the same, we can simply slowly reduce each until we converge upon a stable learned behavior
- In a ***non-stationary*** environment, where things may change at some point, learned solutions may quit working

Non-Stationary Environments

- Suppose environment starts off in one configuration:



- Over time, we can learn a policy for shortest path to goal
- By letting ε and α go to 0, the policy becomes stable

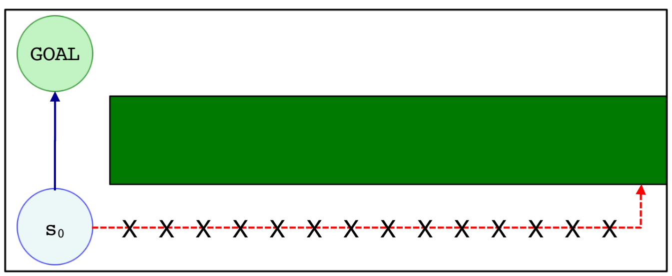
Non-Stationary Environments

- The environment may change, however:

- If ε and α ***stay at*** 0, policy is ***sub-optimal*** from now on

Non-Stationary Environments

- We may be able to tell that environment changes, however



- If value drops off over a long time, we can ***increase*** ε and α again, to resume learning and find new optimal policy

Bellman Equations for Q-values

- Instead of the value of a ***state*** $U(s)$, we can calculate the value of a ***state-action pair*** $Q(s, a)$
- The value of taking action a in state s, and then following the policy π after that:

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

- Similarly, we calculate optimal values $Q^{*}(s, a)$ of taking a in state s, then following ***best*** possible policy after that:

$$Q^{\star}(s,a) = \sum_{s'} P(s,a,s')\ \left[R(s,a,s') + \gamma \max_{a'} Q^{\star}(s',a')\right]$$

- We can do learning for Q-values, too…

TD (*SARSA*) Learning for Q-values

function SARSA-LEARNING(mdp) **returns a policy**
  inputs: mdp, an MDP

  $\forall s \in S, \forall a \in A, Q(s,a) = 0$
  repeat for each episode E:
    set start-state $s \leftarrow s_0$
    set action a, chosen ϵ-greedily based on $Q(s,a)$
    repeat for each time-step t of episode E, until s is terminal:
      take action a
      observe next state s', one-step reward r
      set next action a', chosen ϵ-greedily based on $Q(s',a')$
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$; $a \leftarrow a'$

  return policy π, set greedily for every state $s \in S$, based upon $Q(s,a)$

- Same basic RL method, converging to optimal Q^*
- Called ***SARSA***, due to information used (s, a, r, s', a')

On-Policy Updates

function SARSA-LEARNING(mdp) **returns a policy**
  inputs: mdp, an MDP

  $\forall s \in S, \forall a \in A, Q(s,a) = 0$
  repeat for each episode E:
    set start-state $s \leftarrow s_0$
    set action a, chosen ϵ-greedily based on $Q(s,a)$
    repeat for each time-step t of episode E, until s is terminal:
      take action a
      observe next state s', one-step reward r
      set next action a', chosen ϵ-greedily based on $Q(s',a')$
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$; $a \leftarrow a'$

  return policy π, set greedily for every state $s \in S$, based upon $Q(s,a)$

- Both basic TD and SARSA are on-policy learning/update methods
- We choose our initial action (a) based on current ε-greedy policy

On-Policy Updates

function SARSA-LEARNING(mdp) **returns a policy**
  inputs: mdp, an MDP

  $\forall s \in S, \forall a \in A, Q(s,a) = 0$
  repeat for each episode E:
    set start-state $s \leftarrow s_0$
    set action a, chosen ϵ-greedily based on $Q(s,a)$
    repeat for each time-step t of episode E, until s is terminal:
      take action a
      observe next state s', one-step reward r
      set next action a', chosen ϵ-greedily based on $Q(s',a')$
      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, Q(s',a') - Q(s,a)]$
      $s \leftarrow s'$; $a \leftarrow a'$

  return policy π, set greedily for every state $s \in S$, based upon $Q(s,a)$

- When we do the value update, we ***also*** choose the next action (a') based on the same current ε-greedy policy

The Effect of On-Policy Updates

- When we do this sort of updating, we are ***not*** basing our value calculation on the ***best possible*** policy
- Instead, we are basing it on our ***learning policy***, which means the values that we base our updates and choices on will combine the values that we get from:

1. greedy action selection for exploitation
2. random actions in some states for exploration

- Values we learn can reflect what would happen in a state if we sometimes acted in a non-optimal way
  - For example, on the edge of a cliff, we will sometimes randomly explore jumping off the cliff when learning
  - Edge-states are thus risky, and get lower value than they would really have under the optimal policy (where we only do the best thing, and never jump)

Off-Policy Methods

- One possible solution is to update the values we learn based on the ***best actions only***
- That is, we ***ignore*** rewards and outcomes that come from any of the possible bad actions we take when exploring
- The policy being updated is then ***not*** the current learning version, but the optimal one
  - This is the policy that we wanted to learn in the end, anyway!
- How can we do this?

Q-Learning: Off-Policy Updates

function Q-LEARNING(mdp) **returns a policy**
inputs: mdp, an MDP

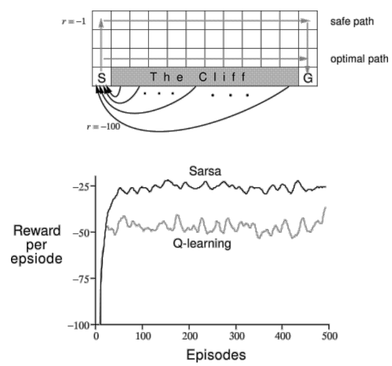
$\forall s \in S, \forall a \in A, Q(s,a) = 0$
repeat for each episode E:
  set start-state $s \leftarrow s_0$
  repeat for each time-step t of episode E, until s is terminal:
    set action a, chosen ϵ-greedily based on $Q(s,a)$
    take action a
    observe next state s', one-step reward r
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$

return policy π, set greedily for every state $s \in S$, based upon $Q(s,a)$

- We still choose actions (a) in an ε-greedy way (so we ***are*** sometimes random)
- However, we update values based upon whatever action would actually be ***best***

Comparing the Methods: Cliff Problem

- Shortest path to the goal goes along edge of a cliff
  - SARSA learns safer path, since edge-states get lower values due to random falls
  - Q-Learning learns best path, since it ***ignores*** random jumps off edge
  - Why does QL do worse in the end? How can we fix this over a period of time?



Example from: Sutton & Barto, 1998

Unifying the Methods

- Both SARSA and Q-Learning can be made to converge to the same optimal policy over time
- By reducing the epsilon-value in our ε-greedy policy, we eventually reduce the randomness
- Thus, the SARSA agent will eventually learn better values even for risky states, and come to use the optimal policy, too (e.g. walking along the cliff's edge)
- So what's the difference?
  - In many cases, Q-Learning can converge on values somewhat ***faster***
  - Doesn't have to spend time "fixing" the values of states where it has ***over-estimated*** negative risk
  - Thus we can reduce the ε-value more rapidly, and learn optimal state-values more quickly
  - What are the potential risks of doing this?

End of Semester

- **Topics**: Reinforcement Learning, Wrap-Up
- **Note**: Last live Q&A this Wednesday (none next week)
- **Project 02**: due Monday, 27 April, 5:00 PM
  - Sentiment analysis in review text
  - Uses two different models of textual data
- **Final Paper**: due Friday, 08 May, 5:00 PM
  - Prompt, rubric, and sample essay on Piazza now
- **Office Hours**:
  - Hours and Zoom links can be found on Piazza and Canvas