CS 188: Artificial Intelligence
MDPs + RL Review

Instructor: Nathan Lambert
University of California, Berkeley

Non-Deterministic Search

Example: Grid World

- A maze-like problem
- The agent lives in a grid
- Walls block the agent's path
- Noise movement: actions do not always go as planned
  - 80% of the time, the action North takes the agent North
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put
- The agent receives rewards each time step
  - Small "living" reward each step (can be negative)
  - Big rewards come at the end (good or bad)
- Goal: maximize sum of rewards

Grid World Actions

Deterministic Grid World

Stochastic Grid World

Markov Decision Processes

- An MDP is defined by:
  - A set of states \( s \in S \)
  - A set of actions \( a \in A \)
  - A transition function \( T(s, a, s') \)
    - Probability that a from \( s \) leads to \( s' \), i.e., \( P(s' \mid s, a) \)
    - Also called the model or the dynamics
  - A reward function \( R(s, a, s') \)
  - Sometimes just \( R(s, a) \)
  - A start state
  - Maybe a terminal state

Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal
- For MDPs, we want an optimal policy \( \pi^* : S \rightarrow A \)
  - A policy \( \pi \) gives an action for each state
  - An optimal policy is one that maximizes expected utility if followed
  - An explicit policy defines a reflex agent
Utilities & Discounting

- Utility: sum of discounted reward
- Each time we descend a level, we multiply in the discount once
- Why discount?
  - Think of it as a gamma chance of ending the process at every step
  - Also helps our algorithms converge
- Example: discount of 0.5
  - \( U([1,2,3]) = 1 \times 1 + 0.5 \times 2 + 0.25 \times 3 \)
  - \( U([1,2,3]) < U([3,2,1]) \)

Optimal Quantities

- The value (utility) of a state \( s \):
  \( V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \)
- The value (utility) of a q-state \( (s,a) \):
  \( Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \)
- The optimal policy:
  \( \pi^*(s) = \text{optimal action from state } s \)

Values of States

- Recursive definition of value:
  \( V^*(s) = \max_a \ Q^*(s,a) \)
  \( Q^*(s,a) = \sum_{s'} T(s,a,s') R(s,a,s') + \gamma V^*(s') \)
  \( V^*(s) = \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V^*(s')] \)

Snapshot of Demo – Gridworld V Values

- Noise = 0.2
- Discount = 0.9
- Living reward = 0

Snapshot of Demo – Gridworld Q Values

- Noise = 0.2
- Discount = 0.9
- Living reward = 0

Solving MDPs
Racing Search Tree

- We're doing way too much work with expectimax!
- Problem: States are repeated
  - Idea: Only compute needed quantities once
- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
- Note: deep parts of the tree eventually don't matter if $\gamma < 1$

Value Iteration

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_0(s)$ values, do one ply of expectimax from each state:
  $$V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V_k(s')]$$
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do

Convergence*

- How do we know the $V_k$ vectors are going to converge?
- Case 1: If the tree has maximum depth $M$, then $V_M$ holds the actual untruncated values
- Case 2: If the discount is less than 1
  - Sketch: For any state $V_k$ and $V_{k+1}$ can be viewed as depth $k+1$ expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, $V_k$ has actual rewards while $V_{k+1}$ has zeros
  - That last layer is at best all $R_{MAX}$
  - It is at worst $R_{MIN}$
  - But everything is discounted by $\gamma$ that far out
  - So $V_k$ and $V_{k+1}$ are at most $(\gamma + \gamma^2 + \cdots)$ different
  - So as $k$ increases, the values converge

Policy Extraction

- Let's imagine we have the optimal values $V^*(s)$
- How should we act?
  - It's not obvious!
- We need to do a mini-expectimax (one step)
  $$\pi^*(s) = \arg \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V^*(s')]$$
- This is called policy extraction, since it gets the policy implied by the values

Computing Actions from Values

- Let's imagine we have the optimal values $V^*(s)$
- How should we act?
  - It's not obvious!
- We need to do a mini-expectimax (one step)
  $$\pi^*(s) = \arg \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V^*(s')]$$
- This is called policy extraction, since it gets the policy implied by the values

Computing Actions from Q-Values

- Let's imagine we have the optimal q-values:
- How should we act?
  - Completely trivial to decide!
  $$\pi^*(s) = \arg \max_a Q^*(s,a)$$
- Important lesson: actions are easier to select from q-values than values!
Problems with Value Iteration

- Value iteration repeats the Bellman updates:
  \[ V_{k+1}(s) = \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V_k(s')] \]
- Problem 1: It’s slow – \( O(S^2A) \) per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values

Policy Iteration

- Alternative approach for optimal values:
  - Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities) until convergence
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal) utilities as future values
  - Repeat steps until policy converges
- This is policy iteration
  - It’s still optimal!
  - Can converge (much) faster under some conditions

Policy Evaluation

- How do we calculate the \( \bar{V} \)'s for a fixed policy \( \pi \)?
  - Idea 1: Turn recursive Bellman equations into updates (like value iteration)
    \[ V_{k+1}^\pi(s) = \zeta \]
    \[ V_{k+1}^\pi(s) = \sum_{s'} T(s,\pi(s),s') [R(s,\pi(s),s') + \gamma V_k^\pi(s')] \]
  - Efficiency: \( O(S^2) \) per iteration
  - Idea 2: Without the maxes, the Bellman equations are just a linear system
    - Solve with Matlab (or your favorite linear system solver)

Policy Iteration

- Evaluation: For fixed current policy \( \pi \), find values with policy evaluation:
  - Iterate until values converge:
    \[ V_{k+1}^\pi(s) = \sum_{s'} T(s,\pi(s),s') [R(s,\pi(s),s') + \gamma V_k^\pi(s')] \]
- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
    \[ \pi_{k+1}(s) = \arg \max_a \sum_{s'} T(s,a,s') [R(s,a,s') + \gamma V_k^\pi(s')] \]

Comparison

- Both value iteration and policy iteration compute the same thing (all optimal values)
- In value iteration:
  - Every iteration updates both the values and (implicitly) the policy
  - We don’t track the policy, but taking the max over actions implicitly recomputes it
- In policy iteration:
  - We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  - After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  - The new policy will be better (or we’re done)
- Both are dynamic programs for solving MDPs
Summary: MDP Equations

- Value iteration equation:
  \[ V_{k+1}(s) = \max_a \sum_s T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Policy evaluation equation:
  \[ V^\pi_k(s) = \sum_a \sum_s T(s, a, s') \left[ R(s, a, s') + \gamma V^\pi_{k-1}(s') \right] \]

- Policy iteration equation:
  \[ \pi_{k+1}(s) = \arg\max_a \sum_s T(s, a, s') \left[ R(s, a, s') + \gamma V^\pi_k(s') \right] \]

Summary: MDP Algorithms

- So you want to…
  - Compute optimal values: use value iteration or policy iteration
  - Compute values for a particular policy: use policy evaluation
  - Turn your values into a policy: use policy extraction (one-step lookahead)

- These all look the same!
  - They basically are – they are all variations of Bellman updates
  - They all use one-step lookahead
  - They differ only in whether we plug in a fixed policy or max over actions

The Bellman Equations

How to be optimal:
- Step 1: Take correct first action
- Step 2: Keep being optimal

Next Time: Reinforcement Learning!

Reinforcement Learning

- Still assume a Markov decision process (MDP):
  - A set of states \( s \in S \)
  - A set of actions (per state) \( A \)
  - A model \( T(s,a,s') \)
  - A reward function \( R(s,a,s') \)
  - Still looking for a policy \( \pi(s) \)

- New twist: don’t know \( T \) or \( R \)
  - I.e. we don’t know which states are good or what the actions do
  - Must actually try actions and states out to learn
Reinforcement Learning

- Basic idea:
  - Receive feedback in the form of rewards
  - Agent’s utility is defined by the reward function
  - Must (learn to) act so as to maximize expected rewards
  - All learning is based on observed samples of outcomes

Environment

Agent

Actions: a

State: s

Reward: r

Model-Based Learning

- Model-Based Idea:
  - Learn an approximate model based on experiences
  - Solve for values as if the learned model were correct
  - Count outcomes s’ for each s, a
  - Normalize to give an estimate \( \hat{T}(s,a,s’) \)
  - Discover each \( \hat{T}(s,a,s’) \) when we experience \( (s,a,s’) \)

- Step 2: Solve the learned MDP
  - For example, use value iteration, as before

Model-Free Learning

Direct Evaluation

- Goal: Compute values for each state under \( \pi \)
- Idea: Average together observed sample values
  - Act according to \( \pi \)
  - Every time you visit a state, write down what the sum of discounted rewards turned out to be
  - Average those samples
  - This is called direct evaluation

Problems with Direct Evaluation

- What’s good about direct evaluation?
  - It’s easy to understand
  - It doesn’t require any knowledge of \( T, R \)
  - It eventually computes the correct average values, using just sample transitions

- What bad about it?
  - It wastes information about state connections
  - Each state must be learned separately
  - So, it takes a long time to learn

Output Values
**Sample-Based Policy Evaluation?**

- We want to improve our estimate of \( V \) by computing these averages:
  \[
  v_{k+1}^n(s) = \sum \mathbb{P}(s, \pi(s), s')[R(s, \pi(s), s') + \gamma v_k^n(s')] \]
- Idea: Take samples of outcomes \( s' \) (by doing the action!) and average
  \[
  \text{sample}_1 = R(s, \pi(s), s'_1) + \gamma v_k^n(s'_1) \\
  \text{sample}_2 = R(s, \pi(s), s'_2) + \gamma v_k^n(s'_2) \\
  \vdots \\
  \text{sample}_n = R(s, \pi(s), s'_n) + \gamma v_k^n(s'_n) \\
  v_{k+1}^n(s) = \frac{1}{n} \sum \text{sample}_i
  \]

**Temporal Difference Learning**

- Big idea: learn from every experience!
  - Update \( V(s) \) each time we experience a transition \((s, a, s', r)\)
  - Likely outcomes \( s' \) still contribute updates more often
- Temporal difference learning of values
  - Policy still fixed, still doing evaluation!
  - More values toward value of whatever successor occurs running average

**Problems with TD Value Learning**

- TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages
- However, if we want to turn values into a (new) policy, we’re sunk:
  \[
  \pi(s) = \arg \max \left( \sum_{s', r} P(s, a, s', r) Q(s, a, s', r) \right)
  \]
- Idea: learn \( Q \)-values, not values
  - Makes action selection model-free too!

**Q-Learning**

- Q-Learning: sample-based \( Q \)-value iteration
  \[
  Q_{k+1}(s, a) = \sum \mathbb{P}(s, a, s') \left[R(s, a, s') + \gamma \max a' Q_k(s', a') \right]
  \]
- Learn \( Q(s, a) \) values as you go
  - Receive a sample \((s, a, s', r)\)
  - Consider your old estimate \( Q(s, a) \)
  - Consider your new sample estimate:
    \[
    \text{sample} = R(s, a, s') + \gamma \max a' Q_k(s', a') \]
  - Incorporate the new estimate into a running average:
    \[
    Q(s, a) = (1 - \alpha) Q(s, a) + \alpha \text{sample}
    \]

**Q-Learning Properties**

- Amazing result: Q-learning converges to optimal policy -- even if you’re acting suboptimally?
- This is called off-policy learning

- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - \( \ldots \) but not decrease it too quickly
  - Basically, in the limit, it doesn’t matter how you select actions! (?)
Exploration vs. Exploitation

How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions (ε-greedy)
    - Every time step, flip a coin
      - With (small) probability ε, act randomly
      - With (large) probability 1-ε, act on current policy
  - Problems with random actions?
    - You do eventually explore the space, but keep thrashing around once learning is done
  - One solution: lower ε over time
  - Another solution: exploration functions

Regret

- Even if you learn the optimal policy, you still make mistakes along the way!
- Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards
- Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal
- Example: random exploration and exploration functions both end up optimal, but random exploration has higher regret

Generalizing Across States

- Basic Q-Learning keeps a table of all q-values
- In realistic situations, we cannot possibly learn about every single state!
- Too many states to visit them all in training
- Too many states to hold the q-tables in memory
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar situations
  - This is a fundamental idea in machine learning, and we’ll see it over and over again

Feature-Based Representations

- Solution: describe a state using a vector of features (properties)
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - 1 / (dist to dot)^2
    - Is Pacman in a tunnel? (0/1)
    - … etc.
  - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)

Linear Value Functions

- Using a feature representation, we can write a q-function (or value function) for any state using a few weights:

\[
V(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s)
\]

\[
Q(s, a) = w_1f_1(s, a) + w_2f_2(s, a) + \ldots + w_nf_n(s, a)
\]

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but actually be very different in value!
Approximate Q-Learning

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- Q-learning with linear Q-functions:
  - transition = \((s, a, r, s')\)
  - difference = \(r + \gamma \max_{\alpha} Q(s', \alpha) - Q(s, a)\)
  - \(Q(s, a) = Q(s, a) + \alpha \text{ difference} f_i(s, a)\)

  Exact Q's:
  - Intuitive interpretation:
    - Adjust weights of active features.
    - E.g., if something unexpectedly bad happens, blame the features that were on and dispose all states with that state's features.
  - Formal justification: online least squares

Linear Approximation: Regression

\[ \hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \]

Prediction:
- \(\hat{y} = \beta_0 + \beta_1 x_1\)
- \(\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2\)

Policy Search

- Problem: often the feature-based policies that work well (win games, maximize utilities) aren’t the ones that approximate V / Q best
  - E.g. your value functions from project 2 were probably horrible estimates of future rewards, but they still produced good decisions.
  - Q-learning’s priority: get Q-values close (modeling)
  - Action selection priority: get ordering of Q-values right (prediction)
  - We’ll see this distinction between modeling and prediction again later in the course

- Solution: learn policies that maximize rewards, not the values that predict them

- Policy search: start with an ok solution (e.g. Q-learning) then fine-tune by hill climbing on feature weights

Next time (with me):
Machine Learning!