CS-GY 6923: Lecture 8
Learning Rates, Stochastic Gradient Descent, Learning Theory, PAC learning

NYU Tandon School of Engineering, Prof. Christopher Musco

FIRST ORDER OPTIMIZATION

First order oracle model: Given a function L to minimize (in our case a loss function), assume we can:

- **Function oracle**: Evaluate $L(\beta)$ for any β.
- **Gradient oracle**: Evaluate $\nabla L(\beta)$ for any β.

Basic Gradient descent algorithm:

- Choose starting point $\beta^{(0)}$.
- For $i = 0, \ldots, T$:
  - $\beta^{(i+1)} = \beta^{(i)} - \eta \nabla L(\beta^{(i)})$
- Return $\beta^{(T)}$.

$\eta > 0$ is the step-size/learning rate parameter.

First justifications for performance:

$$\lim_{\eta \to 0} L(\beta - \eta \mathbf{v}) - L(\beta) = \eta \cdot \left(\frac{\partial L}{\partial \beta_1} v_1 + \frac{\partial L}{\partial \beta_2} v_2 + \ldots + \frac{\partial L}{\partial \beta_d} v_d \right)$$

$$= \eta \cdot \langle \nabla L(\beta), \mathbf{v} \rangle.$$

So if we choose η sufficiently small, setting $\mathbf{v} = \nabla L(\beta)$ should always lead to a reduction in function value.

Second justifications for performance:

Claim (GD Convergence Bound)

If L is a convex loss function (least squares regression, logistic loss, etc.), $\|\nabla L(\beta)\|_2 \leq G$ for all β and $\|\beta^{(0)} - \beta^*\|_2 \leq B$, then after $T = \frac{R^2 G^2}{\epsilon^2}$, we will find some $\hat{\beta}$ such that $L(\hat{\beta}) \leq L(\beta^*) + \epsilon$.

Important practical question: How to set η in practice?

Our theoretical convergence result gives guidance for setting η: it holds when $\eta = \frac{R}{G\sqrt{T}}$. But...

- We don't usually know R or G in advance. We might not even know T.
- Even if we did, setting $\eta = \frac{R}{G\sqrt{T}}$ tends to be a very conservative in practice. The choice 100% leads to convergence, but usually to fairly slow convergence.
- What is L doesn't have bounded gradeints? What if L is not even convex?

(We need different approaches for choosing the step size.)

FIRST APPROACH: EXPONENTIAL GRID SEARCH

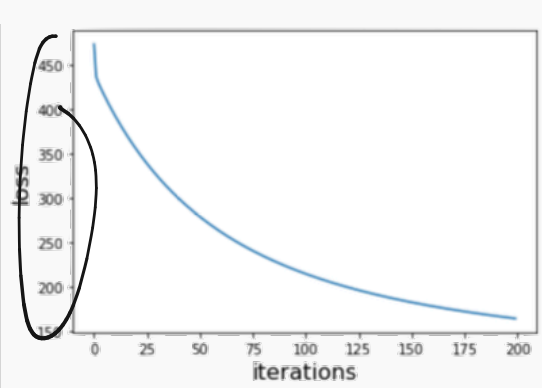
Just as in regularization, search over a grid of possible parameters:

$$\eta = [2^{-5}, 2^{-4}, 2^{-3}, \ldots, 2^{9}, 2^{10}].$$

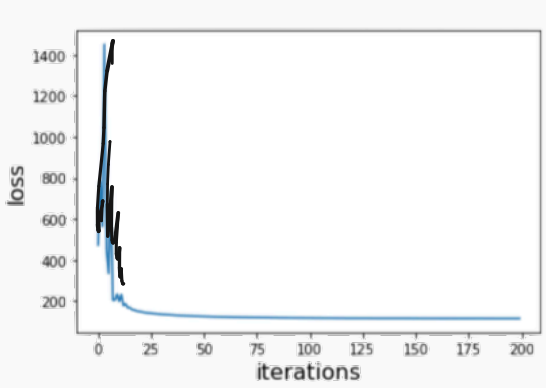
Can manually check if we are converging too slow or undershooting by plotting the optimization curve.

LEARNING RATE

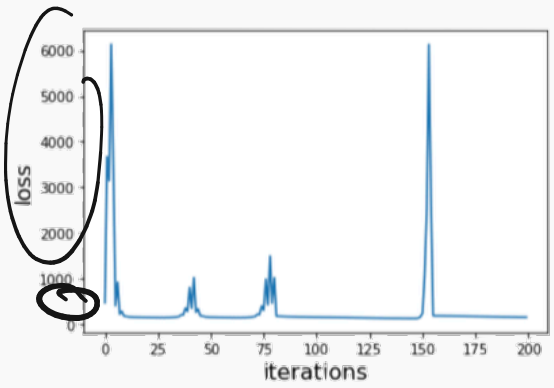
Plot's of loss vs. number of iterations for three difference choices of step size.



small α



just right α



large α

BACKTRACKING LINE SEARCH/ARMIJO RULE

Main idea: If we set $\beta^{(i+1)} \leftarrow \beta^{(i)} - \eta \nabla L(\beta^{(i)})$ then:

$$\begin{aligned} L(\beta^{(i+1)}) &\approx L(\beta^{(i)}) - \eta \left\langle \nabla L(\beta^{(i)}), \nabla L(\beta^{(i)}) \right\rangle \\ &= L(\beta^{(i)}) - \eta \|\nabla L(\beta^{(i)})\|_2^2. \end{aligned}$$

Approximation holds for small η. If it holds, maybe we could get away with a larger η. If it does not hold, we should reduce η.

Choose a tolerance parameter $c < 1$ (typically $c = 1/2$). We will be satisfied if:

$$L(\beta^{(i+1)}) \leq L(\beta^{(i)}) - c \cdot \eta \|\nabla L(\beta^{(i)})\|_2^2.$$

Backtracking line search/Armijo rule

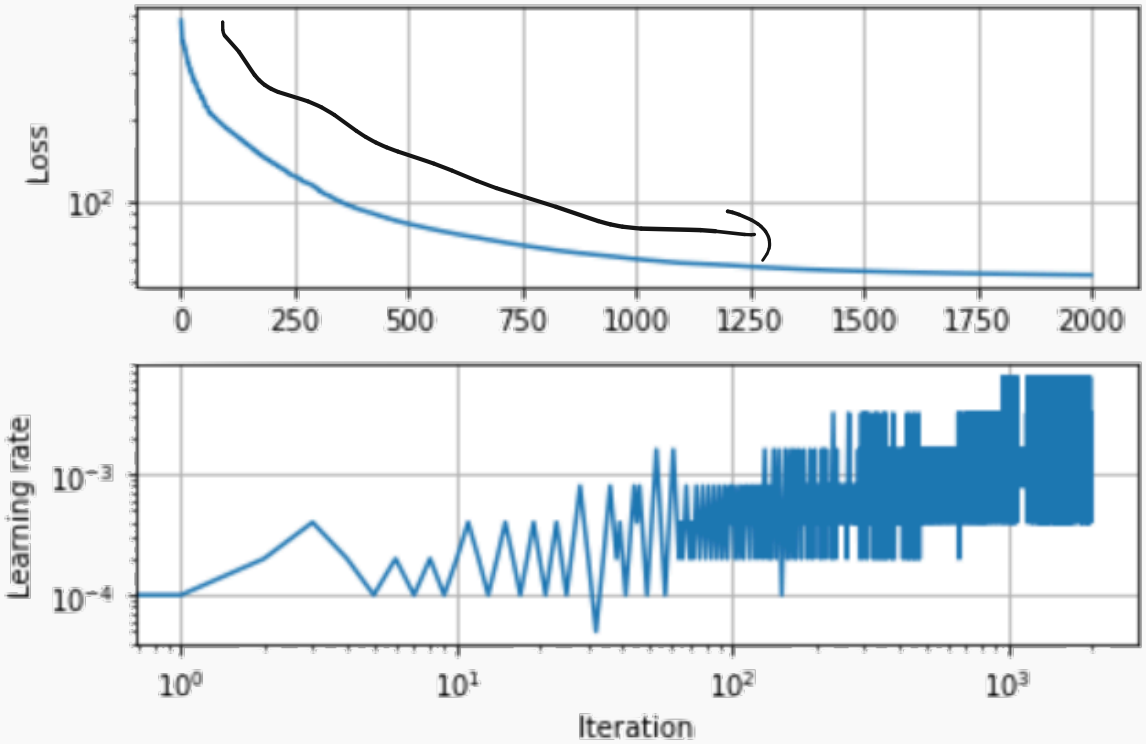
Gradient descent with backtracking line search:

- Choose starting step size η, starting point β
- Choose $c < 1$ (typically $c = 1/2$)
- For $i = 1, \ldots, T$:
  - $\beta^{(new)} = \beta - \eta \nabla L(\beta)$
  - If $L(\beta^{(new)}) \leq L(\beta) - c \cdot \eta \|\nabla L(\beta)\|_2^2$.
    - $\beta \leftarrow \beta^{(new)}$
    - $\eta \leftarrow 2\eta$
  - Else
    - $\eta \leftarrow \eta/2$

Always decreases objective value, works very well in practice.

BACKTRACKING LINE SEARCH/ARMIJO RULE

Gradient descent with backtracking line search:



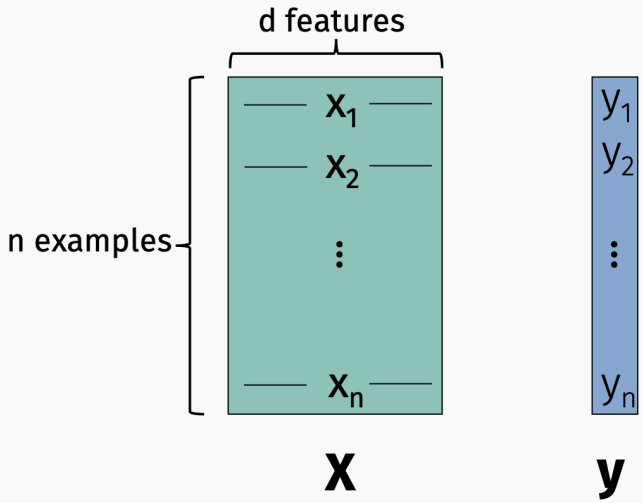
Always decreases objective value, works very well in practice. We will see this in a lab.

COMPLEXITY OF GRADIENT DESCENT

Complexity of computing the gradient will depend on you loss function.

Example 1: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

$$L(\beta) = \|\mathbf{X}\beta - \mathbf{y}\|_2^2 \qquad \nabla L(\beta) = 2\mathbf{X}^T(\mathbf{X}\beta - \mathbf{y})$$



- Runtime of closed form solution $\beta^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$:
- Runtime of one GD step:

COMPLEXITY OF GRADIENT DESCENT

$h(z) = \frac{1}{1+e^{-z}}$

Complexity of computing the gradient will depend on your loss function.

Example 1: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

$$L(\boldsymbol{\beta}) = -\sum_{i=1}^{n} y_i \log(h(\boldsymbol{\beta}^T \mathbf{x}_i)) + (1 - y_i) \log(1 - h(\boldsymbol{\beta}^T \mathbf{x}_i))$$

$$\nabla L(\boldsymbol{\beta}) = \mathbf{X}^T (h(\mathbf{X}\boldsymbol{\beta}) - \mathbf{y})$$

- No closed form solution.
- Runtime of one GD step: $O(nd)$

COMPLEXITY OF GRADIENT DESCENT

Frequently the complexity is $O(nd)$ if you have n data-points and d parameters in your model. This will also be the case for neural networks, no matter how complex.

Not bad, but the dependence on n can be a lot! n might be on the order of thousands, or millions, or trillions.

Gradient computation will be especially slow if the entire dataset does not fit in main memory.

Stochastic Gradient Descent (SGD).

$x_1, \ldots x_n$

$y_1, \ldots y_n$

- Powerful randomized variant of gradient descent used to train machine learning models when n is large and thus computing a full gradient is expensive.

Applies to any loss with finite sum structure:

$$L(\beta) = \sum_{j=1}^{n} \ell(\beta, \mathbf{x}_j, y_j)$$

$$\|X\beta - y\|_2^2 = \sum_{j=1}^{n} (y_j - x_j^T \beta)^2$$

STOCHASTIC GRADIENT DESCENT

Let $L_j(\boldsymbol{\beta})$ denote $\ell(\boldsymbol{\beta}, \mathbf{x}_j, y_j)$.

$$L(\beta) = \sum_{i=1}^{n} L_j(\beta)$$

$$\nabla L(\beta) = \sum_{i=1}^{n} \nabla L_j(\beta)$$

Claim: If $j \in 1, \ldots, n$ is chosen uniformly at random. Then:

$$\mathbb{E}\left[n \cdot \nabla L_j(\boldsymbol{\beta})\right] = \nabla L(\boldsymbol{\beta}).$$

$$= n \cdot \sum_{k=1}^{n} \frac{1}{n} \nabla L_k(\beta) = \sum_{k=1}^{n} \nabla L_k(\beta) = \nabla L(\beta)$$

$\nabla L_j(\boldsymbol{\beta})$ is called a stochastic gradient.

STOCHASTIC GRADIENT DESCENT

$\beta = [\beta_0 \dots \beta_d]$

$L_j(\beta) = \ell(x_j, y_j, \beta)$

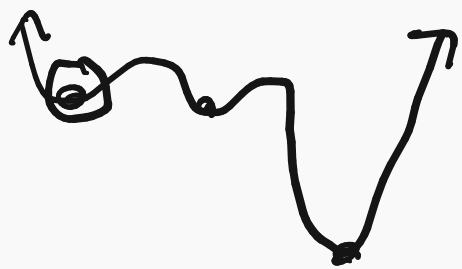
SGD iteration:

- Initialize $\beta^{(0)} = 0$
- For $i = 0, \dots, T-1$:
  - Choose j uniformly at random from $\{1, 2, \dots, n\}$.
  - Compute stochastic gradient $\mathbf{g} = \nabla L_j(\beta^{(i)})$.
  - Update $\beta^{(i+1)} = \beta^{(i)} - \eta \cdot n \cdot \mathbf{g}$

$\nabla L(\beta^{(i)})$

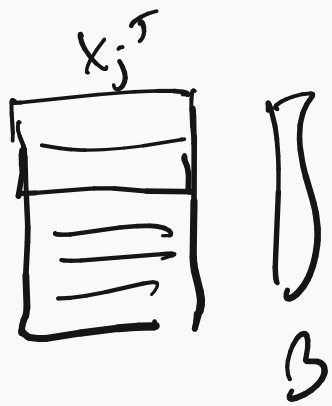
Move in direction of steepest descent in expectation.

Cost of computing g is independent of n!



Example: Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be a data matrix.

$$L(\beta) = \|\mathbf{X}\beta - \mathbf{y}\|_2^2 = \sum_{j=1}^{n} (y_j - \beta^T \mathbf{x}_j)^2$$



$\nabla L(\beta)$ cost $\Theta(nd)$

$L_j(\beta) = (y_j - \beta^T x_j)^2$

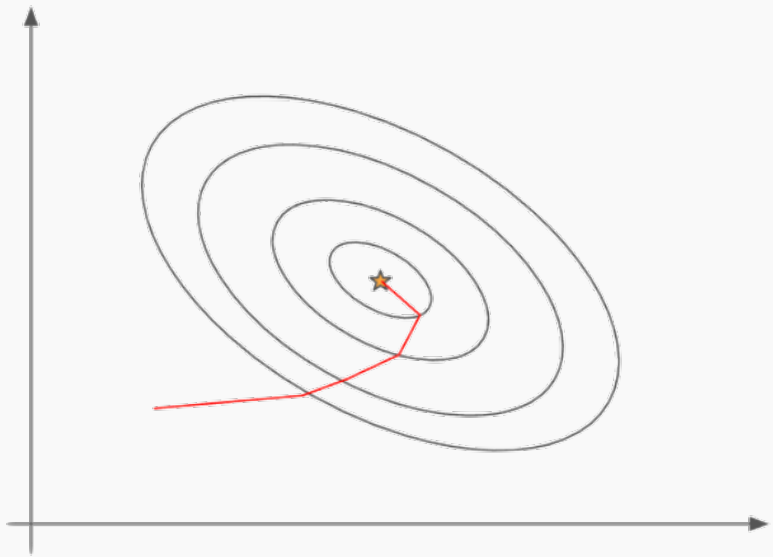
$\nabla L_j(\beta) = \underbrace{2(y_j - \beta^T x_j)}_{\text{scalar}} \cdot \overbrace{(-x_j)}^{\text{vector}}$

- Runtime of one SGD step: $\Theta(d)$

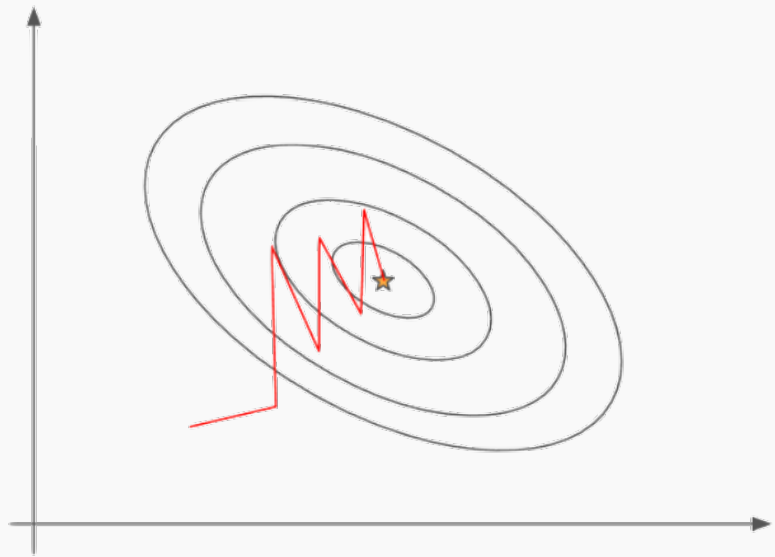
STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Gradient Descent

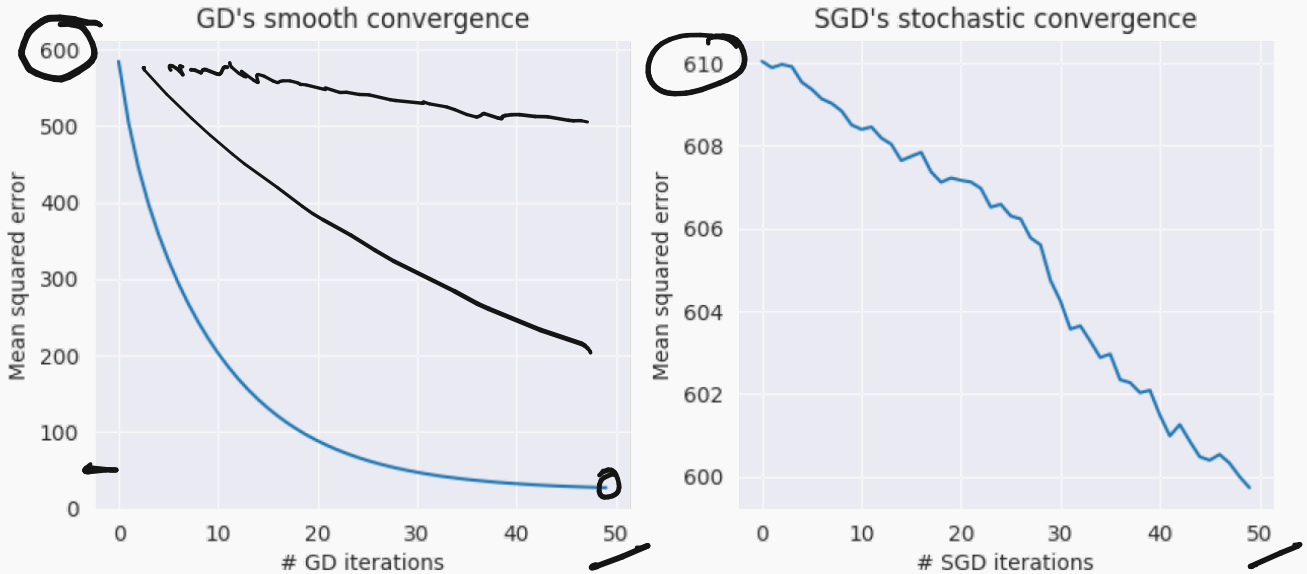


Stochastic Gradient Descent

STOCHASTIC GRADIENT DESCENT

Gradient descent: Fewer iterations to converge, higher cost per iteration.

Stochastic Gradient descent: More iterations to converge, lower cost per iteration.



Typically, SGD has the lower overall runtime. Lots of theory to explain why.

STOCHASTIC GRADIENT DESCENT IN PRACTICE

Typical implementation: **Shuffled Gradient Descent.**

Instead of choosing j independently at random for each iteration, randomly permute (shuffle) data and set $j = 1, \ldots, n$.

After every n iterations, reshuffle data and repeat, or just keep cycling through the data in the same random order.

- Relatively similar convergence behavior to standard SGD.
- **Important term:** one epoch denotes one pass over all training examples: $j = 1, \ldots, j = n$.
- Convergence rates for training ML models are often discussed in terms of epochs instead of iterations.

For gradient descent, 1 epoch = 1 iteration/parameter update, for SGD, 1 epoch = n iterations/parameter updates.

Practical Modification 1: **Mini-batch Gradient Descent.**

Observe that for any batch size s,

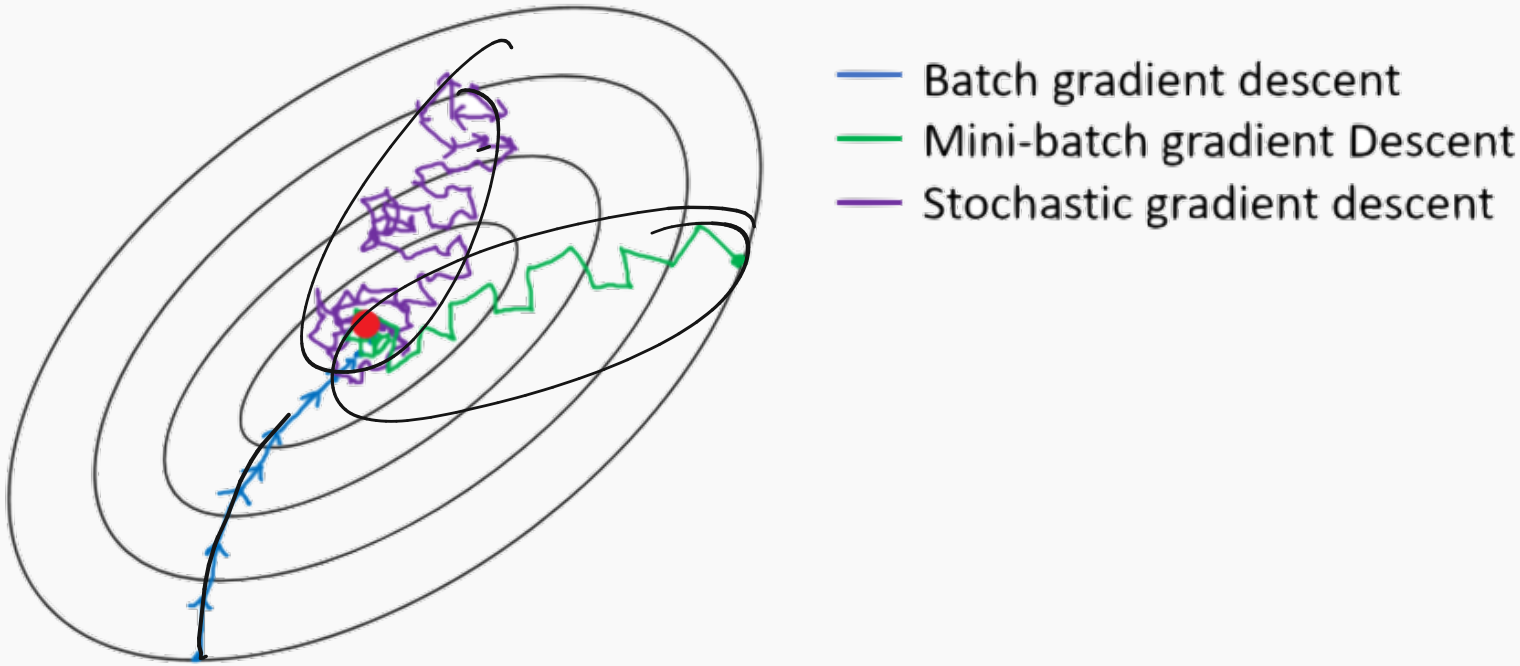
$$\mathbb{E}\left[\frac{n}{s}\sum_{i=1}^{s}\nabla L_{j_i}(\beta)\right] = \nabla L(\beta).$$

if $j_1, \ldots, j_s$ are chosen independently and uniformly at random from $1, \ldots, n$.

Instead of computing a full stochastic gradient, compute the average gradient of a small random set (a mini-batch) of training data examples.

Question: Why might we want to do this?

MINI-BATCH GRADIENT DESCENT



- Overall faster convergence (fewer iterations needed).
- Often not much slower per iteration and regular SGD.

Practical Mod. 2: **Per-parameter adaptive learning rate.**

Let $\mathbf{g} = \begin{bmatrix} g_1 \\ \vdots \\ g_p \end{bmatrix}$ be a stochastic or batch stochastic gradient. Our typical parameter update looks like:

$$\beta^{(t+1)} = \beta^{(t)} - \eta \mathbf{g}.$$

We've already seen a simple method for adaptively choosing the learning rate/step size η.

Practical Mod. 2: **Per-parameter adaptive learning rate.**

In practice, ML loss functions can often be optimized much faster by using "adaptive gradient methods" like Adagrad, Adadelta, RMSProp, and ADAM. These methods make updates of the form:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \begin{bmatrix} \eta_1 \cdot g_1 \\ \vdots \\ \eta_d \cdot g_d \end{bmatrix}$$

So we have a separate learning rate for each entry in the gradient (e.g. parameter in the model). And each $\eta_1, \ldots, \eta_p$ is chosen adaptively.

ADAPTIVE LEARNING RATE METHODS

Roughly, the idea is to normalize the j^{th} entry g_j by its historical average. I.e., $\eta_j \sim \frac{1}{\sqrt{\frac{1}{w}\sum_{i=t-w}^{t}\left(g_j^{(i)}\right)^2}}$. Lots of other things go into these methods though. Take my spring class to learn more!

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

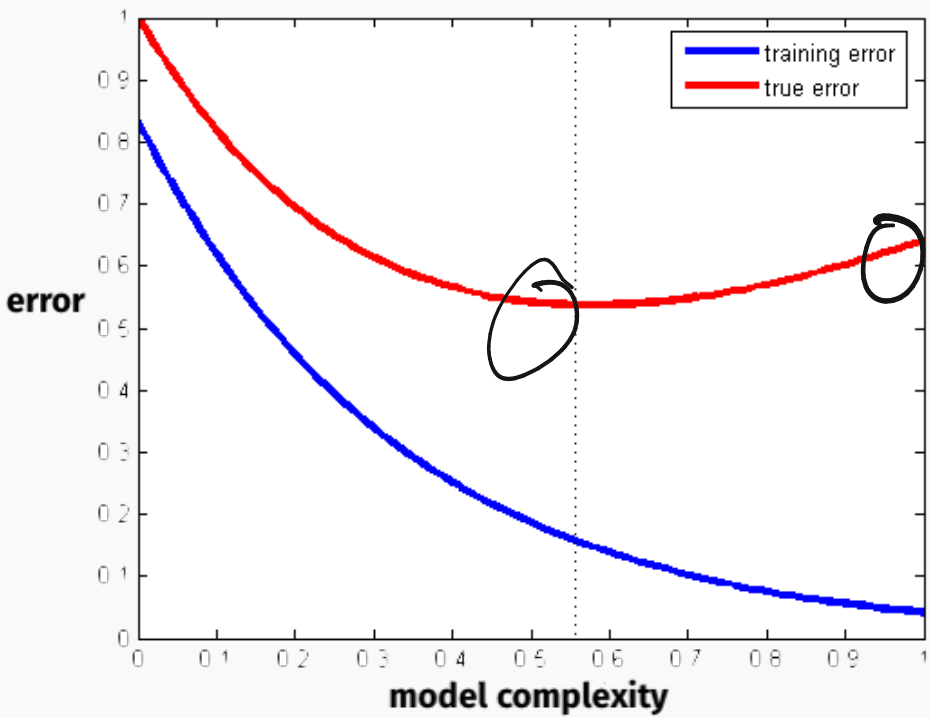
Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

[CITATION] Adam: A method for stochastic optimization
DP Kingma - arXiv preprint arXiv:1412.6980, 2014
☆ Save 99 Cite Cited by 193927 Related articles

LEARNING THEORY

THE FUNDAMENTAL CURVE OF ML

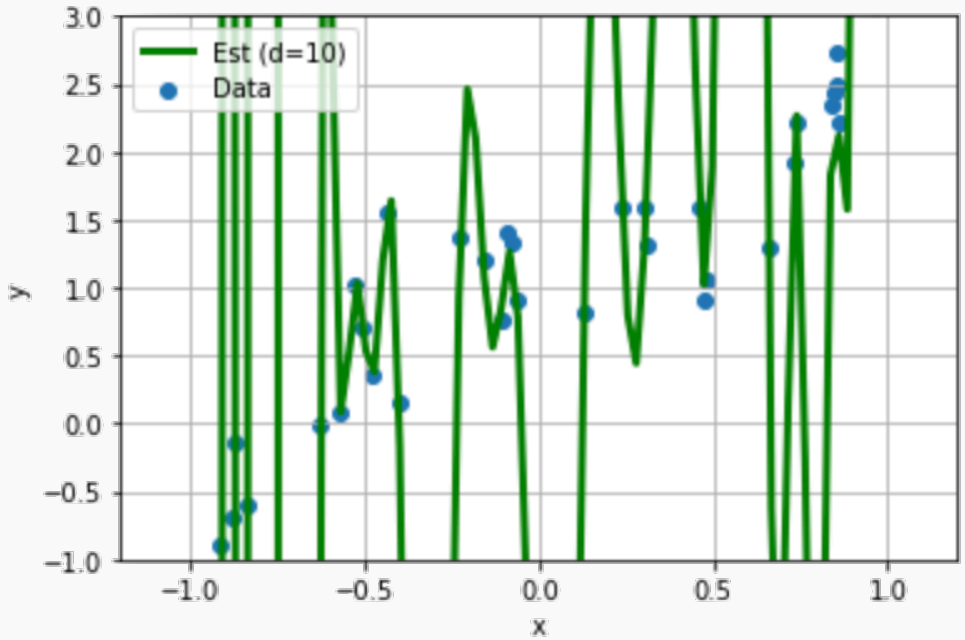
Key Observation: Due to overfitting, more complex models do not always lead to lower test error.



The more complex a model is, the more training data we need to ensure that we do not overfit.

EXAMPLE: POLYNOMIAL REGRESSION

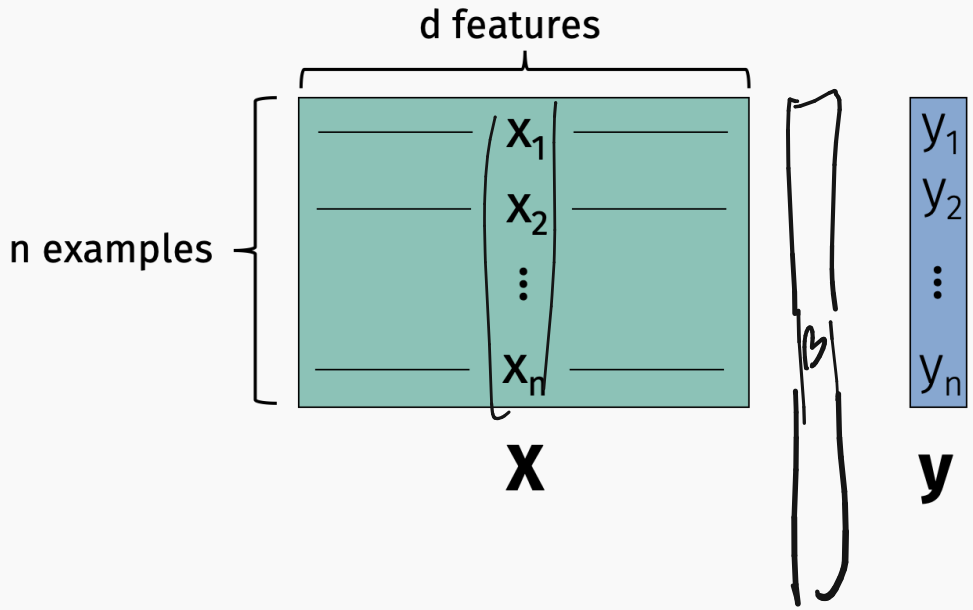
If we want to learn a degree q polynomial model, we will perfectly fit our training data if we have $n \leq q$ examples.



Need $n > q$ samples to ensure good generalization. How much more?

EXAMPLE: LINEAR REGRESSION

If we want to fit a multivariate linear model with d features, we will perfectly fit our training data if we have $n \leq d$ examples.



Need $> d$ samples to ensure good generalization.

How much more?

STATISTICAL LEARNING THEORY

Major goal in learning theory:

Formally characterize how much training data is required to ensure good generalization (i.e., good test set performance) when fitting models of varying complexity.

Statistical Learning Model:

- Assume each data example is randomly drawn from some distribution $(\mathbf{x}, y) \sim \mathcal{D}$.

For today: We will only consider classification problems so assume that $y \in \{0, 1\}$.

Statistical Learning Model:

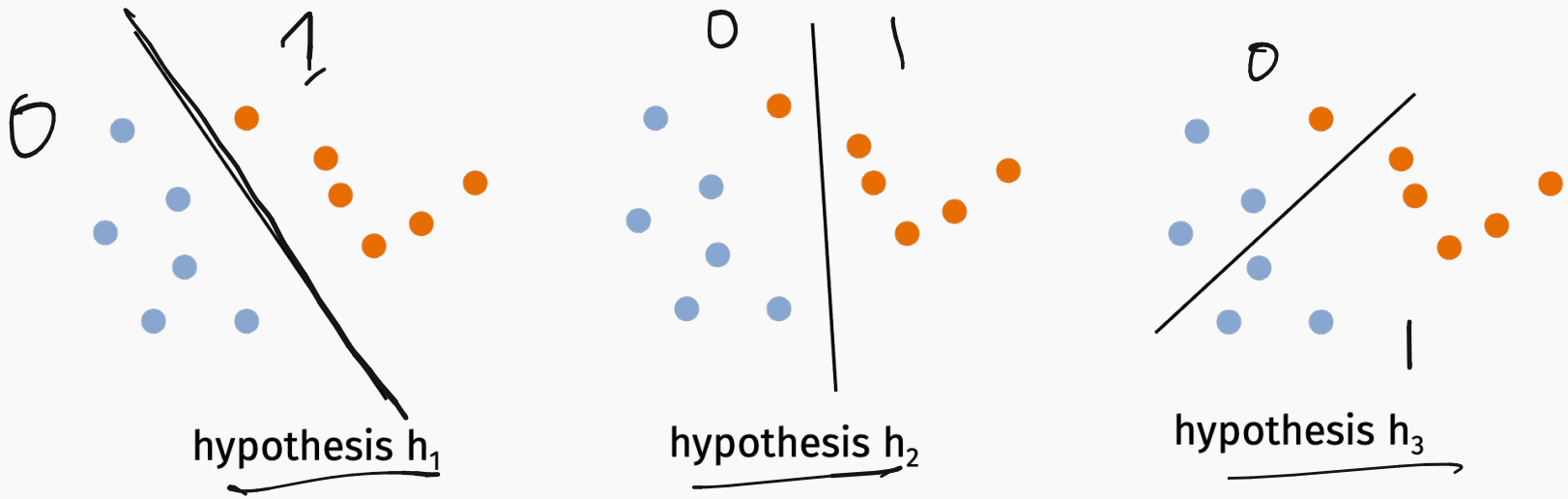
- Assume each data example is randomly drawn from some distribution $(\mathbf{x}, y) \sim \mathcal{D}$.
- Assume we want to fit our data with a function h (a "hypothesis") in some hypothesis class $\mathcal{H}$. For input $\mathbf{x}$, $h(\mathbf{x}) \rightarrow \{0, 1\}$.

Typically, h is just a model, instantiated with a specific set of parameters. I.e., h is the same as $f_{\boldsymbol{\theta}}$ for some choice of $\boldsymbol{\theta}$.

In this case, $\mathcal{H}$ is a set containing all functions of the form $f_{\boldsymbol{\theta}}$ for some choice of $\boldsymbol{\theta}$.

EXAMPLE HYPOTHESIS CLASS

Linear threshold functions:



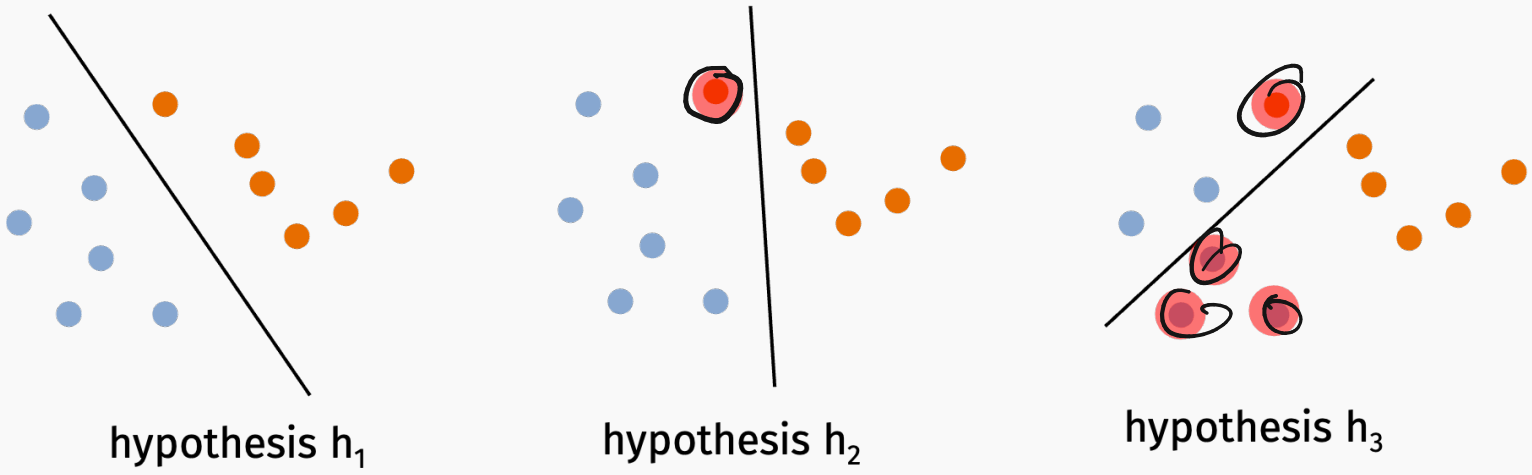
$\mathcal{H}$ contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

.

EXAMPLE HYPOTHESIS CLASS

Linear threshold functions:



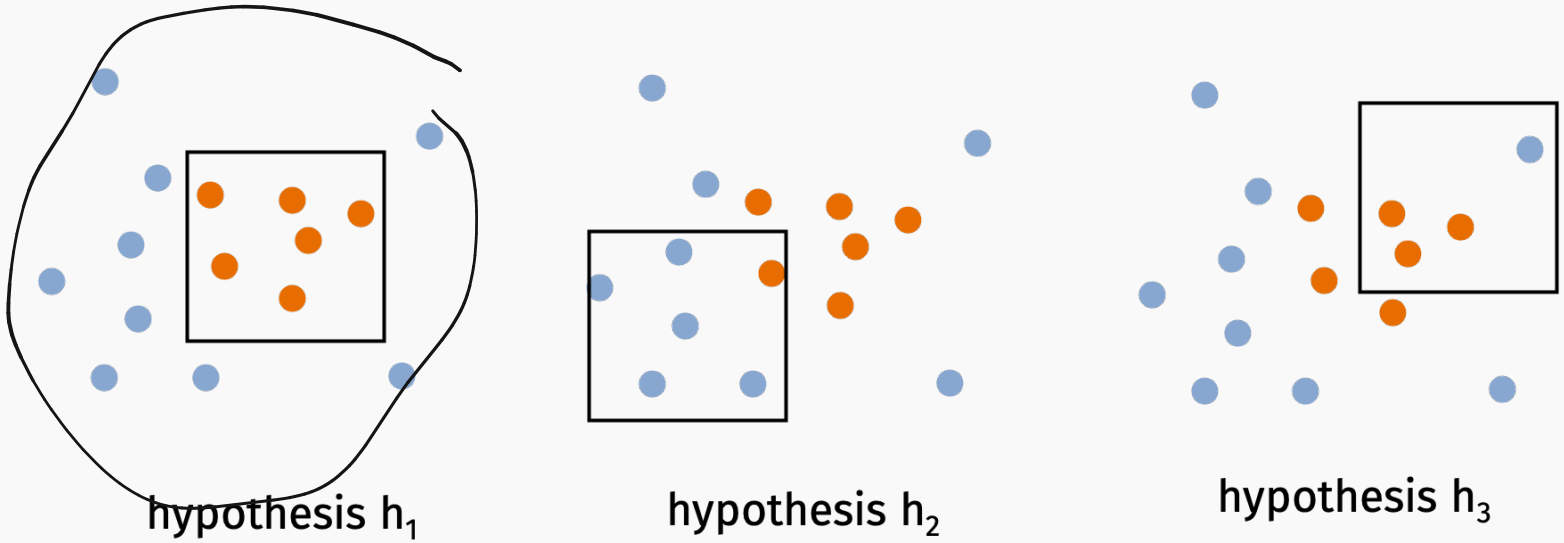
$\mathcal{H}$ contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

.

EXAMPLE HYPOTHESIS CLASS

Axis aligned rectangles:

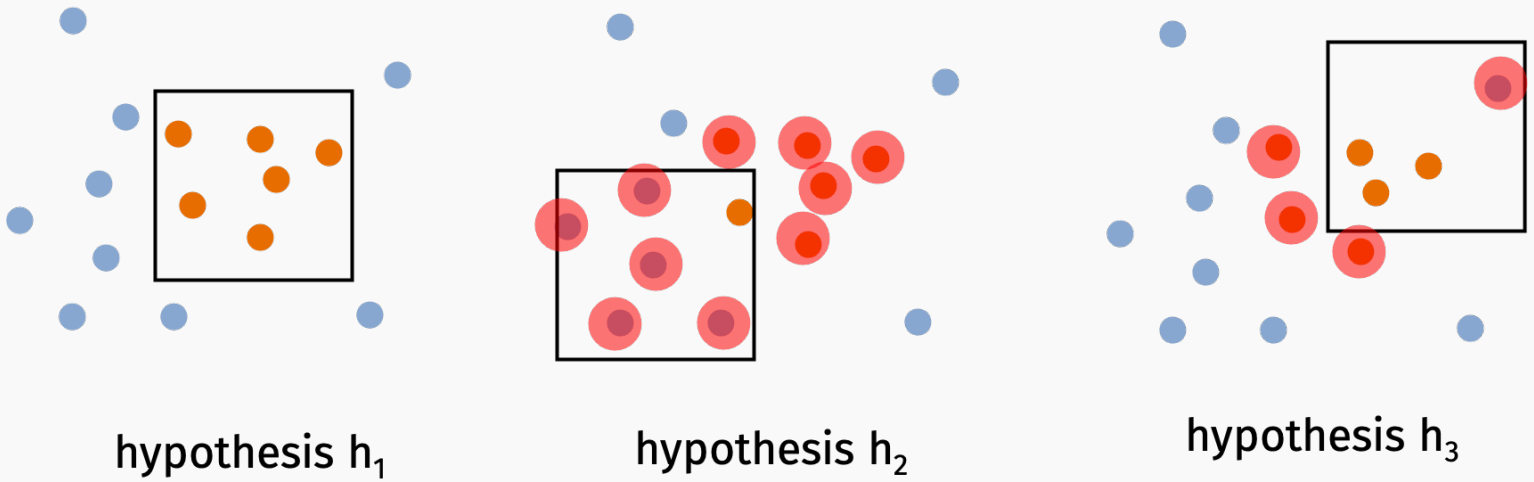


$\mathcal{H}$ contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[l_1 \leq x_1 \leq u_1 \text{ and } l_2 \leq x_2 \leq u_2]$$

EXAMPLE HYPOTHESIS CLASS

Axis aligned rectangles:



$\mathcal{H}$ contains all functions of the form:

$$h(\mathbf{x}) = \mathbb{1}[l_1 \leq x_1 \leq u_1 \text{ and } l_2 \leq x_2 \leq u_2]$$

3-DNF

Disjunctive Normal Form (DNF) formulas:

Assume $\mathbf{x} \in \{0,1\}^d$ is binary.

$\mathcal{H}$ contains functions of the form:

$$h(\mathbf{x}) = (x_1 \wedge \bar{x}_5 \wedge x_{10}) \vee (\bar{x}_3 \wedge x_2) \vee \ldots \vee (\bar{x}_1 \wedge x_2 \wedge x_{10})$$

$\wedge$ = "and", $\vee$ = "or"

k-DNF: Each conjunction has at most k variables.

POPULATION AND EMPIRICAL ERROR

Same as "population risk" for the zero one loss:

$\mathbb{E}[R_{emp}(h)] = R_{pop}(h)$

- **Population ("True") Error**:

$$R_{pop}(h) = \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[h(\mathbf{x}) \neq y]$$

- **Empirical Error**: Given a set of samples $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \sim \mathcal{D}$,

$$R_{emp}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$$

Goal is to find $h \in \mathcal{H}$ that minimizes population error.

Find $h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} R_{pop}(h)$

Let $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \sim \mathcal{D}$ be our training set and let h_{train} be the empirical error minimizer[1]:

$$h_{train} = \arg\min_{h \in H} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$$

Let h^* be the population error minimizer:

$$h^* = \arg\min_{h \in H} R_{pop}(h) = \arg\min_{h} \Pr_{(\mathbf{x},y)\sim\mathcal{D}} [h(\mathbf{x}) \neq y]$$

Goal: Ideally, for some small ϵ, $R_{pop}(h_{train}) - R_{pop}(h^*) \leq \epsilon$.

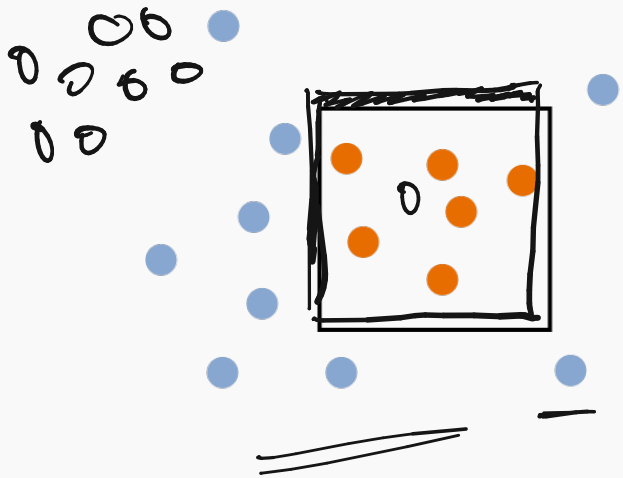
[1] Note that here we are assuming we have an algorithm for computing h_{train}. Might be challenging in practice.

$R_{pop}(h_{train}) \leq R_{pop}(h^*) + \epsilon$

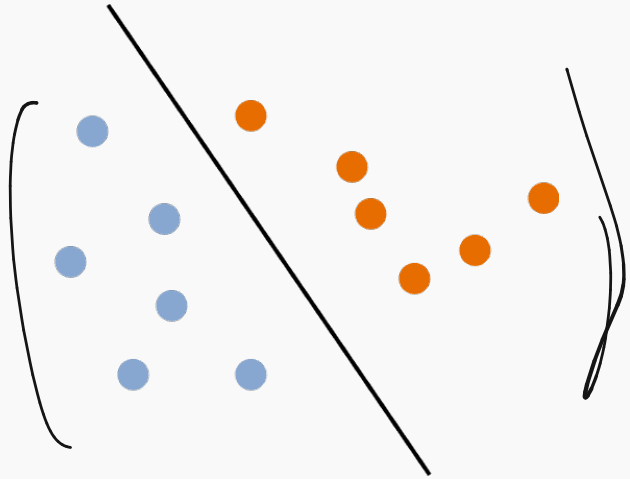
$R_{pop}(h^*) = 0$

Simplification for today: Assume we are in the realizable setting, which means that $R_{pop}(h^*) = 0$. I.e. there is some hypothesis in our class $\mathcal{H}$ that perfectly classifies the data.

Formally, for any $(\mathbf{x}, y)$ such that $\Pr_{\mathcal{D}}[\mathbf{x}, y] > 0$, $h^*(\mathbf{x}) = y$.



hypothesis h*



hypothesis h*

Extending to the case when $R_{pop}(h^*) \neq 0$ is not hard, but the math gets a little trickier. And intuition is roughly the same.

Probably Approximately Correct (PAC) Learning (Valiant, 1984):

For a hypothesis class $\mathcal{H}$, data distribution $\mathcal{D}$, and training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, let $h_{train} = \arg\min_h \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

Main question: In the realizable setting, how many training samples n are required so that, with probability $1 - \delta$,

$$R_{pop}(h_{train}) \leq \epsilon?$$

I.e., how many training samples are required to ensure we probably find an approximately correct hypothesis?

The number of samples n will depend on ϵ, δ, and the complexity of the hypothesis class $\mathcal{H}$. Perhaps surprisingly, it will not depend at all on $\mathcal{D}$.

COMPLEXITY OF HYPOTHESIS CLASS

Many ways to measure complexity of a hypothesis class. Today we will start with the simplest measure: the number of hypotheses in the class, $|\mathcal{H}|$.

Example: What is the number of hypothesis in the class of 3-DNE formulas on d dimensional inputs $\mathbf{x} = [x_1, \ldots, x_d] \in \{0,1\}^d$?

$$h(\mathbf{x}) = (x_1 \wedge \bar{x}_5 \wedge x_{10}) \vee (\bar{x}_3 \wedge x_2) \vee \ldots \vee (\bar{x}_1 \wedge x_2 \wedge x_{10})$$

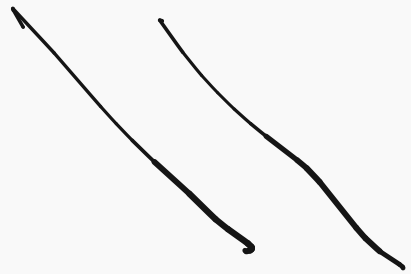
$$\binom{2d}{3} + \binom{2d}{2} + \binom{2d}{1} \leq (2d)^3 + 2d^2 + 2d \leq O(d^3)$$

$$|H| \leq 2^{O(d^3)}$$

COMPLEXITY OF HYPOTHESIS CLASS

$|\mathcal{H}| = C^{d+1}$ $\quad \log(C^{d+1}) = (d+1)\log(C)$

Caveat: Many hypothesis classes are infinitely sized. E.g. the set of linear thresholds



$$h(\mathbf{x}) = \mathbb{1}[\mathbf{x}^T \boldsymbol{\beta} \geq \lambda]$$

But you could imagine approximating $\mathcal{H}$ by a finite hypothesis class. E.g. take values in $\boldsymbol{\beta}, \lambda$ to lie on a finite grid of size C. Then how many hypothesis are there?

Moving from finite to infinite sized hypothesis classes is a huge area of learning theory (VC theory, Rademacher complexity, etc.). Will touch on this if we have time.

Consider the realizable setting with hypothesis class $\mathcal{H}$, data distribution $\mathcal{D}$, training data set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, and $h_{train} = \arg\min_h \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

Theorem

If $n \geq \frac{1}{\epsilon}\left(\log|\mathcal{H}| + \log\frac{1}{\delta}\right)$, *then with probability* $1-\delta$,

$$R_{pop}(h_{train}) \leq \epsilon.$$

Roughly how many training samples are needed to learn 3-DNF formulas? To learn (discretized) linear threshold funtions?

$O(d^3/\epsilon)$ samples to provably learn 3-DNF.

$O(d/\epsilon)$ samples to PAC learn linear classifiers

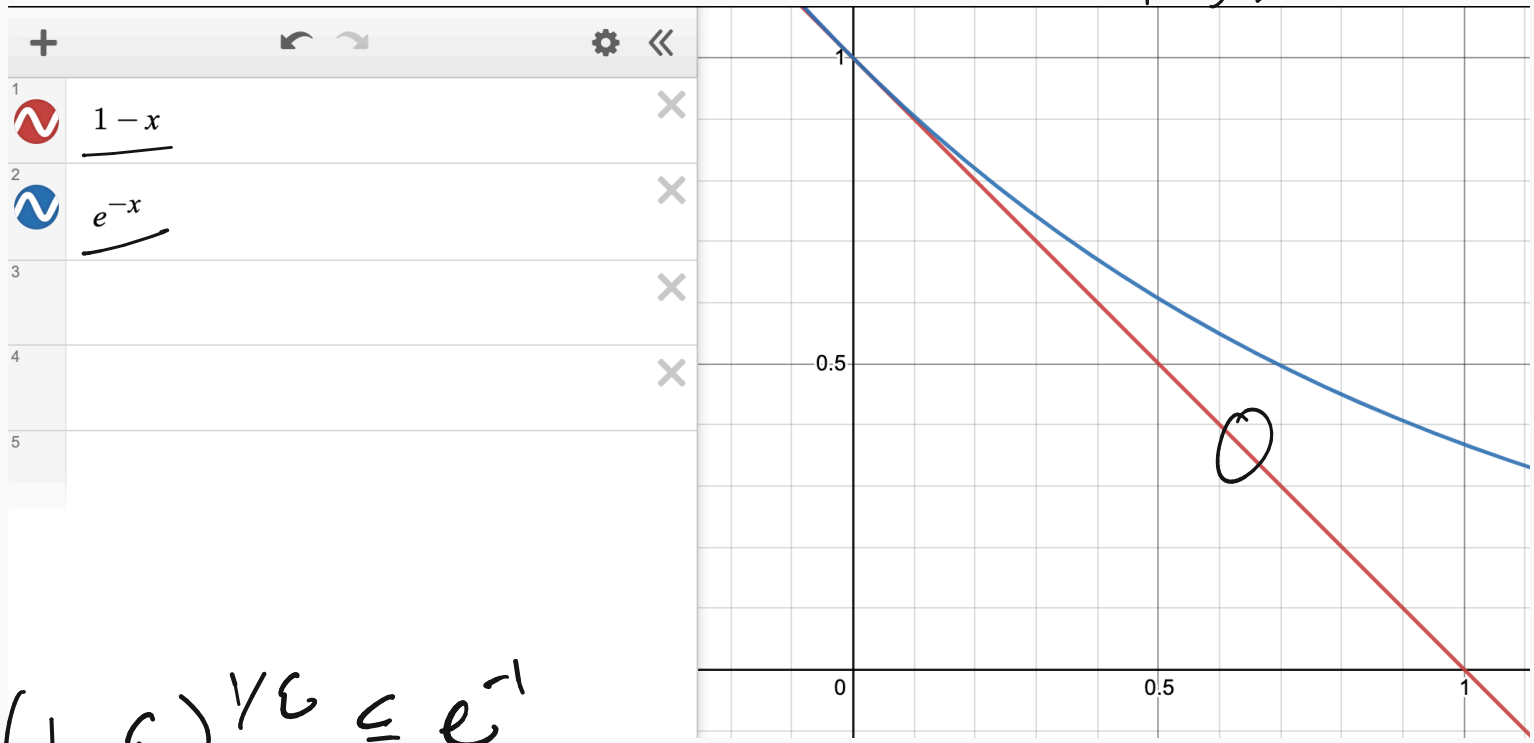
Two ingredients needed for proof:

1. For any $\epsilon \in [0, 1]$, $(1 - \epsilon) \leq e^{-\epsilon}$.
2. Union bound. Basic but important inequality about probabilities.

ALGEBRAIC FACT

For any $\epsilon \in [0, 1]$, $(1 - \epsilon) \leq e^{-\epsilon}$

$$e = \lim_{x \to \infty} \left(1 + \frac{1}{x}\right)^x$$



$(1 - \epsilon)^{1/\epsilon} \leq e^{-1}$

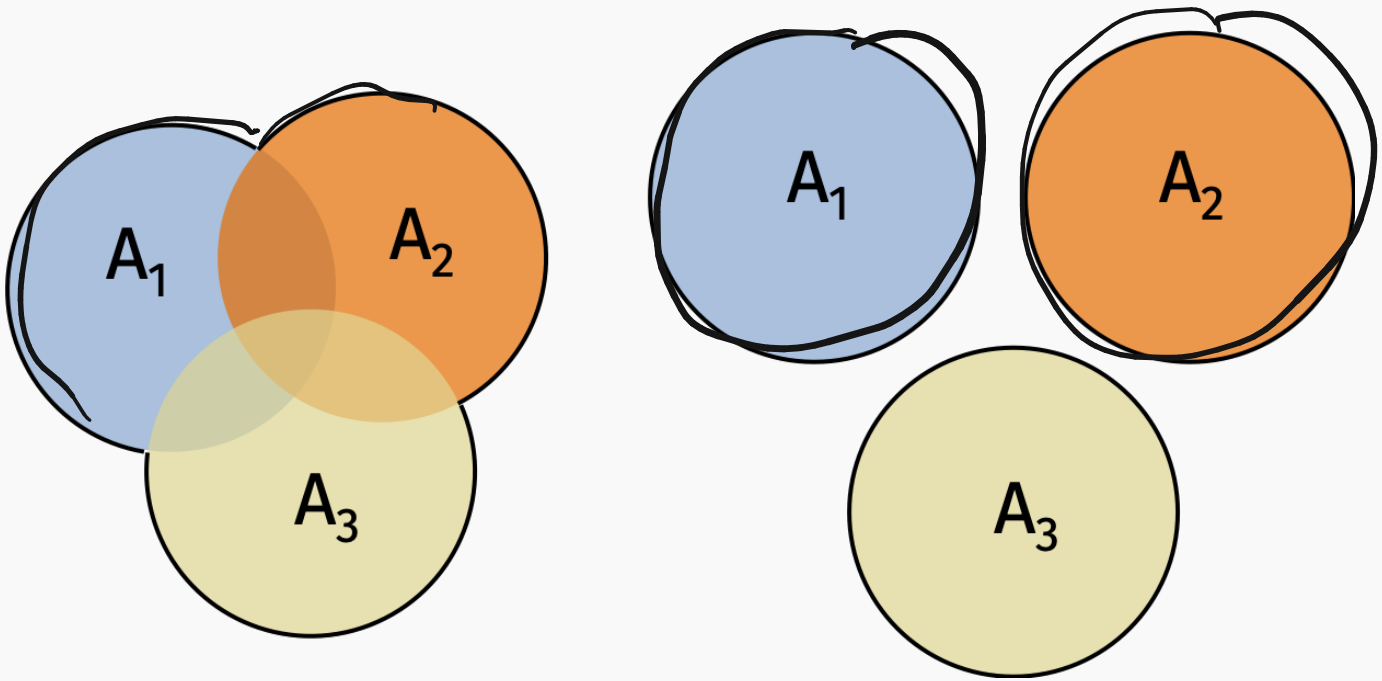
Recall that:

$$\frac{1}{e} = \lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^x = \lim_{\epsilon \to 0} (1 - \epsilon)^{1/\epsilon}$$

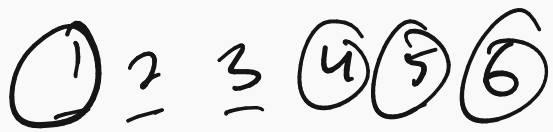
Lemma (Union Bound)

For any random events $A_1, \ldots, A_k$:

$$\Pr[A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_k] \leq \Pr[A_1] + \Pr[A_2] + \ldots + \Pr[A_k].$$



Sometimes written as $\Pr[A_1 \cup A_2 \cup \ldots \cup A_k]$.



Union bound is not tight: What is the probability that a dice roll is odd, or that it is ≤ 2?

$$\frac{4}{6} < \frac{3}{6} + \frac{2}{6}$$

Union bound is tight: What is the probability that a dice roll is 1, or that it is ≥ 4?

$$\frac{4}{6} = \frac{1}{6} + \frac{3}{6}$$

Consider the realizable setting with hypothesis class $\mathcal{H}$, data distribution $\mathcal{D}$, training data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$, and $h_{train} = \arg\min_h \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[h(\mathbf{x}_i) \neq y_i]$.

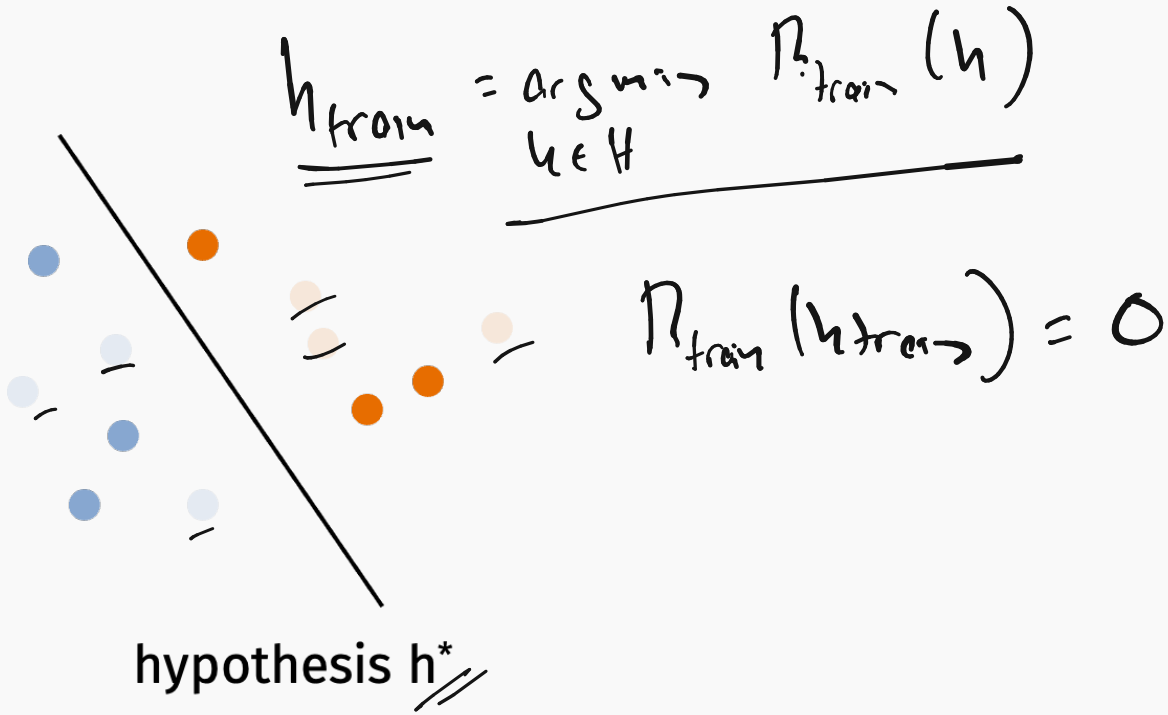
Theorem

If $n \geq \frac{1}{\epsilon} \left(\log |\mathcal{H}| + \log \frac{1}{\delta} \right)$, *then with probability* $1 - \delta$,

$$R_{pop}(h_{train}) \leq \epsilon$$
$$= R_{pop}(h^*) + \epsilon$$

$\log(|H|/\delta) = \log(|H|) + \log(1/\delta)$

First observation: Because we are in the realizable setting, there is always at least one $h \in \mathcal{H}$ such that $h(\mathbf{x}_i) = y_i$ for all $i \in 1, \ldots, n$.



Proof approach: Show that for any fixed hypothesis h^{bad} with $R_{pop}(h^{bad}) > \epsilon$, it is very unlikely that $R_{train}(h^{bad}) = 0$. So with high probability, we will not choose a bad hypothesis.

PROOF

$\mathbb{P}[h^{bad}(x) \neq y] > \epsilon.$

Let h^{bad} be a fixed hypothesis with $R_{pop}(h^{bad}) > \epsilon$. For $(\mathbf{x}, y)$ drawn from $\mathcal{D}$, what is the probability that $h^{bad}(\mathbf{x}) = y$?

$\Pr[h^{bad}(x) = y] \leq (1 - \epsilon)$

$R_{pop}(h) = \mathbb{E}_{(x,y) \sim \mathcal{D}} \mathbb{1}[h^{bad}(x) \neq y]$

What is the probability that for a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)$ drawn from $\mathcal{D}$ that $h^{bad}(\mathbf{x}_i) = y_i$ for all i? I.e. that $R_{train}(h^{bad}) = 0$.

$\leq (1 - \epsilon)^n$

Claim

For any fixed hypthesis h^{bad} with $R_{pop}(h^{bad}) > \epsilon$, the probability that $R_{train}(h) = 0$ can be bounded by:

$$\Pr[R_{train}(h^{bad}) = 0] < e^{-\epsilon n}.$$

$\Pr[R_{train}(h^{bad}) = 0] \leq (1-\epsilon)^n \leq (e^{-\epsilon})^n = e^{-\epsilon n}$.

Set $n \geq \frac{1}{\epsilon}\log(|\mathcal{H}|/\delta)$. Then we have that for any fixed hypthesis h^{bad} with $R_{pop}(h^{bad}) > \epsilon$,

$n \geq \frac{1}{\epsilon}\log(z)$

$$\Pr[R_{train}(h^{bad}) = 0] \leq \frac{\delta}{|\mathcal{H}|}. \quad \epsilon \leq \frac{1}{2}$$

$e^{-\epsilon \cdot \frac{1}{\epsilon}\log(|H|/\delta)} = e^{-\log(|H|/\delta)} = \frac{\delta}{|H|}$

UNION BOUND APPLICATION

Let $h_1^{bad}, \ldots, h_m^{bad}$ be all hypthesis in $\mathcal{H}$ with $R_{pop}(h) > \epsilon$.

$$\begin{aligned}
&\Pr[R_{train}(h_1^{bad}) = 0 \text{ or } \ldots \text{ or } R_{train}(h_m^{bad}) = 0] \leq \ldots \\
&\leq \Pr[R_{train}(h_1^{bad}) = 0] + \ldots + \Pr[R_{train}(h_m^{bad}) = 0] \\
&\leq m \cdot \frac{\delta}{|\mathcal{H}|} < \delta.
\end{aligned}$$

How large can m be? Certainly no more than $|\mathcal{H}|$!

So, the probability that any bad hypothesis has 0 training error is at most δ.

Accordingly, with $\geq (1 - \delta)$ probability, when we choose a hypothesis with 0 training error, we are choosing a good one. I.e. one with $R_{pop}(h) \leq \epsilon$.

TAKE AWAY

Important take-away as we start working with neural networks and other more complex models:

- We expect the amount of training data required to learn a model to scale logarithmically with the size of the model class being fit, $|\mathcal{H}|$.
- Typically, the size of $\mathcal{H}$ grows exponentially with the number of parameters in the model.
- So overall, our training data size should exceed the number of model parameters (and then some).

I.e., our experience from polynomial regression and linear regression is somewhat universal.

INFINITE HYPOTHESIS CLASSES

Ideally we would like to give formal results for infinite hypothesis classes (e.g., any class with real valued parameters) without resorting to discretization. One of the most important tools for doing so is the Vapnik–Chervonenkis (VC) dimension.

Theorem

Let $\mathcal{H}$ be a hypothesis class with VC dimension V. If $n \geq \frac{2\log(1/\epsilon)}{\epsilon}(V + \log\frac{2}{\delta})$, then with probability $1-\delta$,

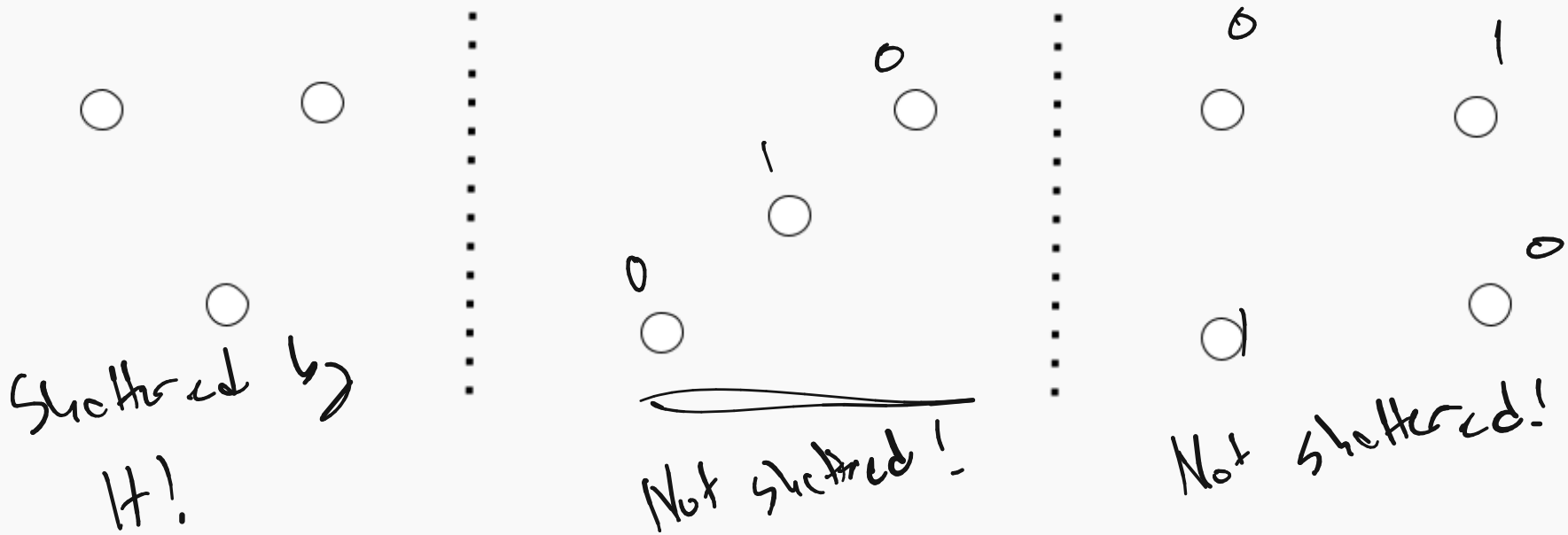
$$R_{pop}(h_{train}) \leq \epsilon.$$

Essentially the same bound as earlier, but $|\mathcal{H}|$ replaced with VC dimension, V.

$q_0 = 3$

We say a hypothesis class $\mathcal{H}$ shatters a set of points $\mathbf{x}_1, \ldots, \mathbf{x}_q \in \mathbb{R}^d$ if there is some hypothesis $h \in \mathcal{H}$ that matches any possible labeling of the data.

Example: Linear classifiers in $d = 2$ dimensions.



Definition (VC dimension)

The VC dimension of a hypothesis class $\mathcal{H}$ over points in $\mathbb{R}^d$ is the size of the largest point set that $\mathcal{H}$ shatters.

What is the VC dimension of the set of linear classifiers in $d = 2$ dimensions?

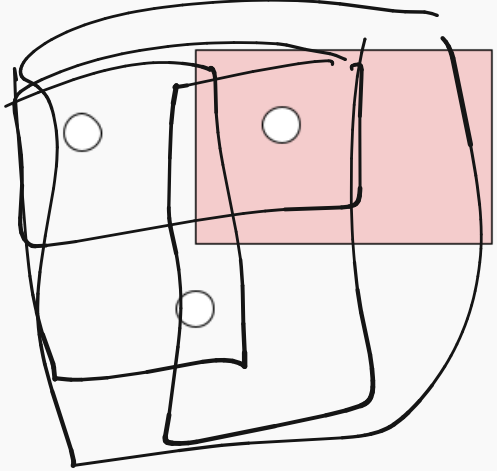
VC dimension = 3

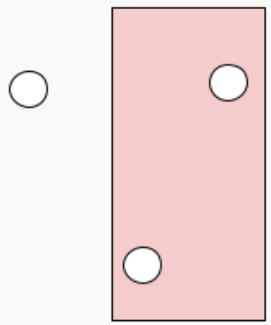
VC dimension = $d + 1$

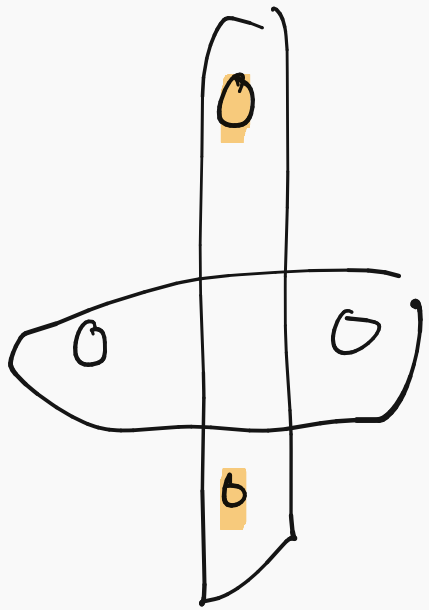
Definition (VC dimension)

The VC dimension of a hypothesis class $\mathcal{H}$ over points in $\mathbb{R}^d$ is the size of the largest point set that $\mathcal{H}$ shatters.

What about axis aligned rectangles?







OTHER IMPORTANT TOPICS

- Generalization of VC dimension to multi-class classification.
- Generalization to regression.
- Tighter bounds that take the distribution $\mathcal{D}$ into account (e.g., via Rademacher complexity).

At the end of the day, the main value of these tools is to improve our understanding of the complexity of different modes/hypothesis classes.

In practice, train/test split is still the major tool for determining if we are overfitting and need more data.