STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 07: Multi-layer Perceptrons, Backprop

September 24, 2026

Today: From Softmax to Multilayer Perceptrons

- **From Binary to Multi-Class Classification**
- Multilayer Perceptron Architecture
- Nonlinear Activation Functions
- Backpropagation
- Practical Considerations for Training MLPs

Last Time: Autograd, End to End

- Specified a distribution (Bernoulli), wrote the likelihood, took −log to get binary cross-entropy
- Derived $\partial L/\partial z = \hat{y} - y$ by hand
  - Confirmed autograd agrees
- Today: We generalize from 2 classes to K classes, then stack layers into an MLP

From Binary to Multi-Class Classification

- **Binary** (Lecture 6): $y \mid x \sim \text{Bernoulli}(\hat{y}), \quad \hat{y} = \sigma(z)$
- **Multi-class**: $y \mid x \sim \text{Categorical}(\hat{y}_1, \ldots, \hat{y}\_K)$, one logit z_k per class
- **Softmax** turns K logits into a valid probability distribution:

$$\hat{y}\_k = \text{softmax}(z)\_k = \exp(z\_k) / \Sigma_j \exp(z_j)$$

- **Sigmoid** is just the K=2 special case of softmax

Likelihood → Cross-Entropy Loss

- **Likelihood** for one example (one-hot y):

$$P(y \mid x) = \prod_k \hat{y}_k^{y_k} = \hat{y}_{\text{true class}}$$

- **Log-likelihood**: $\log P(y \mid x) = \Sigma_k \, y_k \log \hat{y}_k$
- **Loss** = negative log-likelihood = $-\Sigma_k \, y_k \log \hat{y}_k$ (categorical cross-entropy)
- Exact same recipe as Lecture 6, just more classes
- In **PyTorch**: `nn.CrossEntropyLoss` takes raw logits directly, just like `BCEWithLogitsLoss` did

The Gradient: Same Pattern, More Classes

- Just like the binary case $\frac{\partial L}{\partial z} = \hat{y} - y$, the softmax + cross-entropy gradient has a simple closed form:

$$\frac{\partial L}{\partial z_k} = \widehat{y_k} - y_k \qquad \text{(elementwise, y one-hot)}$$

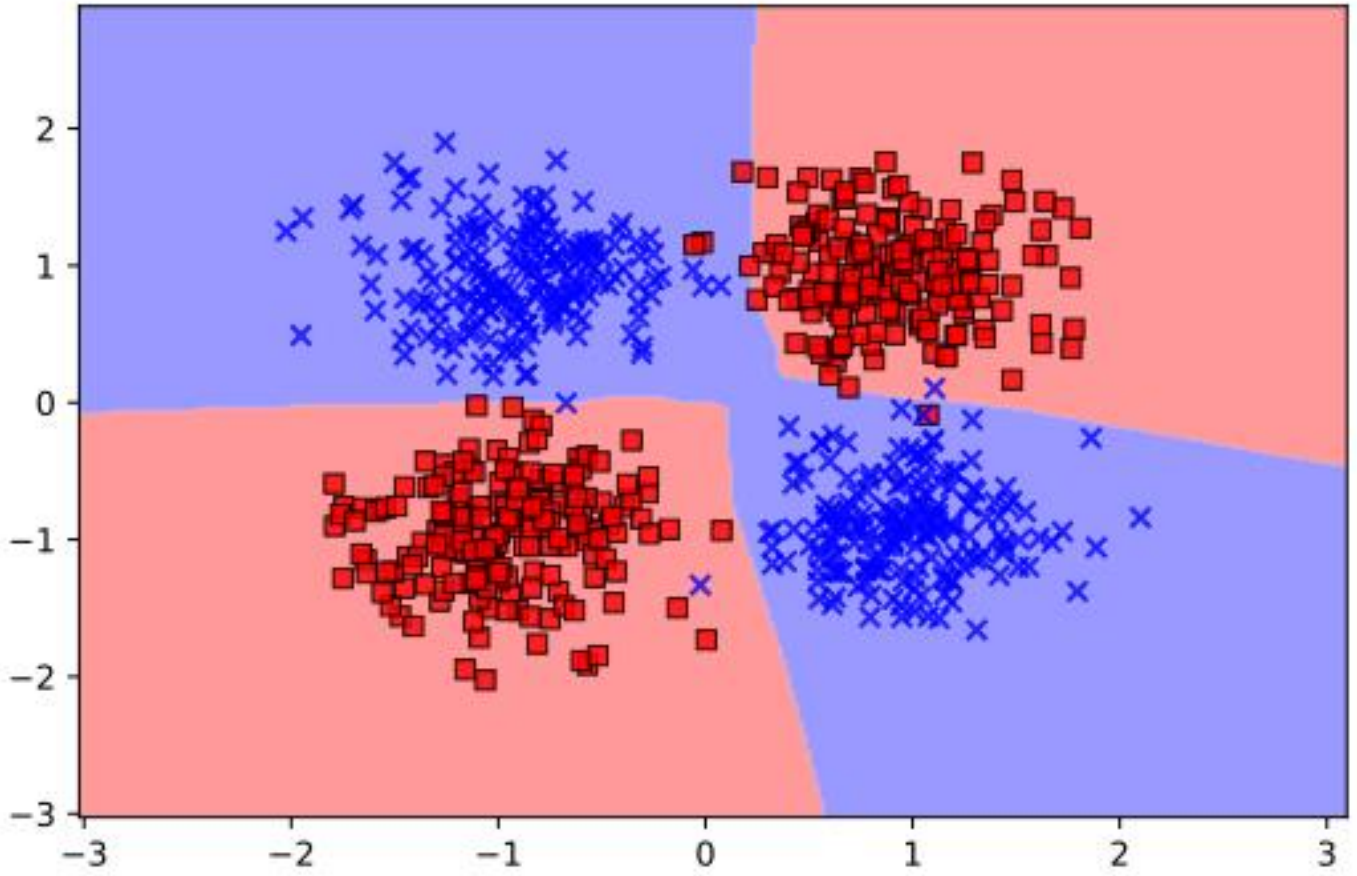
i.e. dL/dz = softmax(z) - onehot(y)

- Same derivation strategy as Lecture 6: chain rule through softmax's own derivative.
- Try it yourself with `torch` the way we checked the binary case last time.

Today: From Softmax to Multilayer Perceptrons

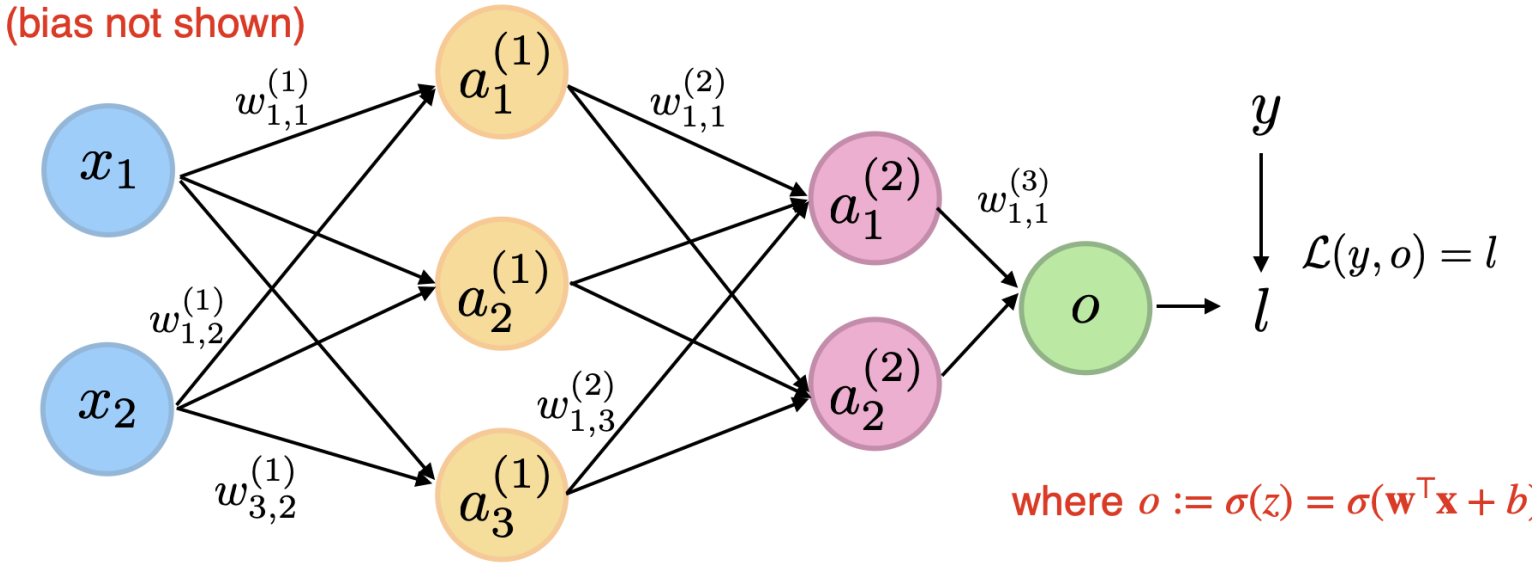
- From Binary to Multi-Class Classification
- **Multilayer Perceptron Architecture**
- Nonlinear Activation Functions
- Backpropagation
- Practical Considerations for Training MLPs

Today: We will finally be able to solve XOR



Multilayer Perceptron

- Computation Graph with Multiple Fully-Connected Layers



where $o := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

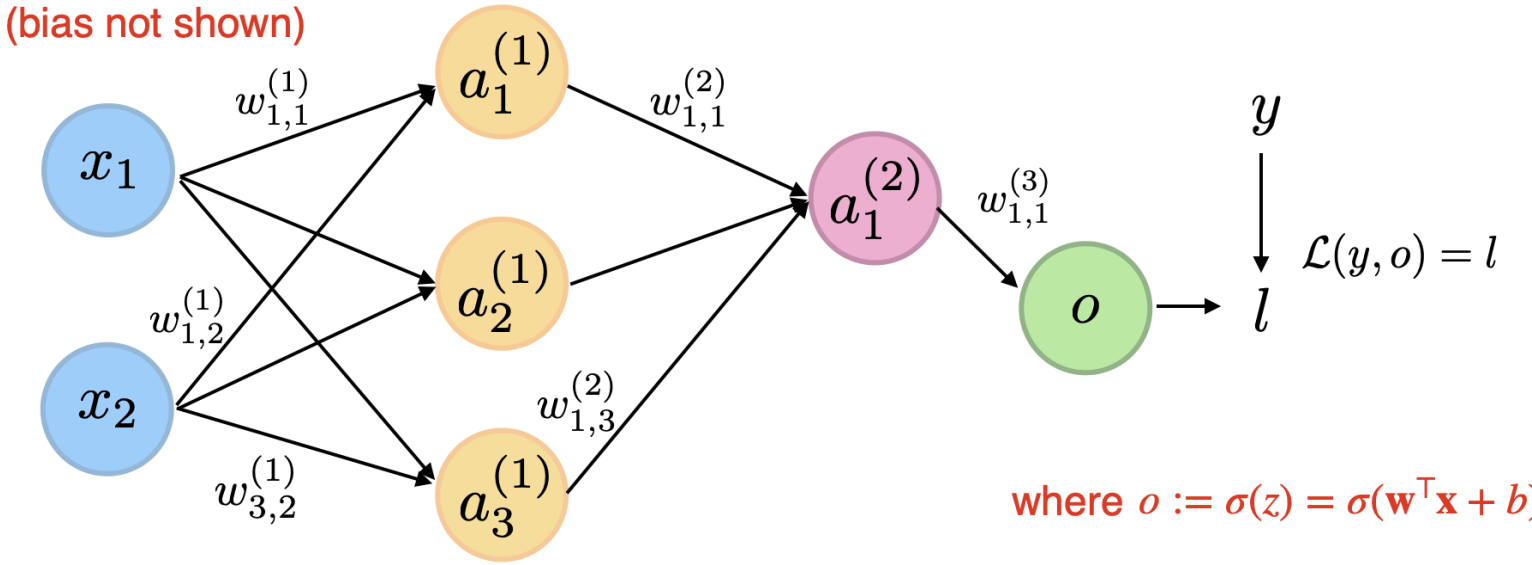
(Assume network for binary classification)

$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Nothing new, really

Multilayer Perceptron

- Computation Graph with Multiple Fully-Connected Layers



where $o := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

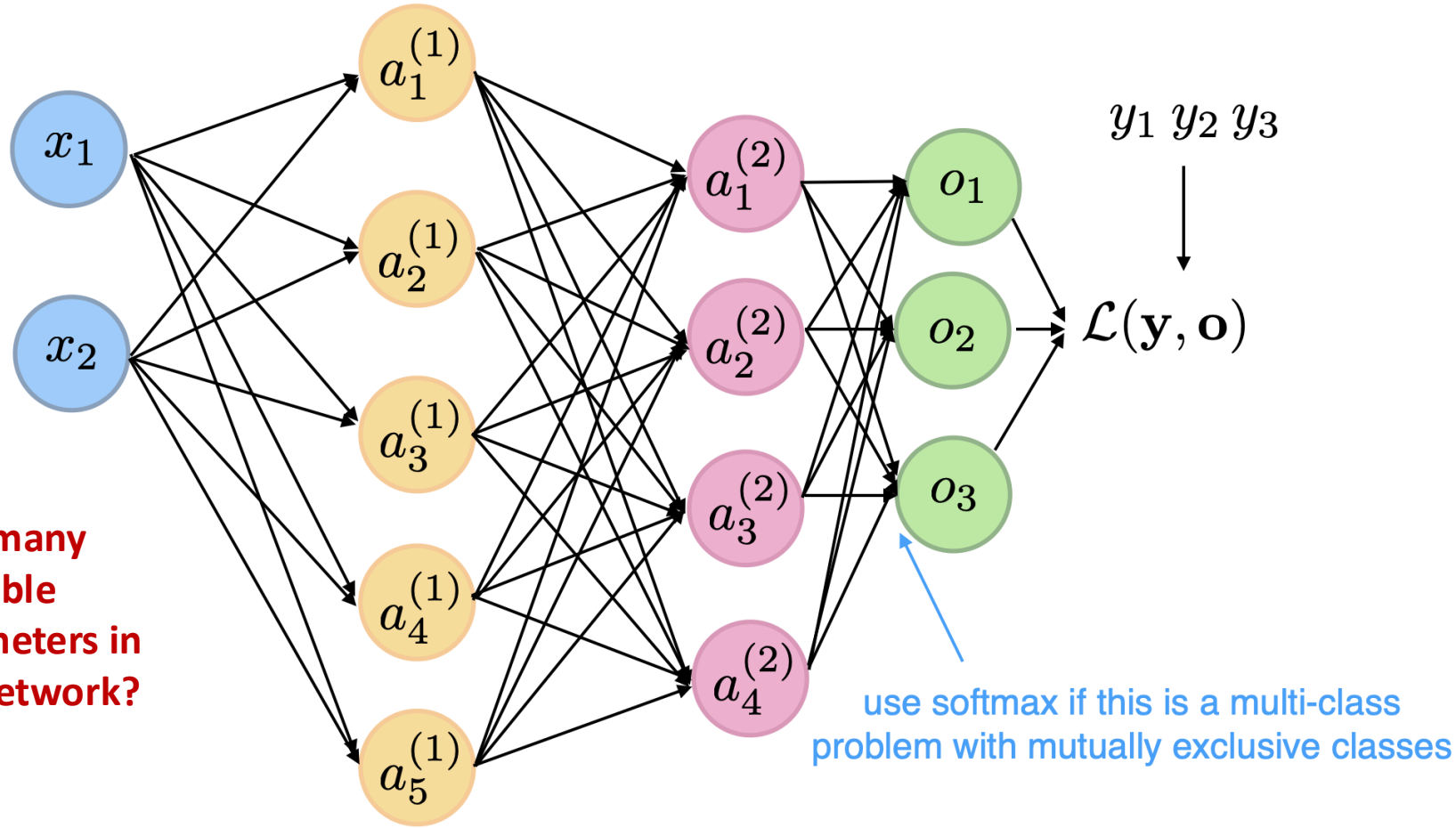
(Assume network for binary classification)

$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}} \cdot \frac{\partial a_2^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}}$$

Nothing new, really

Multilayer Perceptron

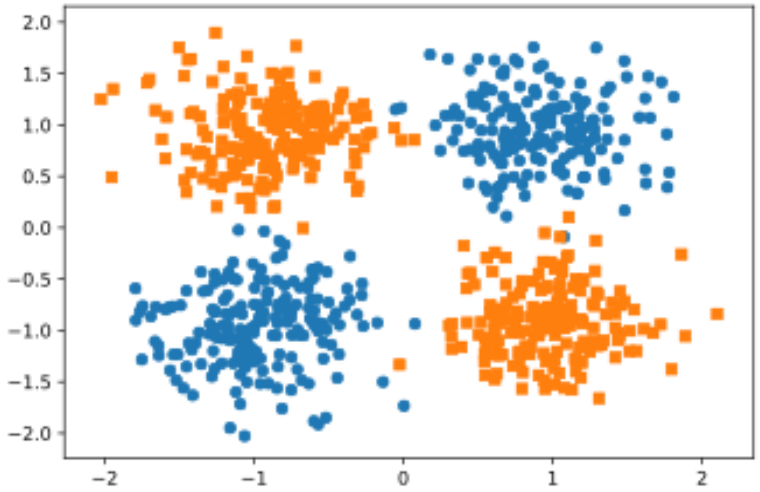
- Computation Graph with Multiple Fully-Connected Layers



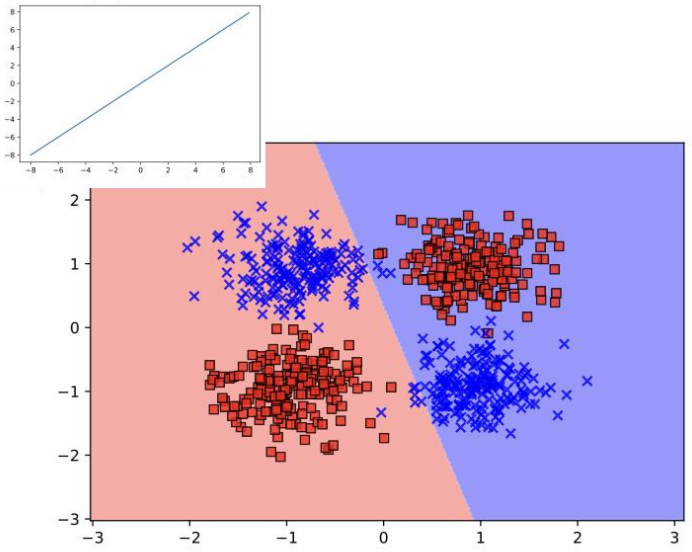
Today: From Softmax to Multilayer Perceptrons

- From Binary to Multi-Class Classification
- Multilayer Perceptron Architecture
- **Nonlinear Activation Functions**
- Backpropagation
- Practical Considerations for Training MLPs

Solving the XOR Problem with Non-Linear Activations

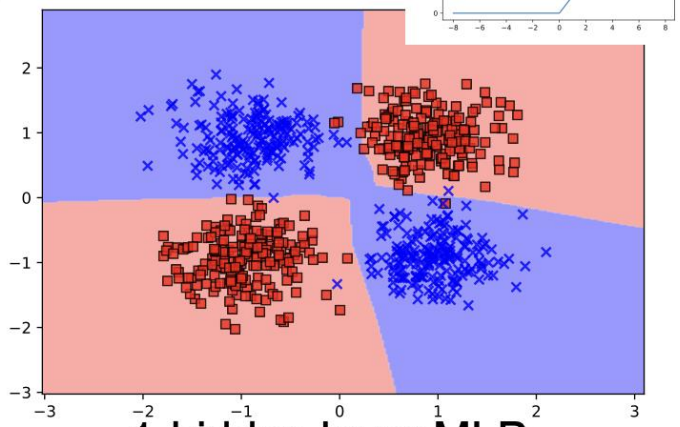


$\sigma(z) = z$



1-hidden layer MLP
with linear activation function

$\sigma(z) = \max(0, z)$

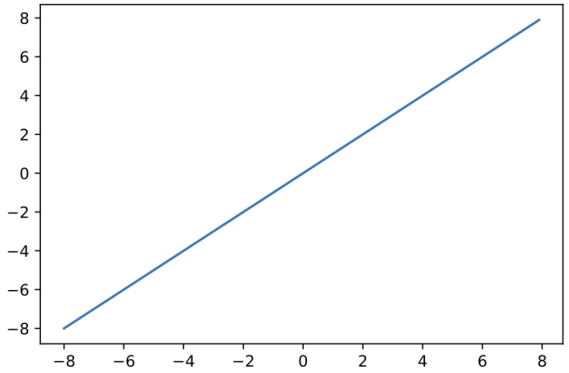


1-hidden layer MLP
with non-linear activation function (ReLU)

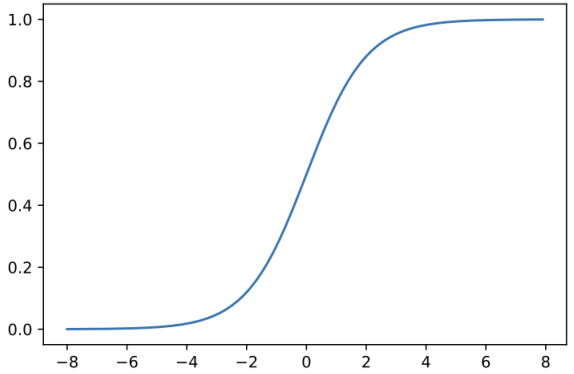
https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L09/code/xor-problem.ipynb

A Selection of Common Activation Functions

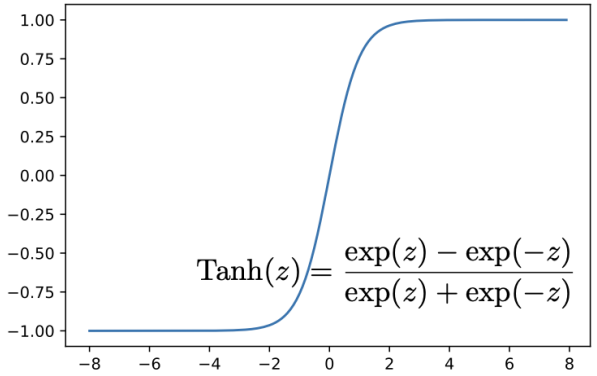
Identity



(Logistic) Sigmoid

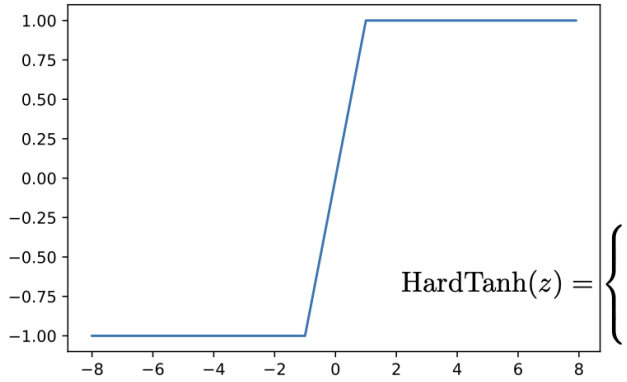


Tanh ("tanH")



$$\text{Tanh}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$

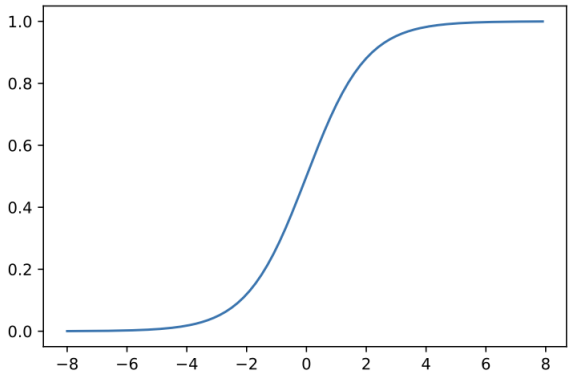
Hard Tanh



$$\text{HardTanh}(z) = \begin{cases} 1 & \text{if } z > 1 \\ -1 & \text{if } z < -1 \\ z & \text{otherwise} \end{cases}$$

A Selection of Common Activation Functions

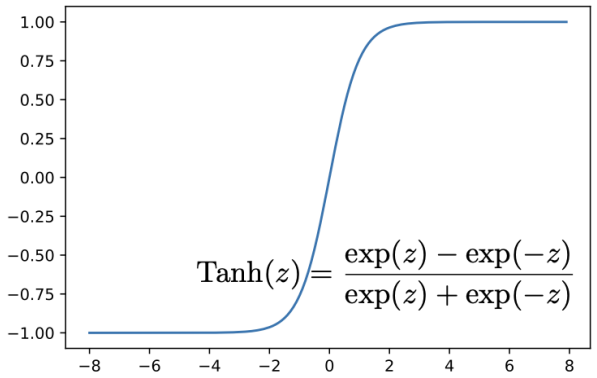
(Logistic) Sigmoid



Advantages of Tanh

- Mean centering
- Positive and negative values
- Larger gradients

Tanh ("tanH")



Also simple derivative:

$$\frac{d}{dz} Tanh(z) = 1 - Tanh(z)^2$$

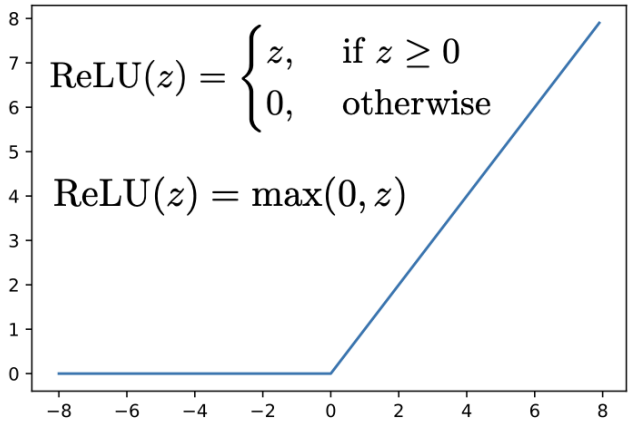
Important to normalize inputs to mean zero and use random weight initialization with **avg. weight centered at zero**

A Selection of Common Activation Functions (cont.)

ReLU (Rectified Linear Unit)

$$\text{ReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

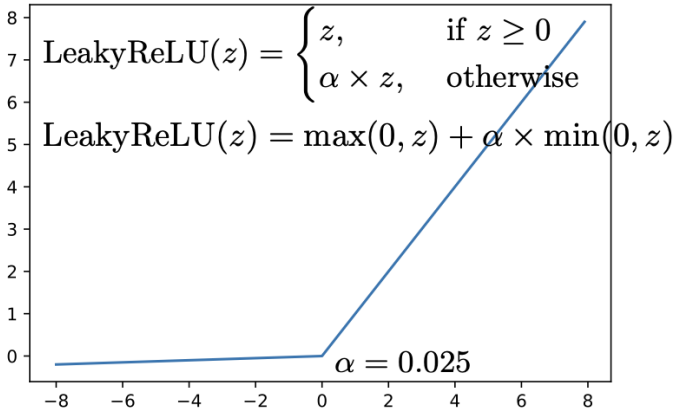
$$\text{ReLU}(z) = \max(0, z)$$



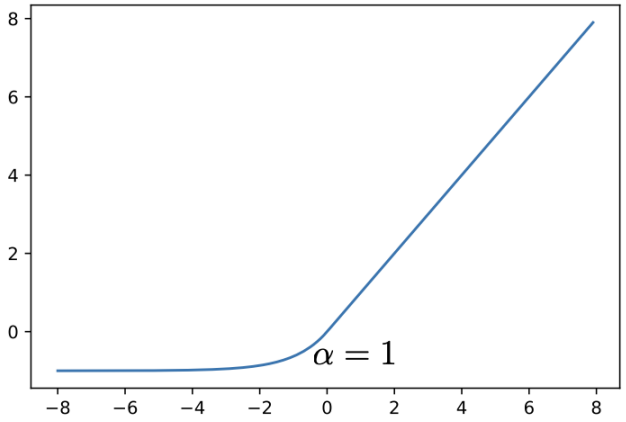
Leaky ReLU

$$\text{LeakyReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha \times z, & \text{otherwise} \end{cases}$$

$$\text{LeakyReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$



ELU (Exponential Linear Unit)



$$\text{ELU}(z) = \max(0, z) + \min(0, \alpha \times (\exp(z) - 1))$$

PReLU (Parameterized Rectified Linear Unit)

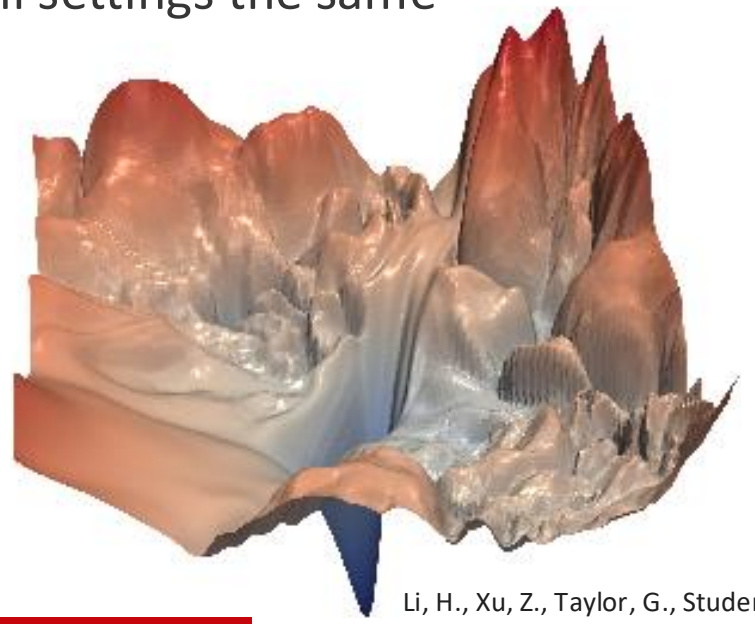
here, alpha is a trainable parameter

$$\text{PReLU}(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{otherwise} \end{cases}$$

$$\text{PReLU}(z) = \max(0, z) + \alpha \times \min(0, z)$$

Note that our Loss is Not Convex Anymore

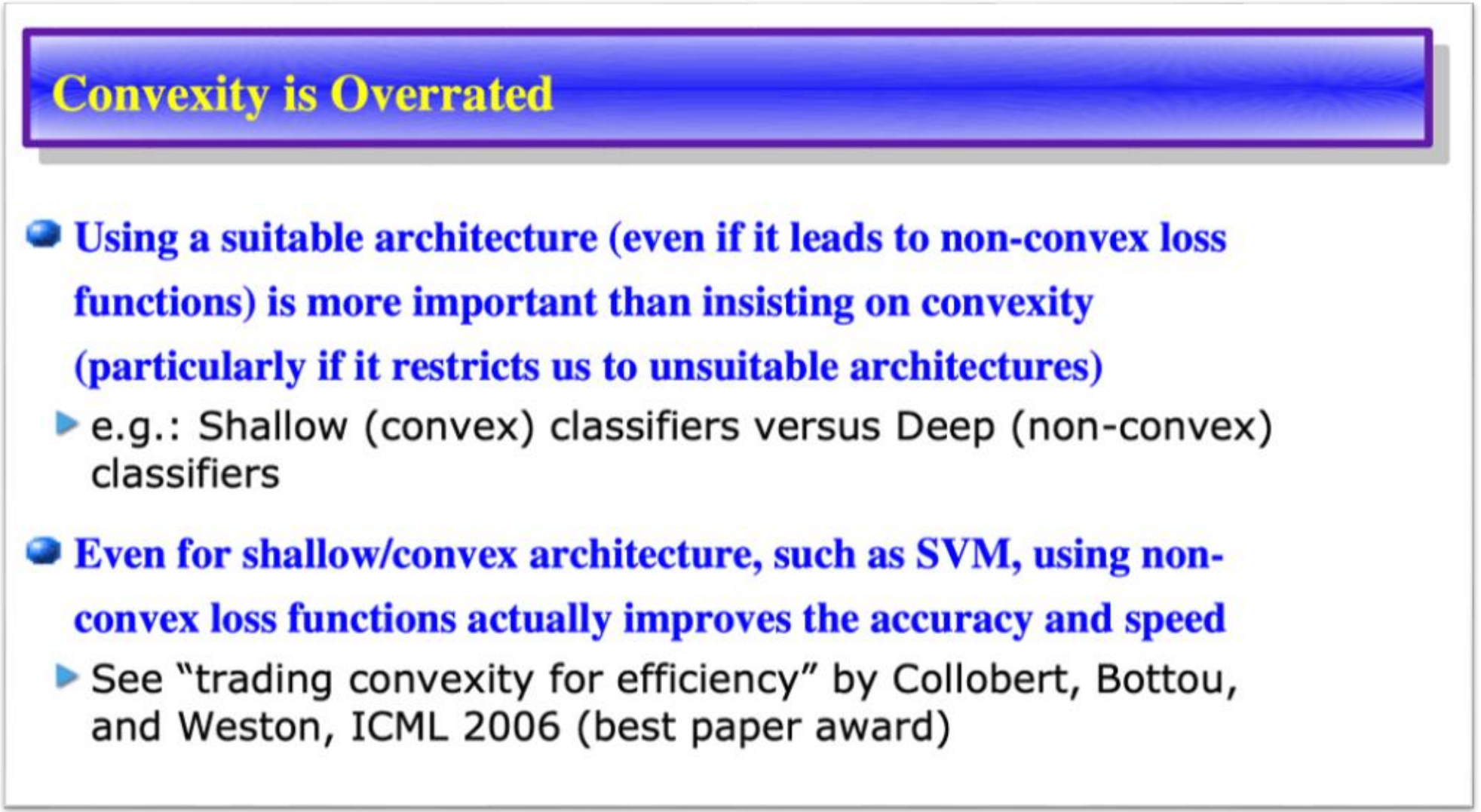
- Linear regression, Adaline, Logistic Regression, and Softmax Regression have convex loss functions
- But our deep loss is no longer convex (most of the time)
  - In practice, we usually end up at different local minima if we repeat the training (e.g. by changing the random seed for weight initialization or shuffling the dataset while leaving all settings the same



Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T., 2018. Visualizing the loss landscape of neural nets. In Advances in Neural Information Processing Systems (pp. 6391-6401).

Death Valley National Park

But…



- Yann LeCun, "Who's afraid of Non-convex loss functions?" – **2007**

Hmm…

What happens if we initialize the multilayer perceptron to all-zero weights?

Answer: The units stay symmetric forever.

This is exactly why we initialize randomly: it breaks the symmetry so different units can learn different features.

Today: From Softmax to Multilayer Perceptrons

- From Binary to Multi-Class Classification
- Multilayer Perceptron Architecture
- Nonlinear Activation Functions
- **Backpropagation**
- Practical Considerations for Training MLPs

Backprop: From L5's Sums to Matrix Form

Recall in Lecture 5, we discussed a fully-connected-layer gradient:
for every weight, you sum the chain rule over every path it reaches the loss through.

That sounds like matrix multiplication. Turns out it is!

Stack all those per-weight sums into vectors, and "sum over paths j" becomes "multiply by the transpose of the weight matrix":

$$\text{delta\_L} = \text{yhat} - \text{y} \qquad \text{(output layer, Lecture 6)}$$

$$\text{delta\_l} = (\text{W\_(l+1)})\text{\^{}T}\ \text{delta\_(l+1)} \qquad \text{(elementwise*)}\ \ \sigma'(\text{z\_l}) \qquad \text{(hidden layers)}$$

$$\text{dL/dW\_l} = \text{delta\_l}\ (\text{a\_(l-1)})\text{\^{}T} \qquad \text{dL/db\_l} = \text{delta\_l}$$

This is why code uses W, delta, and matmul instead of writing out each path by hand.

Backprop by the Numbers: Forward Pass on XOR

Let's define a network:

```
x → [Linear(2,2), ReLU] → h → [Linear(2,1), sigmoid]
→ yhat → BCE loss

W1 = [[1,1],[1,1]],  b1 = [0,-1]
W2 = [1,-2],  b2 = -0.5

x = (1, 0),   y = 1          (XOR(1,0) = 1)
Forward pass:
    z1 = W_1x + b1 = (1, 0)
    h  = ReLU(z1)   = (1, 0)       <- hidden unit 2 is "dead" for this input
    z2 = W_2h + b2  = 1(1) + (-2)(0) - 0.5 = 0.5
    y_hat = sigmoid(0.5) = 0.62,   L = -log(yhat) = 0.47
```

Backprop by the Numbers: Backward Pass

```
dL/dz2 = yhat - y = 0.62 - 1 = -0.38        (Lecture 6's formula)
dL/dW2 = (dL/dz2) h^T = (-0.38, 0)              dL/db2 = -0.38

dL/dh  = (W2)^T (dL/dz2) = (-0.38, 0.75)      (from prev. slide)
dL/dz1 = (dL/dh) (elementwise*) ReLU'(z1) = (-0.38, 0)
dL/dW1 = (dL/dz1) x^T = [[-0.38, 0], [0, 0]]
dL/db1 = (-0.38, 0)
```

Hidden unit 2's gradient is exactly zero — it was "dead" on this example.

Depth Makes Gradients Fragile

Notice, the gradient reaching layer 1 is a PRODUCT of every layer's local derivatives:

```
dL/dz1 = (dL/dzL) (dzL/dz_(L-1)) ... (dz2/dz1)
```

Each factor bakes in an activation derivative:

```
σ'(sigmoid) <= 0.25, σ'(tanh) <= 1,  ReLU' in {0, 1}
```

Mostly-small factors -> product shrinks exponentially with depth (vanishing).
Mostly-large factors (e.g. large weights) -> product grows (exploding).

This is why Tanh's larger gradients matter. We will see more when we discuss Normalization & Initialization (10/6).

Today: From Softmax to Multilayer Perceptrons

- From Binary to Multi-Class Classification
- Multilayer Perceptron Architecture
- Nonlinear Activation Functions
- Backpropagation: Matrices, a Worked Example, and Gradient Flow
- **Practical Considerations for Training MLPs**

Diagnosing loss curves

- Checking the train/test loss/error is one simple diagnosis.
  - Tools such as Weights & Biases (https://wandb.ai) are helpful
- Test/val. error plateau or increases may suggest overfitting
  - Regularization / early stopping is needed (we will discuss next time)
- Train loss may be smoothed over mini-batches
- Some models such as Transformers may be more difficult to train, sharp transition followed by long plateau, e.g. “grokking” (https://arxiv.org/abs/2201.02177)
- Learning rate (LR), LR scheduler, initialization all effect training curves. Lots of trial and error.
  - “Grad student descent”

Parameters vs Hyperparameters

weights (weight parameters)
biases (bias units)

minibatch size
data normalization schemes
number of epochs
number of hidden layers
number of hidden units
learning rates
(random seed, why?)
loss function
various weights (weighting terms)
activation function types
regularization schemes (more later)
weight initialization schemes (more later)
optimization algorithm type (more later)
…

Wide vs Deep Architectures

- MLPs with one (large) hidden layer are universal functions approximators [1-4]
- So why do we want to use deeper architectures?

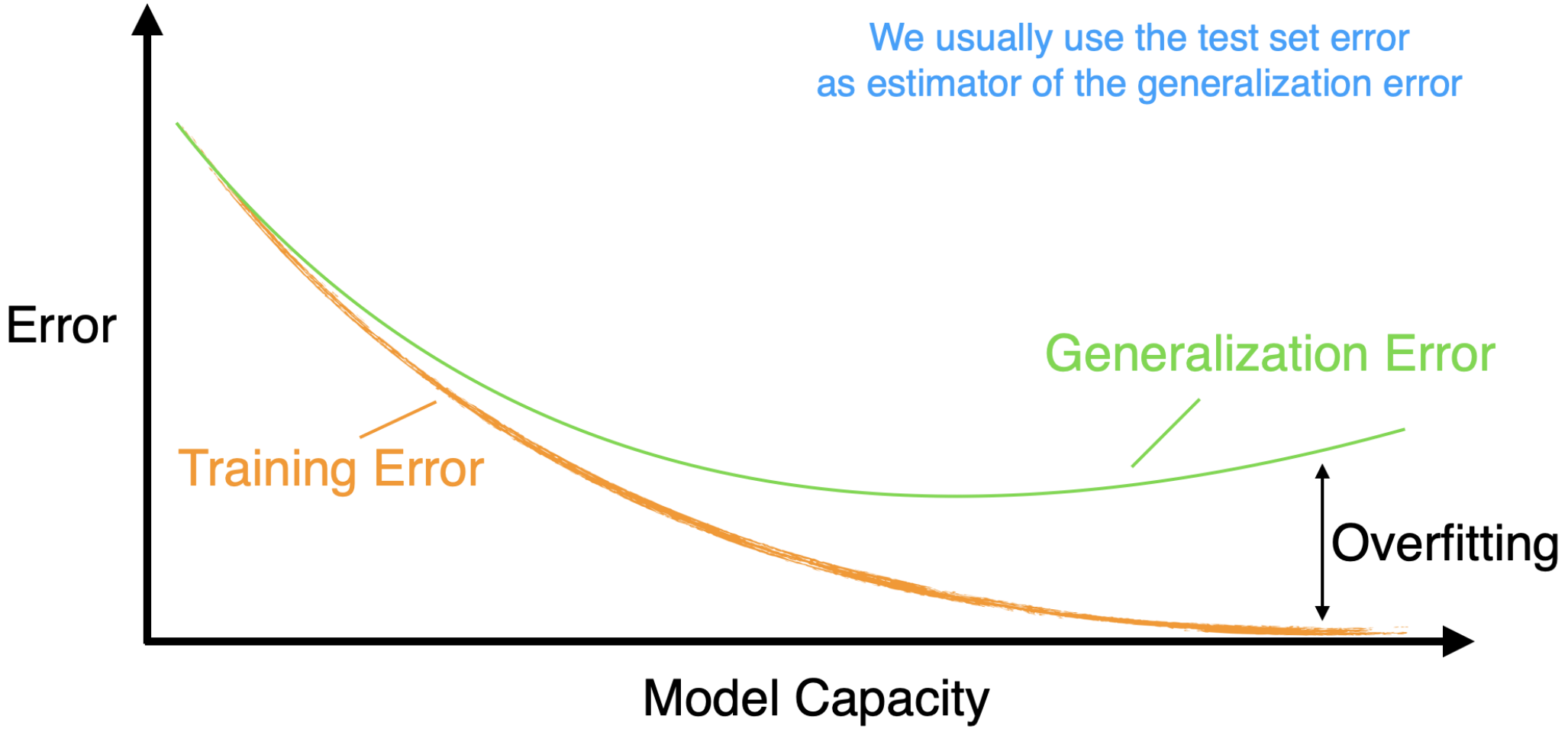
[1] Balázs Csanád Csáji (2001) Approximation with Artificial Neural Networks; Faculty of Sciences; Eötvös Loránd University, Hungary
[2] Barron, Andrew R. "Universal approximation bounds for superpositions of a sigmoidal function." IEEE Transactions on Information theory 39.3 (1993): 930-945.
[3] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2(4), 303–314. doi:10.1007/BF02551274
[4] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, 2(5), 359-366.

Wide vs Deep Architectures

- MLPs with one (large) hidden layer are universal functions approximators [1-4]
- So why do we want to use deeper architectures?

  - Can achieve the same expressiveness with more layers but fewer parameters (combinatorics); fewer parameters => less overfitting?, faster computation
  - Also, having more layers provides some form of regularization: later layers are constrained on the behavior of earlier layers
  - However, more layers => vanishing/exploding gradients
  - Later: different layers for different levels of feature abstraction (DL is really more about feature learning than just stacking multiple layers)
  - Inductive bias? [https://arxiv.org/abs/1608.08225]

Overfitting and Underfitting



Good Practices for Training MLPs

Multilayer Perceptron with Sigmoid Activation and MSE Loss

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mlp-pytorch_sigmoid-mse.ipynb

Multilayer Perceptron with Softmax Activation and Cross-Entropy Loss

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mlp-pytorch_sigmoid-crossentr.ipynb

- Set a seed for reproducibility
- Check that your data/shapes look as expected before training
- Log intermediate values (loss, accuracy) and time your epochs
- If loss behaves strangely (e.g. increases early on), stop and debug before running longer
- Change one hyperparameter at a time and repeat experiments
- See Lecture 6's notebook for a worked PyTorch example of the zero_grad → backward → step loop

From Dense to Structured: Setting Up CNNs

A conv layer is the SAME equation as a fully-connected one: z = W x + b.
The only difference is W is constrained — sparse (local receptive field)
and tied (the same filter reused at every position).

So the backward pass doesn't change either: delta = W^T delta_next (elementwise*)
σ'(z)
What changes: a shared weight now gets a gradient contribution from every
position it was used at, so you SUM them (Lecture 5's weight-sharing slide).

Teaser for next time: backprop through a convolution turns out to be a
convolution too, with the kernel flipped. We'll prove this in Lecture 8.

Next time (9/29): we prove the flipped-kernel claim above, concretely — Structure and weight sharing, CNNs.

Questions?