

# Data 188: Introduction to Deep Learning

## ML Refresher / Softmax Regression

Speaker: Eric Kim  
Lecture 02 (Week 01)  
2026-01-22, Spring 2026. UC Berkeley.

# Announcements

- (Week 1) No discussion, no office hours
- (Week 2) Discussion and office hours starts!
- HW0 is released! Start early!
  - See Ed post for more details: ["HW0 released!"](#)
  - ~2 weeks to complete it
  - Tip: after today's lecture, you should be able to do HW0
    - Q5: we'll cover two-layer NN's in Lecture 03, but you should be able to still do Q5 without it IMO

# Enrollment, Waitlist

- If you're on the waitlist, and you're a DS senior that is graduating in SP26/SU26 and needs Data 188 to fulfill the MLDM requirement: please email [ds-advising@berkeley.edu](mailto:ds-advising@berkeley.edu) to get into the course
- For more info, see Ed post: ["Regarding Enrollment, Waitlists"](#)

# Reminder: important links

[Course Website](#): Course syllabus, lecture slides, discussion notes, homework assignments, staff bios, office hours times.

[Edstem](#): where announcements are posted, and where you can ask questions.

[Gradescope](#): Where you will submit homework assignments. Exam scores will also be here.

[Lecture Zoom link](#): Zoom link for lectures, TuTh 3:30pm-5pm.

[Lecture Recordings](#): lecture recordings will be uploaded within 1-2 days after lecture.

- Tip: this is the bCourses "Media Gallery".

# Outline

Basics of machine learning

Example: softmax regression

# Outline

Basics of machine learning

Example: softmax regresssion

# Machine learning as data-driven programming

Suppose you want to write a program that will classify handwritten drawing of digits into their appropriate category: 0,1,...,9

You *could*, think hard about the nature of digits, try to determine the logic of what indicates what kind of digit, and write a program to codify this logic

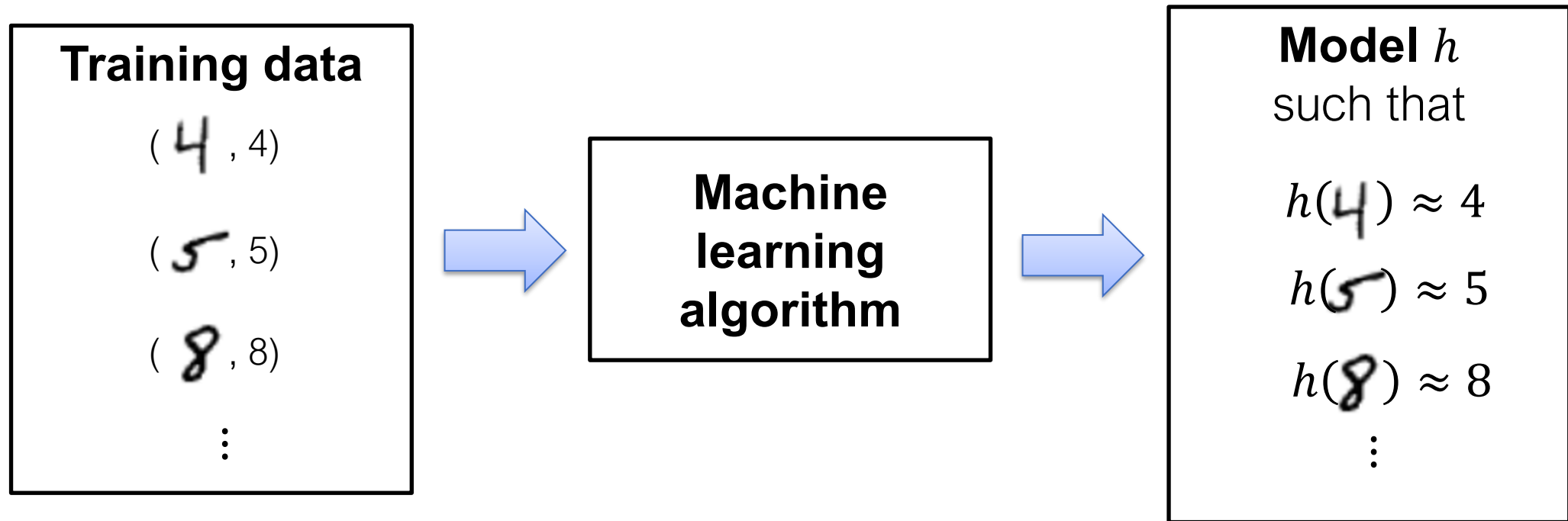
(Despite being a reasonable coder, I don't think I could do this very well)



MNIST Dataset

# Machine learning as data-driven programming

The (supervised) ML approach: collect a *training set* of images with known labels and feed these into a *machine learning algorithm*, which will (if done well), automatically produce a “program” that solves this task





# Three ingredients of a machine learning algorithm

Every machine learning algorithm consists of three different elements:

1. **The hypothesis class:** the “program structure”, parameterized via a set of *parameters*, that describes how we map inputs (e.g., images of digits) to outputs (e.g., class labels, or probabilities of different class labels)
2. **The loss function:** a function that specifies how “well” a given hypothesis (i.e., a choice of parameters) performs on the task of interest
3. **An optimization method:** a procedure for determining a set of parameters that (approximately) minimize the sum of losses over the training set

# Outline

Basics of machine learning

Example: softmax regression

# Multi-class classification setting

Let's consider a *k-class classification setting*, where we have

- Training data:  $x^{(i)} \in \mathbb{R}^n, y^{(i)} \in \{1, \dots, k\}$  for  $i = 1, \dots, m$
- $n$  = dimensionality of the input data
- $k$  = number of different classes / labels
- $m$  = number of points in the training set

Example: classification of 28x28 MNIST digits

- $n = 28 \cdot 28 = 784$  (28x28 pixel images)
- $k = 10$  (digits 0-9)
- $m = 60,000$



# Linear hypothesis function

Our hypothesis function maps inputs  $x \in \mathbb{R}^n$  to  $k$ -dimensional vectors

$$h: \mathbb{R}^n \rightarrow \mathbb{R}^k$$

where  $h_i(x)$  indicates some measure of “belief” in how much likely the label is to be class  $i$  (i.e., “most likely” prediction is coordinate  $i$  with largest  $h_i(x)$ ).

A **linear hypothesis function** uses a *linear* operator (i.e. matrix multiplication) for this transformation

$$h_\theta(x) = \theta^T x$$

for *parameters*  $\theta \in \mathbb{R}^{n \times k}$

# Matrix batch notation


Often more convenient (and this is how you want to code things for efficiency) to write the data and operations in *matrix batch* form

m: batchsize  
n: input dim

$$X \in \mathbb{R}^{m \times n} = \begin{bmatrix} -x^{(1)T} & - \\ \vdots & \\ -x^{(m)T} & - \end{bmatrix}, \quad y \in \{1, \dots, k\}^m = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Then the linear hypothesis applied to this batch can be written as

$$h_{\theta}(X) = \begin{bmatrix} -h_{\theta}(x^{(1)})^T & - \\ \vdots & \\ -h_{\theta}(x^{(m)})^T & - \end{bmatrix} = \begin{bmatrix} -x^{(1)T} \theta & - \\ \vdots & \\ -x^{(m)T} \theta & - \end{bmatrix} = X\theta$$

  
[m, n] [n, k]

# Loss function #1: classification error

A simple loss function to use in classification is the classification error, i.e., whether the classifier makes a mistake or not:

$$\ell_{err}(h(x), y) = \begin{cases} 0 & \text{if } \operatorname{argmax}_i h_i(x) = y \\ 1 & \text{otherwise} \end{cases}$$

We typically use this loss function to assess the *quality* of classifiers

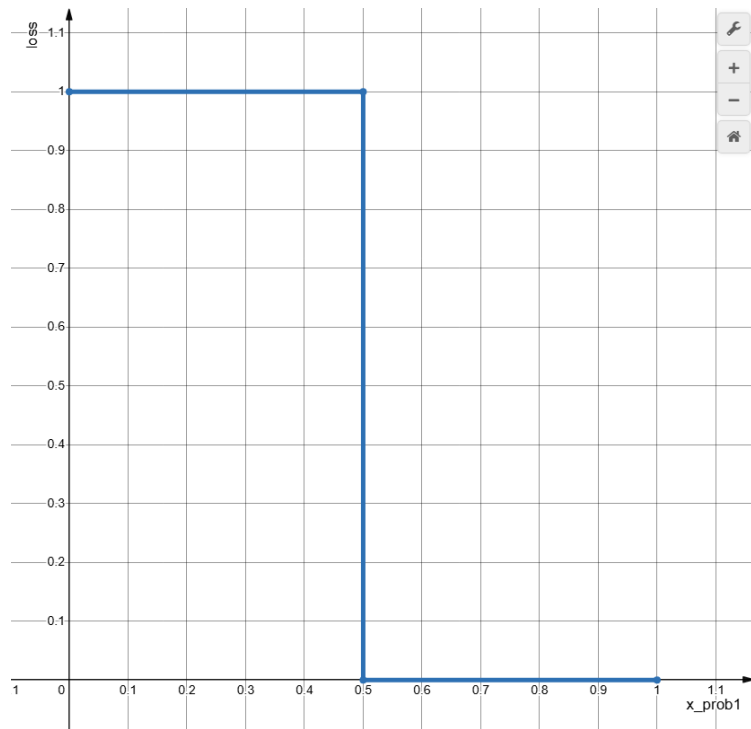
Unfortunately, the error is a bad loss function to use for *optimization*, i.e., selecting the best parameters, because it is not differentiable, and its gradient has little information

Also known as "0/1 loss".

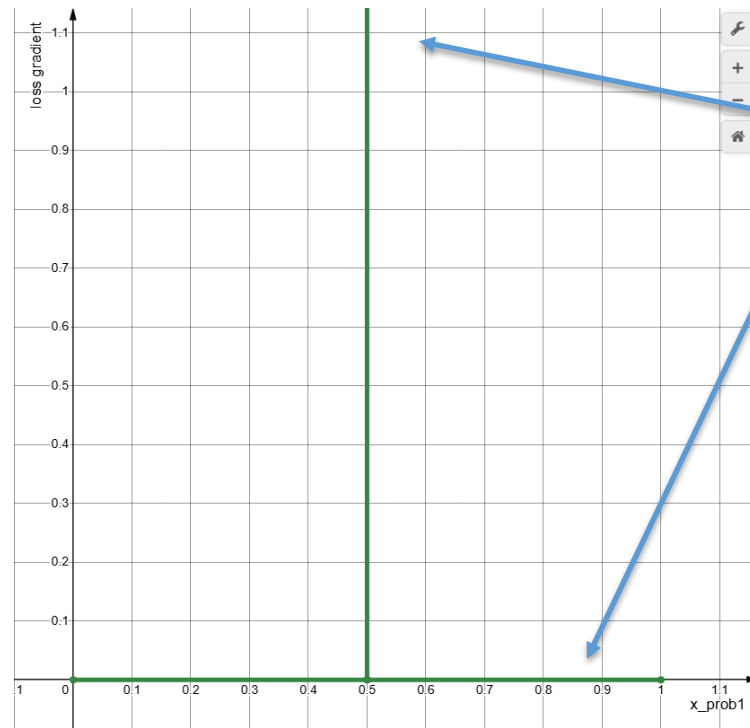
# 0-1 loss gradient

Suppose we have 2 classes (binary classification). Let  $p_{class1}$  denote the probability of class1. Let's view the plots of both the loss and its derivative:

$$\ell_{err}(p_{class1}, y = 1) = \begin{cases} 0 & \text{if } p_{class1} > 0.5 \\ 1 & \text{otherwise} \end{cases}$$



Loss( $p_{class1}$ ,  $y=1$ )



Gradient of Loss

**Problem:** gradient is 0 everywhere, and undefined at  $p_{class1}=0.5$  (shown as infinity for visualization, though technically is undefined). Gradient descent won't work!

## Loss function #2: softmax (aka cross-entropy) loss

Let's convert the hypothesis function to a "probability" by exponentiating and normalizing its entries (to make them all positive and sum to one)

$$z_i = p(\text{label} = i) = \frac{\exp(h_i(x))}{\sum_{j=1}^k \exp(h_j(x))} \Leftrightarrow z \equiv \text{softmax}(h(x))$$

Then let's define a loss to be the (negative) log probability of the true class: this is called *softmax* or *cross-entropy* loss

$$\ell_{ce}(h(x), y) = -\log p(\text{label} = y) = -h_y(x) + \log \sum_{j=1}^k \exp(h_j(x))$$

**Terminology:**  $h(x)$  is called the "logits". They are values output by your model that get normalized by `softmax()` to probabilities.



## Aside: logits

In this equation, the  $h(x)$  are known as **class logits**:

$$p(\text{label} = i) = \frac{\exp(h_i(x))}{\sum_{j=1}^k \exp(h_j(x))} \Leftrightarrow z \equiv \text{softmax}(h(x))$$

Unlike probabilities (range  $[0,1]$ ), logits are unbounded (range  $[-\infty, +\infty]$ ).

Typically, we implement classification models by passing **predicted logits** (rather than probabilities) to loss layers (like pytorch's [CrossEntropyLoss](#)).

Ex: for three classes, suppose our logits are:  $[1, -2, 3]$

```
>>> logits = np.array([[1, -2, 3]])
>>> probs = softmax_normalize(logits)
>>> probs
array([[0.11849965, 0.00589975, 0.8756006 ]])
>>> probs.sum() # always sums to 1
np.float64(1.0)
```

# Binary Cross Entropy (BCE)

Special case: for binary classification (positive/negative class), can simplify expressions. Let  $h(x)$  be the predicted logits for the positive class:

$$p(\text{label} = \text{pos}) = \frac{1}{1 + e^{-h(x)}} \quad \text{Aka "sigmoid" or "logistic" function}$$

Then, binary cross-entropy loss is:

$$\ell_{bce}(h(x), y = \text{pos}) = -\log p(\text{label} = \text{pos}) = -\log\left(\frac{1}{1 + e^{-h(x)}}\right)$$

# BCE loss gradient

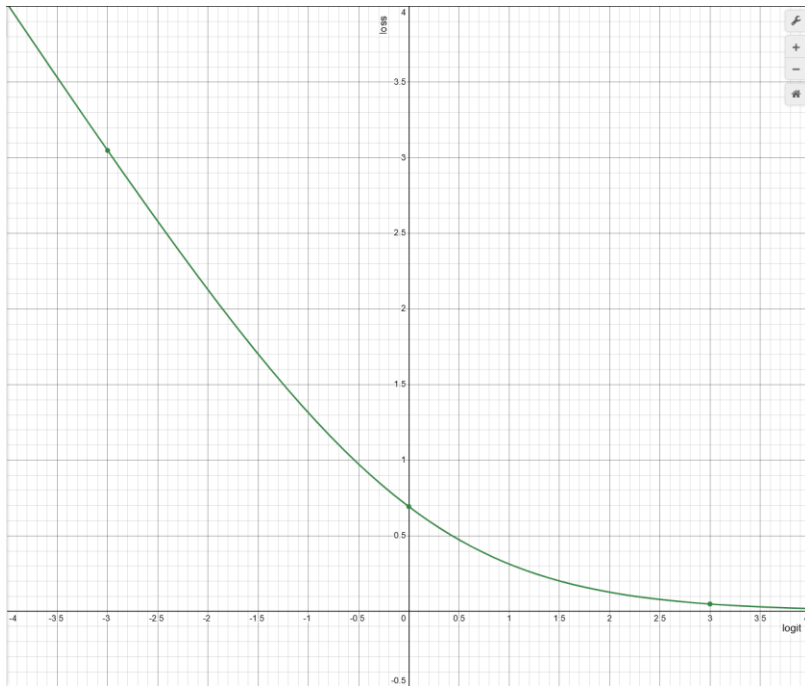
Let's view the BCE loss and its gradient:

$$\ell_{bce}(h(x), y) = -\log\left(\frac{1}{1 + e^{-h(x)}}\right)$$

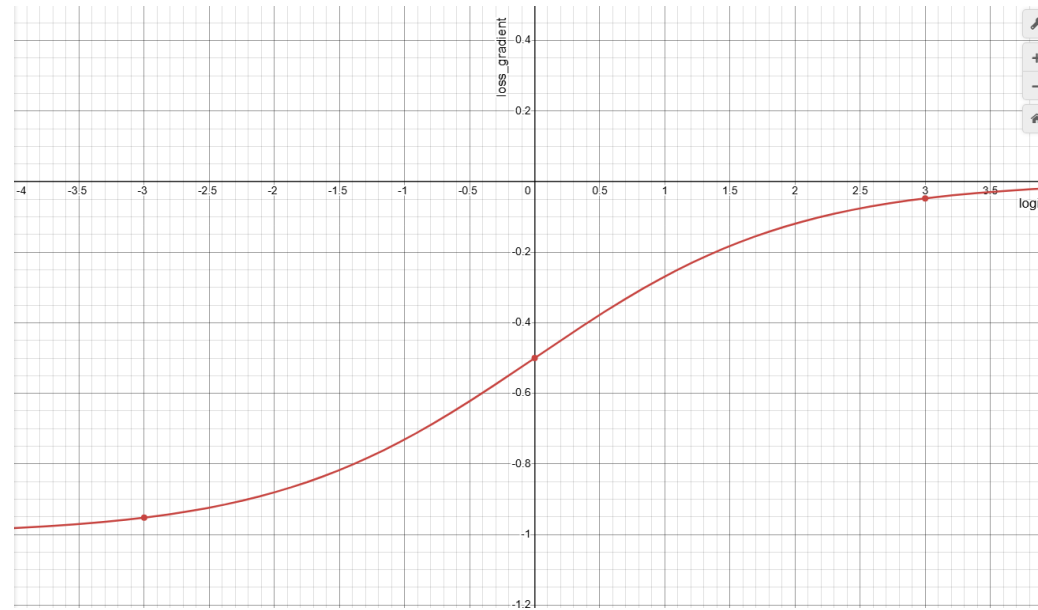
$$\frac{\partial}{\partial h(x)} \ell_{bce}(h(x), y = pos) = -\frac{1}{e^{h(x)} + 1}$$

Gradient is smooth, continuous, and provides a "healthy" training signal! Gradient descent will work well here.

Observation: gradient is large for **confidently wrong preds**, and small for **correct preds**. Good property!



Loss(p\_class1, y=1)



Gradient of Loss

Note: dots indicate logit vals where prob=5%, 50%, and 95% confidence.

# The softmax regression optimization problem

The third ingredient of a machine learning algorithm is a method for solving the associated optimization problem, i.e., the problem of minimizing the average loss on the training set

$$\underset{\theta}{\text{minimize}} \quad \frac{1}{m} \sum_{i=1}^m \ell(\underbrace{h_{\theta}(x^{(i)})}_{\text{model prediction}}, \underbrace{y^{(i)}}_{\text{ground truth label}})$$

**Means:** find the model parameters  $\theta$  that minimizes the training loss

For softmax regression (i.e., linear hypothesis class and softmax loss):

$$\underset{\theta}{\text{minimize}} \quad \frac{1}{m} \sum_{i=1}^m \ell_{ce}(\theta^T x^{(i)}, y^{(i)})$$

So how do we find  $\theta$  that solves this optimization problem?

# Definition: Jacobians

For a multi-input function, multi-output function  $f: \mathbb{R}^n \rightarrow \mathbb{R}^k$ , the **Jacobian** of  $f$  is defined as the collection of all partial derivatives:

$$\nabla f(x) \in \mathbb{R}^{k \times n} = \begin{bmatrix} \frac{\partial f(x)_1}{\partial x_1} & \frac{\partial f(x)_1}{\partial x_2} & \dots & \frac{\partial f(x)_1}{\partial x_n} \\ \frac{\partial f(x)_2}{\partial x_1} & \dots & \dots & \dots \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(x)_k}{\partial x_1} & \vdots & \dots & \frac{\partial f(x)_k}{\partial x_n} \end{bmatrix}$$

Note: this uses the "[numerator](#)" convention for the Jacobian, eg  $[k, n]$ . The "**denominator**" convention instead uses  $[n, k]$ .

Different places can use different conventions, so take care!

**In this course, we'll use the "numerator" convention.**

Ex: if  $f(x_1, x_2) = [x_1 + 3x_2, 2x_1x_2, 42]$ ,  $f: \mathbb{R}^2 \rightarrow \mathbb{R}^3$ , then its Jacobian would be:

$$\nabla f(x) \in \mathbb{R}^{3 \times 2} = \begin{bmatrix} \frac{\partial(x_1 + 3x_2)}{\partial x_1} & \frac{\partial(x_1 + 3x_2)}{\partial x_2} \\ \frac{\partial(2x_1x_2)}{\partial x_1} & \frac{\partial(2x_1x_2)}{\partial x_2} \\ \frac{\partial(42)}{\partial x_1} & \frac{\partial(42)}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2x_2 & 2x_1 \\ 0 & 0 \end{bmatrix}$$

**Intuition:** how much does  $f(x)$  change with respect to each of its inputs?

**In ML:** how much does my loss  $f(x)$  change with respect to each of its model parameters?

# Tip: Jacobian "proper" shapes

**Warning:** in this class, we will often work with functions that accept matrices as input/output. When we define the Jacobian of these matrix-valued functions, the "true" shape of the Jacobian is still 2D, not 3D/4D.

Example: suppose  $f$  takes an  $[n \times k]$  matrix, and outputs a scalar. Its Jacobian  $\nabla f(x)$  has shape  $[n \cdot k, 1]$ , where the first dimension is "flattened" (eg row-wise).

Notably, the shape of  $\nabla f(x)$  is NOT  $[n, k]$ .

However, for convenience, people sometimes represent  $\nabla f(x)$  as a matrix (shape= $[n, k]$ ) rather than as a tall vector (shape= $[n \cdot k, 1]$ ). This is fine, as long as you recognize when this happens!

This will be important later, where it's important to keep track of matrix/Jacobian shapes.

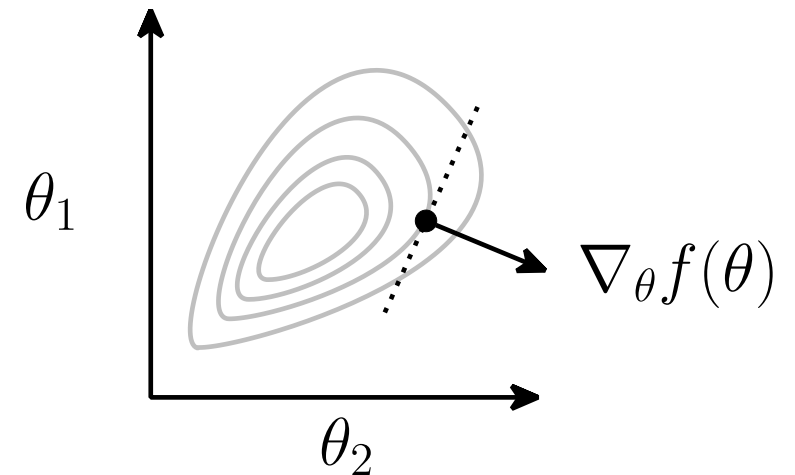
(Apologies if this doesn't make much sense right now, but it will in a few days...)

# Optimization: gradient descent (1/2)

For a matrix-input, scalar output function  $f: \mathbb{R}^{n \times k} \rightarrow \mathbb{R}$ , the *gradient* is defined as the matrix of partial derivatives

$$\nabla_{\theta} f(\theta) \in \mathbb{R}^{n \times k} = \begin{bmatrix} \frac{\partial f(\theta)}{\partial \theta_{11}} & \dots & \frac{\partial f(\theta)}{\partial \theta_{1k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(\theta)}{\partial \theta_{n1}} & \dots & \frac{\partial f(\theta)}{\partial \theta_{nk}} \end{bmatrix}$$

Note: the shape is actually  $[n \times k, 1]$ , but visualized here as  $[n, k]$  for visual clarity



Gradient points in the direction that most *increases*  $f$  (locally)

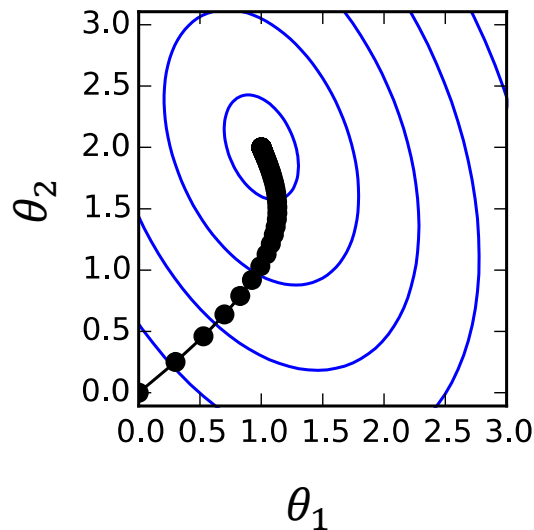
In ML,  $\theta$  is our model parameters, and  $f$  is our loss function. Thus,  $\nabla_{\theta} f(\theta)$  tells us: if I want to minimize the loss  $f$ , in which direction should I adjust my model parameters  $\theta$ ?

# Optimization: gradient descent (2/2)

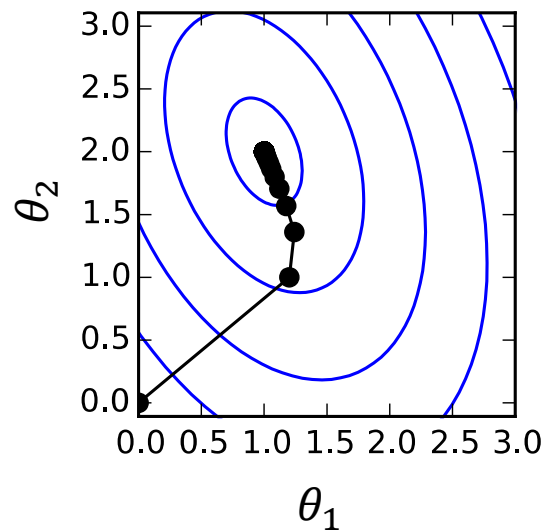
To *minimize* a function, the gradient descent algorithm proceeds by iteratively taking steps in the direction of the negative gradient

$$\theta := \theta - \alpha \nabla_{\theta} f(\theta)$$

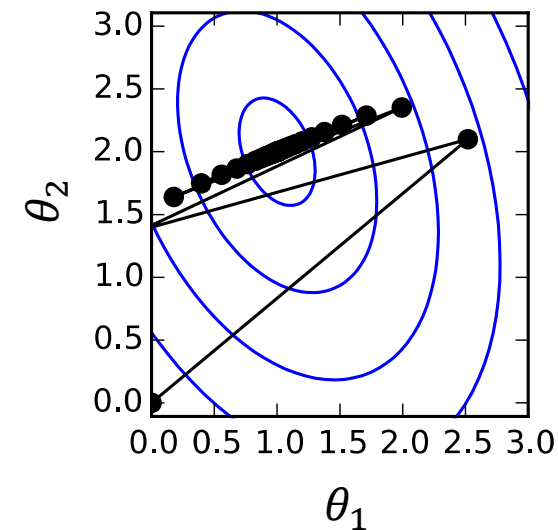
where  $\alpha > 0$  is a *step size* or *learning rate*



$\alpha = 0.05$



$\alpha = 0.2$



$\alpha = 0.42$

The art of picking a good learning rate:  
**Too small:** overly long training jobs.  
**Too large:** unstable training, overshooting.

**In practice:** set empirically (a hyperparameter), eg "try a bunch of step sizes, choose the one that performs best".



# "Stochastic" gradient descent (aka mini-batches)

If our objective (as is the case in machine learning) is the *sum* of individual losses, we typically don't want to compute the gradient using all examples to make a single update to the parameters

Instead, take many gradient steps each based upon a *minibatch* (small partition of the data), to make many parameter updates using a single "pass" over data

Repeat:

Sample a minibatch of data  $X \in \mathbb{R}^{B \times n}, y \in \{1, \dots, k\}^B$

Update parameters  $\theta := \theta - \frac{\alpha}{B} \sum_{i=1}^B \nabla_{\theta} \ell(h_{\theta}(x^{(i)}), y^{(i)})$

Jargon: a pass over the entire dataset is called an "epoch"

# The gradient of the softmax objective

So, how do we compute the gradient for the softmax objective?

$$\nabla_{\theta} \ell_{ce}(\theta^T x, y) = ?$$

One way: "brute force". Write out  $\ell_{ce}(\theta^T x, y)$ , and directly (by hand!) differentiate it with respect to model parameters  $\theta$ . Aka "plug and chug". Labor intensive!

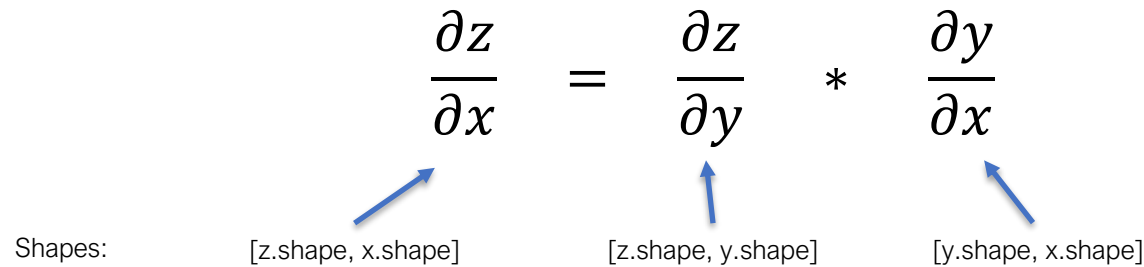
A more convenient way: utilize the **(multivariate) chain rule** to simplify calculations.

# Multivariate chain rule

Suppose we have  $z = f(y), y = g(x)$ . Then:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} * \frac{\partial y}{\partial x}$$

Shapes:  $[z.\text{shape}, x.\text{shape}]$   $[z.\text{shape}, y.\text{shape}]$   $[y.\text{shape}, x.\text{shape}]$



**Note:** unlike scalar chain rule, these quantities are **matrices**, and this is **matrix multiplication**. See how the shapes are compatible with each other! Important to keep track of shapes.

**Intuition:**  $x$  changes  $z$  indirectly via intermediate function  $g(x)$ . To find out how changes in  $x$  modify  $z$  ( $\frac{\partial z}{\partial x}$ ): first find out how changes in  $y$  affect  $z$  ( $\frac{\partial z}{\partial y}$ ), then scale it by how changes in  $x$  affect  $y$  ( $\frac{\partial y}{\partial x}$ ).

Decomposes  $\frac{\partial z}{\partial x}$  in terms of two (hopefully easier) derivatives  $\frac{\partial z}{\partial y}, \frac{\partial y}{\partial x}$ .

**Spoiler alert:** this will be very useful for deep learning ("backprop!")

# Softmax gradient: chain rule (first term)

Let  $h = \theta^T x$  (aka pred logits). Using the chain rule, the softmax loss gradient is:

$$\frac{\partial \ell_{ce}(h, y)}{\partial \theta} = \frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta}$$

$[1, \dim(\theta)]$                        $[1, k]$                        $[k, \dim(\theta)]$

Let's start by computing the first term:

$$\frac{\partial \ell_{ce}(h, y)}{\partial h_i} = \frac{\partial}{\partial h_i} \left( -h_y + \log \sum_{j=1}^k \exp h_j \right)$$

$[1, 1]$

$$= -1\{i = y\} + \frac{\exp h_i}{\sum_{j=1}^k \exp h_j}$$

Means: 1 if  $i=y$ , 0 otherwise. Also written as:  $\delta_{iy}$

$e_y$  is the **one-hot encoding** of  $y$ ,  
eg vector of all 0's except a 1 at  
index for ground-truth class  $y$ .

In vector form:  $\frac{\partial \ell_{ce}(h, y)}{\partial h} = (z - e_y)^T$ , where  $z = \text{softmax}(h)$

$[1, k]$

# Softmax gradient: chain rule (second term)

Recall:  $\theta$  shape is  $[n, k]$ ,  $x$  shape is  $[n, 1]$ ,  $k$  is num classes.

To compute the second term  $\frac{\partial h}{\partial \theta}$  (where  $h = \theta^T x$ ):

$$\theta^T x = \begin{bmatrix} \theta^T[0, :] * x \\ \theta^T[1, :] * x \\ \vdots \\ \theta^T[k, :] * x \end{bmatrix}$$

Observation:  $h[i]$  only depends on the  $i$ -th row of  $\theta^T$ .



$$\frac{\partial h[0]}{\partial \theta} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \quad \frac{\partial h[1]}{\partial \theta} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ 0 & x & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \quad \dots$$

Note: shape is  $[1, d*k]$ , but for clarity I've reshaped to  $[d, k]$

Tip:  $X[i, :]$  means "the  $i$ -th row", eg numpy-like notation

Technically, at this point we are "done"! We can calculate each of the terms of this chain rule expression to get what we want. What could go wrong...?

$$\frac{\partial \ell_{ce}(h, y)}{\partial \theta} = \frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta}$$

Shape:

$[1, \dim(\theta)]$

$[1, k]$

$[k, \dim(\theta)]$

**Issue:**  $\frac{\partial h}{\partial \theta}$  is large! Ex:

ImageNet-1k has 1000 classes. Classification with >10K classes is quite common now. Can we do better?

Recall:  $\theta$  shape is  $[n, k]$ ,  $x$  shape is  $[n, 1]$ ,  $k$  is num classes.

# Trick: exploit structure

Let's see if we can calculate  $\frac{\partial \ell_{ce}(h, y)}{\partial \theta}$  without having to explicitly create  $\frac{\partial h}{\partial \theta}$ :

$$\frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta} = \sum_{i=1}^k \frac{\partial \ell_{ce}}{\partial h[i]} * \frac{\partial h[i]}{\partial \theta} = \left( \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ c_0 x & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} + \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ 0 & c_1 x & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} + \dots \right)$$

Where  $c_i = \frac{\partial \ell_{ce}}{\partial h[i]} = z - e_y$

$$= \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ c_0 x & c_1 x & \cdots & c_k x \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Tip: matrix-vector mult  $A * x$  can be expressed as scaling the  $i$ -th column of  $A$  by the  $i$ -th entry in  $x$ , and summing each scaled column:

$$A * x = \sum_{i=1}^k A[:, i] * x[i]$$

$$= x * \left( \frac{\partial \ell_{ce}}{\partial h} \right) = x * (z - e_y)^T$$

Observe that:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} * \begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 \\ x_2 y_1 & x_2 y_2 \\ x_3 y_1 & x_3 y_2 \end{bmatrix} = \begin{bmatrix} \vdots & \vdots \\ y_1 x & y_2 x \\ \vdots & \vdots \end{bmatrix}$$

Much easier to calculate this!

Shape check:  $[n, 1] * [1, k] \rightarrow [n, k]$  (which, when flattened row-wise) leads to  $[1, n * k]$ !

**What we've done:** we've taken advantage of the structure of  $\frac{\partial \ell_{ce}(h, y)}{\partial h}$  and  $\frac{\partial h}{\partial \theta}$  to calculate  $\frac{\partial \ell_{ce}(h, y)}{\partial \theta}$  in an efficient manner.

# Time, memory complexity

k classes, n input  
dimensionality,  
 $\text{shape}(\theta)=[n, k]$

$$\frac{\partial \ell_{ce}(h, y)}{\partial \theta} = \frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta}$$

Shape:

$[1, n * k]$

$[1, k]$

$[k, n * k]$

**Approach 1:** "naive". Directly instantiate each term, and do matrix multiply.

**Approach 2:** Exploit structure to calculate efficiently.

	Memory	Time
Approach 1	$O(k^2n + k)$	$O(k^2n)$
Approach 2	$O(nk)$	$O(kn)$
Reduction factor	$k + \frac{1}{n} \cong k$	$k$

**Recall:** matrix multiplication  
of  $[m,n]$  and  $[n,p]$  matrix is  
 $O(m*n*p)$

Approach 2 is k  
times better than  
Approach 1 (in both  
time and space!)

# Data point: ResNet-50, ImageNet-1k

[ResNet-50](#): standard convnet (2015)

[ImageNet-1k](#): standard image classification dataset.

k=1000 classes

n=1000 (fc6)

shape( $\theta$ )=[n, k]=[1000, 1000]

$$\frac{\partial \ell_{ce}(h, y)}{\partial \theta} = \frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta}$$

Shape: [1, n \* k]

[1, k]

[k, n \* k]

**Approach 1:** "naive". Directly instantiate each term, and do matrix multiply.

**Approach 2:** Exploit structure to calculate efficiently.

	Memory (fp32), batchsize=1	Time
Approach 1	$O(k^2n + k) = 4 \text{ GB}$	$O(k^2n)$
Approach 2	$O(nk) = 4 \text{ MB}$	$O(kn)$
<b>Reduction factor</b>	$k + \frac{1}{n} \cong k = 1000x$	$k = 1000x$

Approach 2 is 1000x times better than Approach 1 (in both time and space!)

To [train a resnet50 model on imagenet1k](#) in pytorch: ~306 MB GPU memory (batchsize=1), (~14 GB batchsize=96)


4 GB to store  $\frac{\partial h}{\partial \theta}$  is way too expensive! batchsize=96 would require 4\*96=384 GB (!)

Top-of-the-line GPU cards (server) max out at **80 GB memory** (Nvidia H100, ~\$25k each as of Jan 2026)



# Softmax gradient: recap

Let  $h = \theta^T x$  (aka pred logits). The softmax loss gradient is:

$$\frac{\partial \ell_{ce}(h, y)}{\partial \theta} = \frac{\partial \ell_{ce}(h, y)}{\partial h} * \frac{\partial h}{\partial \theta} = x * (z - e_y)^T$$


Note: shape should be  $[1, \dim(\theta)]$ , but here it is  $[n, k]$  for clarity.

Same process works if we use “matrix batch” form of the loss

$$\nabla_{\theta} \ell_{ce}(X\theta, y) \in \mathbb{R}^{B \times k} = X^T (Z - I_y), \quad Z = \text{softmax}(X\theta)$$

Where B is the number of samples in our batch.

$I_y$  is the  $[B, k]$  matrix of one-hot encodings stacked on top of each other.

X has shape  $[B, n]$ .

$\theta$  shape is  $[n, k]$

# Putting it all together

Despite a (somewhat long) derivation, it's neat how *simple* the final algorithm is

- Repeat until parameters / loss converges
  1. Iterate over minibatches  $X \in \mathbb{R}^{B \times n}$ ,  $y \in \{1, \dots, k\}^B$  of training set
  2. Update the parameters  $\theta := \theta - \frac{\alpha}{B} X^T (Z - I_y)$

Step size

This is:  $\frac{\partial \ell_{ce}(h, y)}{\partial \theta}$ , aka  $\nabla_{\theta} \text{loss}(h, y)$

That is the entirety of the softmax regression algorithm

As you will see on the homework, this gets less than 8% error in classifying MNIST digits, runs in a couple seconds

Up next time: neural networks (a.k.a. fancier hypothesis classes)