Fitted Dynamic Programming

Lucas Janson and Sham Kakade

CS/Stat 184: Introduction to Reinforcement Learning
Fall 2023

Today

- Feedback from last lecture
- Recap
- Neural networks
- Fitted value iteration
- Fitted policy iteration

Feedback from feedback forms

Feedback from feedback forms

1. Thank you to everyone who filled out the forms!

Feedback from feedback forms

1. Thank you to everyone who filled out the forms!
2.

Today

- Feedback from last lecture
- Recap
- Neural networks
- Fitted value iteration
- Fitted policy iteration

Recap

Recap

To approximate $\mathbb{E}[y \mid x]$ from data, can use Empirical Risk Minimization (ERM):

$$\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Recap

To approximate $\mathbb{E}[y \mid x]$ from data, can use Empirical Risk Minimization (ERM):

$$\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Choose $\mathcal{F}$ based on approximation, complexity, and optimization criterion

Recap

To approximate $\mathbb{E}[y \mid x]$ from data, can use Empirical Risk Minimization (ERM):

$$\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Choose $\mathcal{F}$ based on approximation, complexity, and optimization criterion

Optimize via gradient descent (GD) or stochastic gradient descent (SGD)

Recap

To approximate $\mathbb{E}[y \mid x]$ from data, can use Empirical Risk Minimization (ERM):

$$\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Choose $\mathcal{F}$ based on approximation, complexity, and optimization criterion

Optimize via gradient descent (GD) or stochastic gradient descent (SGD)

Linear regression parameterizes $f(x)$ as $x^\top \theta$ and can work well when $\mathbb{E}[y \mid x]$ very smooth, high-dimensional (penalties like ridge/lasso help here), and/or there is a good featurization $\phi(x)$

Today

- ✓ Feedback from last lecture
- ✓ Recap
- Neural networks
- Fitted value iteration
- Fitted policy iteration

Neural network model

Neural network model

Building blocks:

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x + b_1) + b_2$. Can think of as:

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2 \sigma(W_1 x + b_1) + b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a,0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x+b_1)+b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,
2. Linearly transform with $W_1 \in \mathbb{R}^{m\times d}$ and $b_1 \in \mathbb{R}^m$ to get $W_1x+b_1 \in \mathbb{R}^m$

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a,0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x+b_1)+b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,
2. Linearly transform with $W_1 \in \mathbb{R}^{m\times d}$ and $b_1 \in \mathbb{R}^m$ to get $W_1x+b_1 \in \mathbb{R}^m$
3. Apply (element-wise) the nonlinearity σ to get $\sigma(W_1x+b_1) \in \mathbb{R}^m$

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x + b_1) + b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,
2. Linearly transform with $W_1 \in \mathbb{R}^{m \times d}$ and $b_1 \in \mathbb{R}^m$ to get $W_1x + b_1 \in \mathbb{R}^m$
3. Apply (element-wise) the nonlinearity σ to get $\sigma(W_1x + b_1) \in \mathbb{R}^m$
4. Linearly transform with $W_2 \in \mathbb{R}^{m \times 1}$ and $b_2 \in \mathbb{R}$ to get $W_2\sigma(W_1x + b_1) + b_2 \in \mathbb{R}$

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x + b_1) + b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,
2. Linearly transform with $W_1 \in \mathbb{R}^{m\times d}$ and $b_1 \in \mathbb{R}^m$ to get $W_1x + b_1 \in \mathbb{R}^m$
3. Apply (element-wise) the nonlinearity σ to get $\sigma(W_1x + b_1) \in \mathbb{R}^m$
4. Linearly transform with $W_2 \in \mathbb{R}^{m\times 1}$ and $b_2 \in \mathbb{R}$ to get $W_2\sigma(W_1x + b_1) + b_2 \in \mathbb{R}$

With p layers: $f(x) = W_p\sigma(W_{p-1}\sigma(\cdots\ \sigma(W_1x + b_1)\ \cdots) + b_{p-1}) + b_p$

Neural network model

Building blocks:

1. Linear transformation (multiplication by matrix W, then addition by vector b)
2. Nonlinear transformation σ, e.g., ReLU $\sigma(a) = \max(a, 0)$, applied element-wise

Simplest nontrivial NN is $f(x) = W_2\sigma(W_1x + b_1) + b_2$. Can think of as:

1. Start with input $x \in \mathbb{R}^d$,
2. Linearly transform with $W_1 \in \mathbb{R}^{m \times d}$ and $b_1 \in \mathbb{R}^m$ to get $W_1x + b_1 \in \mathbb{R}^m$
3. Apply (element-wise) the nonlinearity σ to get $\sigma(W_1x + b_1) \in \mathbb{R}^m$
4. Linearly transform with $W_2 \in \mathbb{R}^{m \times 1}$ and $b_2 \in \mathbb{R}$ to get $W_2\sigma(W_1x + b_1) + b_2 \in \mathbb{R}$

With p layers: $f(x) = W_p\sigma(W_{p-1}\sigma(\cdots\ \sigma(W_1x + b_1)\ \cdots) + b_{p-1}) + b_p$

Parameter vector θ concatenates all W's and b's; $\dim(\theta)$ scales as width $\times$ depth

Optimizing the neural network

Optimizing the neural network

Computing gradients, even stochastic gradients $\nabla_\theta L_i(\theta)$, is daunting

Optimizing the neural network

Computing gradients, even stochastic gradients $\nabla_\theta L_i(\theta)$, is daunting

A trick called backpropagation allows such gradients to be computed efficiently

Optimizing the neural network

Computing gradients, even stochastic gradients $\nabla_\theta L_i(\theta)$, is daunting

A trick called backpropagation allows such gradients to be computed efficiently

Too notationally cumbersome to cover here, but basically the hierarchical structure of neural networks plays very nicely with the chain rule for derivatives (see Wikipedia or many other sources on internet for more)

Optimizing the neural network

Computing gradients, even stochastic gradients $\nabla_\theta L_i(\theta)$, is daunting

A trick called backpropagation allows such gradients to be computed efficiently

Too notationally cumbersome to cover here, but basically the hierarchical structure of neural networks plays very nicely with the chain rule for derivatives (see Wikipedia or many other sources on internet for more)

Unfortunately, $L(\theta)$ is non-convex, i.e., it will in general have many local optima

Optimizing the neural network

Computing gradients, even stochastic gradients $\nabla_\theta L_i(\theta)$, is daunting

A trick called backpropagation allows such gradients to be computed efficiently

Too notationally cumbersome to cover here, but basically the hierarchical structure of neural networks plays very nicely with the chain rule for derivatives (see Wikipedia or many other sources on internet for more)

Unfortunately, $L(\theta)$ is non-convex, i.e., it will in general have many local optima

We hope that SGD finds a good one… in practice there are optimization tricks that are like SGD but perform better, e.g., one very popular one is called Adam

Notes/Paradoxes on Neural Networks

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
    a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
    b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)
   a) This is true, though larger / more complex NNs also need more data to train

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)
   a) This is true, though larger / more complex NNs also need more data to train
   b) The number of NN parameters is not a good measure of its "complexity"

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)
   a) This is true, though larger / more complex NNs also need more data to train
   b) The number of NN parameters is not a good measure of its “complexity”
3. Are highly non-convex, breaking criterion 3 (optimization)

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)
   a) This is true, though larger / more complex NNs also need more data to train
   b) The number of NN parameters is not a good measure of its “complexity”
3. Are highly non-convex, breaking criterion 3 (optimization)
   a) The optimizers used for NNs don’t find arbitrary solutions, they actually find “low-complexity” solutions!

Notes/Paradoxes on Neural Networks

1. Work well for all problems, breaking criterion 1 (approximation)
   a) Actually, NNs need a lot of data, and are often worse than classical methods on smaller data sets
   b) Many of the most famous / impressive NNs, such as CNNs for vision or AlphaFold for protein structure, heavily incorporate problem-specific structure into their models
2. Work better when larger / more complex, breaking criterion 2 (complexity)
   a) This is true, though larger / more complex NNs also need more data to train
   b) The number of NN parameters is not a good measure of its "complexity"
3. Are highly non-convex, breaking criterion 3 (optimization)
   a) The optimizers used for NNs don't find arbitrary solutions, they actually find "low-complexity" solutions!

Practical Neural Networks are very far from "just" ERM

Today

- ✓ Feedback from last lecture
- ✓ Recap
- ✓ Neural networks
- Fitted value iteration
- Fitted policy iteration

Recall: Value Iteration Algorithm (infinite horizon, discounted)

Recall: Value Iteration Algorithm (infinite horizon, discounted)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V^\star(s) = \max_a \left\{ r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^\star(s') \right] \right\}, \quad \forall s$$

Recall: Value Iteration Algorithm (infinite horizon, discounted)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V^\star(s) = \max_a \left\{ r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V^\star(s') \right] \right\}, \quad \forall s$$

And the VI algorithm is a fixed-point algorithm to find $V^\star$:

1. Initialization: $V^0(s) = 0, \ \forall s$
2. For $t = 0, \ldots T-1$

   $$V^{t+1}(s) = \max_a \left\{ r(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^t(s') \right\}, \ \forall s$$
3. Return: $V^T(s)$

   $$\pi(s) = \arg\max_a \left\{ r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} V^T(s') \right\}$$

Recall: Dynamic Programming for $V^\star$ (finite horizon)

Recall: Dynamic Programming for $V^\star$ (finite horizon)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V_h^\star(s) = \max_a \left\{ r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right\}, \quad \forall s, h$$

Recall: Dynamic Programming for $V^\star$ (finite horizon)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V_h^\star(s) = \max_a \left\{ r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right\}, \quad \forall s, h$$

- Initialize: $V_H^\pi(s) = 0 \ \forall s \in S$

  For t= $H - 1, \ldots 0$, set:

  - $V_h^\star(s) = \max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$
  - $\pi_h^\star(s) = \arg\max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$

Recall: Dynamic Programming for $V^\star$ (finite horizon)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V_h^\star(s) = \max_a \left\{ r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right\}, \quad \forall s, h$$

- Initialize: $V_H^\pi(s) = 0 \ \forall s \in S$

  For t= $H-1, \ldots 0$, set:
  - $V_h^\star(s) = \max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$
  - $\pi_h^\star(s) = \arg\max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$

Notation: Now relabel $V_h(s) =: V(s,h)$ (just move subscript to an explicit argument)

Recall: Dynamic Programming for $V^\star$ (finite horizon)

Recall that Bellman equations state that the optimal value function $V^\star(s)$ satisfies:

$$V_h^\star(s) = \max_a \left\{ r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right\}, \quad \forall s, h$$

- Initialize: $V_H^\pi(s) = 0 \ \forall s \in S$

  For t= $H - 1, \ldots 0$, set:
  - $V_h^\star(s) = \max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$
  - $\pi_h^\star(s) = \arg\max_a \left[r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[V_{h+1}^\star(s') \right] \right], \forall s \in S$

Notation: Now relabel $V_h(s) =: V(s,h)$ (just move subscript to an explicit argument)

The above DP algorithm can just be seen as solving SH (Bellman) equations for the SH different values of $V(s,h)$, but doing so in an exact, efficient way via DP

Q-Value Dynamic Programming Algorithm:

Q-Value Dynamic Programming Algorithm:

Recall from HW1, Problem 2, the Bellman equations for $Q^\star$:

$$Q_h^\star(s,a) = r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\max_{a'} Q_{h+1}^\star(s',a') \right]$$

Q-Value Dynamic Programming Algorithm:

Recall from HW1, Problem 2, the Bellman equations for $Q^\star$:

$$Q_h^\star(s,a) = r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[\max_{a'} Q_{h+1}^\star(s',a')\right]$$

Analogous Q-value DP, with same notational change as previous slide: h as argument

1. Initialization: $Q(s,a,H) = 0 \quad \forall s, a$
2. Solve (via dynamic programming):

$$Q(s,a,h) = r(s,a) + \mathbb{E}_{s' \sim P(s,a)}\left[\max_{a' \in A} Q(s',a',h+1)\right] \quad \forall s, a, h$$

3. Return:

$$\pi_h(s) = \arg\max_a \left\{Q(s,a,h)\right\}$$

What if we can’t just evaluate the expectations?

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

This is now full RL!!

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

Suppose:

This is now full RL!!

- We have N trajectories $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

  Each trajectory is of the form $\tau_i = \{s_0^i, a_0^i, \ldots s_{H-1}^i, a_{H-1}^i, s_H^i\}$
- π_{data} is often referred to as our data collection policy.

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

Suppose:

This is now full RL!!

- We have N trajectories $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

  Each trajectory is of the form $\tau_i = \{s_0^i, a_0^i, \ldots s_{H-1}^i, a_{H-1}^i, s_H^i\}$

- π_{data} is often referred to as our data collection policy.

Want: $Q(s, a, h) \approx r(s, a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

Suppose:

This is now full RL!!

- We have N trajectories $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

  Each trajectory is of the form $\tau_i = \{s_0^i, a_0^i, \ldots s_{H-1}^i, a_{H-1}^i, s_H^i\}$
- π_{data} is often referred to as our data collection policy.

Want: $Q(s, a, h) \approx r(s, a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$

Since we're trying to approximate conditional expectations, seems like it kind of fits into supervised learning—can we use an approach like that?

What if we can't just evaluate the expectations?

If S and/or A are very large, computing expectations could be very expensive

We may not have a way to directly compute those expectations, but instead only have access to a simulator (or the real world), where we can collect data

Suppose:

This is now full RL!!

- We have N trajectories $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

  Each trajectory is of the form $\tau_i = \{s_0^i, a_0^i, \ldots s_{H-1}^i, a_{H-1}^i, s_H^i\}$
- π_{data} is often referred to as our data collection policy.

Want: $Q(s, a, h) \approx r(s, a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$

Since we're trying to approximate conditional expectations, seems like it kind of fits into supervised learning—can we use an approach like that? Yes!

Connection to Supervised Learning

$$Q(s,a,h) \approx r(s,a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$$

Then we'd be happy if we found a

$$Q(s_h, a_h, h) = f(x) = \mathbb{E}[y \mid x] = \mathbb{E}\left[r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1) \,\middle|\, s_h, a_h, h \right]$$

Connection to Supervised Learning

$$Q(s,a,h) \approx r(s,a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$$

What are the y and x?

Then we'd be happy if we found a

$$Q(s_h, a_h, h) = f(x) = \mathbb{E}[y \mid x] = \mathbb{E}\left[r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1) \middle| s_h, a_h, h \right]$$

Connection to Supervised Learning

$$Q(s,a,h) \approx r(s,a) + \mathbb{E}_{s' \sim P(s,a)}\left[\max_{a' \in A} Q(s',a',h+1)\right] \quad \forall s,a,h$$

What are the y and x?

Note that the RHS can also be written as

$$\mathbb{E}\left[r(s_h,a_h) + \max_{a'} Q(s_{h+1},a',h+1) \middle| s_h, a_h, h\right]$$

Then we'd be happy if we found a

$$Q(s_h,a_h,h) = f(x) = \mathbb{E}[y\,|\,x] = \mathbb{E}\left[r(s_h,a_h) + \max_{a'} Q(s_{h+1},a',h+1) \middle| s_h, a_h, h\right]$$

Connection to Supervised Learning

$$Q(s,a,h) \approx r(s,a) + \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a' \in A} Q(s', a', h+1) \right] \quad \forall s, a, h$$

What are the y and x?

Note that the RHS can also be written as

$$\mathbb{E}\left[r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1) \middle| s_h, a_h, h \right]$$

This suggests that $y = r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1)$ and $x = (s_h, a_h, h)$

Then we'd be happy if we found a

$$Q(s_h, a_h, h) = f(x) = \mathbb{E}[y \mid x] = \mathbb{E}\left[r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1) \middle| s_h, a_h, h \right]$$

Connection to Supervised Learning (cont’d)

Connection to Supervised Learning (cont'd)

We can convert our data $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$, into (y, x) pairs; how many?

Connection to Supervised Learning (cont'd)

We can convert our data $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$, into (y, x) pairs; how many? NH

Connection to Supervised Learning (cont'd)

We can convert our data $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$, into (y, x) pairs; how many? NH

BUT, to compute each y, we need to already know Q!

Connection to Supervised Learning (cont'd)

We can convert our data $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$, into (y, x) pairs; how many? NH

BUT, to compute each y, we need to already know Q!

Setting that aside for the moment, to fit supervised learning, we'd minimize a least-squares objective function: $\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{NH} (y_i - f(x_i))^2$

Connection to Supervised Learning (cont'd)

We can convert our data $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$, into (y, x) pairs; how many? NH

BUT, to compute each y, we need to already know Q!

Setting that aside for the moment, to fit supervised learning, we'd minimize a least-squares objective function: $\hat{f}(x) = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{NH} (y_i - f(x_i))^2$

Then if we have enough data, choose a good $\mathcal{F}$, and optimize well,

$$Q(s_h, a_h, h) := \hat{f}(x) \approx \mathbb{E}[y \mid x] = \mathbb{E}\left[r(s_h, a_h) + \max_{a'} Q(s_{h+1}, a', h+1) \middle| s_h, a_h, h \right]$$

Fitted (Q-)Value Iteration

To address the circularity problem of not knowing Q for computing the y, we have an algorithmic tool… what is it?

Fitted (Q-)Value Iteration

To address the circularity problem of not knowing Q for computing the y, we have an algorithmic tool… what is it?

Hint: we used it for another VI algorithm before…

Fitted (Q-)Value Iteration

To address the circularity problem of not knowing Q for computing the y, we have an algorithmic tool… what is it?

Hint: we used it for another VI algorithm before…

Fixed point iteration! Initialize, then at each step, pretend Q is known by plugging in the previous time step's Q to compute the y's, and then use that to get next Q

Fitted (Q-)Value Iteration

To address the circularity problem of not knowing Q for computing the y, we have an algorithmic tool… what is it?
Hint: we used it for another VI algorithm before…

Fixed point iteration! Initialize, then at each step, pretend Q is known by plugging in the previous time step's Q to compute the y's, and then use that to get next Q

Input: **offline dataset** $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

1. Initialize fitted Q function at f_0
2. For $k = \not{0},1,\ldots,K$:

$$f_k = \arg\min_{f \in \mathscr{F}} \sum_{i=1}^{N} \sum_{h=1}^{H-1} \left(f(s_h^i, a_h^i, h) - \left(r(s_h^i, a_h^i) + \max_a f_{k-1}(s_{h+1}^i, a, h+1) \right) \right)^2$$

3. With f_K as an estimate of $Q^\star$, return $\pi_h(s) = \arg\max_a \left\{ f^K(s, a, h) \right\}$

Fitted (Q-)Value Iteration

To address the circularity problem of not knowing Q for computing the y, we have an algorithmic tool… what is it?
Hint: we used it for another VI algorithm before…

Fixed point iteration! Initialize, then at each step, pretend Q is known by plugging in the previous time step's Q to compute the y's, and then use that to get next Q

Input: **offline dataset** $\tau_1, \ldots \tau_N \sim \rho_{\pi_{data}}$

1. Initialize fitted Q function at f_0
2. For $k = 0,1,\ldots,K$:

$$f_k = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \sum_{h=1}^{H-1} \left(f(s_h^i, a_h^i, h) - \left(r(s_h^i, a_h^i) + \max_a f_{k-1}(s_{h+1}^i, a, h+1) \right) \right)^2$$

3. With f_K as an estimate of $Q^\star$, return $\pi_h(s) = \arg\max_a \left\{ f^K(s, a, h) \right\}$

Q-Learning is an online version, i.e., draw new trajectories at each k based on f_k as Q-function

Today

- ✓ Feedback from last lecture
- ✓ Recap
- ✓ Neural networks
- ✓ Fitted value iteration
- Fitted policy iteration

Recall: Policy Iteration (PI)

- Initialization: choose a policy $\pi^0 : S \mapsto A$
- For $k = 0,1,\ldots$
  1. **Policy Evaluation**: Solve (via dynamic programming):
     $$Q^{\pi^k}(s,a,h) = r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[Q^{\pi^k}(s', \pi^k(a), h+1)\right] \quad \forall s,a,h$$
  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a Q^{\pi^k}(s,a,h)$

Recall: Policy Iteration (PI)

- Initialization: choose a policy $\pi^0 : S \mapsto A$
- For $k = 0,1,\ldots$
  1. **Policy Evaluation**: Solve (via dynamic programming):
     $$Q^{\pi^k}(s, a, h) = r(s, a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[Q^{\pi^k}(s', \pi^k(a), h+1) \right] \quad \forall s, a, h$$
  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a Q^{\pi^k}(s, a, h)$

Again: what if we're in full RL setting where we can't just evaluate expectations?

Recall: Policy Iteration (PI)

- Initialization: choose a policy $\pi^0 : S \mapsto A$
- For $k = 0,1,\ldots$
  1. **Policy Evaluation**: Solve (via dynamic programming):

     $$Q^{\pi^k}(s,a,h) = r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[Q^{\pi^k}(s', \pi^k(a), h+1)\right] \quad \forall s,a,h$$

  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a Q^{\pi^k}(s,a,h)$

Again: what if we're in full RL setting where we can't just evaluate expectations?

This breaks the Policy Evaluation step, so can we do a fitted version?

Recall: Policy Iteration (PI)

- Initialization: choose a policy $\pi^0 : S \mapsto A$
- For $k = 0,1,\ldots$
  1. **Policy Evaluation**: Solve (via dynamic programming):
     $$Q^{\pi^k}(s,a,h) = r(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)}\left[Q^{\pi^k}(s', \pi^k(a), h+1)\right] \quad \forall s,a,h$$
  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a Q^{\pi^k}(s,a,h)$

Again: what if we're in full RL setting where we can't just evaluate expectations?

This breaks the Policy Evaluation step, so can we do a fitted version?

Yes! RHS can be written as $\mathbb{E}\left[r(s_h,a_h) + Q^{\pi^k}(s_{h+1}, \pi^k(a_h), h+1) \middle| s_h, a_h, h\right]$

Recall: Policy Iteration (PI)

- Initialization: choose a policy $\pi^0 : S \mapsto A$
- For $k = 0,1,\ldots$
  1. **Policy Evaluation**: Solve (via dynamic programming):

     $$Q^{\pi^k}(s,a,h) = r(s,a) + \mathbb{E}_{s'\sim P(\cdot|s,a)}\left[Q^{\pi^k}(s', \pi^k(a), h+1)\right] \quad \forall s,a,h$$

  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a Q^{\pi^k}(s,a,h)$

Again: what if we're in full RL setting where we can't just evaluate expectations?

This breaks the Policy Evaluation step, so can we do a fitted version?

Yes! RHS can be written as $\mathbb{E}\left[r(s_h,a_h) + Q^{\pi^k}(s_{h+1}, \pi^k(a_h), h+1) \,\middle|\, s_h, a_h, h\right]$

Spot the difference!

Fitted Policy Evaluation

Use exact same strategy as before: **fixed point iteration**

Fitted Policy Evaluation

Use exact same strategy as before: **fixed point iteration**

Input: policy π, **dataset** $\tau_1, \ldots \tau_N \sim \rho_\pi$

1. Initialize fitted Q^π function at f_0
2. For $k = 0,1,\ldots,K$:

$$f_k = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \sum_{h=1}^{H-1} \left(f(s_h^i, a_h^i, h) - \left(r(s_h^i, a_h^i) + f_{k-1}(s_{h+1}^i, \pi(s_h^i), h+1) \right) \right)^2$$

3. Return the function f_K as an estimate of Q^π

Fitted Policy Iteration:

- Initialization: choose a policy $\pi^0 : S \mapsto A$ and a sample size N
- For $k = 0,1,\ldots$
  1. **Fitted Policy Evaluation**: Using N sampled trajectories $\tau_1, \ldots \tau_N \sim \rho_{\pi^k}$, obtain approximation $\hat{Q}^{\pi^k} \approx Q^{\pi^k}$
  2. **Policy Improvement**: set $\pi_h^{k+1}(s) := \arg\max_a \hat{Q}^{\pi^k}(s, a, h)$

(Another) Fitted Policy Evaluation option

(Another) Fitted Policy Evaluation option

Using the definition of the Q function, can do a non-iterative fitted policy evaluation

$$Q^\pi(s, a, h) = \mathbb{E}\left[\sum_{t=h}^{H-1} r(s_t, a_t) \,\middle|\, (s_h, a_h) = (s, a)\right]$$

(Another) Fitted Policy Evaluation option

Using the definition of the Q function, can do a non-iterative fitted policy evaluation

$$Q^\pi(s, a, h) = \mathbb{E}\left[\sum_{t=h}^{H-1} r(s_t, a_t) \,\middle|\, (s_h, a_h) = (s, a), h\right]$$

Input: policy π, **dataset** $\tau_1, \ldots \tau_N \sim \rho_\pi$

Return:

$$\hat{Q}^\pi = \arg\min_{f \in \mathcal{F}} \sum_{i=1}^{N} \sum_{h=1}^{H-1} \left(f(s_h^i, a_h^i, h) - \sum_{t=h}^{H-1} r(s_t^i, a_t^i)\right)^2$$

Today

- Feedback from last lecture
- Recap
- Neural networks
- Fitted value iteration
- Fitted policy iteration

Summary:

- Neural Networks work well for complex function approximation with big data
- Incorporating supervised learning into PI and VI makes them RL algorithms!

Attendance:

bit.ly/3RcTC9T

Feedback:

bit.ly/3RHtlxy