How to Explore?

- Several schemes for forcing exploration
  - Simplest: random actions ($\epsilon$-greedy)
  - Every time step, flip a coin
  - With small probability $\epsilon$, act randomly
  - With large probability $1-\epsilon$, 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 $\epsilon$ over time
  - Another solution: exploration functions

Exploration Functions

- When to explore?
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established, eventually stop exploring

- Exploration function
  - Takes a value estimate $u$ and a visit count $n$, and returns an optimistic utility, e.g. $f(u, n) = u + \frac{k}{n}$
  - Regular Q-Update: $Q(s, a) \leftarrow \max_{a'} Q(s', a') + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$
  - Modified Q-Update: $Q(s, a) \leftarrow \max_{a'} Q(s', a') + \gamma \max_{a'} f(Q(s', a'), N(s', a'))$
  - Note: this propagates the “bonus” back to states that lead to unknown states as well
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.

Approximate Q-Learning

- 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 all in training.
  - Too many states to hold the q-table 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.

Example: Pacman

Let’s say we discover through experience that this state is bad:

In naïve q-learning, we know nothing about this state:

Or even this one!

Video of Demo Q-Learning Pacman – Tiny – Watch All
Feature-Based Representations

- Solution: describe a state using a vector of features (aka "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
  - % (state to empty)
  - Is Pacman in a tunnel? (0/1)
  - ... etc.
- Is it the exact state on this slide?
- 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-learning with linear Q-functions:
  \[ Q(s, a) = w_1f_1(s, a) + w_2f_2(s, a) + \ldots + w_nf_n(s, a), \]
- Intuitive interpretation:
  - Adjust weights of active features.
  - E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state’s features.
- Formal justification: online least squares.

Example: Q-Pacman

- Exact Q’s
  \[ Q(s, a) = 4.0f_{DORT}(s, a) - 1.0f_{GOST}(s, a), \]
  \[ f_{DORT}(s, NORTH) = 0.5 \]
  \[ f_{GOST}(s, NORTH) = 1.0 \]
  \[ Q(s, a) = w_1f_1(s, a) + \gamma \max_{a'} Q(s', a') \]
  \[ Q(s, a) = -\gamma \max_{a'} Q(s', a') \]
- Approximate Q’s
  \[ Q(s, a) = 3.0f_{DORT}(s, a) - 3.0f_{GOST}(s, a), \]
Video of Demo Approximate Q-Learning -- Pacman

Q-Learning and Least Squares

Linear Approximation: Regression*

- Prediction: $\hat{y} = w_0 + w_1 f_1(x)$

Optimization: Least Squares*

- Total error: $\sum (y_i - \hat{y}_i)^2 = \sum \left(y_i - \sum w_i f_i(x_i)\right)^2$

Minimizing Error*

Imagine we had only one point $x$, with features $f(x)$, target value $y$, and weights $w$:

- Error: $\sum (y_i - \hat{y}_i)^2$
- Gradient: $\frac{\partial \text{error}}{\partial w_n} = \left[ \sum f_i(x) \right] f_n(x)$
- Update: $w_n \leftarrow w_n + \alpha \left[ \sum f_i(x) \right] f_n(x)$

Overfitting: Why Limiting Capacity Can Help*

Approximate $q$ update explained:

$$w_n \leftarrow w_n + \alpha \left[ \gamma \max_{Q'} Q'(x', a) - Q(x, a) \right] f_n(x, a)$$

"target"  "prediction"
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)

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.

Simplest policy search:
- Start with an initial linear value function or Q-function
- Nudge each feature weight up and down and see if your policy is better than before

Problems:
- How do we tell the policy got better?
- Need to run many sample episodes!
- If there are a lot of features, this can be impractical

Better methods exploit lookahead structure, sample wisely, change multiple parameters...

PILCO (Probabilistic Inference for Learning Control)

- Model-based policy search to minimize given cost function
- Policy: mapping from state to control
- Rollout: plan using current policy and GP dynamics model
- Policy parameter update via CG/BFGS
- Highly data efficient

Demo: Standard Benchmark Problem

- Swing pendulum up and balance in inverted position
- Learn nonlinear control from scratch
- 4D state space, 300 controller parameters
- 7 trials/17.5 sec experience
- Control freq.: 10 Hz
Controlling a Low-Cost Robotic Manipulator

• Low-cost system ($500 for robot arm and Kinect)
• Very noisy
• No sensor information about robot’s joint configuration used
• Goal: Learn to stack tower of 5 blocks from scratch
  • Kinect camera for tracking block in end-effector
  • State: coordinates (3D) of block center (from Kinect)
  • 4 controlled DoF
  • 20 learning trials for stacking 5 blocks (5 seconds long each)
  • Account for system noise, e.g.,
    • Robot arm
    • Image processing

Deepmind AI Playing Atari

That’s all for Reinforcement Learning!

• Very tough problem: How to perform any task well in an unknown, noisy environment!
• Traditionally used mostly for robotics, but becoming more widely used
• Lots of open research areas:
  • How to best balance exploration and exploitation?
  • How to deal with cases where we don’t know a good state/feature representation?

Conclusion

• We’ve done with Part I: Search and Planning!
• We’ve seen how AI methods can solve problems in:
  • Search
  • Constraint Satisfaction Problems
  • Games
  • Markov Decision Problems
  • Reinforcement Learning
• Next up: Part II: Uncertainty and Learning!