Data 188: Introduction to Deep Learning

Manual Neural Networks

Speaker: Eric Kim
Lecture 03 (Week 02)
2026-01-27, Spring 2026. UC Berkeley.

Acknowledgement: slides are based on CMU's Deep Learning Systems course (10-414/714, Fall 2025). Thanks to: Prof. Zico Kolter, Prof. Tianqi Chen, Prof. Tim Dettmers.

Announcements

- HW0 continues! Due Feb 5th (~1.5 weeks away)
    - Submit on Gradescope
- Discussion and office hours starts this week! See the course page for times and locations.
- Ask questions on Edstem!

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Outline

From linear to nonlinear hypothesis classes

Neural networks

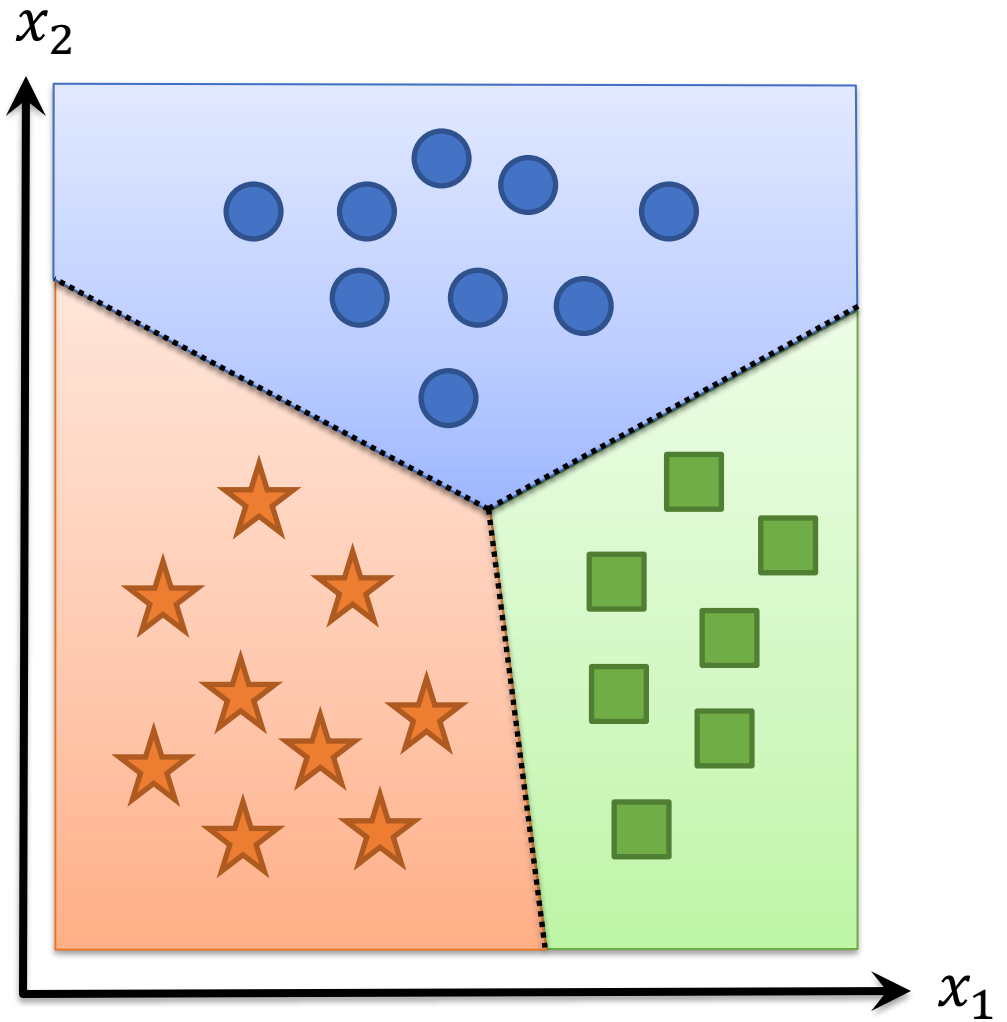
Backpropagation (i.e., computiing gradients)

The trouble with linear hypothesis classes

Recall that we needed a hypothesis function to map inputs in $\mathbb{R}^n$ to outputs (class logits) in $\mathbb{R}^k$, so we initially used the linear hypothesis class

$$h_\theta(x) = \theta^T x, \qquad \theta \in \mathbb{R}^{n \times k}$$

This classifier essentially forms k linear functions of the input and then predicts the class with the largest value: equivalent to partitioning the input into k linear regions corresponding to each class



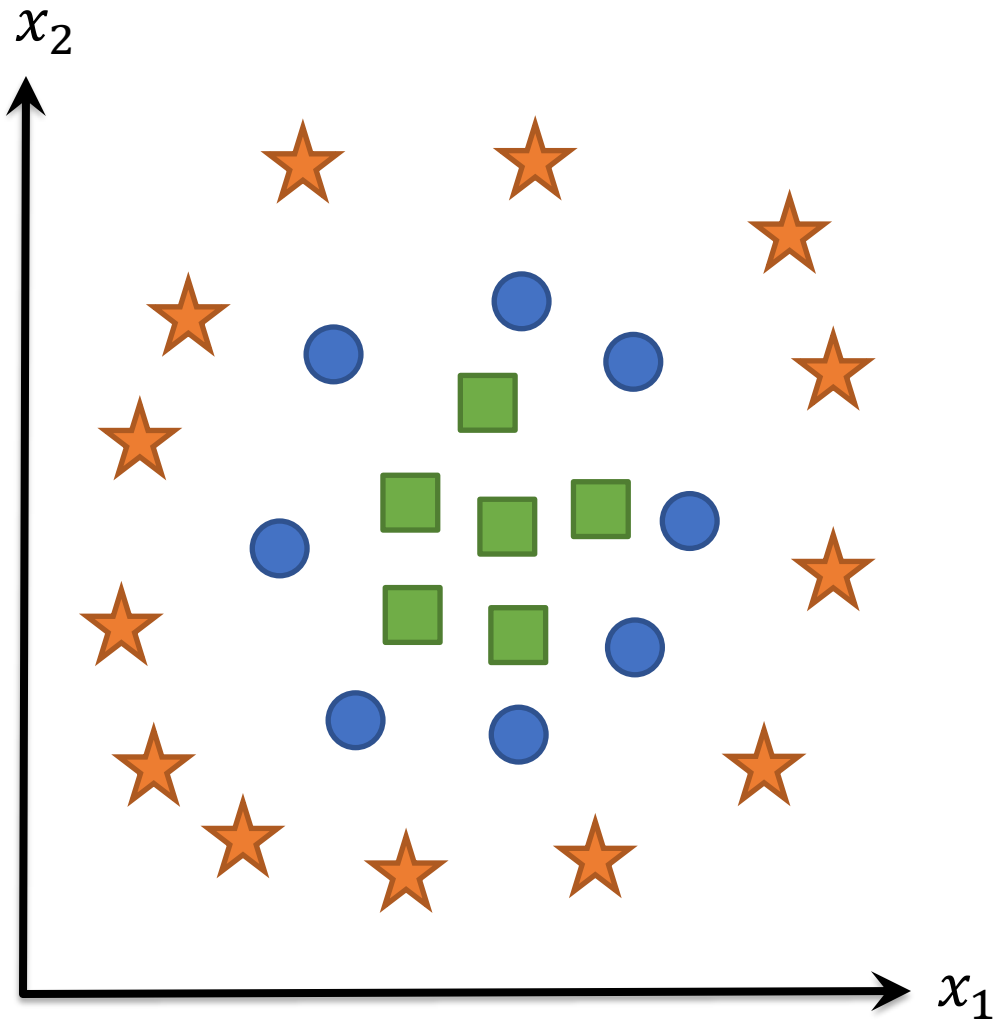
What about nonlinear classification boundaries?

What if we have data that cannot be separated by a set of linear regions?

We want some way to separate these points via a nonlinear set of class boundaries

One idea: apply a linear classifier to some (potentially higher-dimensional) *features* of the data

$$h_\theta(x) = \theta^T \phi(x)$$
$$\theta \in \mathbb{R}^{d \times k}, \phi: \mathbb{R}^n \to \mathbb{R}^d$$



How do we create features?

How can we create the feature function ϕ?

1. Through manual engineering of features relevant to the problem (the "old" way of doing machine learning)
2. In a way that itself is learned from data (the "new" way of doing ML)

Question: what if we just again use a linear function for ϕ? Good idea/bad idea?

$$\phi(x) = W^T x$$

Answer: Bad idea, because it is just equivalent to another linear classifier. We haven't increased the model's expressiveness/power:

$$h_\theta(x) = \theta^T \phi(x) = \theta^T W^T x = \tilde{\theta} x$$

Recall: the composition of linear functions is another linear function

Nonlinear features

But what does work? … essentially *any* nonlinear function of linear features

$$\phi(x) = \sigma(W^T x)$$

where $W \in \mathbb{R}^{n \times d}$, and $\sigma: \mathbb{R}^d \to \mathbb{R}^d$ is essentially *any* nonlinear function (often an element-wise operation for simplicity)

Example: let W be a (fixed) matrix of random Gaussian samples, and let σ be the cosine function $\Longrightarrow$ "random Fourier features" (work great for many problems)

But maybe we want to train W to minimize loss as well? Or maybe we want to compose multiple features together?

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Neural networks / deep learning

A *neural network* refers to a particular type of hypothesis class, consisting of **multiple, parameterized differentiable functions (a.k.a. "layers") composed together in any manner to form the output**

The term stems from biological inspiration, but at this point, literally any hypothesis function of the type above is referred to as a neural network

"Deep network" is just a synonym for "neural network," and "deep learning" just means "machine learning using a neural network hypothesis class" (let's cease pretending that there is any requirements on depth beyond "just not linear")

- But it's also true that modern neural networks involve composing together a *lot* of functions, so "deep" is typically an appropriate qualifier

The “two layer” neural network

We can begin with the simplest form of neural network, basically just the nonlinear features proposed earlier, but where both sets of weights are learnable parameters

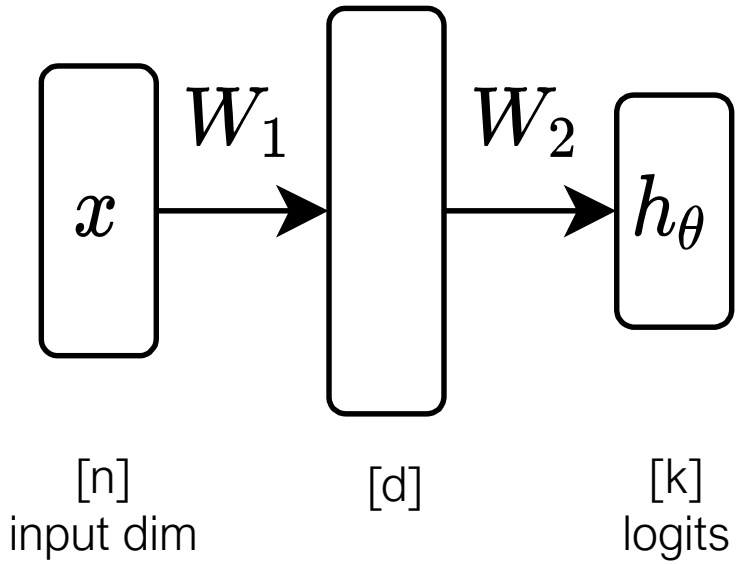
$$h_\theta(x) = W_2^T \sigma(W_1^T x)$$
$$\theta = \{W_1 \in \mathbb{R}^{n \times d}, W_2 \in \mathbb{R}^{d \times k}\}$$

where $\sigma: \mathbb{R} \to \mathbb{R}$ is a nonlinear function applied *elementwise* to the vector (e.g. sigmoid, ReLU)

Written in batch matrix form

$$h_\theta(X) = \sigma(XW_1)W_2$$

X.shape=[batchsize, n]
W_1.shape=[n, d]
W_2.shape=[d, k]
$h_\theta(X)$.shape=[batchsize, k]



Jargon: d is commonly called the "hidden dim" of the model.

Universal function approximation (1/2)

Theorem (1D case): Given any smooth function $f: \mathbb{R} \to \mathbb{R}$, closed region $\mathcal{D} \subset \mathbb{R}$, and $\epsilon > 0$, we can construct a one-hidden-layer neural network $\hat{f}$ such that

$$\max_{x \in \mathcal{D}} \left| f(x) - \hat{f}(x) \right| \leq \epsilon$$

Proof: Select some dense sampling of points $\left(x^{(i)}, f\left(x^{(i)}\right)\right)$ over $\mathcal{D}$. Create a neural network that passes exactly through these points (see next slide). Because the neural network function is piecewise linear, and the function f is smooth, by choosing the $x^{(i)}$ close enough together, we can approximate the function arbitrarily closely.

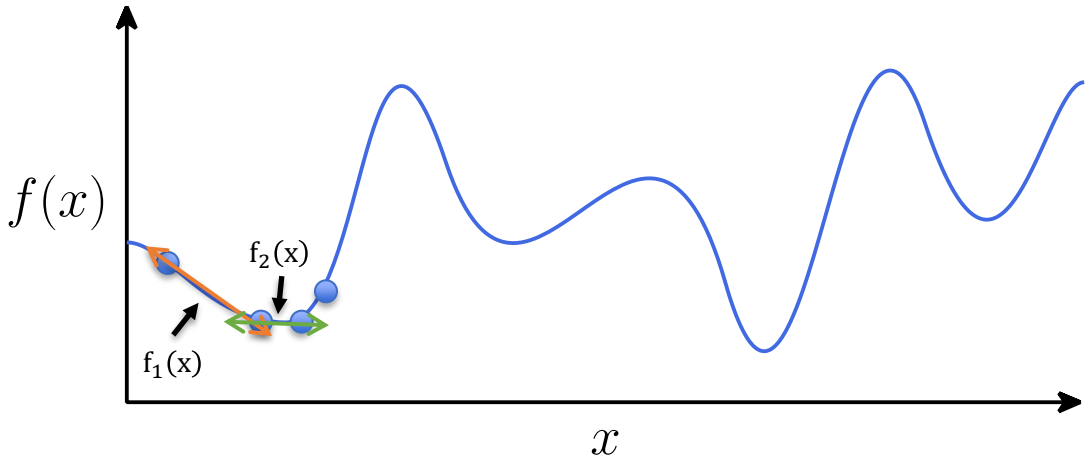
Universal function approximation (2/2)

Assume one-hidden-layer ReLU network (w/ bias):

$$\hat{f}(x) = \sum_{i=1}^{d} \pm \max\{0, w_i x + b_i\}$$

Visual construction of approximating function.

Idea: given N points, fit (N-1) functions $\mathrm{f_i(x)} = \max\{0, w_i x + b_i\}$ such that sum of these (N-1) functions yields a piecewise-linear fit to the N points.
Note: one can construct $\max\{0, w_i x + b_i\}$ to connect two points.



Aside: this proof/result is not very practical, so don't read too much into it.
Boils down to "abusing" NN's into a nearest-neighbor interpolator.

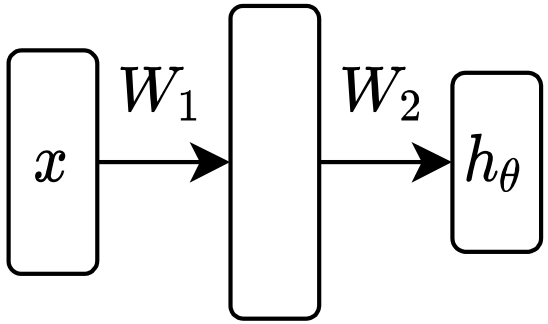
Main takeaway: a sufficiently wide (and/or deep) neural network can fit any* function with arbitrary precision.

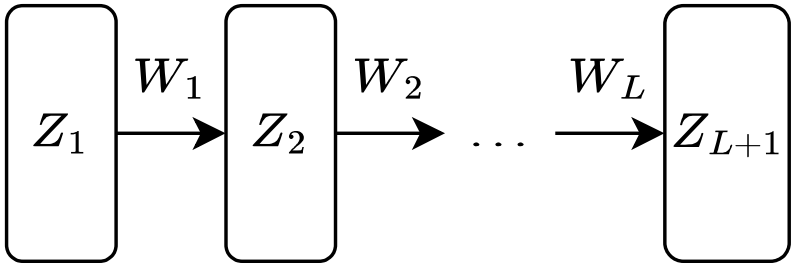
Gives us (some) confidence that pursuing NN's is a justified path forward for learning highly complicated functions (eg image classification models).

* for smooth, continuous functions

(optional) For an alternate visual + interactive visualization, see "A visual proof that neural nets can compute any function".

Fully-connected deep networks





Idea: generalize two-layer network to L-layers ("keep stacking layers!")

A.k.a: "Multi-layer perceptron" (MLP), feedforward network, fully-connected network. In batch form:

$$Z_{i+1} = \sigma_i(Z_i W_i), i = 1, \ldots, L$$
$$Z_1 = X,$$
$$h_\theta(X) = Z_{L+1}$$
$$[Z_i \in \mathbb{R}^{b \times n_i}, W_i \in \mathbb{R}^{n_i \times n_{i+1}}]$$

with nonlinearities $\sigma_i: \mathbb{R} \to \mathbb{R}$ applied elementwise, and parameters

$$\theta = \{W_1, \ldots, W_L\}$$

(Can also optionally add bias term) $z_{i+1} = \sigma_i(z_i W_i + b_i)$

"bias", aka learned offset term

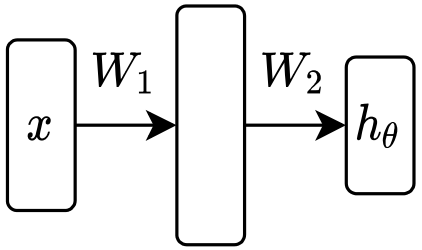
Scaling NN's: width vs depth

Two ways to make an MLP "bigger":

1. Increase depth (eg add more layers)

2. Increase width

$h_\theta(X) = \sigma(XW_1)W_2$

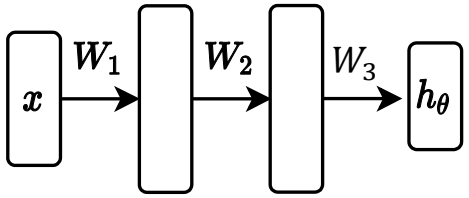


W_1.shape=[n, d=100]
W_2.shape=[d=100, k]
X.shape=[b, n]

More depth

More width

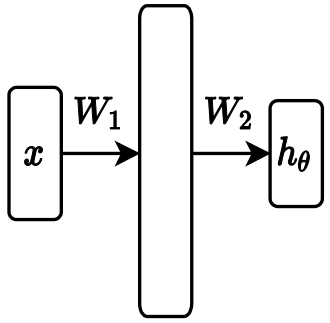
$h_\theta(X) = \sigma(\sigma(XW_1)W_2)W_3$



W_1.shape=[n, d1=100]
W_2.shape=[d1=100, d2=100]
W_3.shape=[d2=100, k]

Note: in theory d1 doesn't have to equal d2!

$h_\theta(X) = \sigma(XW_1)W_2$

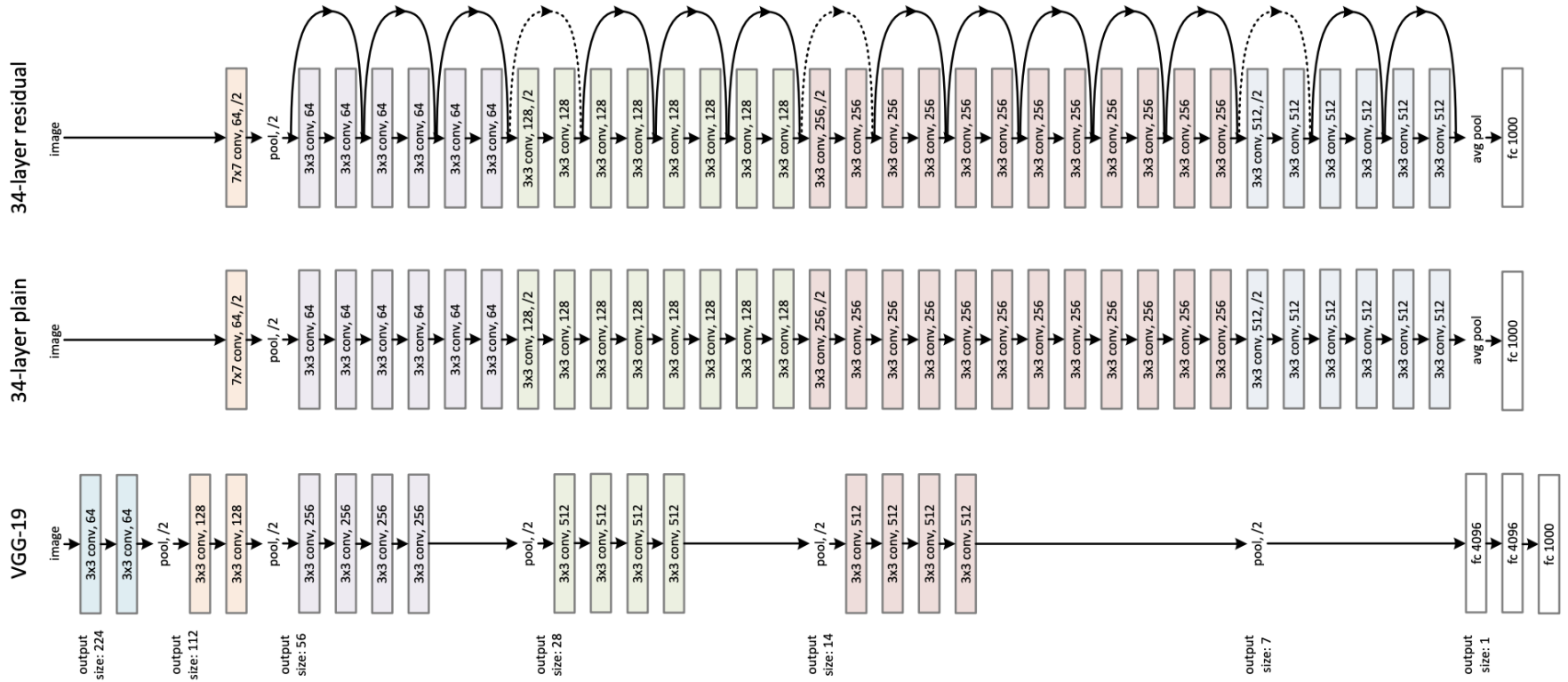


W_1.shape=[n, d=200]
W_2.shape=[d=200, k]

Aka "double the width"

Winner: depth

The deep learning field has generally gravitated towards deeper model architectures rather than wider. Ex: ResNet-50, 101, 152 have 50/101/152 layers!

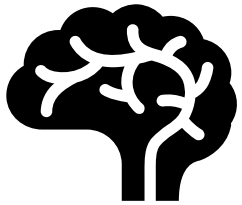


Empirically: deeper (rather than wider) networks generally gives you better results while keeping compute/num-parameters smaller.

Context: VGG-19 was a previous state-of-the-art image classifier, surpassed by resnet and its significantly more layers (increased depth)

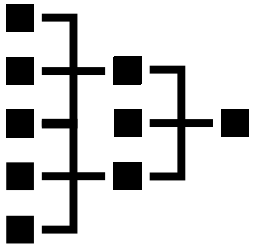
Figure from original Resnet paper, "Deep Residual Learning for Image Recognition".

Why deep networks?



They work like the brain!

… no they don't!



Deep circuits are provably more efficient!

… at representing functions neural networks cannot actually learn (e.g. parity)!

?

Empirically it seems like they work better for a fixed parameter count!

… okay!

Aside: deep learning is largely driven by **empirical findings**. Very little formal guarantees. Aka "we tried out X, and it worked really well!"

In reality: "we tried X on dataset Y. No guarantee that X will work on a different dataset Z!"

Outline

From linear to nonlinear hypothesis classes

Neural networks

Backpropagation (i.e., computiing gradients)

Neural networks in machine learning

Recall that neural networks just specify one of the "three" ingredients" of a machine learning algorithm, also need:

- Loss function: still cross entropy loss, like last time
- Optimization procedure: still SGD, like last time

In other words, we still want to solve the optimization problem

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

using SGD, just with $h_\theta(x)$ now being a neural network

Requires computing the gradients $\nabla_\theta \ell_{ce}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$ for each element of θ

The gradient(s) of a two-layer network (W2)

Let's do the gradient w.r.t. W_2…

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_2} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_2} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

Note: unlike previous single-layer derivation, here I am writing x as a row vector (not as a column vector!). This is to make extending to batched X easier.

The gradient w.r.t. W_2 looks identical to the softmax regression case.

From previous lecture's derivation, the gradient is

$$\nabla_{W_2} \ell_{ce}(\sigma(xW_1)W_2, y) = z_1^T(\text{softmax}(z_2) - e_y)$$

When extended to minibatch X (shape=[batchsize, n]), we get:

$$\nabla_{W_2} \ell_{ce}(\sigma(XW_1)W_2, y) = Z_1^T(\text{softmax}(Z_2) - I_y)$$
$$= \sigma(XW_1)^T(S - I_y)$$

$S = \text{softmax}(\sigma(XW_1)W_2)$

I_y is one-hot vectors, stacked

The gradient(s) of a two-layer network (W1) (1/4)

Deep breath and let's do the gradient w.r.t. W_1…

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

Note: x is a row vector (not as a column vector!)

Approach: I'm going to calculate **left to right** (sort of like `reduce()` fn...)

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} = (\text{softmax}(z_2) - e_y)$$

$$\frac{\partial z_2}{\partial z_1} = \frac{\partial z_1 W_2}{\partial z_1} = W_2^T$$

Short proof:

$$z_1 W_2 = [z_1 W_2[:,0] \quad z_1 W_2[:,1] \quad \ldots]$$

Observation: i-th entry of $z_1 W_2$ only depends on i-th column of W_2

$$\frac{\partial (z_1 W_2)}{\partial z_1}[0,:] = [W_2[:,0]^T]$$
$$\frac{\partial (z_1 W_2)}{\partial z_1}[1,:] = [W_2[:,1]^T]$$
$$\ldots$$

$$\frac{\partial (z_1 W_2)}{\partial z_1} = W_2^T$$

The gradient(s) of a two-layer network (W1) (2/4)

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

So far, we've computed:

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} = (\text{softmax}(z_2) - e_y) \qquad \frac{\partial z_2}{\partial z_1} = \frac{\partial z_1 W_2}{\partial z_1} = W_2^T$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_1}$$

Observe this identity (due to the chain rule). So we can simplify (reduce) the main equation by one term:

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} = (\text{softmax}(z_2) - e_y) W_2^T$$

The gradient(s) of a two-layer network (W1) (3/4)

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

Next, let's proceed and multiply the left-most two terms...

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0}} \cdot \frac{\partial z_0}{\partial W_1}$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial \sigma(z_0)}{\partial z_0}$$

$$= \sum_{i=1}^{d} \left(\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \right)[0, i] \cdot \left(\frac{\partial \sigma(z_0)}{\partial z_0} \right)[i, :]$$

$\frac{\partial \sigma(z_0)}{\partial z_0}$ has nice structure:

$$\frac{\partial \sigma(z_0)}{\partial z_0}[0, :] = [\acute{\sigma}(z_0[0]) \quad 0 \quad \dots \quad 0]$$

$$\frac{\partial \sigma(z_0)}{\partial z_0}[1, :] = [0 \quad \acute{\sigma}(z_0[1]) \quad \dots \quad 0]$$

$$\dots$$

$$= \frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \circ \acute{\sigma}(z_0)$$

$\acute{\sigma}(z_0)$ is the elementwise application of this scalar sigmoid derivative: $\acute{\sigma}(x) = \sigma(x)(1 - \sigma(x))$

where $\circ$ denotes elementwise multiplication

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} = (\text{softmax}(z_2) - e_y) W_2^T$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} = (\text{softmax}(z_2) - e_y) W_2^T \circ \acute{\sigma}(z_0)$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_0}$$

The gradient(s) of a two-layer network (W1) (4/4)

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

Next, let's proceed and multiply the left-most two terms...

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}}$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial xW_1}{\partial W_1}$$

$$= x^T \left(\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0}\right)$$

(by similar argument for single-layer softmax derivation)

where ∘ denotes elementwise multiplication

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} = \left(\text{softmax}(z_2) - e_y\right) W_2^T$$

$$\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} = \left(\text{softmax}(z_2) - e_y\right) W_2^T \circ \acute{\sigma}(z_0)$$

$$\Rightarrow \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = x^T \left(\text{softmax}(z_2) - e_y\right) W_2^T \circ \acute{\sigma}(z_0)$$

$$\Rightarrow \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1}$$

Idea: you may have seen some patterns here. Is it possible to generalize these calculations, say to different model architectures?

Extend to minibatch (shape=[batchsize, n])

$$\Rightarrow \frac{\partial \ell_{ce}(Z_2, y)}{\partial W_1} = X^T \left(\text{softmax}(Z_2) - I_y\right) W_2^T \circ \acute{\sigma}(Z_0)$$

(...phew!)

$Let\ Z_0 = XW_1, Z_1 = \sigma(Z_0), Z_2 = Z_1 W_2$

Abstraction V0 (1/2)

Observation: many terms in $\frac{\partial \ell_{ce}(z_2,y)}{\partial W_1}$ is $\frac{\partial output}{\partial input}$ for each operator ("layer"):

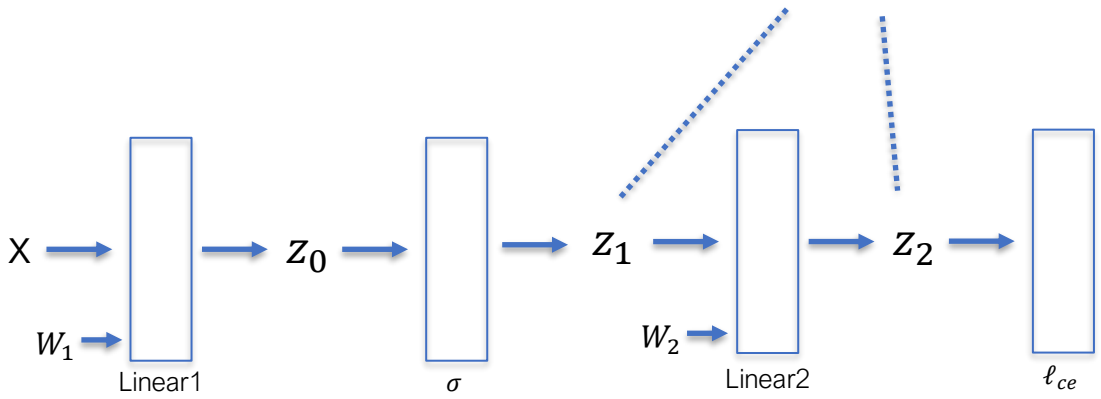
(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

Ex: for "Linear2", $\frac{\partial z_2}{\partial z_1}$ is $\frac{\partial output}{\partial input}$

```
class Linear(Layer):
    def gradient_v0(self) -> np.ndarray:
        # calculate d_out/d_in
        dout_din = self.W.T
        return dout_din
```



Abstraction idea: for each operation type ("layer"), define how to compute $\frac{\partial output}{\partial input}$.

Ex: for Linear layer $z = xW$:

(derived previously)

$$\frac{\partial z}{\partial x} = W^T$$

Abstraction V0 (2/2)

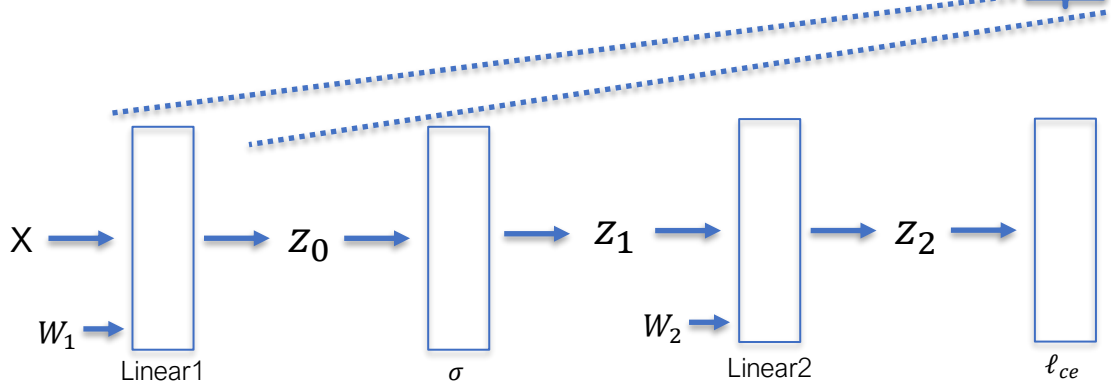
Observation: each term in $\frac{\partial \ell_{ce}(z_2,y)}{\partial W_1}$ is $\frac{\partial output}{\partial input}$ for each operator ("layer"):

(optional, rewrite to clean things up)

$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

```
class Linear(Layer):
    def gradient_v0(self) -> np.ndarray:
        # calculate d_out/d_in
        dout_din = self.W.T
        return dout_din
        # calculate dout/dW
        dout_dW = np.zeros(
            (len(z), math.prod(W.shape))
        )
        # ...calculate big term...
        return dout_din, dout_dW
```



Important: for layers with model parameters, we need to also define how to compute $\frac{\partial output}{\partial param}$

Abstraction idea: for each operation type ("layer"), define how to compute $\frac{\partial output}{\partial input}$.

Ex: for Linear layer $z = xW$:

(derived from previous lecture)

$$\frac{\partial z[0]}{\partial W} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ x & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \quad \frac{\partial z[1]}{\partial W} = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ 0 & x & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix} \quad \cdots$$

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

...but previously we saw that directly instantiating $\frac{\partial output}{\partial param}$ can be **too slow to be practical** (ex: Linear layer!)

Efficient abstraction: vector-Jacobian product

Optimization: calculate product "left to right" (`reduce`-like).
Always involves "vector x matrix -> vector" products (aka "vector-Jacobian product").

(optional, rewrite to clean things up)
$Let\ z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1}} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

$$= \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0}} \cdot \frac{\partial z_0}{\partial W_1}$$

$$= \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}}$$

$$= \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1}$$

dloss_dout is called the **"upstream" gradient**, and is the left-hand-side term in the chain rule reduction

```
class Linear(Layer):
    def gradient(self, dloss_dout: np.ndarray) -> tuple[np.ndarray]:
        # calculate dloss_dout @ dout/dinput
        dloss_dout = dloss_dout @ self.W.T
        # calculate dloss_dout @ dout/dW
        dloss_dW = x.T @ dloss_dout
        return dloss_dout, dloss_dW
```

Performant abstraction: for each layer, compute:

$$\frac{\partial loss}{\partial input} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input}$$

If the layer has model parameters, also compute:

$$\frac{\partial loss}{\partial param} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial param}$$

Linear layer gradient "recipe"

$$\frac{\partial loss}{\partial output} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input} = \frac{\partial loss}{\partial output} W^T$$

$$\frac{\partial loss}{\partial W} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial W} = x^T \left(\frac{\partial loss}{\partial output}\right)$$

Backprop pseudocode: two layer MNIST classifier

```
# initialize layers
linear1 = Linear(in=28*28, out=64)
linear2 = Linear(in=64, out=10)
elem_sigmoid = ElemSigmoid()
loss_ce = CrossEntropyLoss(num_classes=10)

# forward
x = training_images[:batchsize, :]
y = training_labels[:batchsize]
z0 = linear1.forward(x)
z1 = elem_sigmoid.forward(z0)
z2 = linear2.forward(z1)  # logits
loss_val = loss_ce.forward(z2, y)

# backward
dloss_dz2 = loss_ce.backward()
dloss_dz1, dloss_dW2 = linear2.backward(dloss_dz2)
dloss_dz0 = elem_sigmoid.backward(dloss_dz1)
dloss_dx, dloss_dW1 = linear1.backward(dloss_dz0)

# update params
W1 = W1 - stepsize * dloss_dW1
W2 = W2 - stepsize * dloss_dW2
```

(optional, rewrite to clean things up)

Let $z_0 = xW_1, z_1 = \sigma(z_0), z_2 = z_1 W_2$

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_1} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1} = \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1}} \cdot \frac{\partial z_1}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}$$

$$= \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_1} \cdot \frac{\partial z_1}{\partial z_0}} \cdot \frac{\partial z_0}{\partial W_1}$$

$$= \underbrace{\frac{\partial \ell_{ce}(z_2, y)}{\partial z_0} \cdot \frac{\partial z_0}{\partial W_1}}$$

$$= \frac{\partial \ell_{ce}(z_2, y)}{\partial W_1}$$

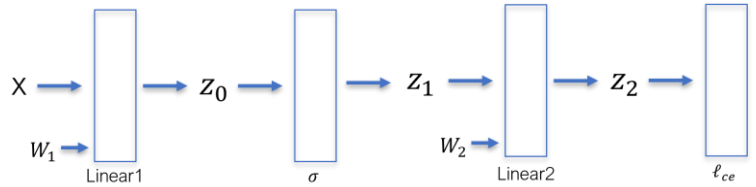
Performant abstraction: for each layer, compute:

$$\frac{\partial loss}{\partial input} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial input}$$

If the layer has model parameters, also compute:

$$\frac{\partial loss}{\partial param} = \frac{\partial loss}{\partial output} \cdot \frac{\partial output}{\partial param}$$

$$\frac{\partial \ell_{ce}(\sigma(xW_1)W_2, y)}{\partial W_2} = \frac{\partial \ell_{ce}(z_2, y)}{\partial W_2} = \frac{\partial \ell_{ce}(z_2, y)}{\partial z_2} \cdot \frac{\partial z_2}{\partial W_2}$$



Goal: understand the mapping between code and math!

Next: we will generalize this idea to arbitrary model architectures, via "computation graphs"!