Policy Search
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
Policy Search

- 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...
Q-Learning and Least Squares
Linear Approximation: Regression

Prediction:
\[ \hat{y} = w_0 + w_1 f_1(x) \]

Prediction:
\[ \hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x) \]
Optimization: Least Squares

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left( y_i - \sum_k w_k f_k(x_i) \right)^2$$
Loss Function

\[
\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left( y_i - \sum_k w_k f_k(x_i) \right)^2
\]
Gradients

\[ y = x_1^2 + x_2^2 \]
\[ \frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \]
\[ \frac{\partial y}{\partial x_1} = 2x_1 \]
\[ \frac{\partial y}{\partial x_2} = 2x_2 \]

- Partial derivative: the immediate change of output for a change of input, or slope, or rate.
- Gradient: vector of partial derivatives, one per input scalar.
- Defines tangent plane.
- Gradient points in the direction of fastest increase.
  - Actually, on the tangent plane, so only in a region around the dot for the actual function.
Gradients

\[ \Delta y = \nabla y \ast \Delta x \]

\[ \Delta y = \left[ \frac{\partial y}{\partial x_1} \right] \ast \left[ \frac{\Delta x_1}{\Delta x_2} \right] \]

\[ \Delta y = \nabla y \ast (\Delta_1 x + \Delta_2 x) \]

\[ \Delta y = \nabla y \ast \Delta_1 x + \nabla y \ast \Delta_2 x \]

\[ \Delta y = \nabla y \ast \Delta_1 x + 0 \]
Minimizing Error

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

\[
\text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2
\]

\[
\frac{\partial \text{error}(w)}{\partial w_m} = -\left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

\[
w_m \leftarrow w_m + \alpha \left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

Approximate q update explained:

\[
w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a') - Q(s, a) \right] f_m(s, a)
\]

“target”  “prediction”
Updating w

Prediction:
\[ \hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x) \]
Updating $w$

- "Rotating" $w$

Prediction:

$$\hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x)$$
Updating w

- Bias
Updating w

- Bias
Overfitting: Why Limiting Capacity Can Help*
Actor-Critic Algorithms

- Analogous to Policy Iteration, we will have $V^\pi$ and $\pi$.
- $\pi$ is no longer deterministic. Policy is now $P(a|s)$.

1. take action $a$ based on $\pi(a|s)$, get $(s, a, s', r)$
2. update $V^\pi$ based on $r + \gamma V^\pi(s')$
3. update $\pi$ based on $r + \gamma V^\pi(s') - V^\pi(s)$ weighted by $\nabla \log \pi(a|s)$
   1. pretend it’s weighted by features of $\pi(a|s)$
Optimism
Double Deep Q-Network

- Deep Q-Network = Deep Neural Network estimating Q values.

1. checkpoint DQN into Q’
2. iterate:
   1. collect samples \((s, a, s', r)\)
      1. ideally, some are based on Q
   2. update Q based on \(r + \gamma Q(s', a = \text{argmax } Q') - Q(s, a)\)
Generalization

online RL setting

offline RL setting
Why Off-Policy

**standard real-world RL process**
1. instrument the task so that we can run RL
   - safety mechanisms
   - autonomous collection
   - rewards, resets, etc.
2. wait a long time for online RL to run
3. change the algorithm in some small way
4. throw it all in the garbage and start over for the next task

**offline RL process**
1. collect initial dataset
   - human-provided
   - scripted controller
   - baseline policy
   - all of the above
2. Train a policy offline
3. change the algorithm in some small way
4. collect more data, add to growing dataset
5. keep the dataset and use it again for the next project!
Why Multiple Algorithms

- simpler optimization math
- less finicky hyperparameters
- faster update steps
- on-policy

More efficient (fewer samples)

- model-based shallow RL
- model-based deep RL
- off-policy Q-function learning
- actor-critic style methods

Less efficient (more samples)

- on-policy policy gradient algorithms
- evolutionary or gradient-free algorithms
Iteration 0
RL: Learning Manipulation

Levine*, Finn*, Darrell, Abbeel, JMLR 2016
OpenAI: Dactyl
Trained with domain randomization