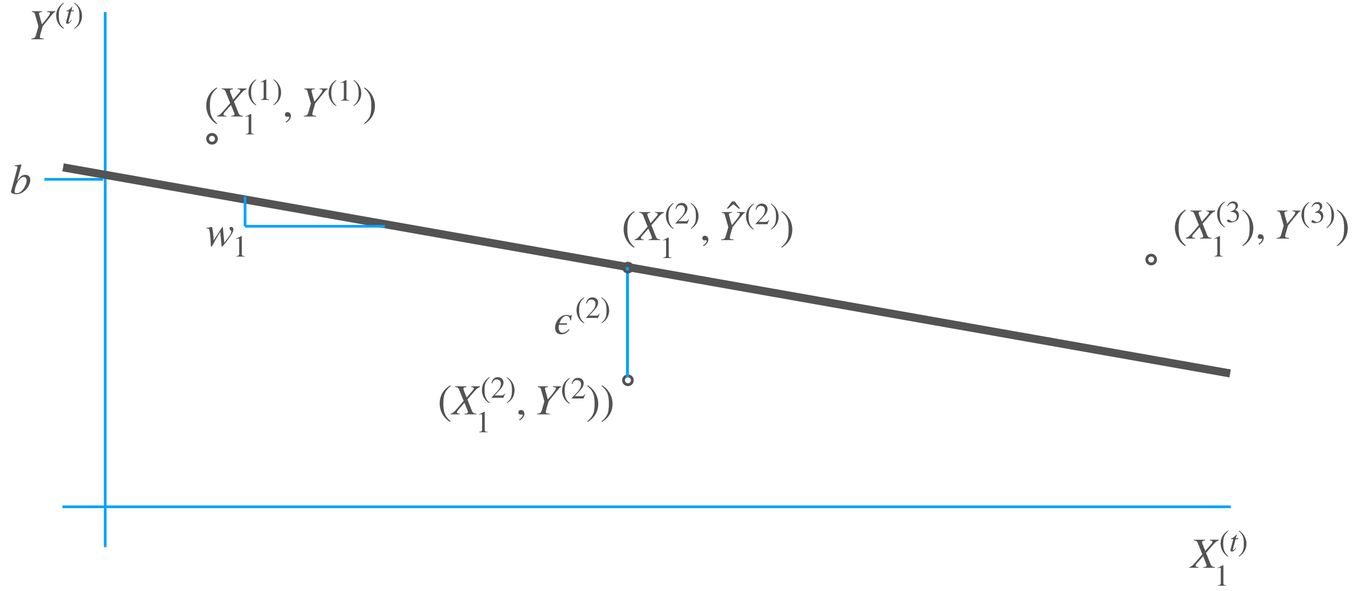
Linear regression



- Regression = supervised prediction of *real* labels
  - data: features $X^{(t)} \in \mathbb{R}^d$, labels $Y^{(t)} \in \mathbb{R}$, $t = 1 : T$
- *Linear* regression = predict a linear function of features
  - model: predict $\hat{Y}^{(t)} = w \cdot X^{(t)} = \sum_i w_i X_i^{(t)}$ or $w \cdot X^{(t)} + b$
  - can omit b if we have a constant feature

Linear regression

- Objective:
  - minimize sum of squared errors:

$$\frac{1}{2}\sum_{t=1}^{T}(\underbrace{Y^{(t)}-\hat{Y}^{(t)}}_{\epsilon^{(t)}})^2$$

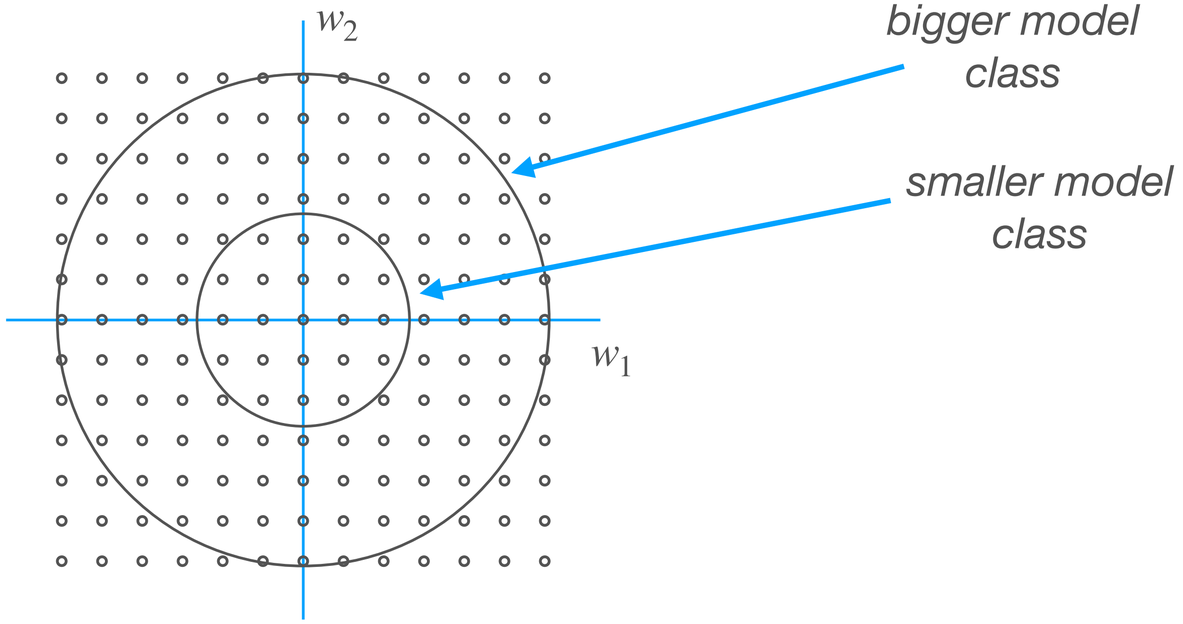
  - plus optional regularizer (see next slides)
- E.g., minimize

$$L(w)=\frac{1}{2}\sum_{t=1}^{T}(Y^{(t)}-\hat{Y}^{(t)})^2+\frac{\lambda}{2}\|w\|_2^2$$

Model complexity

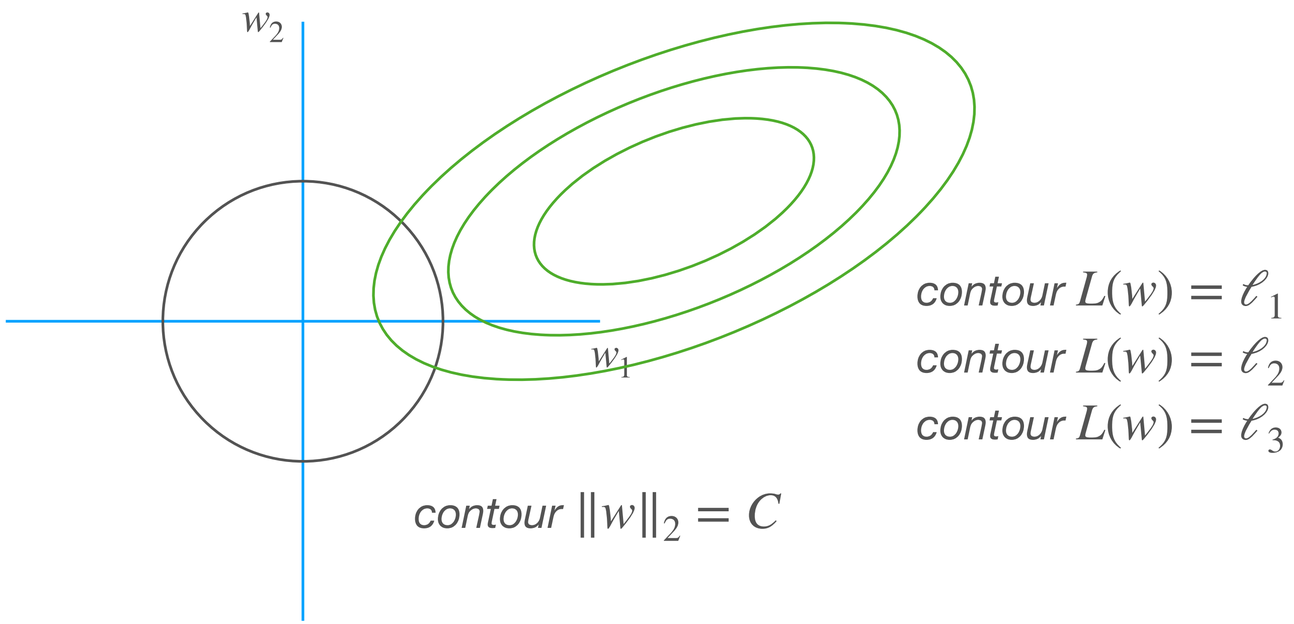
*Terminology: **model** can mean either the structure or the structure plus fitted parameters*
*To distinguish, **model class** vs. **trained/fitted model***

Trained model = hypothesis
One or more model classes make up hypothesis space



- Model complexity (for model selection):
  - how many features/weights
  - how many *nonzero* weights
  - *size* of weights (there are more effectively-distinct weight vectors in $\{w \mid \|w\| \leq 100\}$ than in $\{w \mid \|w\| \leq 1\}$)

Model selection

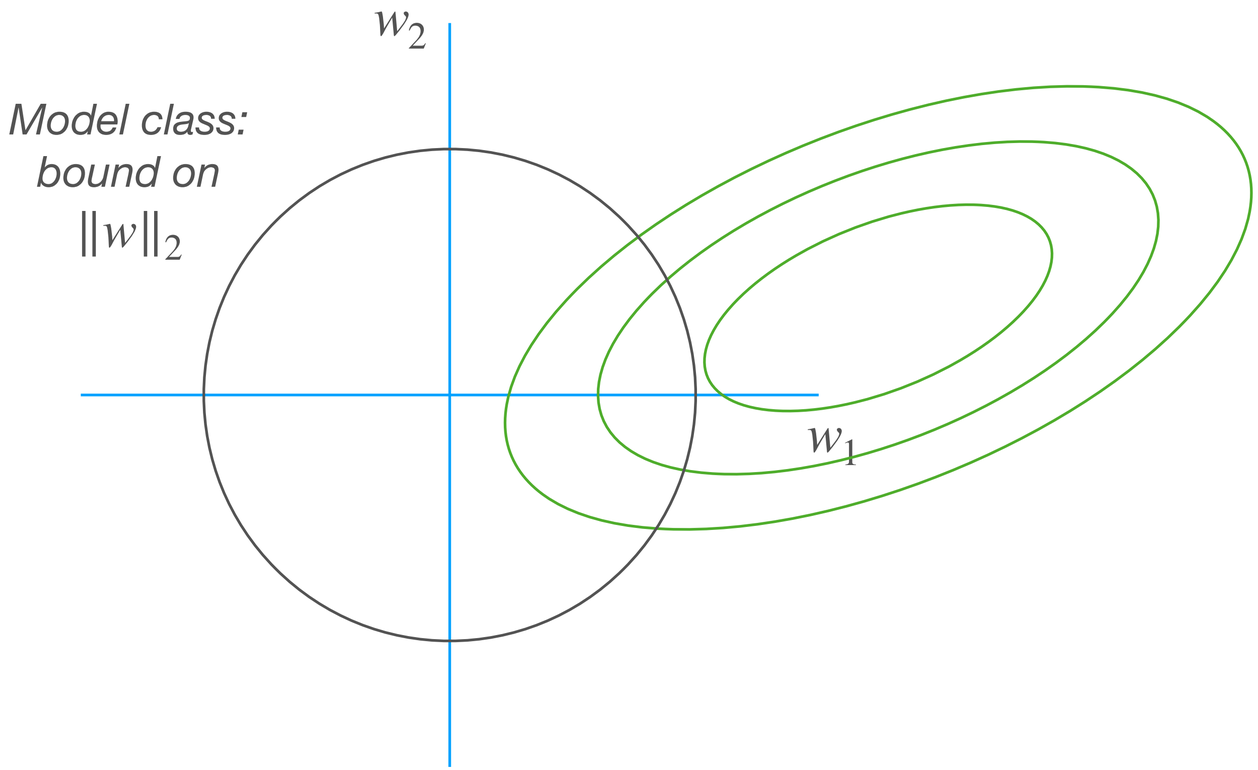


- To limit complexity, can limit (e.g.) norm of weight vector
- Corresponding model fitting problem:

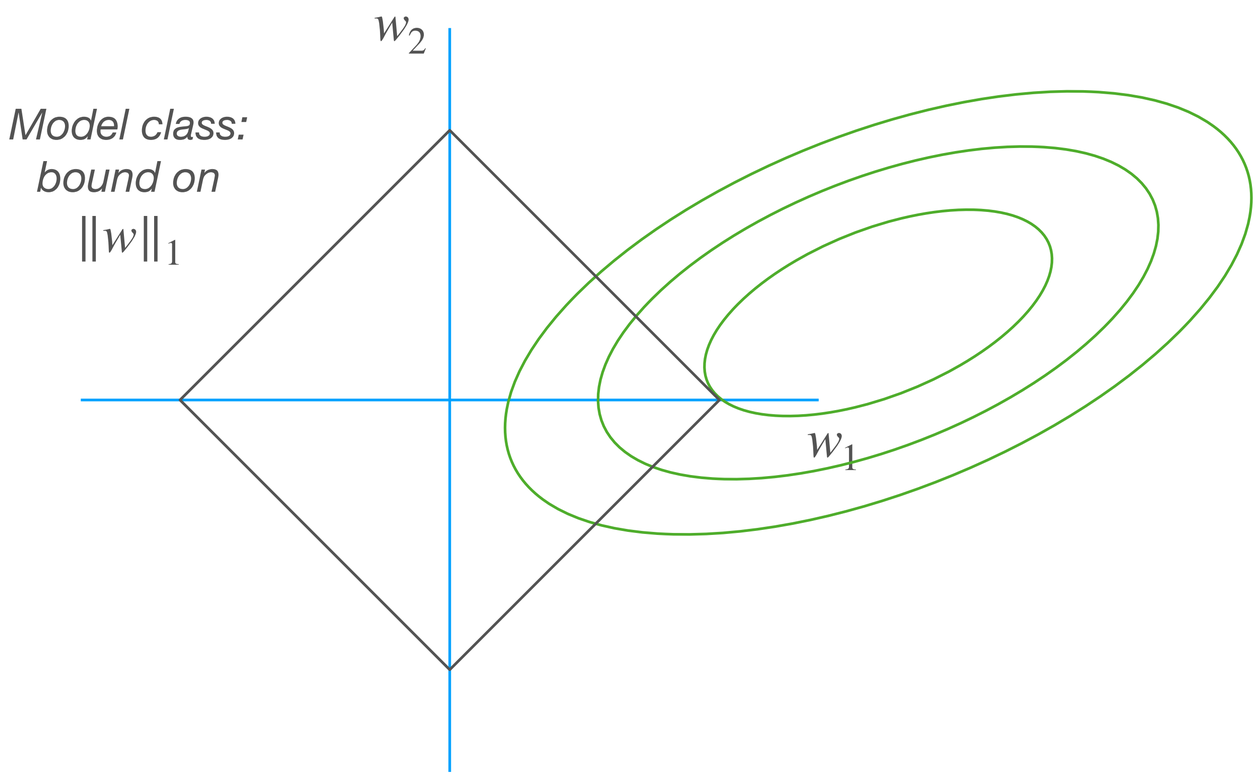
$$\min_w L(w) = \frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - X^{(t)} \cdot w)^2 \quad \text{s.t.} \quad \|w\|_2 \le C$$

- Model selection = picking C

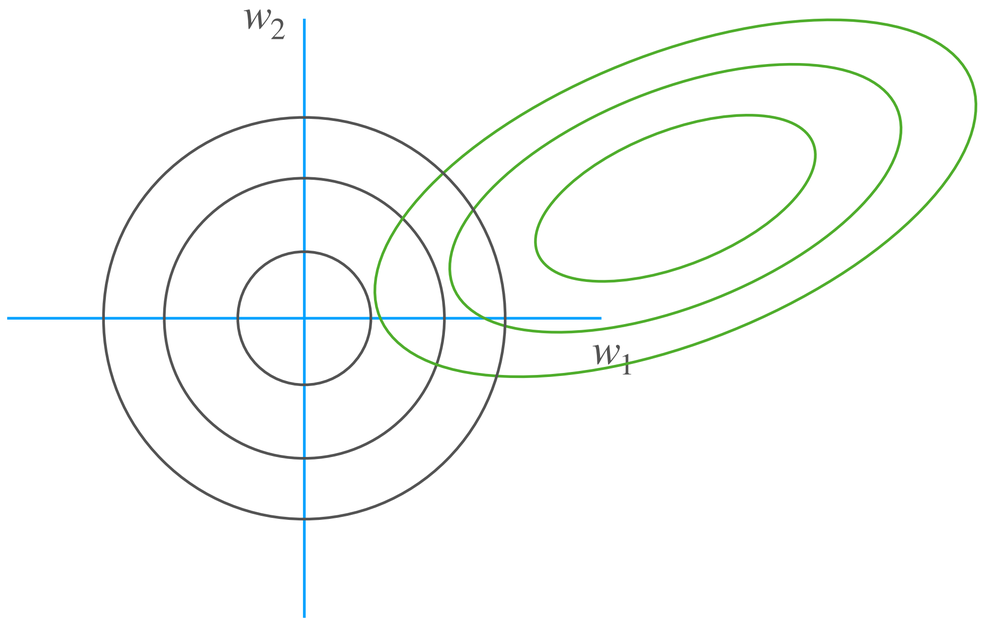
Size (norm) and count of nonzeros are related



Size (norm) and count of nonzeros are related



Regularization by penalty vs. constraint



- Instead of constrained minimization

$$\min_w L(w) = \frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - X^{(t)} \cdot w)^2 \quad \text{s.t.} \quad \|w\|_2 \le C$$

- we can try to directly trade off $L(w)$ and $\|w\|_2$ in optimizer (avoid outer model selection loop):

$$\min_w L(w) = \frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - X^{(t)} \cdot w)^2 + \frac{\lambda}{2}\|w\|_2^2$$

Bias-variance tradeoff

- Regularization parameters λ, C trade off underfitting vs. overfitting
  - underfitting is often called ***bias:*** the correct answer is not an option, so we pick something that might be far from it
  - overfitting shows up as ***variance*** of parameters w: more choices $\rightarrow$ more variability in which one we select

Overfitting, bias, variance example

```
n = 10
k = 7
X = np.random.normal(size=(n, k))
Y = np.random.normal(size=(n, 1))
w, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
(np.linalg.norm(w), np.mean((Y - X@w)**2))
```

- What is the smallest MSE this code should return if there were no selection bias?
- What is the actual best w?

Overfitting, bias, variance example

```
n = 10
k = 7
X = np.random.normal(size=(n, k))
Y = np.random.normal(size=(n, 1))
w, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
(np.linalg.norm(w), np.mean((Y - X@w)**2))
```

- What is the smallest MSE this code should return if there were no selection bias?
- What is the actual best w?

```
# three runs (n=10, k=7):

(1.729258520013365, 0.0940395032823642 1)
(1.9706932994777069, 0.07379394277037196)
(3.591128626783351, 0.47238275606461944)
```

Overfitting, bias, variance example

```
n = 10
k = 7
X = np.random.normal(size=(n, k))
Y = np.random.normal(size=(n, 1))
w, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
(np.linalg.norm(w), np.mean((Y - X@w)**2))
```

- What is the smallest MSE this code should return if there were no selection bias?
- What is the actual best w?

```
# three runs (n=10, k=7):

(1.729258520013365, 0.0940395032823642 1)
(1.9706932994777069, 0.07379394277037196)
(3.591128626783351, 0.47238275606461944)

# three runs (n=20, k=2):

(0.20006773436081007, 0.8932440732695813)
(0.3765035073938464, 0.9200354120567253)
(0.576420124635776 1, 0.7614382010070438)
```

Linear regression: summary of objective

- Objective:
  - minimize sum of squared errors: $\frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - \hat{Y}^{(t)})^2$
  - plus optional regularizer: e.g., L_2, L_1, both
  - $\|w\|_2^2$ called *ridge regression*
  - $\|w\|_1$ called *LASSO*
  - if we use both, *elastic net*
- E.g., minimize

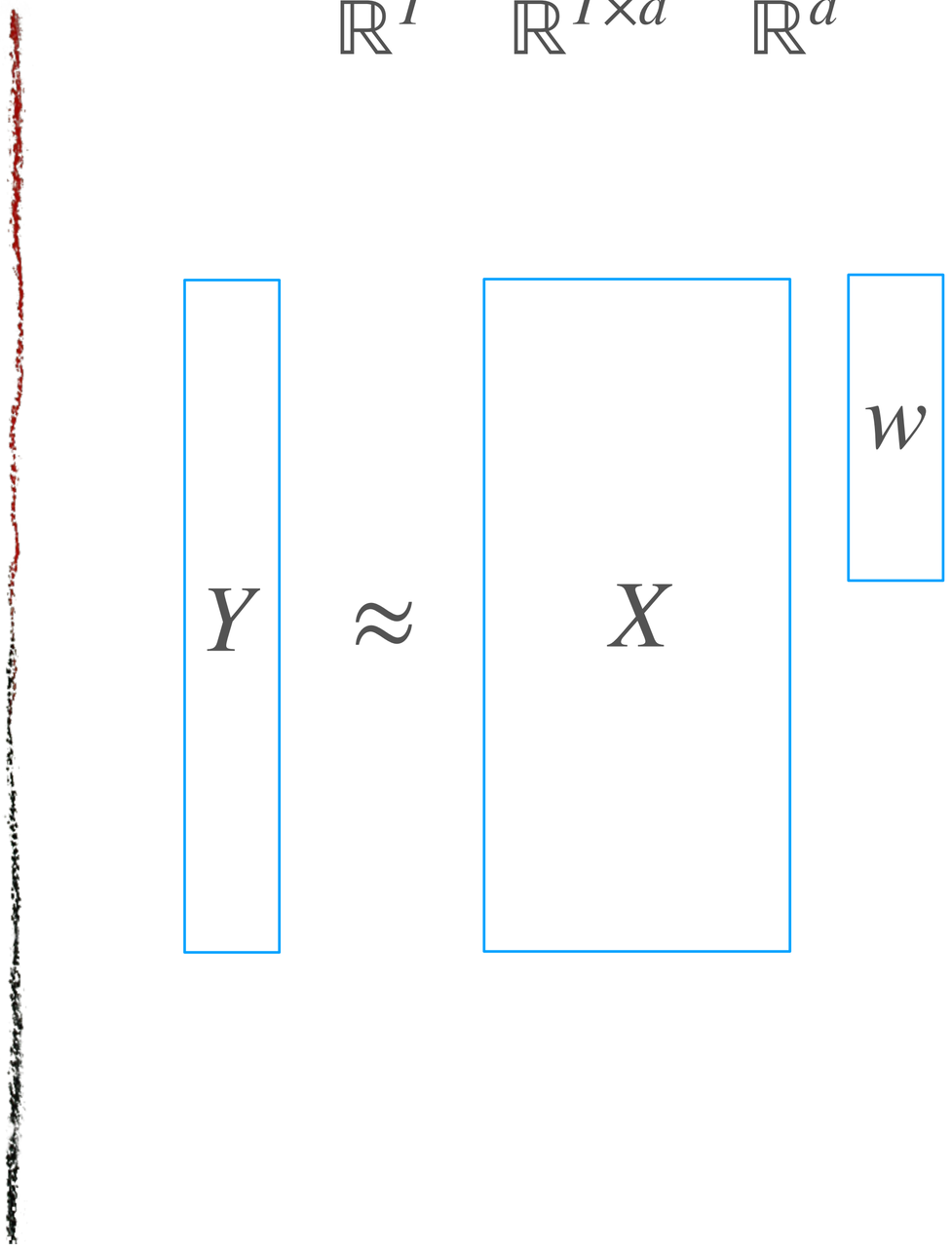
$$L(w) = \frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - \hat{Y}^{(t)})^2 + \frac{\lambda}{2}\|w\|_2^2 + \beta\|w\|_1$$

Solving it

- To find best w, several options:
  - minimize symbolically by setting $\nabla_w L$ to 0
    - sub-choices: QR, SVD, Cholesky
  - iterative optimizer
    - sub-choices: stochastic vs. full gradient
    - 1st-order like SGD or Adam
    - 2nd-order like Newton
    - in-between like L-BFGS
    - tools like momentum, rescaling
  - choice of covariance form or kernel form

Schematic

$\mathbb{R}^T \quad \mathbb{R}^{T \times d} \quad \mathbb{R}^d$



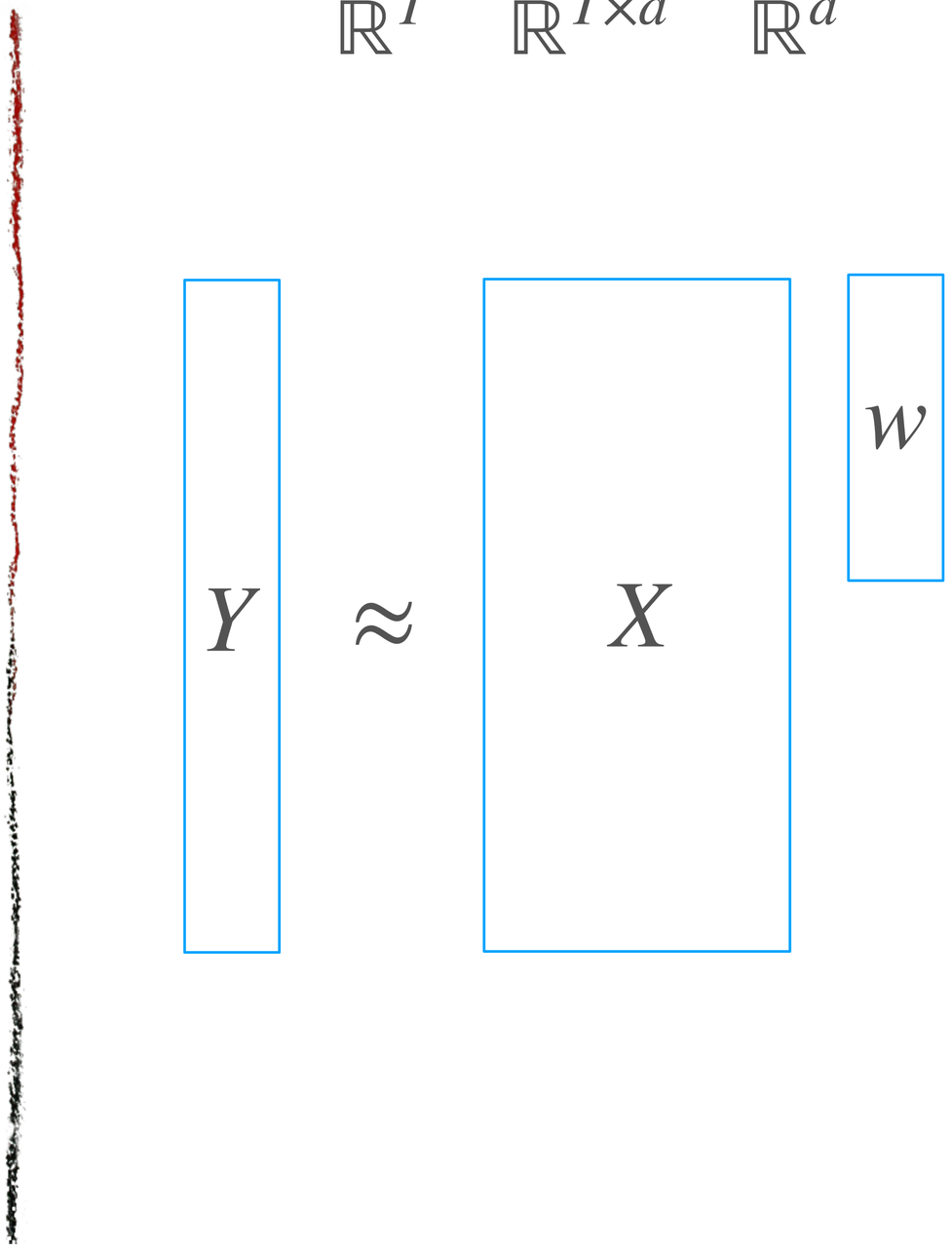
features X labels y

Family History	Resting Blood Pressure	Cholesterol	Cost (k$)
Yes	Low	Normal	0.1
No	Medium	Normal	0.2
No	Low	Abnormal	10.3
Yes	Medium	Normal	8.7
Yes	High	Abnormal	13.1
No	Low	Normal	0.1
No	Medium	Normal	0.4
Yes	Medium	Abnormal	9.6

- data matrix X, label vector Y; rows are $X^{(t)}$, $Y^{(t)}$

Schematic

$\mathbb{R}^T \quad \mathbb{R}^{T \times d} \quad \mathbb{R}^d$



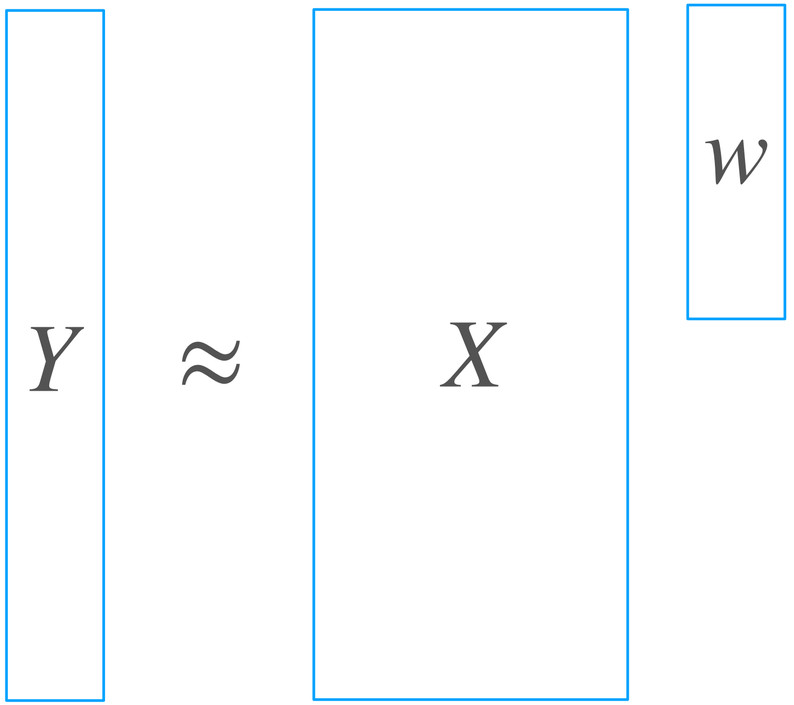
features X labels y

	Family History	Resting Blood Pressure	Cholesterol	Cost (k$)
1	Yes	Low	Normal	0.1
1	No	Medium	Normal	0.2
1	No	Low	Abnormal	10.3
1	Yes	Medium	Normal	8.7
1	Yes	High	Abnormal	13.1
1	No	Low	Normal	0.1
1	No	Medium	Normal	0.4
1	Yes	Medium	Abnormal	9.6

- data matrix X, label vector Y; rows are $X^{(t)}$, $Y^{(t)}$

Schematic

$\mathbb{R}^T \quad \mathbb{R}^{T \times d} \quad \mathbb{R}^d$



features X labels y

	Family History	Resting Blood Pressure	Cholesterol	Cost (k$)
1	Yes	Low	Normal	0.1
1	No	Medium	Normal	0.2
1	No	Low	Abnormal	10.3
1	Yes	Medium	Normal	8.7
1	Yes	High	Abnormal	13.1
1	No	Low	Normal	0.1
1	No	Medium	Normal	0.4
1	Yes	Medium	Abnormal	9.6

$$L(w) = \|Y - Xw\|_F^2 \quad [+ R(w)]$$

- data matrix X, label vector Y; rows are $X^{(t)}$, $Y^{(t)}$

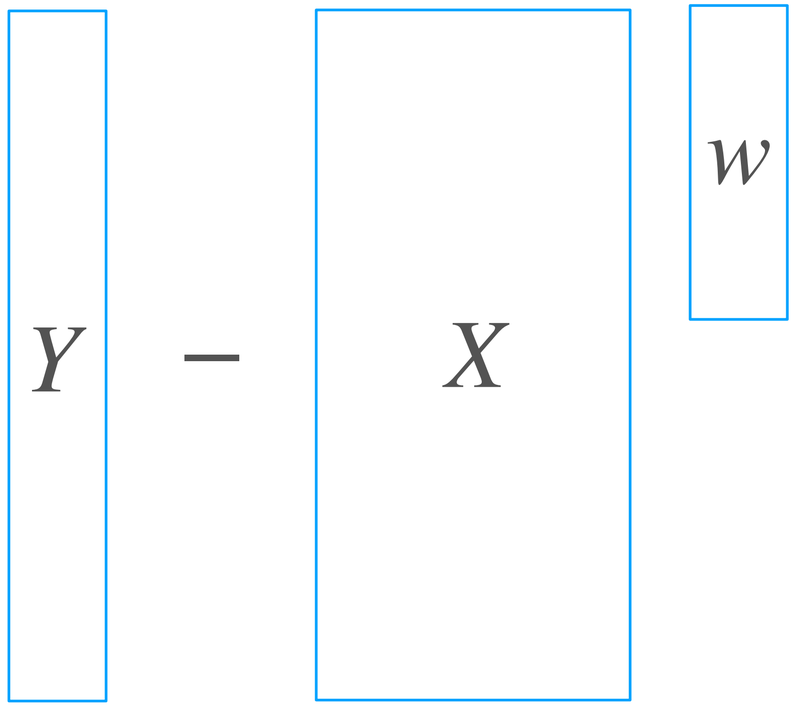
MOO!

- Previous slides illustrate *model, objective, optimizer* for linear regression
- Often three good questions to ask when learning a new ML method — maybe not complete, but a good place to start

MOO examples

- Linear regression
  - model = linear functions of features
  - objective = sum of squared errors (+ regularizer)
  - optimizer = exact or SGD
- ID3
  - model = decision trees
  - objective = mutual information
  - optimizer = greedy top-down induction

Gradients



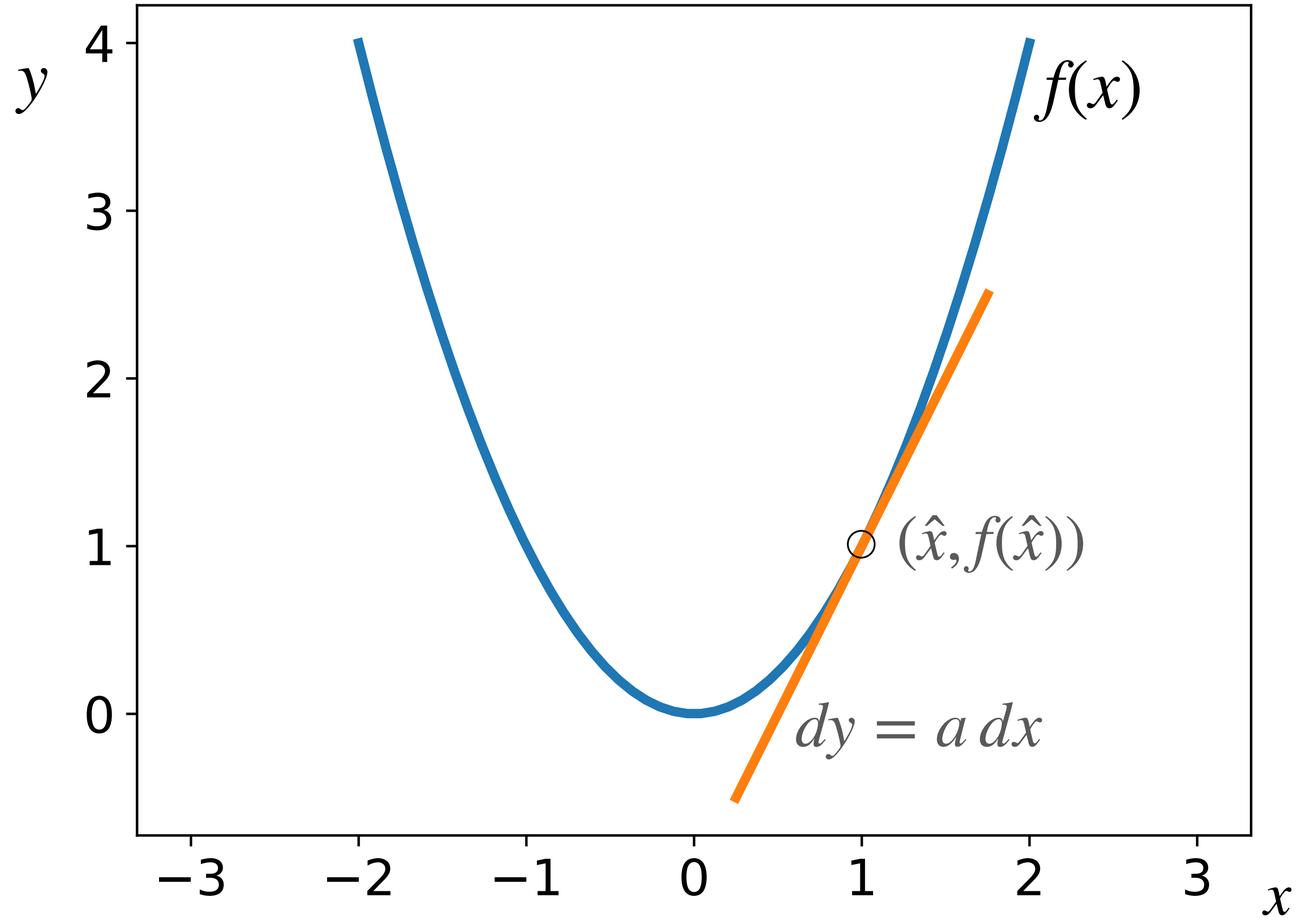
- For any of the above, need gradients
  - $L(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{2}\sum_{t=1}^{T}(Y^{(t)} - w \cdot X^{(t)})^2$

    $= \frac{\lambda}{2}\|w\|^2 + \frac{1}{2}\|Y - Xw\|^2$
  - $\nabla L(w) =$

Derivatives everywhere

- The way we derive many ML methods:
  - start from objective to minimize (here, sum of squared errors plus optional regularizer)
  - take a few derivatives ($0 = d\,\text{objective} / dw$)
  - find optimum (e.g., by setting to 0 and solving, or SGD)
  - done!
- Will continue to be a common pattern
- Review of intuition and tools for high-d derivatives, so we can do it fast

Derivatives notation



$m = n = 1$

$dy = y - f(\hat{x})$

$dx = x - \hat{x}$

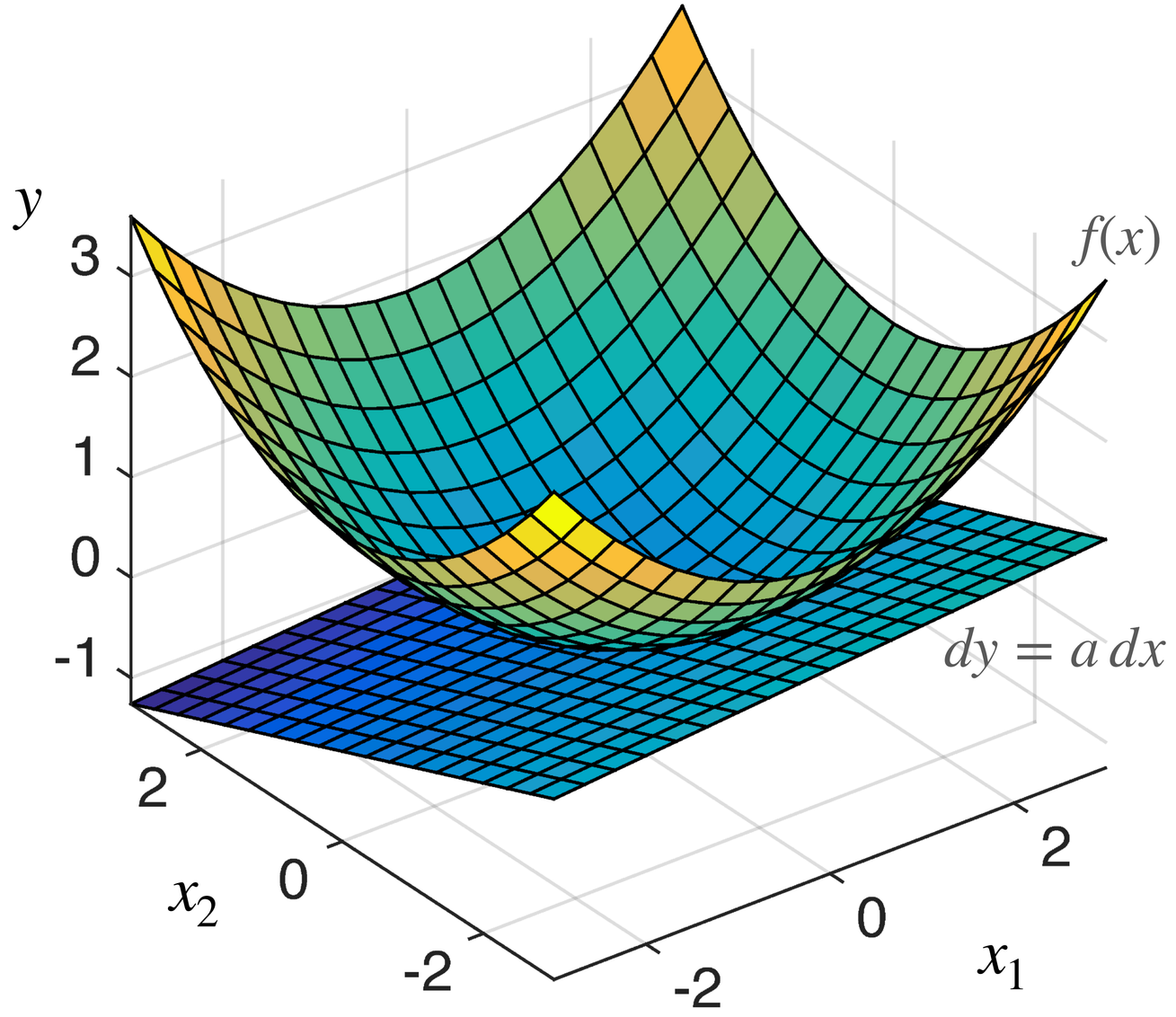
dx, dy are called ***differentials***

- Function $y = f(x)$
  - $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $f \in \mathbb{R}^n \to \mathbb{R}^m$
- Derivative is a *linear fn* that *locally* approximates f near $\hat{x}$
  - $dy = a\,dx$, where $a \in \mathbb{R}$ is the derivative

Compare older vs. newer notation

- Older version:
  - $\frac{dy}{dx}$ is atomic: derivative of y wrt x
  - dx and dy are meaningless on their own
  - focused on infinitesimals: change in x or $y \to 0$
- Newer version:
  - dx and dy have separate meaning: terms in a linear function (the Taylor expansion)
  - dx and dy can be any size: linear function is well defined even for large changes in x or y (though accuracy of Taylor approximation can be bad if dx, dy are big)
- Heuristic to convert: “divide through” or “multiply through” by dx (only makes sense in scalar case, but often succeeds in converting notation anyway)

Higher dimensions

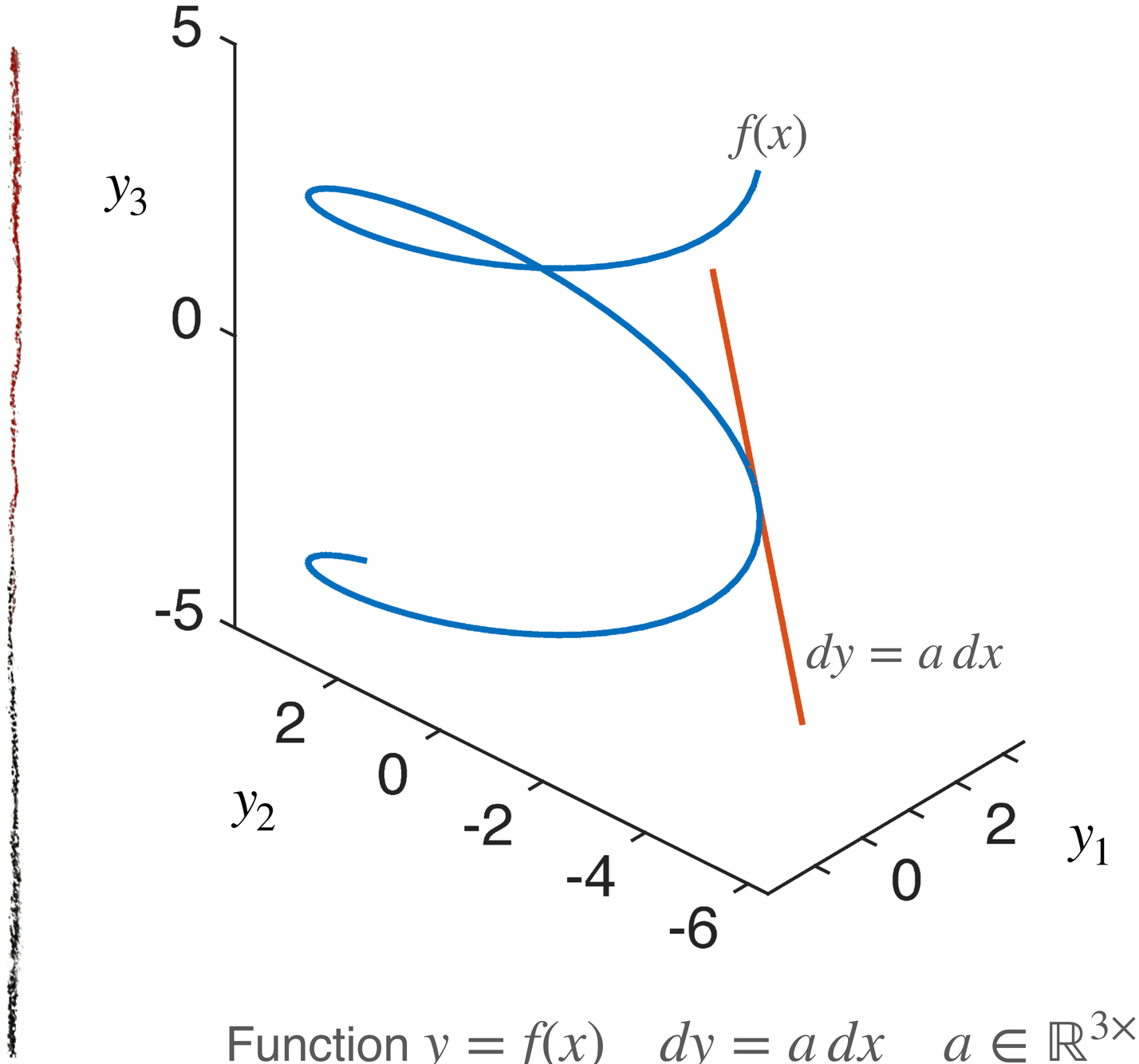


$\mathbb{R}^n \to \mathbb{R}^m$

$m = 1, n = 2$

Function $y = f(x)$ $dy = a\,dx$ $a \in \mathbb{R}^{1\times 2}$

Higher dimensions



$\mathbb{R}^n \to \mathbb{R}^m$

$m = 3, n = 1$

Function $y = f(x)$ $dy = a\,dx$ $a \in \mathbb{R}^{3\times 1}$

Notation for linear functions

output shape (derivative of)

	input shape (with respect to) Scalar	Vector	Matrix
Scalar	$ds = a\,dt$	$ds = u^T dv$	$ds = \mathrm{tr}(M^T dN)$
Vector	$du = v\,ds$	$du = M\,dv$	$\times$
Matrix	$dM = N\,ds$	$\times$	$\times$

- a, s, t: scalars u, v: column vectors M, N: matrices
- note: $ds = a\,dt$ is a synonym for $a = \frac{ds}{dt}$, etc.
- $\times$ means no common notation; see torch.einsum

Notation for linear functions

output shape (derivative of) \ input shape (with respect to)	Scalar	Vector	Matrix
Scalar	$ds = a\,dt$	gradient	$ds = \mathrm{tr}(M^T dN)$
Vector	velocity	Jacobian	×
Matrix	$dM = N\,ds$	×	×

- a, s, t: scalars u, v: column vectors M, N: matrices
- note: $ds = a\,dt$ is a synonym for $a = \frac{ds}{dt}$, etc.
- × means no common notation; see torch.einsum

Inner product

- With scalar output s, for any shape of input X, can write $ds = \langle A, dX \rangle$ where A, X, dX all have same shape
  - $\langle \cdot, \cdot \rangle$ is ***inner product***, in our case $\sum_{\text{index lists } I} A_I \, dX_I$
  - e.g., if A, dX are $m \times n$ matrices, then

$$\langle A, dX \rangle = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} \, dX_{ij}$$

- Expressions on previous slide are equivalent to above
  - most complex is $\text{tr}(M^\top dN)$
  - tr is sum of diagonal elements of a square matrix

$$\left[M^\top dN\right]_{jk} =$$

  - sum of diagonal elements means set $j = k$, sum over j:

Shape conventions

- Derivative = coefficient of ds, dt, dv, dN on RHS
- Ambiguity: can write $\langle M, dN \rangle$ or $\text{tr}(M^\top dN)$
  - is the derivative M or $M^\top$?
- Convention: x, y, and derivative $\frac{dy}{dx}$ are ***tensors*** (multidimensional arrays of numbers)
- In derivative $\frac{dy}{dx}$:
  - the dimensions of y come first, then x
  - in same order as dimensions of y and x
- For example: if y is 3×2 and x is $4 \times 6 \times 5$ then $\frac{dy}{dx}$ is $3 \times 2 \times 4 \times 6 \times 5$

Special cases

- If y is a scalar, then $\frac{dy}{dx}$ is the same shape as x
  - derivative of a scalar wrt a 5×3 matrix is a 5×3 matrix
- If x and y are vectors, $\frac{dy}{dx}$ (the Jacobian) is the same shape as a linear function from x to y
  - derivative of a 3-vector wrt a 6-vector is a 3×6 matrix

Useful identities

compatible dimensions:

- *2nd dimension of A = 1st dimension of B*
- *2nd dimension of B = 1st dimension of C*
- *2nd dimension of C = 1st dimension of A*

- scalar = scalar$^\top$ = tr(scalar)
- Trace rotation
  - $\mathrm{tr}(ABC) = \mathrm{tr}(CAB)$ when A, B, C have compatible dimensions
- Transpose-Hadamard
  - $A \circ B =$ Hadamard (componentwise) product,

    $$[A \circ B]_{ij} = A_{ij} B_{ij}$$
  - $A^\top (B \circ C) = (A \circ B)^\top C$

Tools for derivatives: linearity, chain, product

- Linearity:
  - $d(ax + b) = a\,dx + b, \quad d(AX + B) = A\,dX + B, \ldots$
- Chain rule for $f(g(x))$
  - old notation: $\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$ ("cancel the dg")
  - new: if $df = A\,dg$, $dg = B\,dx$ then $df =$
- Product rule for $f(x)\,g(x)$
  - $d(fg) = df\,g + f\,dg$
  - works for any kind of product (matrix product, vector cross product, elementwise (Hadamard) product, vector convolution, Khatri-Rao product …)
  - unlike scalar case, order can matter (products are not necessarily commutative)

Partial vs. total derivatives

- In a function of several variables, partial derivative imagines changing one of them, holding others fixed
  - e.g., $\frac{\partial}{\partial x}(x+y)^2 =$ holding y fixed
- New notation: any variable that appears with d in RHS is changing, others are fixed
  - e.g., $d(x+y+z)^2 =$ holding z fixed

Example: chain rule with multiple variables

- Ex: $y = (u + v)^3, \;\; u = \cos x, \;\; v = x^2$

- Sometimes called ***law of total derivatives***:

$$df(u(x), v(x), w(x)) = f^{(1)}(u, v, w)\, u'(x)\, dx + f^{(2)}(u, v, w)\, v'(x)\, dx + f^{(3)}(u, v, w)\, w'(x)\, dx$$

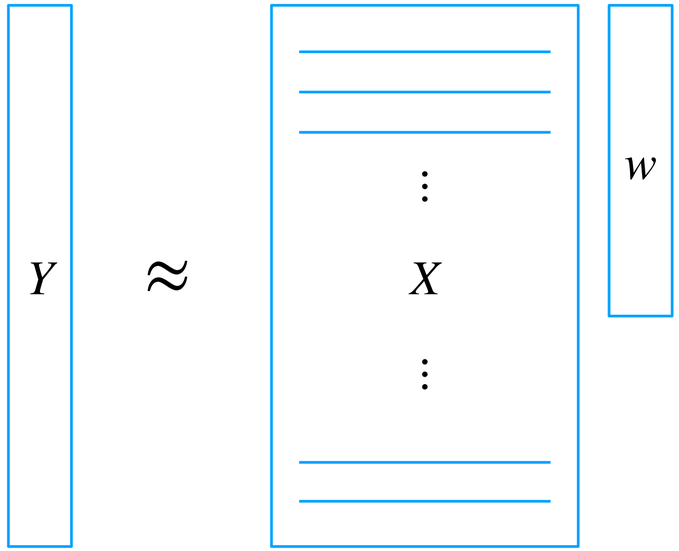
Gradient practice: Linear regression in one slide

$$L = \|Y - Xw\|^2 = (Y - Xw)^\top (Y - Xw)$$

$$L = Y^\top Y - 2Y^\top Xw + w^\top X^\top Xw$$

Exact solution for ridge regression

$X \in \mathbb{R}^{T \times d}$
$Y \in \mathbb{R}^{T}$
$w \in \mathbb{R}^{d}$



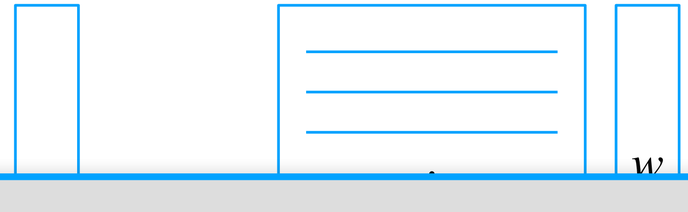
- Predictions are $\hat{Y} = Xw$
- Objective is $\frac{\lambda}{2}\|w\|^2 + \frac{1}{2}\|Y - Xw\|^2$
- Gradient is $\lambda w + X^\top(Xw - Y)$
- Exact solution is $(\lambda I + X^\top X)w = X^\top Y$
  - implemented in `torch.linalg.lstsq`

Exact solution for ridge regression

$X \in \mathbb{R}^{T \times d}$

$Y \in \mathbb{R}^{T}$

$w \in \mathbb{R}^{d}$



- Pred
- Obje
- Grad
- Exac
  - implemented in `torch.linalg.lstsq`

Why use torch.linalg.lstsq? Handles (many) special cases: over- or under-determined, rank-deficient, poor conditioning

It would be several days of lectures to describe algorithms and best practices for all of these

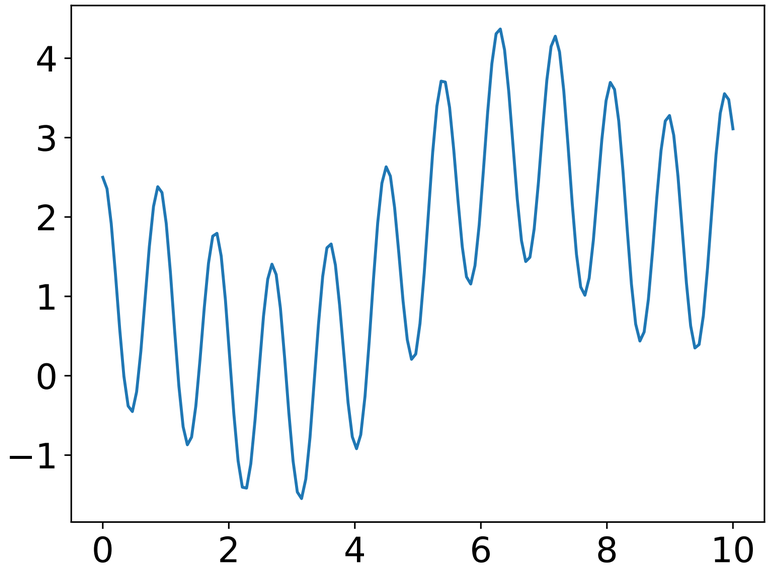
Please remember: don't write inv() unless you are sure none of the special cases will bite you

High-dimensional regression

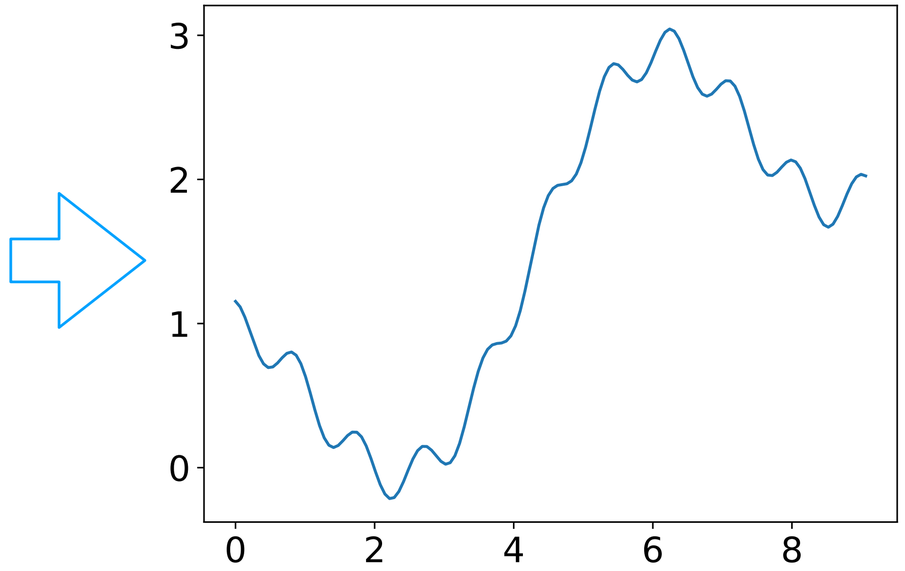
- In ML we often need to learn predictors for high-dimensional inputs and outputs
  - means we need to work with high-dimensional tensors (algebra, differentiation)
- We'll go through an example using just linear layers to illustrate: ***signal processing*** or prediction from audio, images, video, …

Signal processing

low-pass filter



air pressure v. time



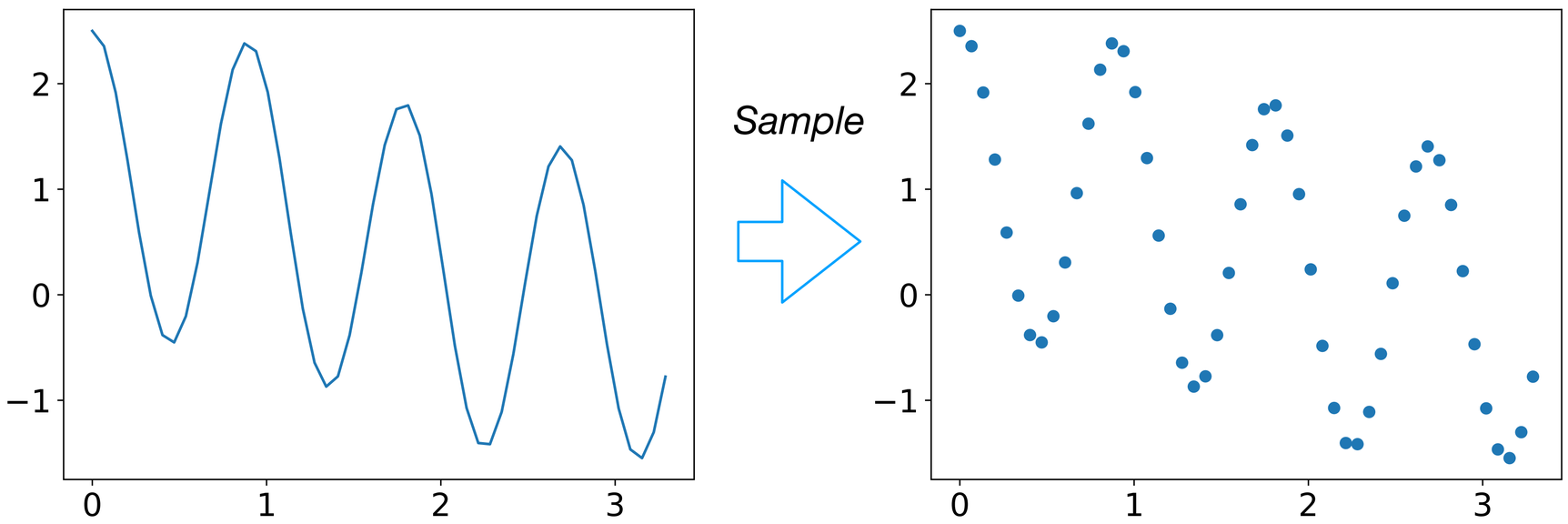
air pressure v. time

image credit: ChatGPT

Fewer knobs

image credit: ChatGPT

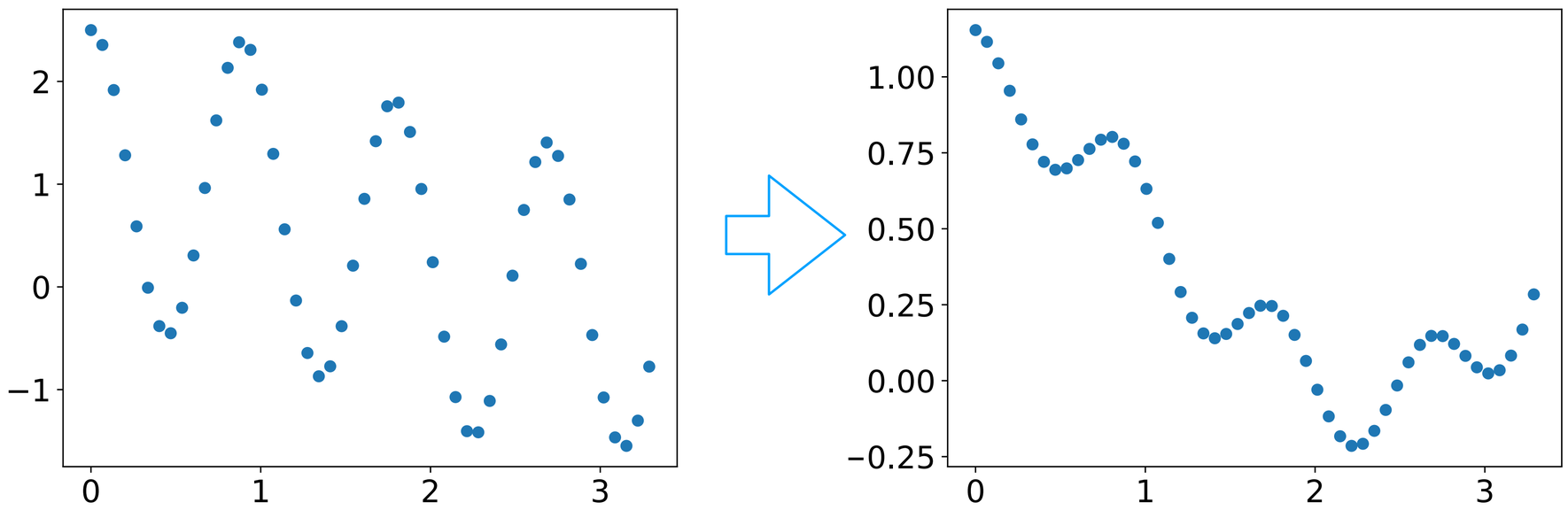
Sampled signal



- Continuous-time signal $\xrightarrow{\textit{sampling}}$ vector in $\mathbb{R}^{50}$
  - instead of $f(0.134) = 1.92$, we have $x_3 = 1.92$
- Think of f as an infinite-d vector, x as its finite-d version

Filter = linear function

low-pass filter



- Each coordinate of filtered signal is a linear function of input signal: $y = Wx$
  - $x, y \in \mathbb{R}^{50}$
  - $W \in \mathbb{R}^{50 \times 50}$

Time invariance

- Audio filters are special matrices: behavior of filter stays fixed over time
  - i.e., the linear function that gives us y_{23} as a function of x is "the same as" the linear function that gives us y_{42}
  - "same" means the coefficients are the same but shifted in time

$$\begin{pmatrix} 0.33 & 0.33 & 0.33 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0.33 & 0.33 & 0.33 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0.33 & 0.33 & 0.33 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

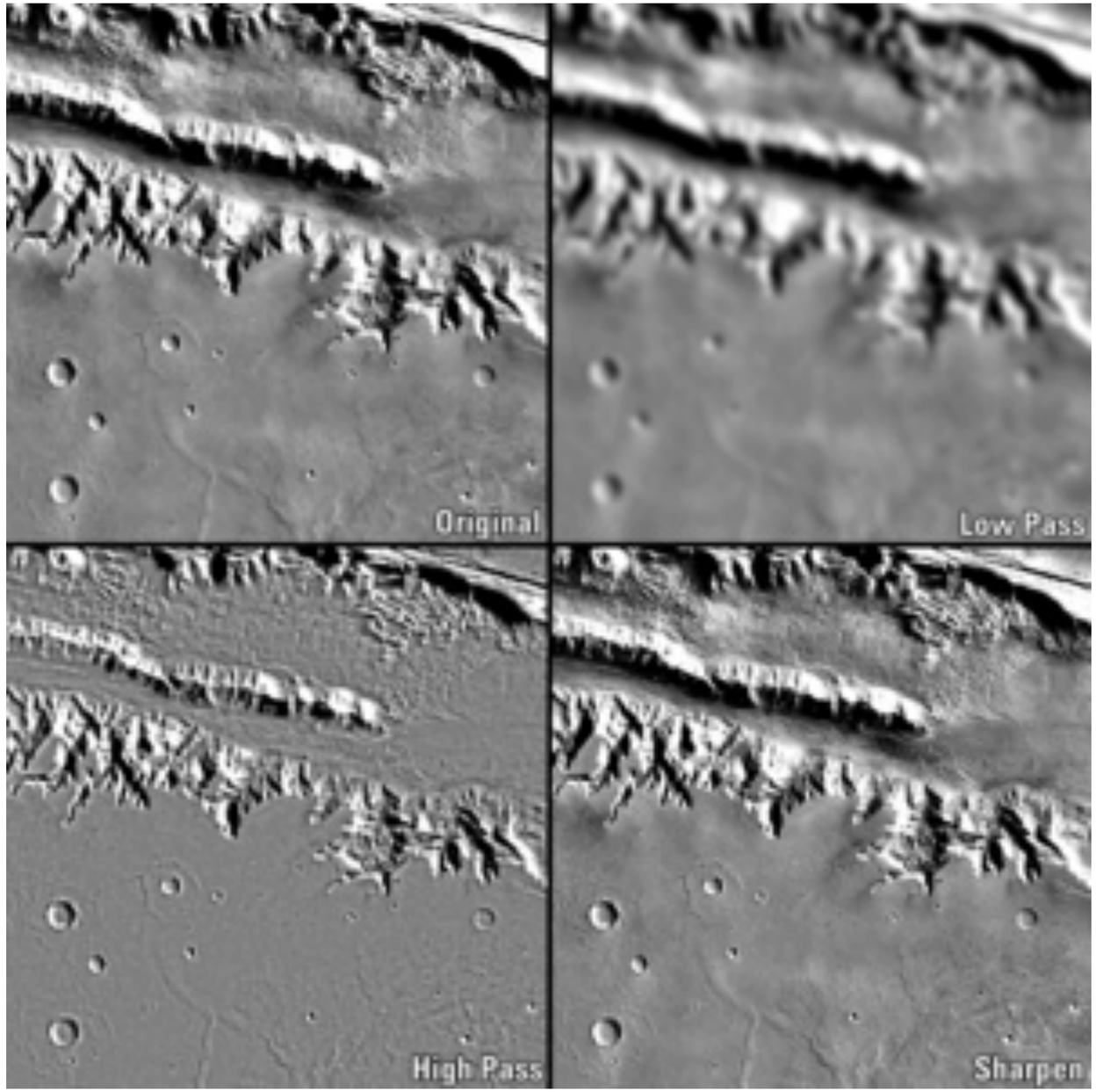
- This operation is called *convolution*, and we can learn its weights from examples

i.e., each output (say y_{17}) is a target (label); corresponding features are a window of nearby inputs (say $(x_{15}, x_{16}, \dots, x_{19})$)

n-d signals

- Beyond having graphs (1d signals) as inputs, common to have 2d, 3d, or higher
  - e.g., temperature vs. location across US (2d)
  - e.g., picture of fluffy kitten downloaded from internet (2d if grayscale, 3d if color)
  - e.g., gas pressure throughout a reaction vessel (3d)
  - e.g., value function for a control problem where state is the position of a rigid body (6d)

2-d convolution



credit: USGS

Signals as tensors

- An n-dimensional signal is represented as a tensor with n indices
  - e.g., grayscale image: 2 indices (pixel i, j)
  - e.g., color image: 3 indices (last is channel: R, G, or B)
  - e.g., pressure in a volume: 3 indices (voxel i, j, k)
- Each index also called a ***mode***
  - helps with overloading of the word “dimension”
- List of index ranges: the ***shape*** of the tensor
  - e.g., RGB image at VGA resolution: (640, 480, 3)
  - shape (3, 4, 2, 5) tensor has 120 total elements
  - first index can be 0, 1, 2; second 0, 1, 2, 3; …

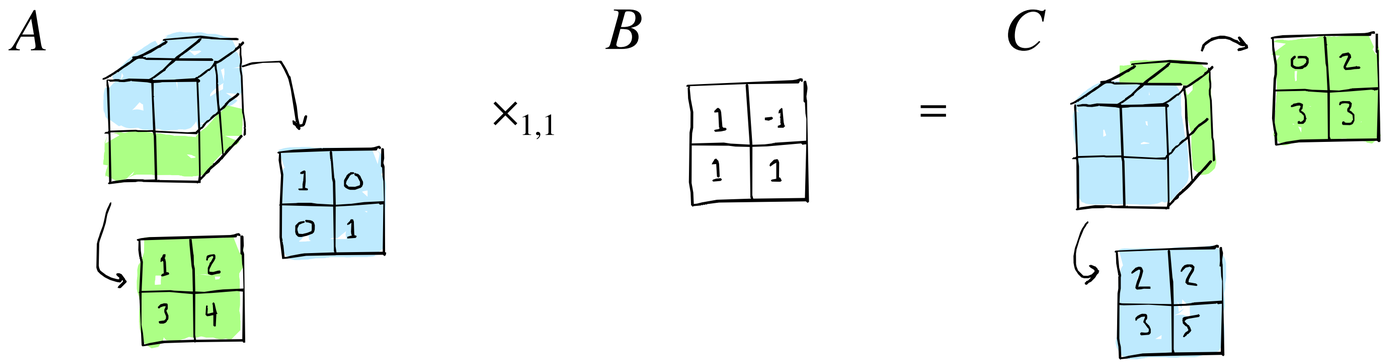
Tensor operations

- Componentwise $+$, $\times$, $\cos$, …
  - with *broadcasting:*

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 & 3 & 4 \\ 6 & 7 & 8 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 9 \\ 4 & 10 & 18 \end{pmatrix}$$

Tensor operations

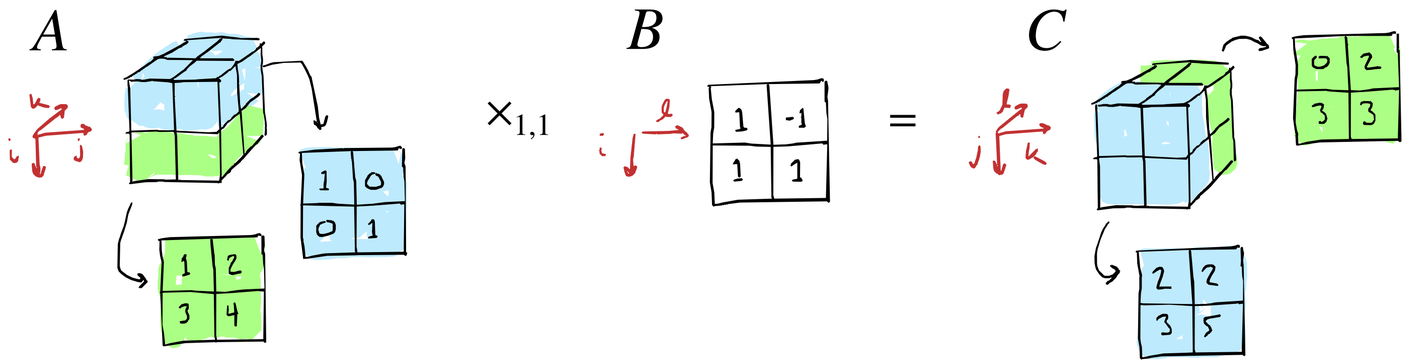


- Contraction: e.g., $A \times_{3,2} B$ or $A \times_{1,1} B$
  - like matrix multiplication (precise definition below), but using mode 3 of A and mode 2 of B — regular matrix multiplication is $\times_{2,1}$
  - modes of $A \times_{3,2} B$: modes of A (except 3) then modes of B (except 2)
  - by convention, all of A's modes first (in order) then B's
- Represents a linear function of B with parameters A
  - or a linear function of A with parameters B

Contraction is a special case of Einstein summation

- Convention: a_{ij} means a matrix whose i, j element is a_{ij}
  - range of i, j can be left implicit or specified separately: e.g., $i \in \{1 \ldots 5\}, j \in \{1 \ldots 7\}$
- Similarly, a_{ijkl} means a 4-mode tensor
- So does $a_{ijk} b_{jkl}$ — the i, j, k, l element is $a_{ijk}\, b_{jkl}$
- To represent contraction, we raise one instance of index
  - e.g., $a_{ijk} b^k_{jl}$ is a 3-mode tensor
  - element i, j, l is $\sum_k a_{ijk} b_{jkl}$
- In general (Einstein summation), any index that appears both raised and lowered gets summed out
  - $a^k_{ij} b^l_{jk} c_{lm}$ has indices i, j, m, elements $\sum_k \sum_l a_{ijk} b_{jkl} c_{lm}$

Tensor contraction as Einstein summation



- Can write the tensor contraction from above as an Einstein summation: $[A_{ijk}B^i_l] = [C_{jkl}]$
- Notation difference: instead of numbering the modes, we name them by their indices

Derivative as contraction/ einsum

- Revisiting convention from earlier: if we have two tensors y_{ij} and x_{klm} then derivative $\frac{dy}{dx}$ is a 5-mode tensor representing a linear function from dx to dy
- We can write this linear function as a tensor contraction or Einstein summation:

$$dy_{ij} = A_{ij}^{klm}\, dx_{klm}$$

  - where A is the 5-mode tensor of coefficients
- Indices are $ijklm$, with last 3 indices (the ones corresponding to input x) raised
- Ex: Jacobian is $dy_i = A_i^j\, dx_j$

Tensor example: factored linear network for image regression

- Input: RGB image patch x, size $32 \times 32 \times 3$
  - comes from satellite image of Earth
- Label y: $\ln(\text{population})$ for imaged area $\in \mathbb{R}$
- Model
  - filter each channel with learned filter f, size 11×11, indices –5…5
  - collapse channels by weighted sum, learned $\mu \in \mathbb{R}^3$
  - linear model on result, weights w, size 32×32
- Output $\hat{y}$:

Tensor example: factored linear network for image regression

- Input: RGB image patch x, size $32 \times 32 \times 3$
  - comes from satellite image of Earth
- Label y: $\ln(\text{population})$ for imaged area $\in \mathbb{R}$
- Model
  - filter each channel with learned filter f, size 11×11, indices –5…5
  - collapse channels by weighted sum, learned $\mu \in \mathbb{R}^3$
  - linear model on result, weights w, size 32×32
- Output $\hat{y}$:

$$\hat{y} = x_{i+k,j+l,c} f^{kl} \mu^c w^{ij}$$

convention: if $i \leq 0$ or $i > 32$ then $x_{ijc} = 0$

(similarly for j)

Factored linear network

$$\hat{y} = x_{i+k,j+\ell,c} f^{kl} \mu^{c} w^{ij}$$

- Prediction $\hat{y}$ is a *linear* function of x
  - with coefficients $f^{kl} \mu^{c} w^{ij}$ (a 5-mode tensor)
  - inductive bias: three learnable factors f, μ, w
  - like a matrix factorization, *way* fewer parameters than if we used a general tensor of the same shape
- Could learn by SGD or by alternately solving for f, μ, w
  - to differentiate: note contraction is a product (in the sense of product rule)
  - and any einsum of AB is linear in A when B is fixed (and vice versa)

Working with linear models

- Slides so far are enough to fit linear models
- Remaining slides: some intuition about how to design and use linear models

Pre-processing

- Linear models are extremely useful, but can be sensitive to how the features are expressed
  - also true of nonlinear models
- Extremely common to modify features before training our model — called *pre-processing*
- Large toolbox of possible ways to pre-process
  - each tool is designed to fix a potential problem
  - use any or all of them, as the situation calls for
  - but don’t use them unless they’re needed
- Bundle together into a *feature transform*

Categorical features

- Categorical = list of discrete possible values, no obvious relationship among values
  - e.g., weather = {rainy, sunny, windy}
- Tool: *one-hot* encoding
  - create one numerical feature for each possible value
  - set one feature to 1, rest to 0 (metaphor: like making one wire “hot”, i.e., with an applied voltage)
  - rainy $\rightarrow (1,0,0)$ sunny $\rightarrow (0,1,0)$ windy $\rightarrow (0,0,1)$
- Variant: pick one value as default
  - if k values, create $k-1$ features
  - non-default values get a one-hot encoding
  - default $\rightarrow (0,0,\ldots,0)$

Ordered features

- Sometimes discrete features have a built-in order
  - {elementary school, middle school, high school, undergrad, MS, PhD}
- Could use one-hot, but it would erase the order
- Instead, *thermometer code*:
  - use k features for k values, but turn on the first j of them to encode the jth value
  - e.g., high school $\rightarrow (1,1,1,0,0,0)$
  - analogy to the mercury rising in an old-fashioned thermometer
- Can also omit a feature for the first value (assign all zeros) and have $k-1$ features

Linearly dependent features

- Sometimes a feature will be accidentally duplicated (e.g., two sources provided it w/ different names, or we added a constant feature when one was there already)
- More subtly, a feature might be the sum or difference of two others — e.g., see one-hot encodings
- This is no problem w/ regularization
  - L_2: if k copies of a feature, we'll assign $\frac{1}{k}$ weight to each
  - L_1: any way to split weight is optimal, often prefer sparse
  - other: will usually do something sane
- But without regularization, optimum might be undefined!
- Fix by finding features to drop until no more redundancy
  - diagnose by testing rank of feature matrix X

Feature range

- Features can have very different typical range of values
  - e.g., a yes/no question vs. the population of a city
  - e.g., units: measure distance in mm vs. Mpc
- If no regularization, linear models don't care
  - multiply feature by 73, regression will divide weight by 73
- But regularization changes this (why?)
- Two common fixes: location and scale

Location

- Remove location differences by subtracting feature means
  - feature X_i^t becomes $X_i^t - \hat{\mu}_i$ (called ***centering***)
  - here $\hat{\mu}_i$ is an estimate of the mean (or could use other statistics such as median)
- Note: must compute $\hat{\mu}_i$ from *training data only*
  - don't use validation or test data: can't in deployed version, so doing it during development gives wrong answers
  - $\hat{\mu}_i$ becomes part of our model
- Alternate approach: omit regularization on constant coefficient b in $w \cdot X^t + b$
  - then b is free to compensate for differences in mean

Scale

- Remove scale differences by dividing by feature scale
  - feature X_i^t becomes $\frac{1}{\sigma_i} X_i^t$ where σ_i is estimate of scale
  - e.g., the standard deviation (square root of variance)
  - or the max absolute value, interquartile range, …
- As above, must compute σ_i on *training data only*, and σ_i becomes part of our model
- If we're transforming location and scale, do location first
  - scale is more meaningful on centered data

Outliers

- Sometimes a single example will completely dominate the size of a feature
  - e.g., weight: most examples are mice, one is elephant
  - or typo: forgot separator, so “030, 001” → “030001”
- Possible fixes:
  - filter out (delete) the outlying examples
  - clip the feature (project onto an estimate of the “reasonable” range, e.g., elephant → heaviest mouse)
  - CDF transform (rank transform): replace feature value by its rank in the training set, then scale to $[0,1]$
  - whichever we use, has to depend only on training set, becomes part of our model

And more…

- Feed examples through a foundation model
- Text → word part embeddings
- Random projection
- Vector quantization
- Positional encoding
- …

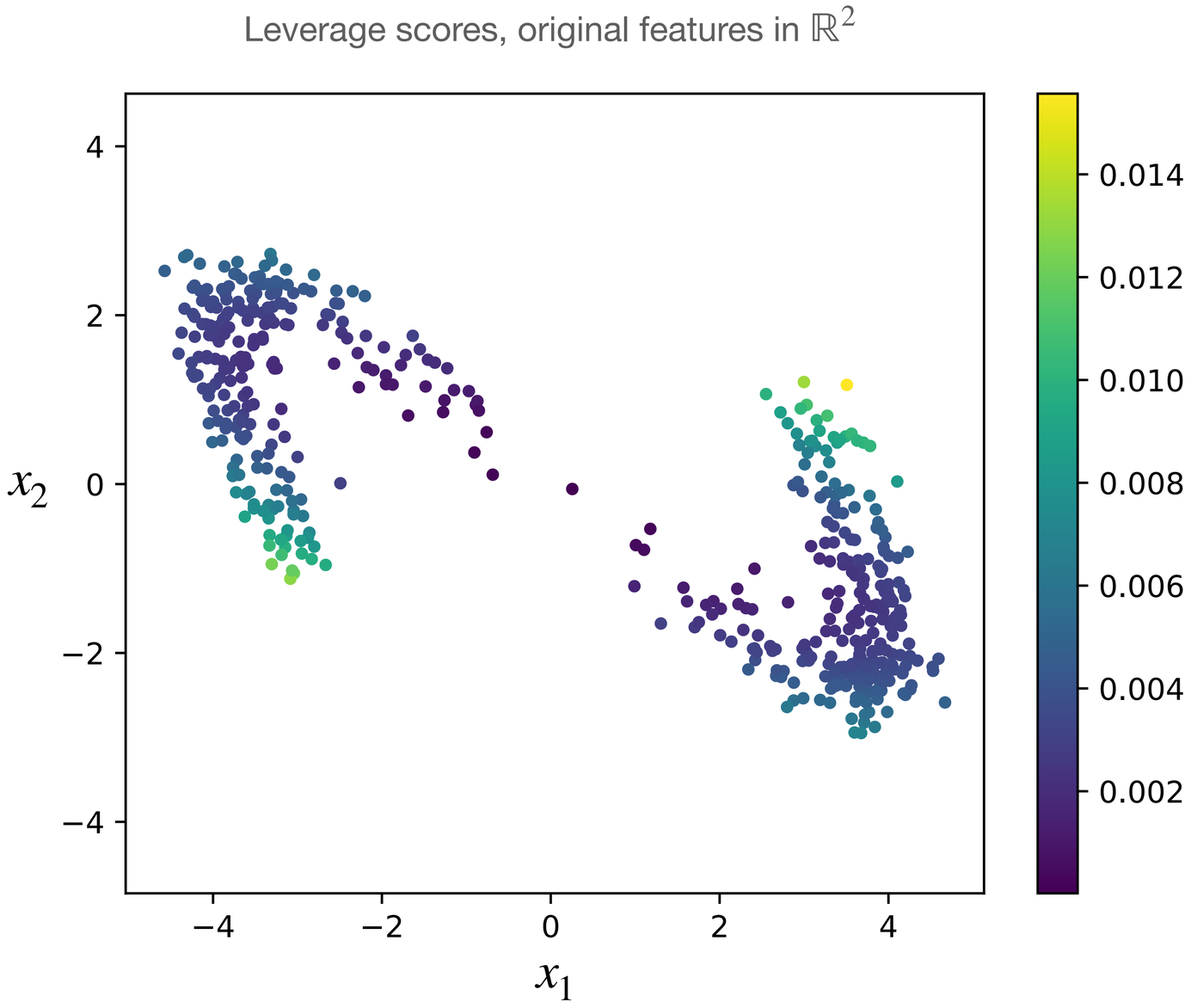
To process or not to process?

- Tempting to apply every possible preprocessing trick just to be sure
- Unfortunately it's not this simple
  - can lead to overfitting (parameters: location, scale, CDF)
  - can erase useful information: maybe the feature that didn't vary much is actually not very informative
  - interpretability
- Probably best to be conservative:
  - look for problems that influence the fit: outliers, significant variation in scale, …
  - fix at the source if possible
  - pre-process when necessary

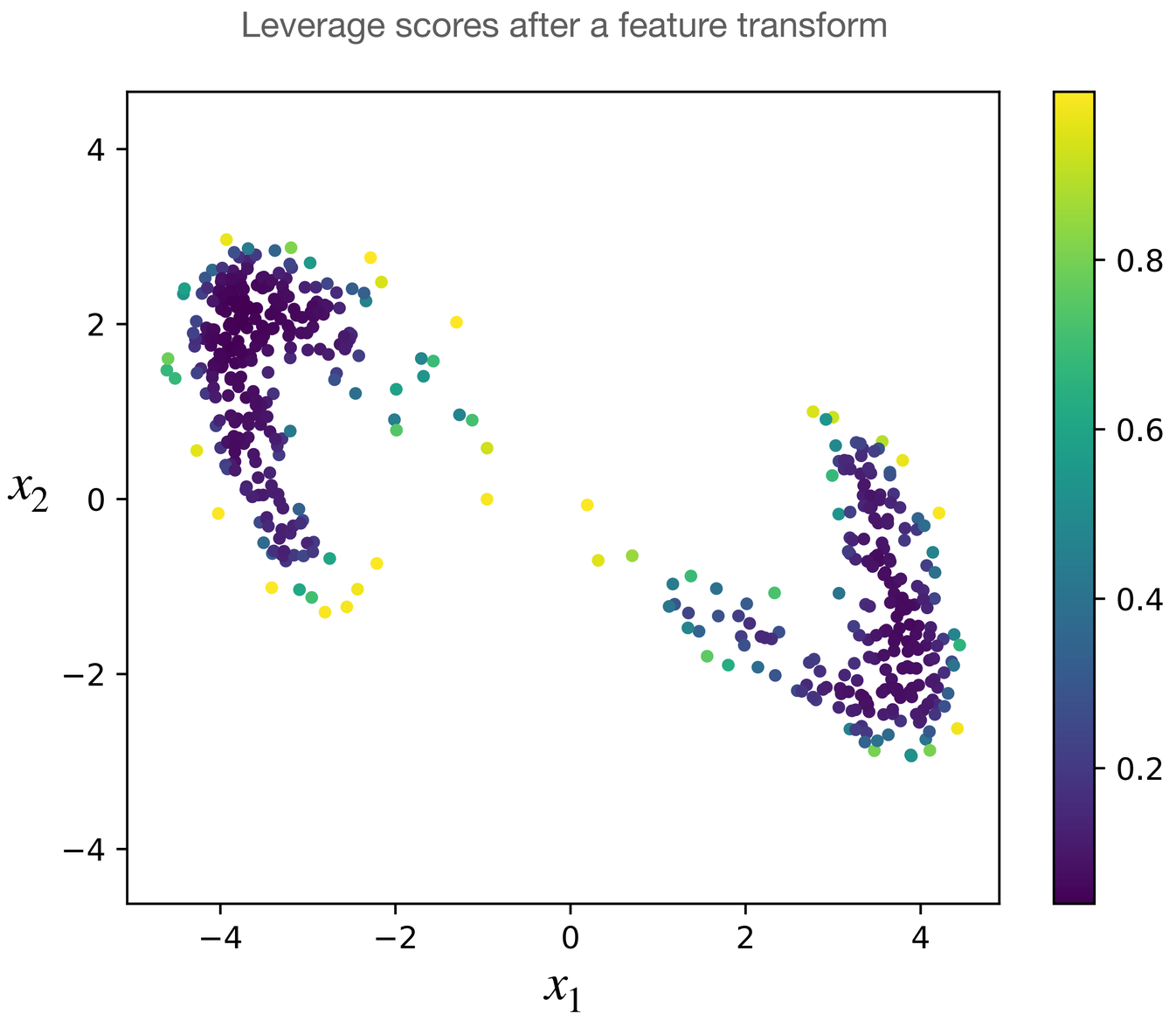
Leverage

- Some data points influence the fit more than others
  - big derivative of predictions $\hat{Y}$ wrt training labels Y, called *leverage*
- For linear regression, measure with *leverage scores*:
  - tth leverage score = $\frac{d\hat{Y}^t}{dY^t}$
  - Large leverage = contributes a lot to the fit
    (so $\frac{d\hat{Y}^s}{dY^t}$ tends to be large, even if not measured directly)
  - All leverage scores are in $[0,1]$
  - Sum of leverages = feature dimension k

Leverage example



Leverage example



Leverage example