# Supervised Learning (in 1 Lecture)

### Lucas Janson and Sham Kakade CS/Stat 184: Introduction to Reinforcement Learning Fall 2023

- Feedback from last lecture
- Recap
- Supervised learning setup
- Linear regression
- Neural networks



### Feedback from feedback forms

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





- Recap
- Supervised learning setup
- Linear regression
- Neural networks

- Thompson sampling is a good heuristic for bandits
- bandits)

### Recap

• Couple more slides on it, then we move on (rest of today unrelated to

What could go wrong for smaller T? Suppose K = 2 and T = 3, and:

- t = 0:  $a_0 = 1$ ,  $r_0 = 1$
- $t = 1: a_1 = 2, r_1 = 0$
- t = 2 (last time step, with  $\hat{\mu}_2^{(1)} = 1$  and

one sample from each arm, Thompson sampling isn't sure which arm is best.

no reason to explore rather than exploit.

Thompson sampling doesn't know this, and neither does UCB (although UCB) wouldn't happen to make the same mistake in this case).

Thompson sampling in practice (cont'd) So Thompson sampling is basically exactly optimal for large T

nd 
$$\hat{\mu}_2^{(2)} = 0$$
):  $a_2 = ?$ 

Thompson sampling has a decent probability of choosing  $a_2 = 2$ , since with just

But  $a_2 = 1$  is clear right choice here: there is no future value to learning more, i.e.,



# Thompson sampling in practice (cont'd)

Fix: add a tuning parameter to make it more greedy. Some possibilities:

- Update the Beta parameters by  $1 + \epsilon$  instead of just 1 each time
- Instead of just taking one sample of  $\mu$  and computing the greedy action with respect to it, take *n* samples, compute the greedy action with respect to each, and pick the *mode* of those greedy actions

that have worked well so far), as opposed to arms that may be good

- For small T, Thompson sampling is not greedy enough

- All of these favor arms that the algorithm has more confidence are good (i.e., arms
- Such tuning can improve Thompson sampling's performance even for reasonably large T (the asymptotic optimality of vanilla TS is very asymptotic)



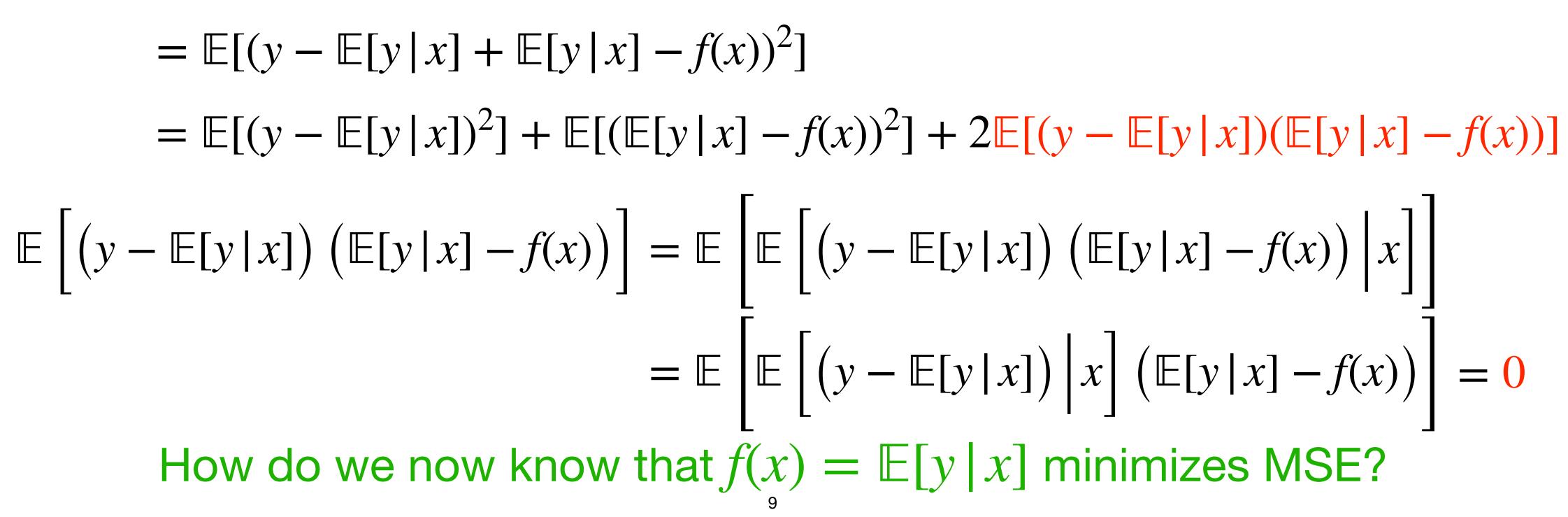
### Feedback from last lecture

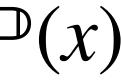
- Recap
  - Supervised learning setup
  - Linear regression
  - Neural networks



- Data: i.i.d. pairs  $(y_1, x_1), \dots, (y_n, x_n)$  drawn from distribution  $\mathbb{P}(y, x) = \mathbb{P}(y | x)\mathbb{P}(x)$ 
  - Goal: learn a good predictor f(x) of y
  - Note:  $\mathbb{E}[y | x]$  minimizes mean squared error
  - $\mathsf{MSE}(f) = \mathbb{E}[(y f(x))^2]$ 
    - $= \mathbb{E}[(y \mathbb{E}[y | x] + \mathbb{E}[y | x] f(x))^2]$

# Supervised learning setup





# Empirical risk minimization (ERM)

- This fact both motivates  $\mathbb{E}[y | x]$  as a target for learning, and suggests how to do it
  - Law of large numbers:  $\mathbb{E}[(y f(x))^2]$

Empirical risk minimization (ERN

- E.g.,  $f(x) = \sum y_i 1_{\{x=x_i\}}$  achieves zero training error (as long as no ties in the  $x_i$ 's) i=1

Fact:  $\mathbb{E}[y | x] = \arg\min_{f} \mathbb{E}[(y - f(x))^2]$ 

$$\approx \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2 =: \text{training error c}$$

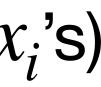
M): 
$$\hat{f}(x) = \arg\min_{f} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Seems great, but if we allow f in the argmin to range over all functions, we can get ridiculous solutions. Can anyone think of one?

But it predicts 0 at every x value not in the training data, regardless of the data!









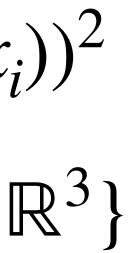
### Function classes

Need to constrain ERM to a function

- E.g. (if x scalar) quadratic functions:  $\mathcal{F}$
- How to choose  $\mathcal{F}$ ? Three main high-level criteria:
- 1. Approximation:  $\mathbb{E}[y|x] \approx \arg\min_{x \in \mathbb{Z}} \mathbb{E}[(y f(x))^2]$
- 2. Complexity:  $\mathcal{F}$  doesn't contain "too many" functions/dimensions 3. Optimizable: need to be able to compute the argmin (or something like it)

class 
$$\mathscr{F}: \widehat{f}(x) = \arg\min_{f \in \mathscr{F}} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))$$
  
 $\mathscr{F} = \{f(x) = ax^2 + bx + c : (a, b, c) \in A\}$ 

<u>Statistical learning theory</u>: the ERM optimum (criterion 3)  $\hat{f}$  will perform well if  $\mathcal{F}$ 's approximation error (criterion 1) and complexity (criterion 2) are low





# Optimization

Parameterized ERM optimization: 
$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2;$$
  $\hat{f} = \frac{1}{n} \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2;$   $L(\theta) = \frac{1}{n} \sum_{i=1}^n L_i(\theta),$  gradient operator

- Downside: computing  $\nabla_{\theta} L(\theta^{(i)})$  at each step expensive for big data

Typically our function class  $\mathscr{F}$  is parameterized by a parameter vector  $\theta \in \mathbb{R}^d$ , i.e., every  $f \in \mathcal{F}$  can be written as  $f_{\theta}(x)$  for some  $\theta \in \mathbb{R}^d$ 

Gradient descent: initialize at  $\theta_0$ , update via  $\theta^{(i+1)} = \theta^{(i)} - \eta \nabla_{A} L(\theta^{(i)})$ 

Stochastic gradient descent: initialize at  $\theta_0$ , update via  $\theta^{(i+1)} = \theta^{(i)} - \eta \nabla_{\theta} L_i(\theta^{(i)})$ Can do multiple passes of data, or uses batch size b > 1 at each step <u>Main takeaway</u>: this works (for good choices of b and  $\eta$ , which may vary with i)





- Recap
- Supervised learning setup
  - Linear regression
  - Neural networks



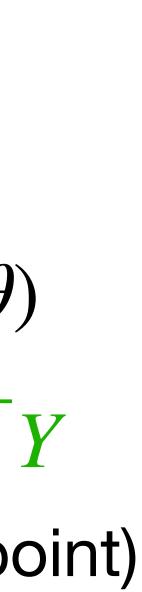
### Linear model

- Linear model (if  $d = \dim$ 
  - ERM optimization:  $\hat{\theta}$  =
- Let  $Y := (y_1, \dots, y_n) \in \mathbb{R}^n$  and X := $\hat{\theta} = \arg \min$  $\theta \in |$ Let  $L(\theta) = \frac{1}{2} \|Y - X\theta\|^2$ :  $\nabla_{\theta} L(\theta) = X^{\top} (Y - X\theta), \quad \nabla_{\theta} L_i(\theta) = x_i (y_i - x_i^{\top} \theta)$ If  $n < d, X^{\top}X$  non-invertible; many solutions exists (think: fitting line through 1 point)

$$\begin{split} \mathsf{n}(x), & \text{let } \theta \in \mathbb{R}^d \text{):} f_{\theta}(x) = x^{\mathsf{T}} \theta \\ = & \arg\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (y_i - x_i^{\mathsf{T}} \theta)^2 \\ & (x_1^{\mathsf{T}}; \dots; x_n^{\mathsf{T}}) \in \mathbb{R}^{n \times d}, \text{ can rewrite ERM as:} \\ & \inf_{\mathbb{R}^d} \frac{1}{2} \|Y - X\theta\|^2 \end{split}$$

Instead of (S)GD,  $\nabla_{\theta} L(\theta) = 0$  leads to closed-form solution  $\hat{\theta} = (X^{\top} X)^{-1} X^{\top} Y$ 

Surprising fact: GD initialized at 0 finds solution with smallest norm!



- 1. Can work surprisingly well in practice, especially in high dimensions
- 2. Need good features
- - j = 1b) Lasso penalty: add  $\lambda \sum_{j=1}^{n} |\theta_j|$  to training loss to encourage sparse  $\hat{\theta}$ *j*=1

### Notes on linear models

a) Linear functions approximate smooth functions pretty well, if very smooth

a) Can use domain knowledge to construct transformation  $\phi(x)$  which can be higher- or lower-dimensional than x, and then just use linear model in  $\phi(x)$ 3. Adding penalty to ERM objective can help a lot, especially in high dimensions

a) <u>Ridge</u> penalty: add  $\lambda \sum \theta_i^2$  to training loss to discourage huge  $\hat{\theta}$  entries





- Recap
- Supervised learning setup
- Linear regression
  - Neural networks



# Neural network model

### Building blocks:

- 1. Linear transformation (multiplication by matrix W, then addition by vector b) 2. Nonlinear transformation  $\sigma$ , 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  $x \in \mathbb{R}^d$ ,

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

With p layers:  $f(x) = W_p \sigma(W_{p-1} \sigma(\cdots \sigma(W_1 x + b_1) \cdots) + b_{p-1}) + b_p$ 

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







# 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 (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 optimal
- 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 on NNs

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!









- Recap
- Supervised learning setup
- Linear regression
- Neural networks



### Summary:

- Given data comprised of a bunch of (y, x) pairs, there exists a huge toolbox (a whole field's worth) to approximate the function  $\mathbb{E}[y | x]$
- Generally, we write down a squared-error loss function for a parameterized function class and optimize it via (possibly stochastic) gradient descent Attendance:

bit.ly/3RcTC9T



### Feedback:

### bit.ly/3RHtlxy



