Lecture 09

Multilayer Perceptrons

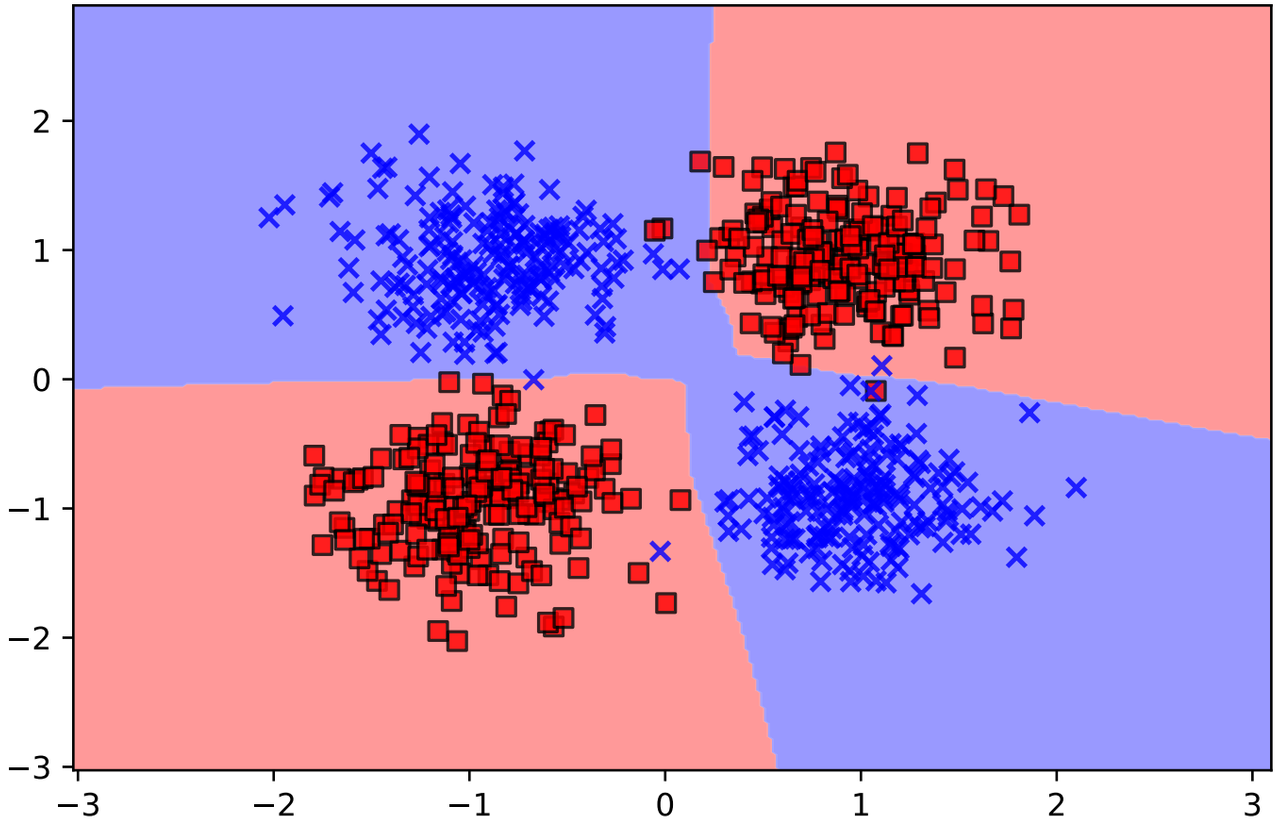
STAT 453: Deep Learning, Spring 2020

Sebastian Raschka

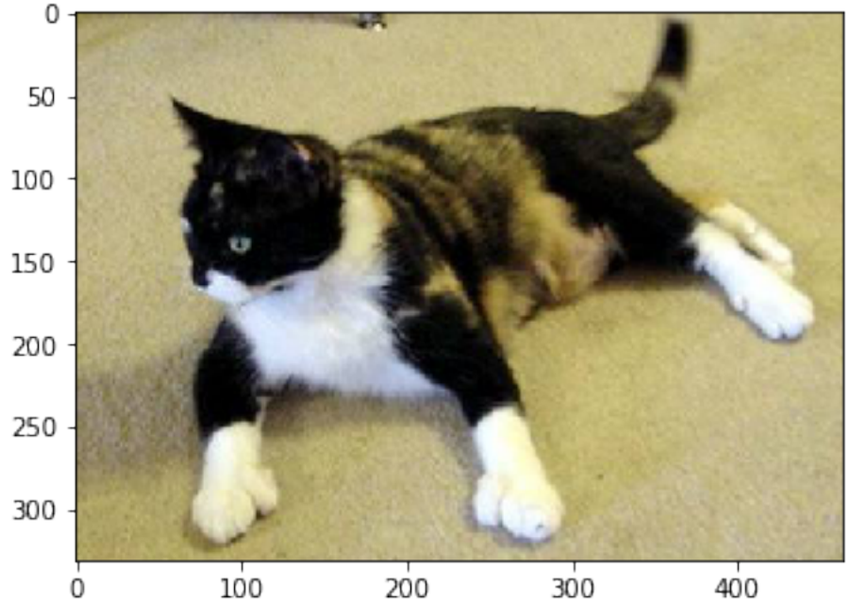
http://stat.wisc.edu/~sraschka/teaching/stat453-ss2020/

Today

We will finally be able to solve the XOR problem ...



... and talk about cats!



Topics

Multilayer Perceptron Architecture

Nonlinear Activation Functions

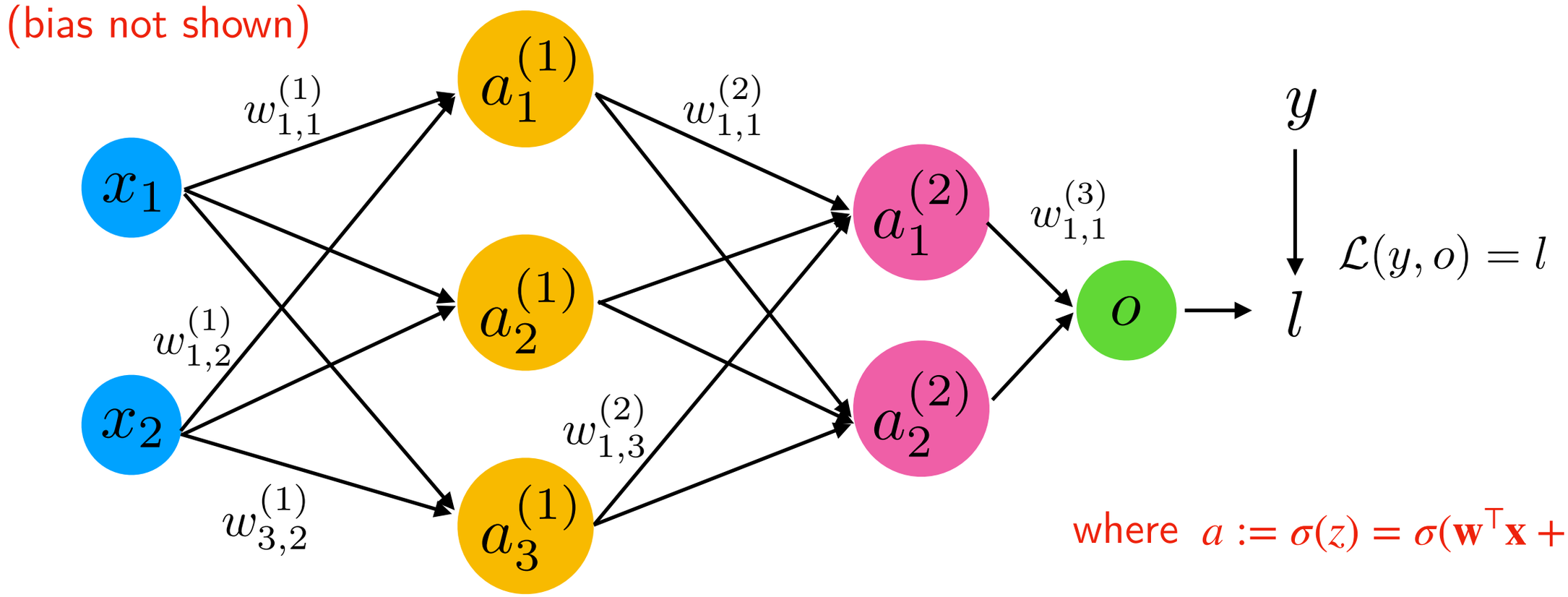
Multilayer Perceptron Code Examples

Overfitting and Underfitting

Cats & Dogs and Custom Data Loaders

Graph with Fully-Connected Layers = Multilayer Perceptron

Nothing new, really

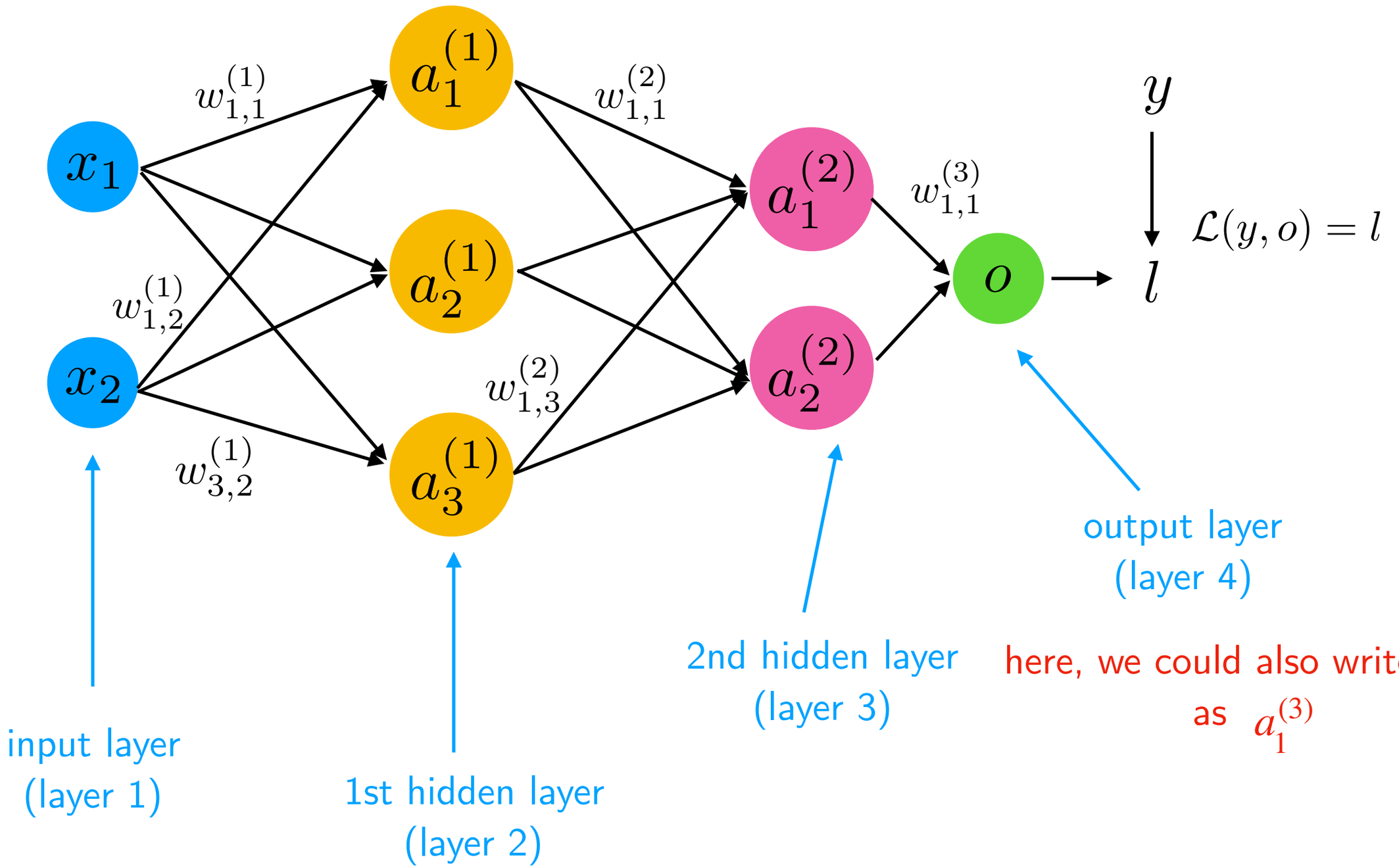


where $a := \sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$

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

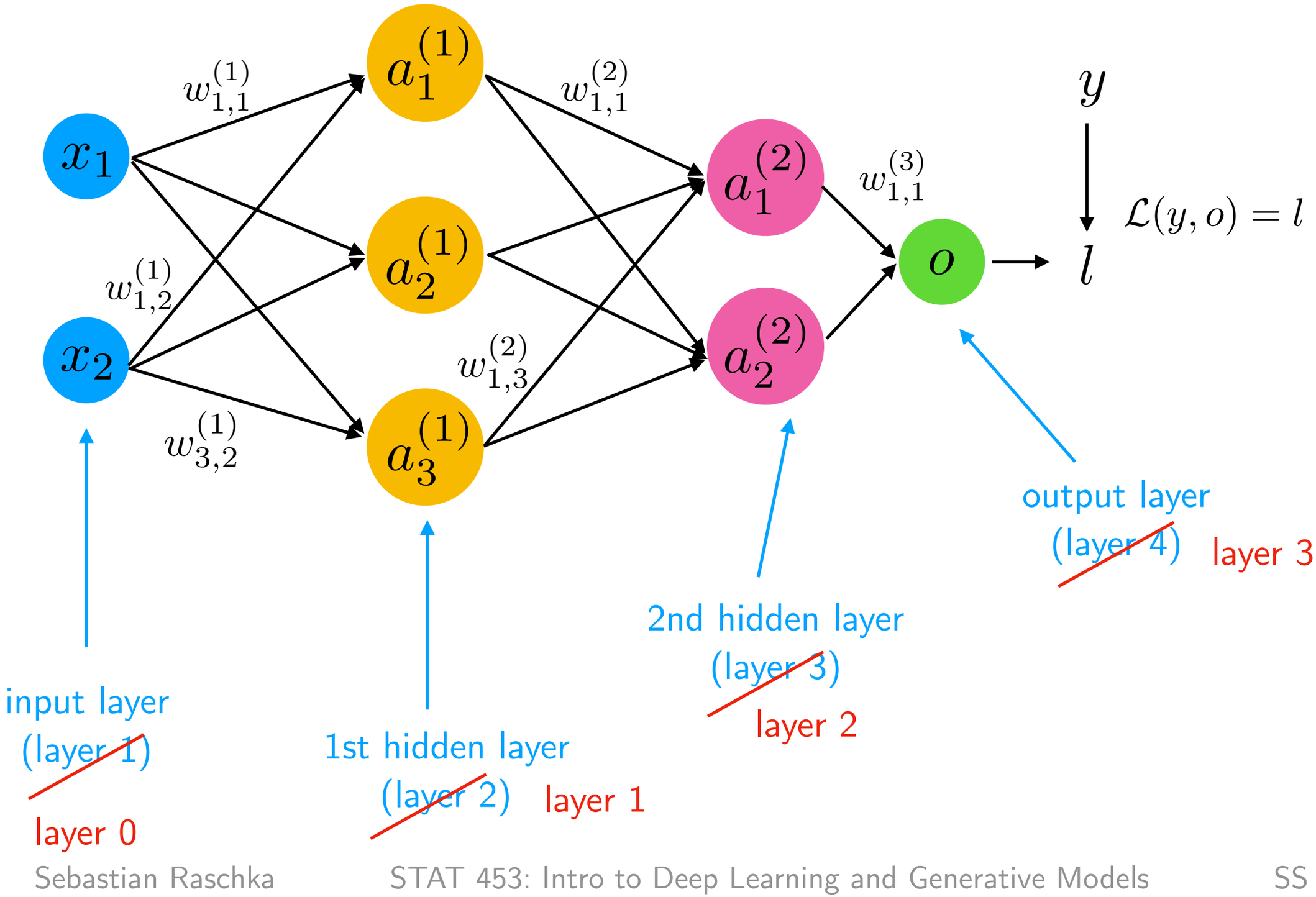
(Assume network for binary classification)

Graph with Fully-Connected Layers = Multilayer Perceptron

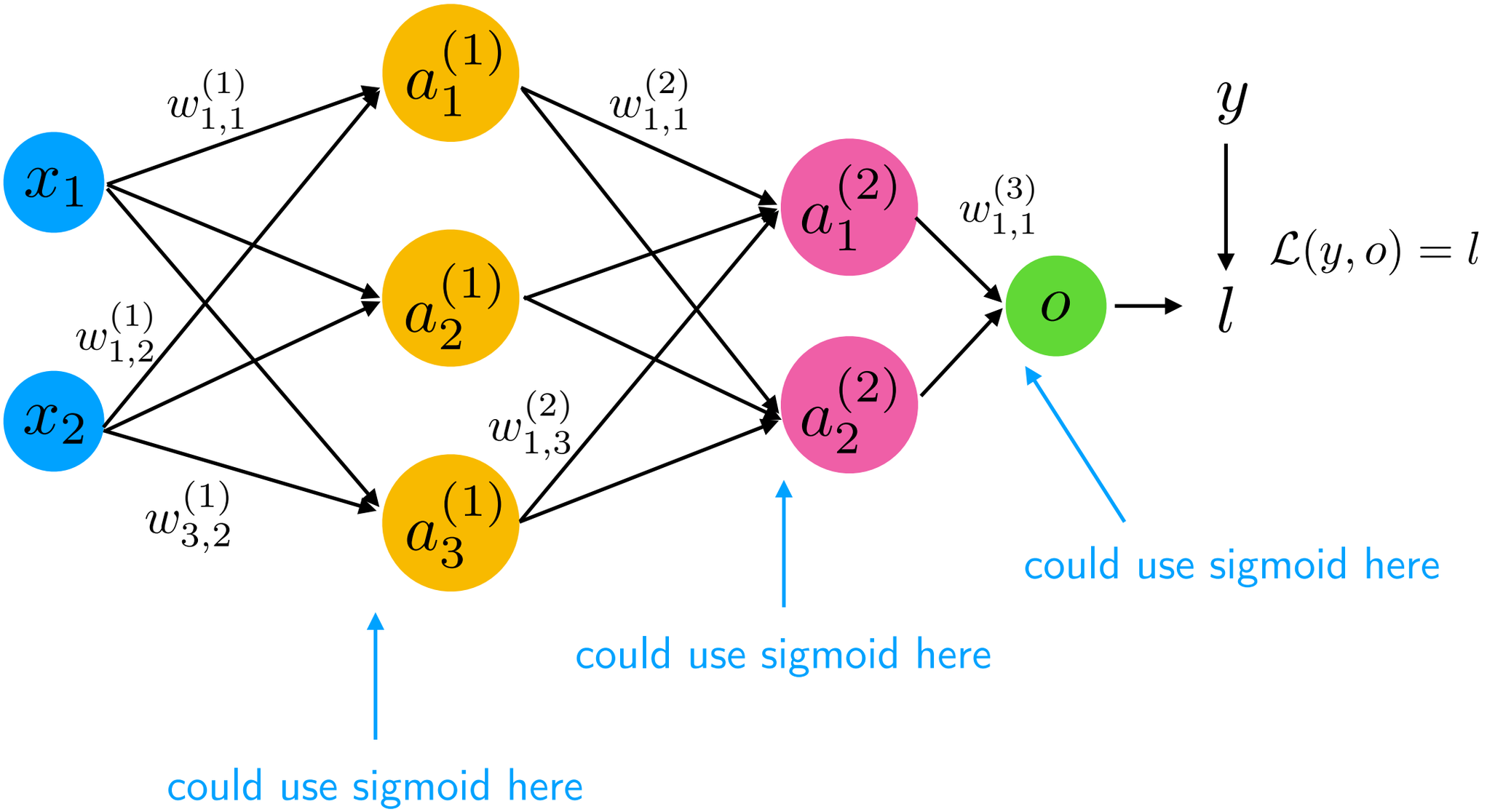


Graph with Fully-Connected Layers
= Multilayer Perceptron

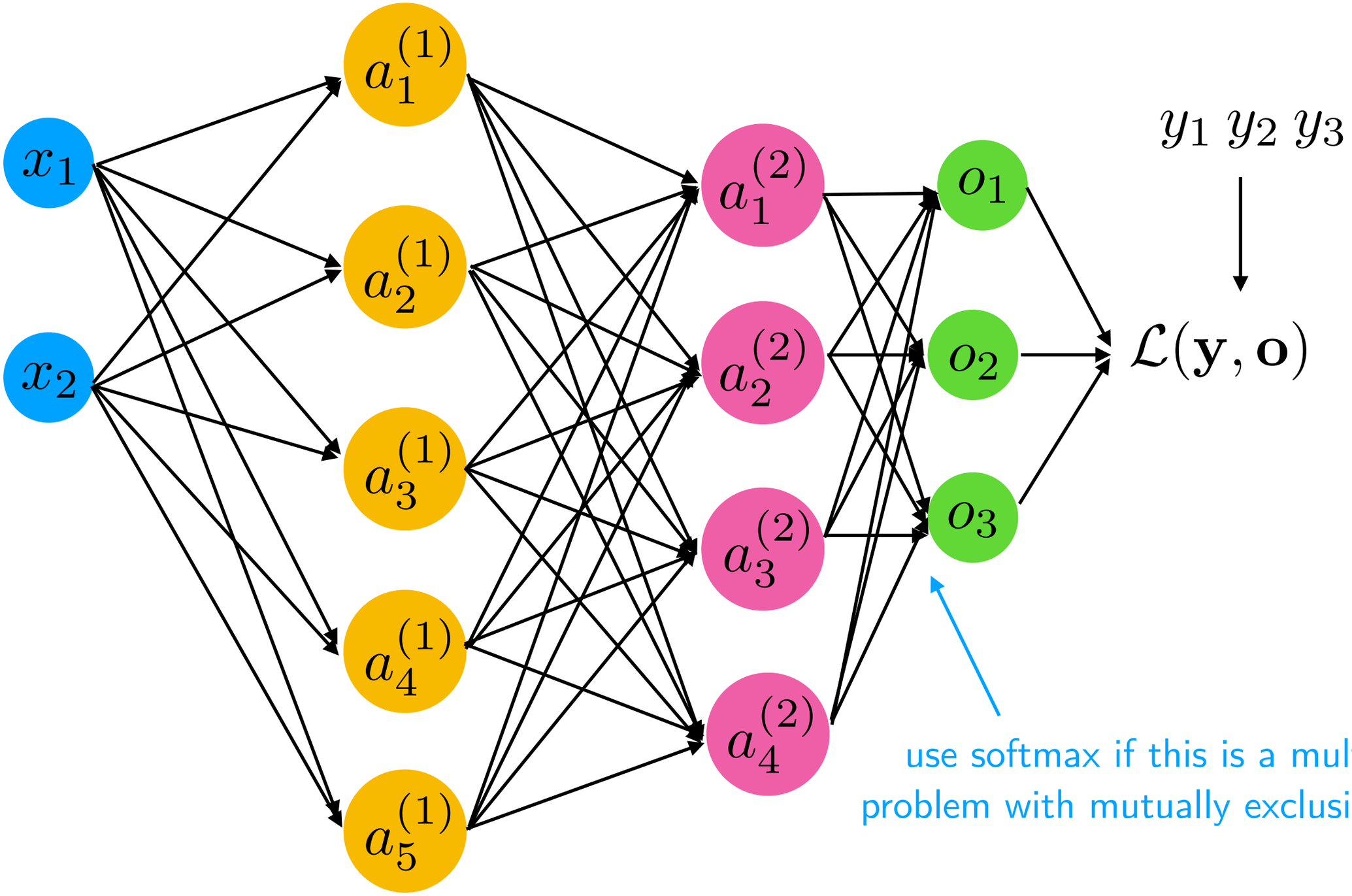
A more common counting/naming scheme, because then a perceptron/Adaline/ logistic regression model can be called a "1-layer neural network"



Graph with Fully-Connected Layers = Multilayer Perceptron



Graph with Fully-Connected Layers = Multilayer Perceptron

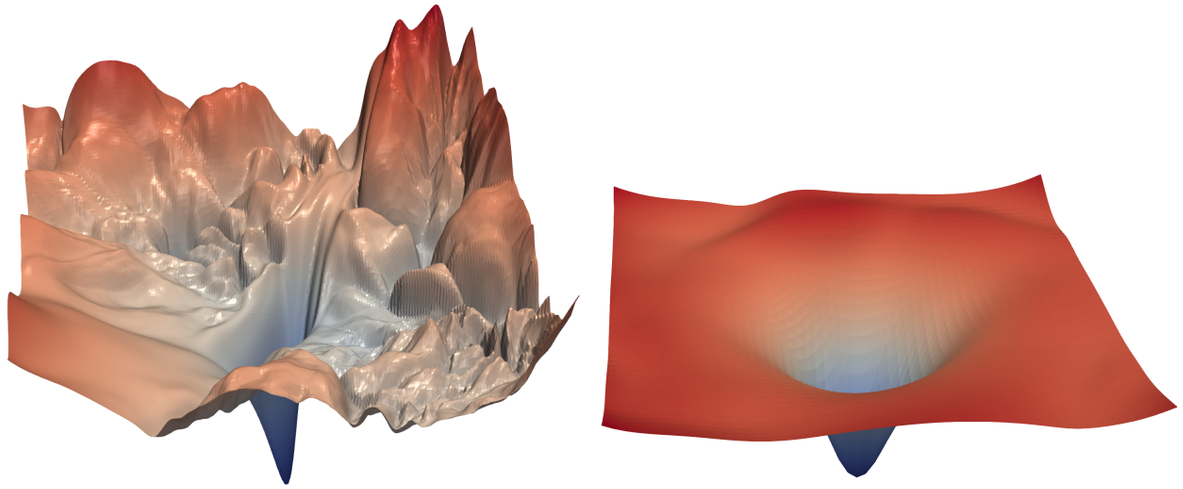


Note That the Loss is Not Convex Anymore

- Linear regression, Adaline, Logistic Regression, and Softmax Regression had convex loss functions with respect to the weights

- This is not the case anymore; 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 setting the same)

- In practice though, we WANT to explore different starting weights, however, because some lead to better solutions than others



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

Image Source: 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).

About Softmax & Sigmoid in the Output Layer and Issues with MSE

- Sigmoid activation + MSE has the problem of very flat gradients when the output is very wrong i.e., 10^{-5} probability and class label 1

$$\frac{\partial \mathcal{L}}{\partial w_j} = -\frac{2}{n}(\mathbf{y} - \mathbf{a}) \odot \sigma(\mathbf{z}) \odot (1 - \sigma(\mathbf{z}))\mathbf{x}_j^\top$$ (derivative for sigmoid + MSE neuron)

- Softmax (forces network to learn probability distribution over labels) in output layer is better than sigmoid because of the mutually exclusive labels as discussed in the Softmax lecture; hence, in output layer, softmax is usuall better than sigmoid

Activation Functions

Question: What happens if we don't use non-linear activation functions?

<START> Brief PyTorch Recap from L06

Objected-Oriented vs Functional* API

*Note that with "functional" I mean "functional programming" (one paradigm in CS)

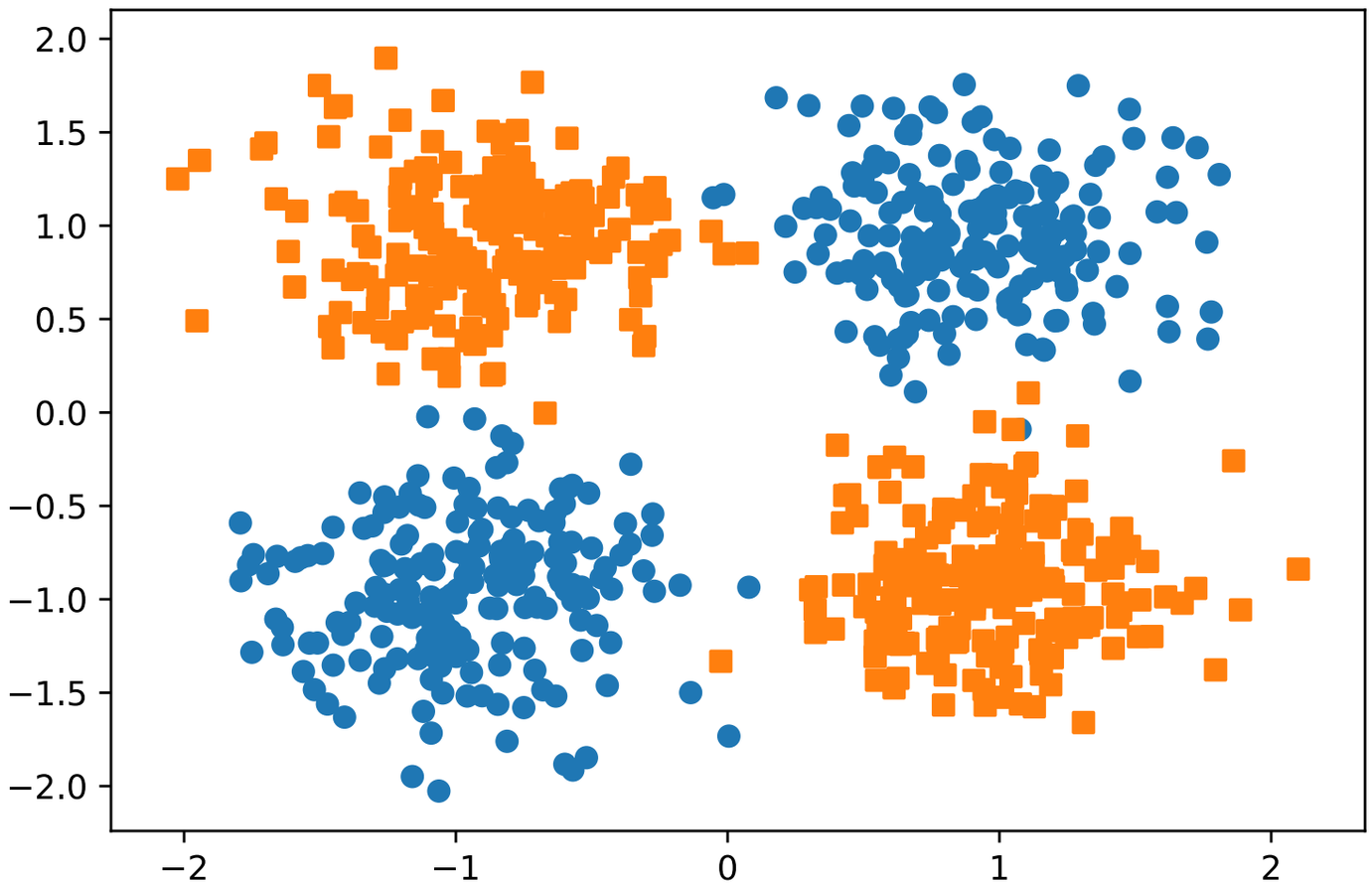
```
import torch.nn.functional as F


class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                          num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Unnecessary because these functions don't need to store a state but maybe helpful for keeping track of order of ops (when implementing "forward")

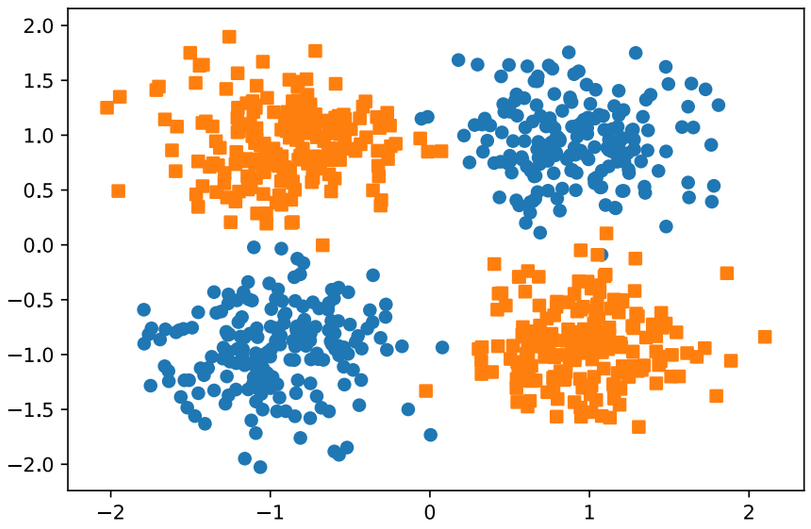
```
class MultilayerPerceptron(torch.nn.Module):

    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)

        self.relu1 = torch.nn.ReLU()

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)

        self.relu2 = torch.nn.ReLU()

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                          num_classes)

        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        out = self.linear_1(x)
        out = self.relu1(out)
        out = self.linear_2(out)
        out = self.relu2(out)
        logits = self.linear_out(out)
        probas = self.softmax(logits, dim=1)
        return logits, probas
```

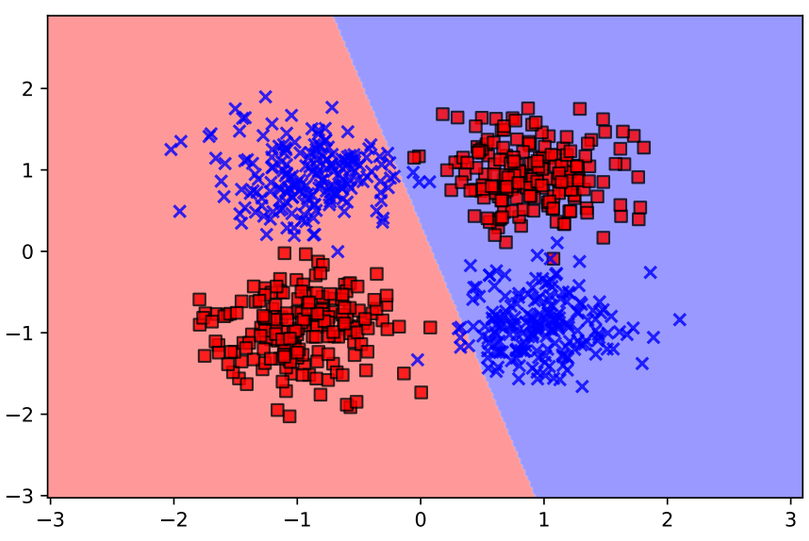
Brief PyTorch Recap <END>

Solving the XOR Problem with Non-Linear Activations

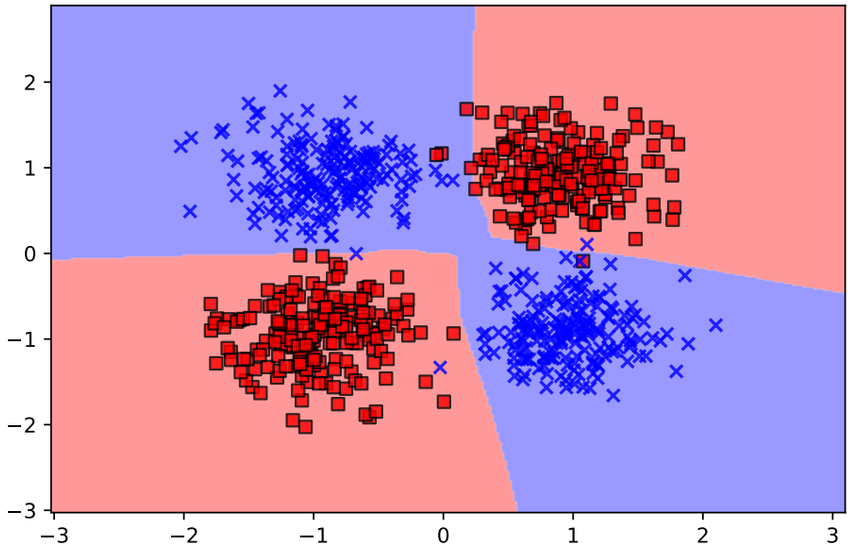


Solving the XOR Problem with Non-Linear Activations





1-hidden layer MLP
with linear activation function

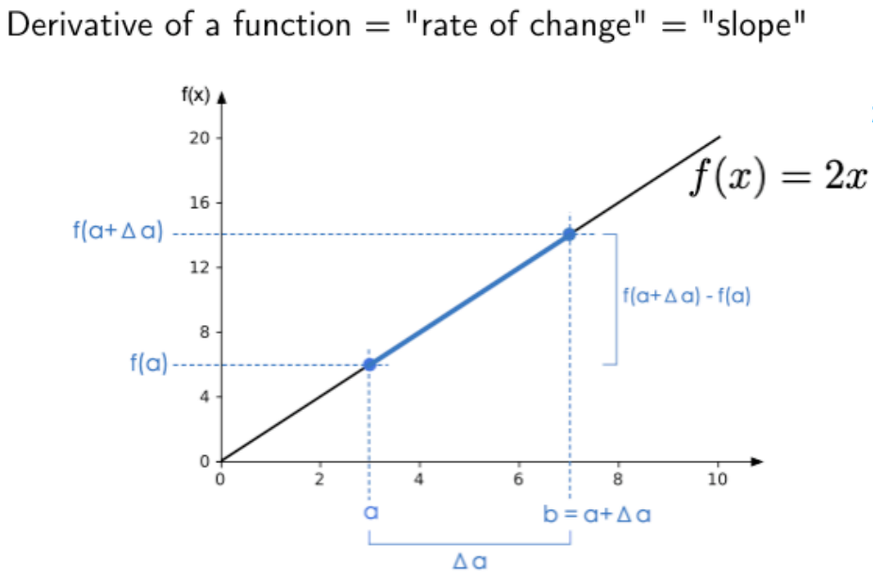


1-hidden layer MLP
with non-linear activation function (ReLU)

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

Gradient Checking

- Back in the day, we usually checked our gradients manually during debugging (note that this is super slow!)



$$\text{Slope} = \frac{f(a+\Delta a) - f(a)}{a + \Delta a - a} = \frac{f(a+\Delta a) - f(a)}{\Delta a}$$

(remember this from the calculus refresher section?)

Usually, a centered version works better, where epsilon is a very small value:

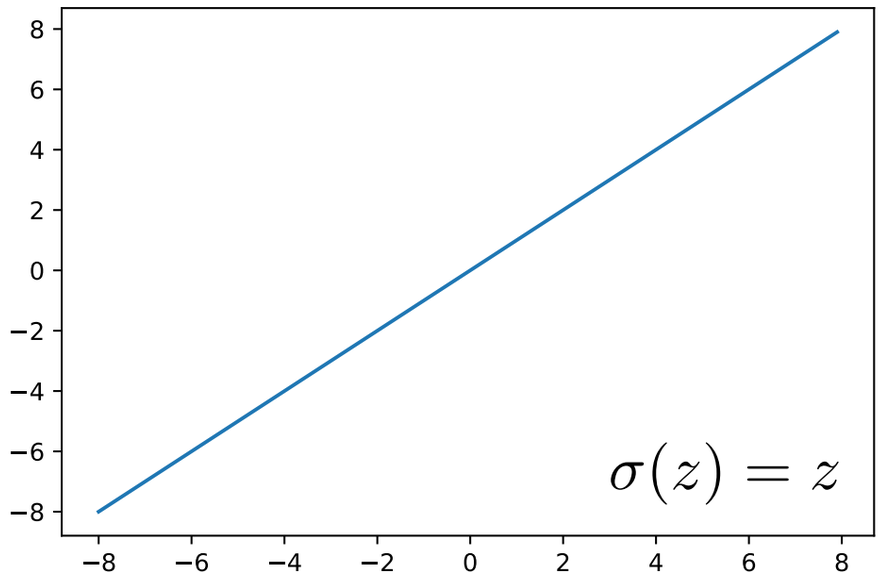
$$\frac{\mathcal{L}\left(w_{i,j}^{(l)} + \varepsilon\right) - \mathcal{L}\left(w_{i,j}^{(l)} - \varepsilon\right)}{2\varepsilon}$$

(then compare this with the symbolic gradient and compute the difference, e.g., via L2 norm)

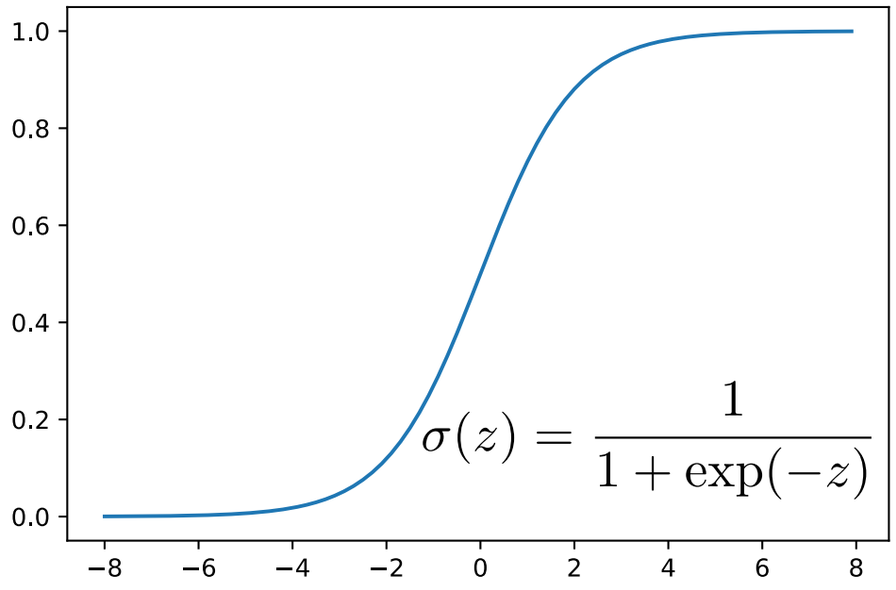
- Rarely done in practice anymore because we usually nowadays use autograd anyway, due to the complexity of deep neural networks

A Selection of Common Activation Functions (1)

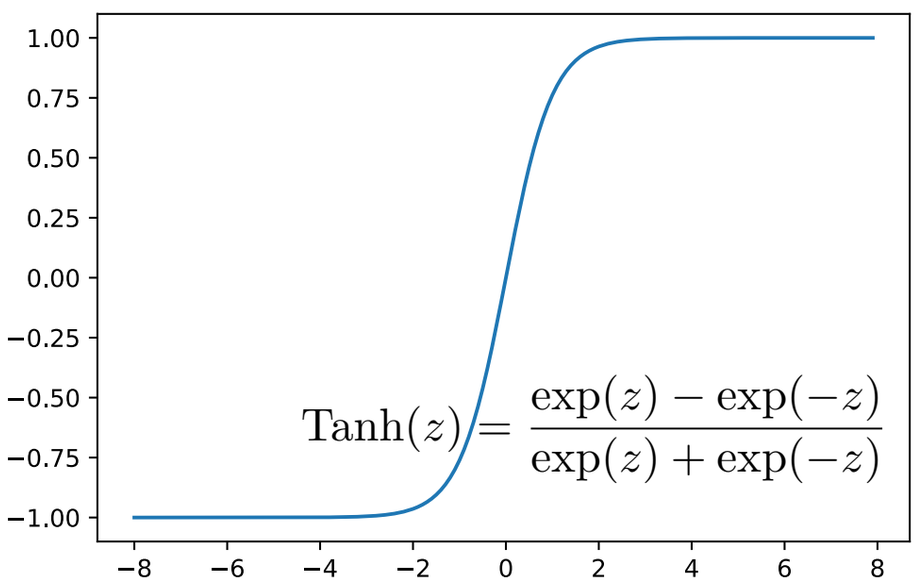
Identity



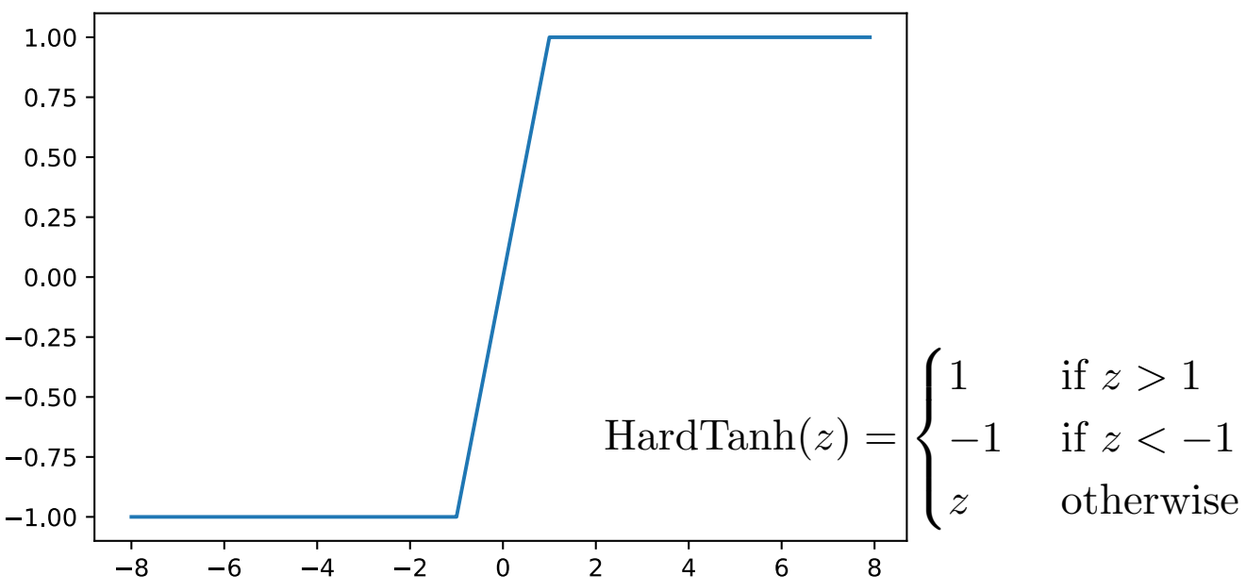
(Logistic) Sigmoid



Tanh ("tanH")



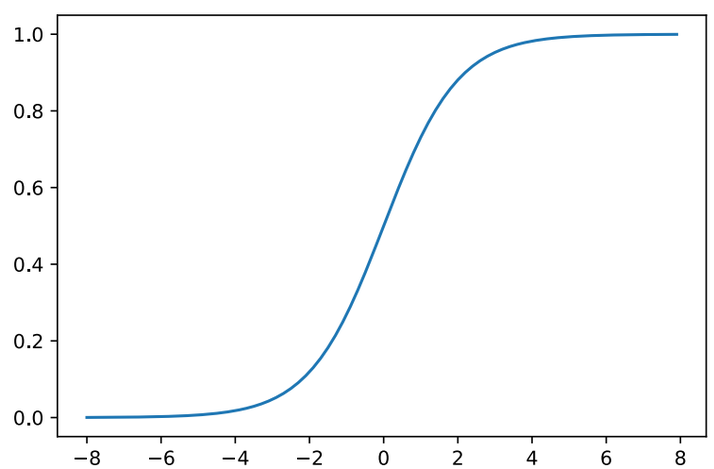
Hard Tanh



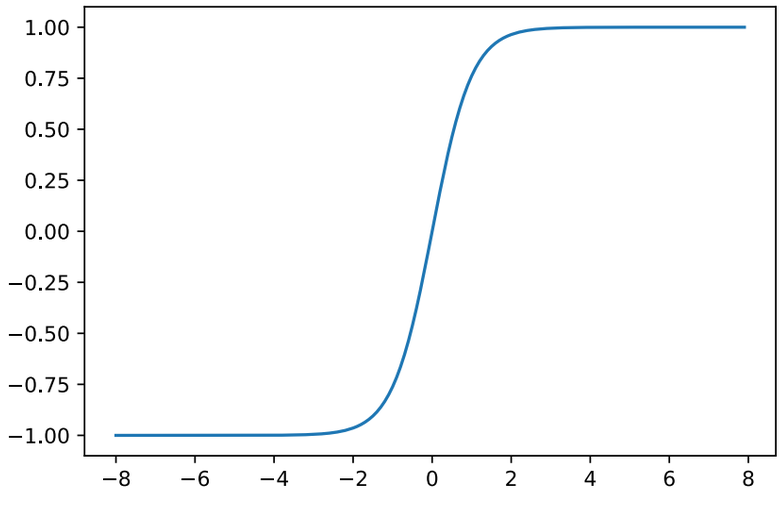
A Selection of Common Activation Functions (1)

- Advantages of Tanh
- Mean centering
- Positive and negative values
- Larger gradients

(Logistic) Sigmoid



Tanh ("tanH")



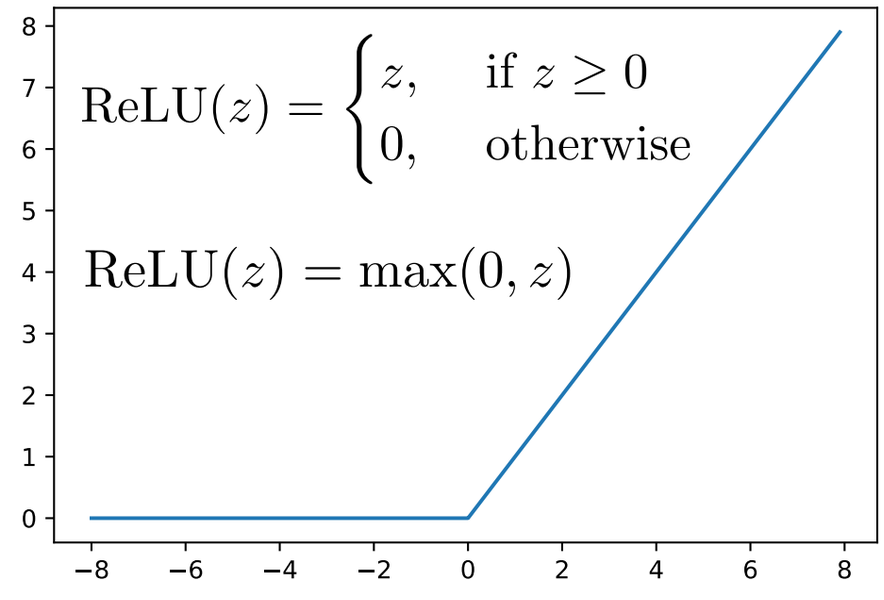
Additional tip: Also good to normalize inputs to mean zero and use random weight initialization with avg. weight centered at zero

Also simple derivative:

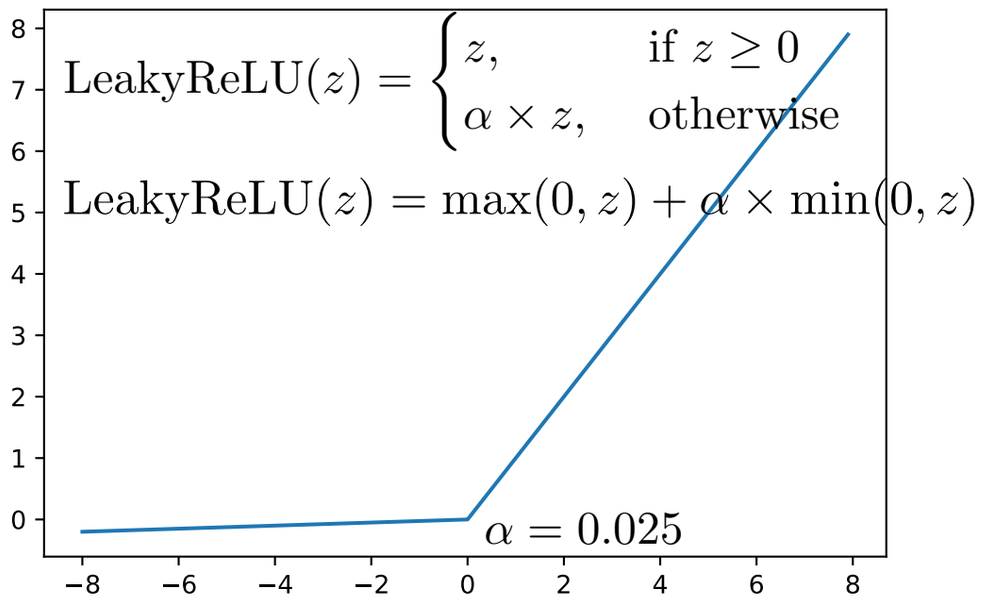
$$\frac{d}{dz}Tanh(z) = 1 - Tanh(z)^2$$

A Selection of Common Activation Functions (2)

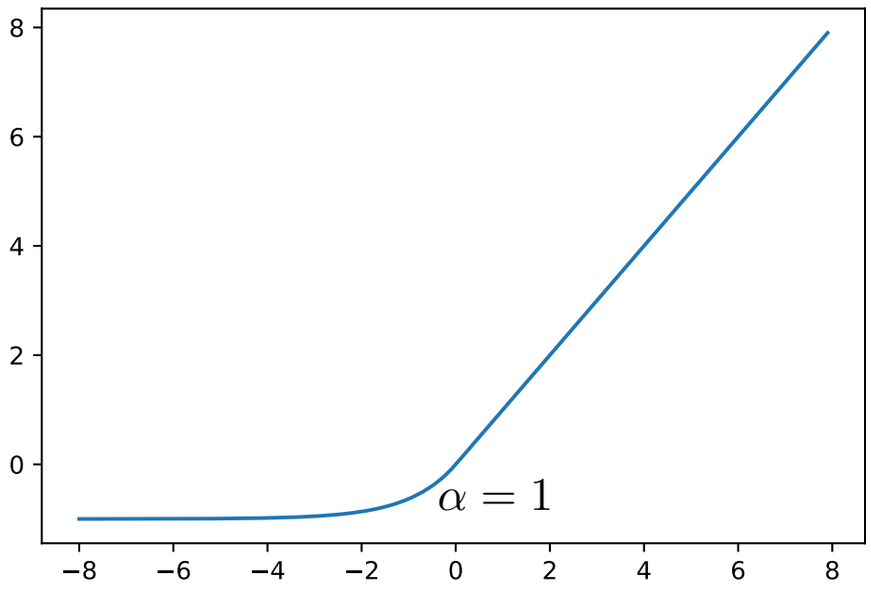
ReLU (Rectified Linear Unit)



Leaky ReLU



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

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

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

Multilayer Perceptron Architecture

Nonlinear Activation Functions

Multilayer Perceptron Code Examples

Overfitting and Underfitting

Cats & Dogs and Custom Data Loaders

Multilayer Perceptron with Sigmoid Activation and MSE Loss (from scratch)

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

```
def backward(self, x, a_1, a_2, y):
```

```
    # Part 1: dLoss/dOutWeights
    ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
    ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
    ## for convenient re-use

    # input/output dim: [n_examples, n_classes]
    dloss_da2 = 2.*(a_2 - y_onehot) / y.size(0)

    # input/output dim: [n_examples, n_classes]
    da2_dz2 = a_2 * (1. - a_2) # sigmoid derivative

    # output dim: [n_examples, n_classes]
    delta_out = dloss_da2 * da2_dz2 # "delta (rule) placeholder"

    # gradient for output weights

    # [n_examples, n_hidden]
    dz2__dw_out = a_1

    # input dim: [n_classlabels, n_examples] dot [n_examples, n_hidden]
    # output dim: [n_classlabels, n_hidden]
    dloss__dw_out = torch.mm(delta_out.t(), dz2__dw_out)
    dloss__db_out = torch.sum(delta_out, dim=0)


    ###################################
    # Part 2: dLoss/dHiddenWeights
    ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet * dHiddenNet/dWeight

    # [n_classes, n_hidden]
```

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

VS

Multilayer Perceptron with Softmax Activation and Cross Entropy Loss

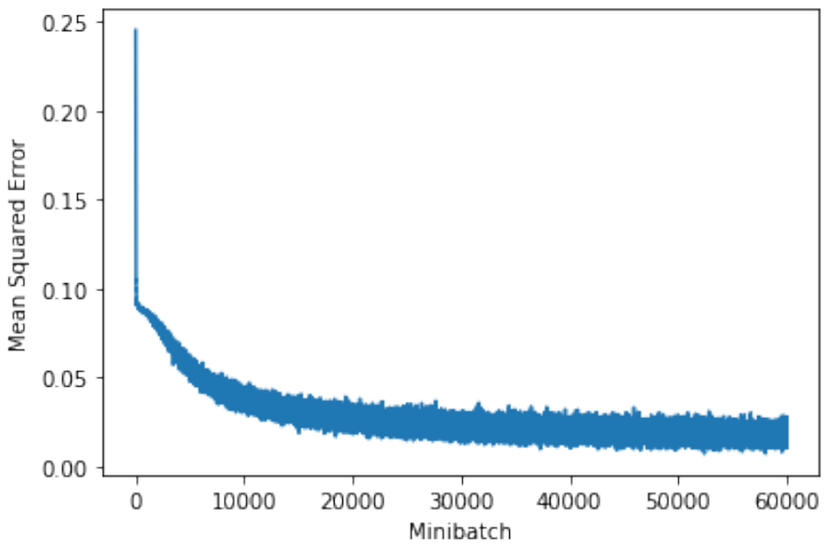
https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mlp-pytorch_sigmoid-crossentr.ipynb

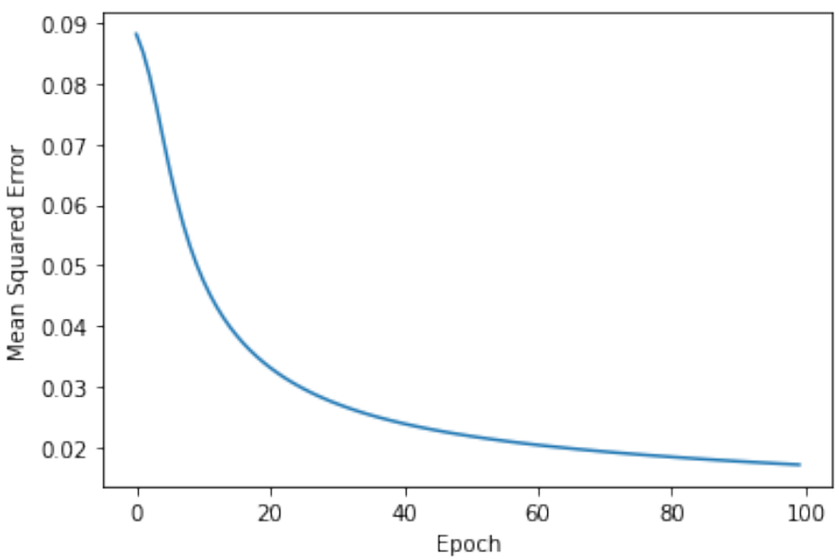
Multilayer Perceptron with Sigmoid Activation and MSE Loss

vs

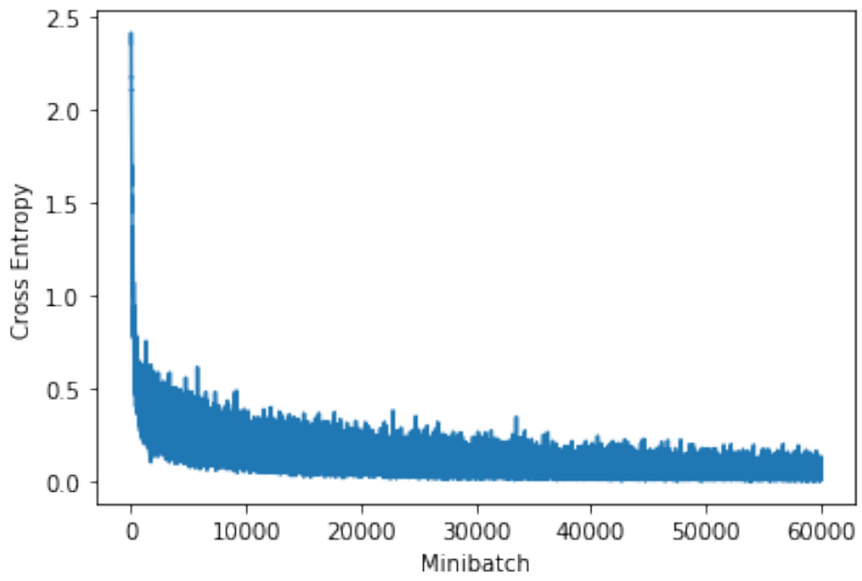
Multilayer Perceptron with Softmax Activation and Cross Entropy Loss

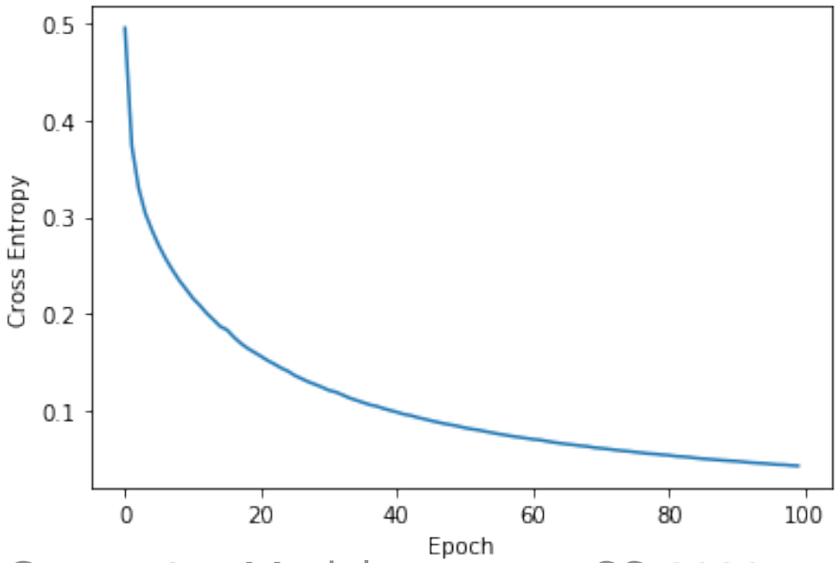
https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mlp-pytorch_sigmoid-mse.ipynb





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



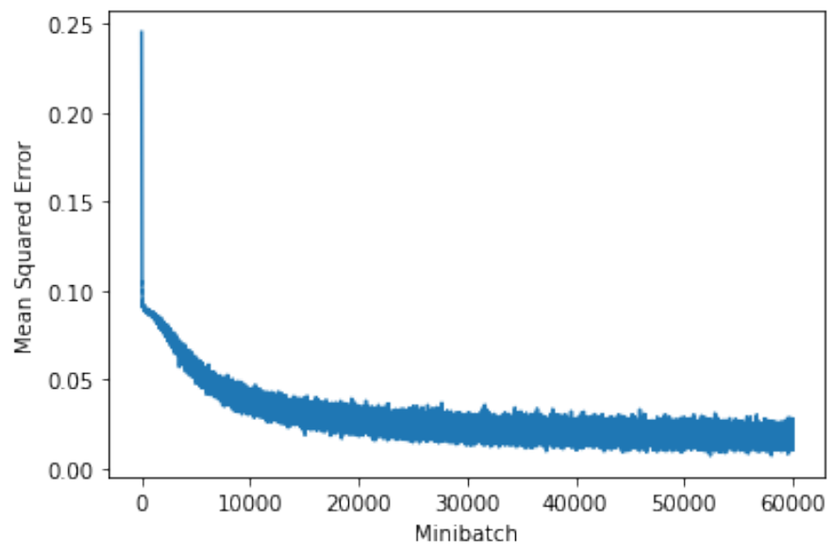


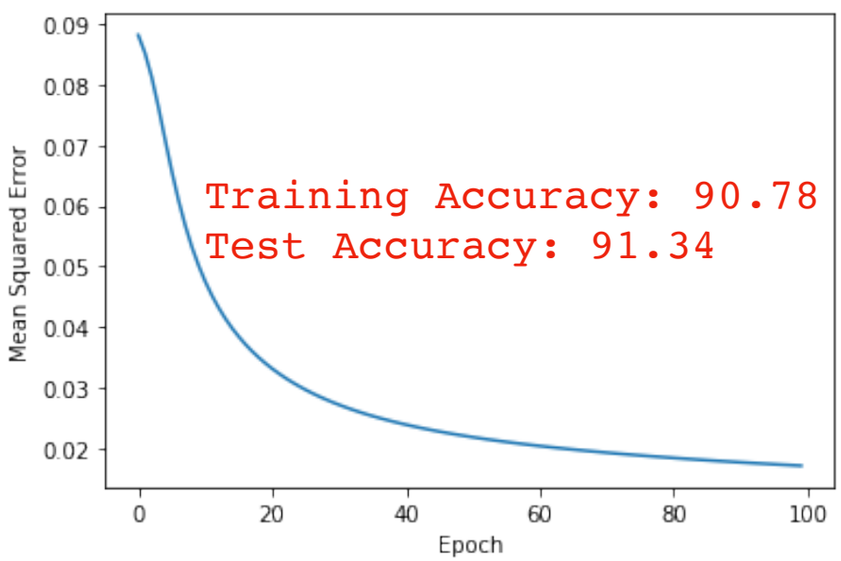
Multilayer Perceptron with Sigmoid Activation and MSE Loss

vs

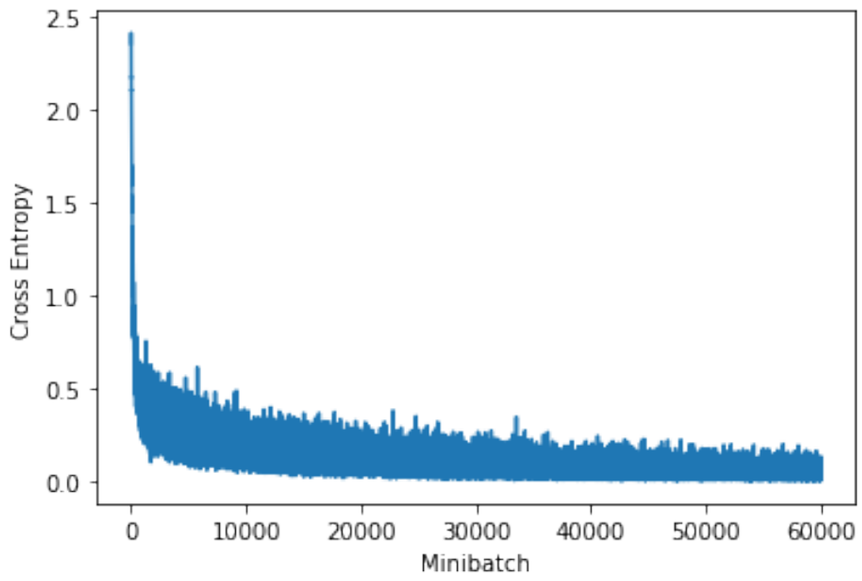
Multilayer Perceptron with Softmax Activation and Cross Entropy Loss

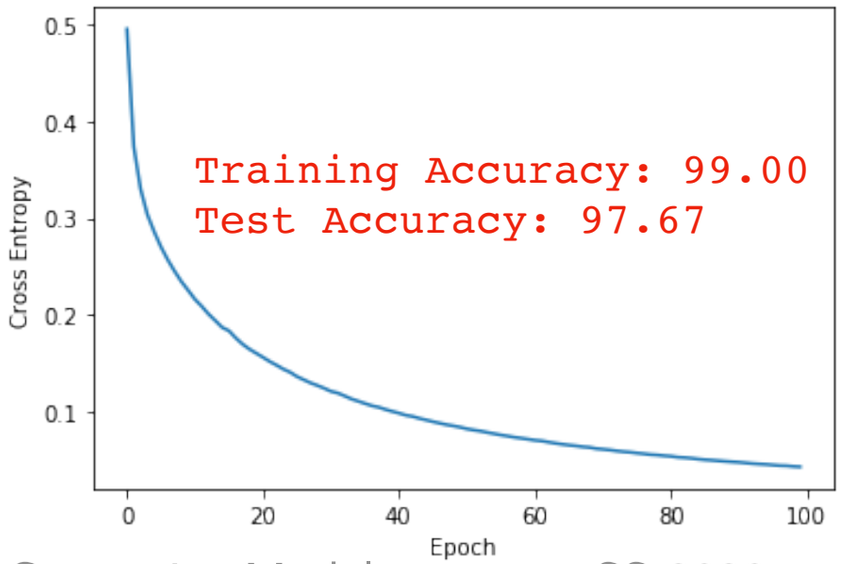
https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mlp-pytorch_sigmoid-mse.ipynb





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





Dead Neurons

- ReLU is probably the most popular activation function (simple to compute, fast, good results)
- But esp. ReLU neurons might "die" during training
- Can happen if, e.g., input is so large/small that net input is so small that ReLUs never recover (gradient 0 at $x < 0$)
- Not necessarily bad, can be considered as a form of regularization
- (compared to sigmoid/Tanh, ReLU suffers less from vanishing gradient problem but can more easily "explode")

White vs Deep Architectures (Breadth vs Depth)

MLP's with one (large) hidden unit are universal function approximators [1-3] already 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] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2(4), 303–314. doi:10.1007/BF02551274

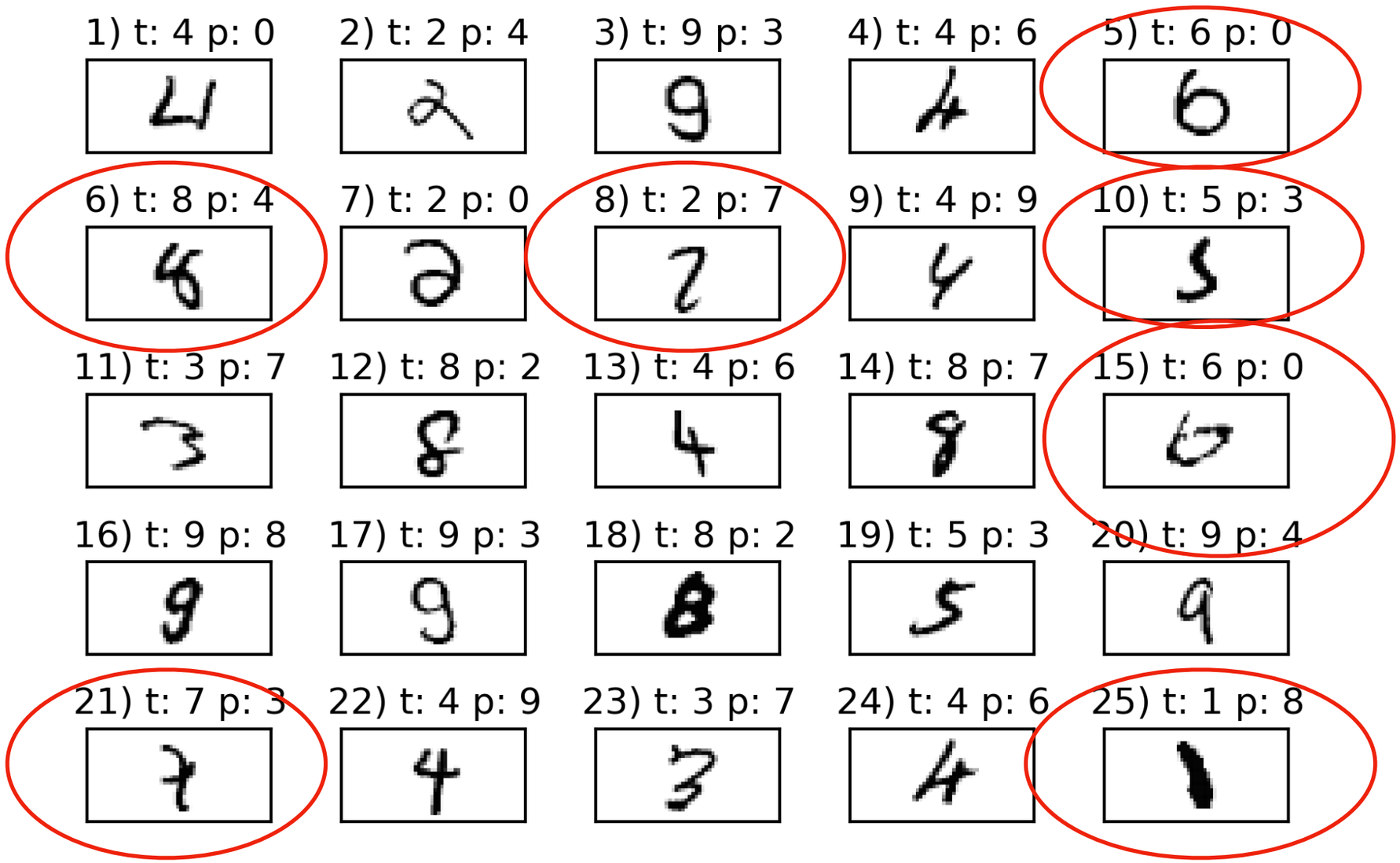
[3] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, *2*(5), 359-366.

White vs Deep Architectures (Breadth vs Depth)

- **Can achieve the same expressiveness with more layers but fewer parameters (combinatorics); fewer parameters => less overfitting**
- **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)**

Model Evaluation

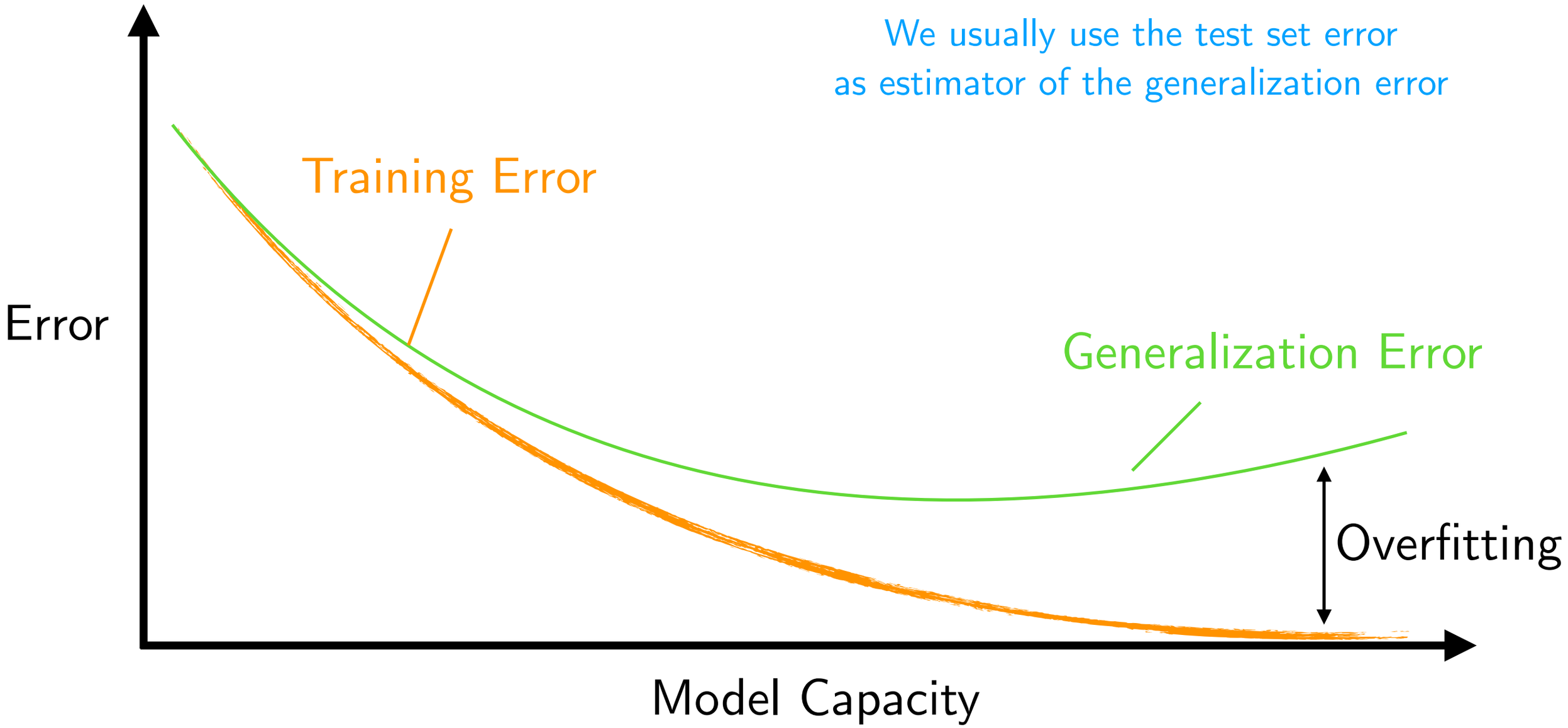
Recommended Practice: Looking at Some Failure Cases



Failure cases of a ~93% accuracy (not very good, but beside the point)
2-layer (1-hidden layer) MLP on MNIST
(where *t=target* class and *p=predicted class*)

Overfitting and Underfitting

We usually use the test set error
as estimator of the generalization error



Bias-Variance Decomposition

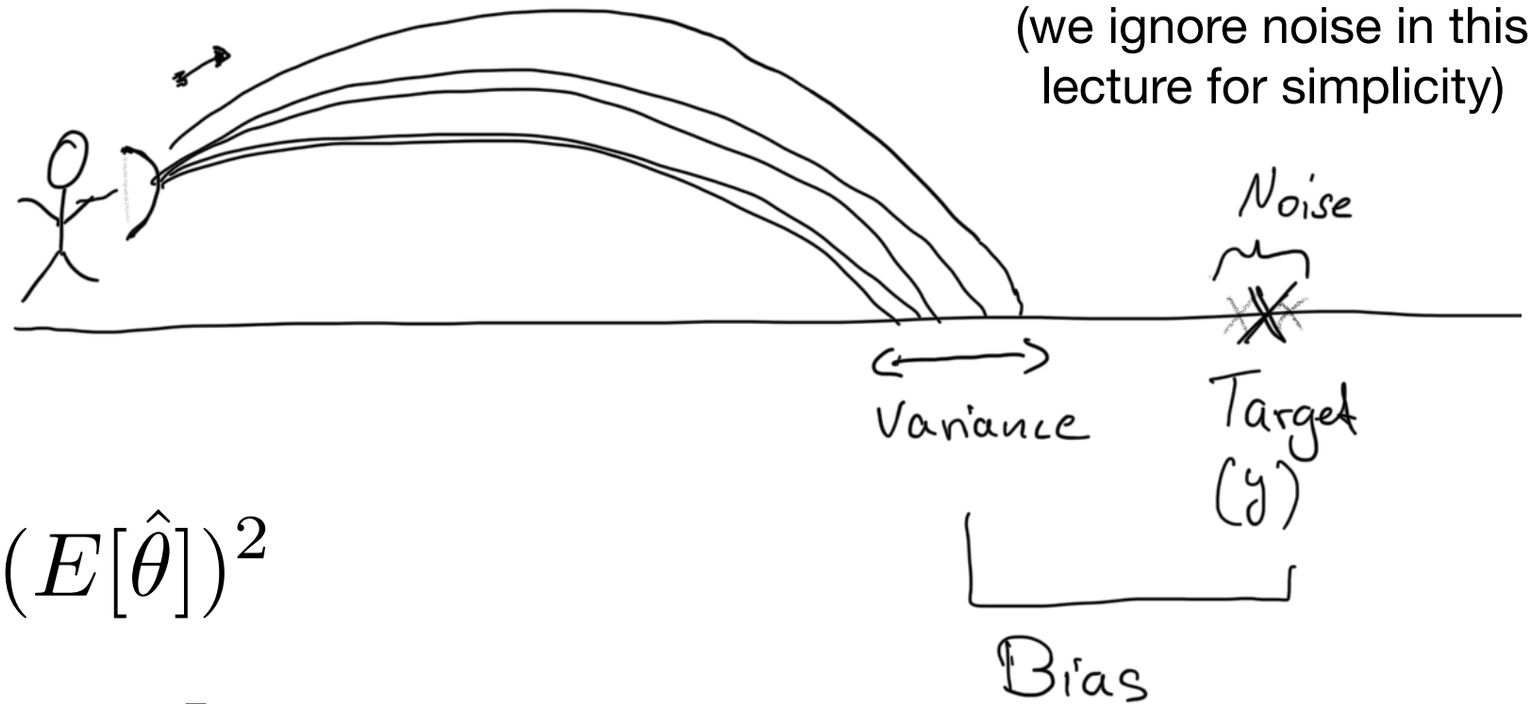
General Definition:

$$\text{Bias}_\theta[\hat{\theta}] = E[\hat{\theta}] - \theta$$

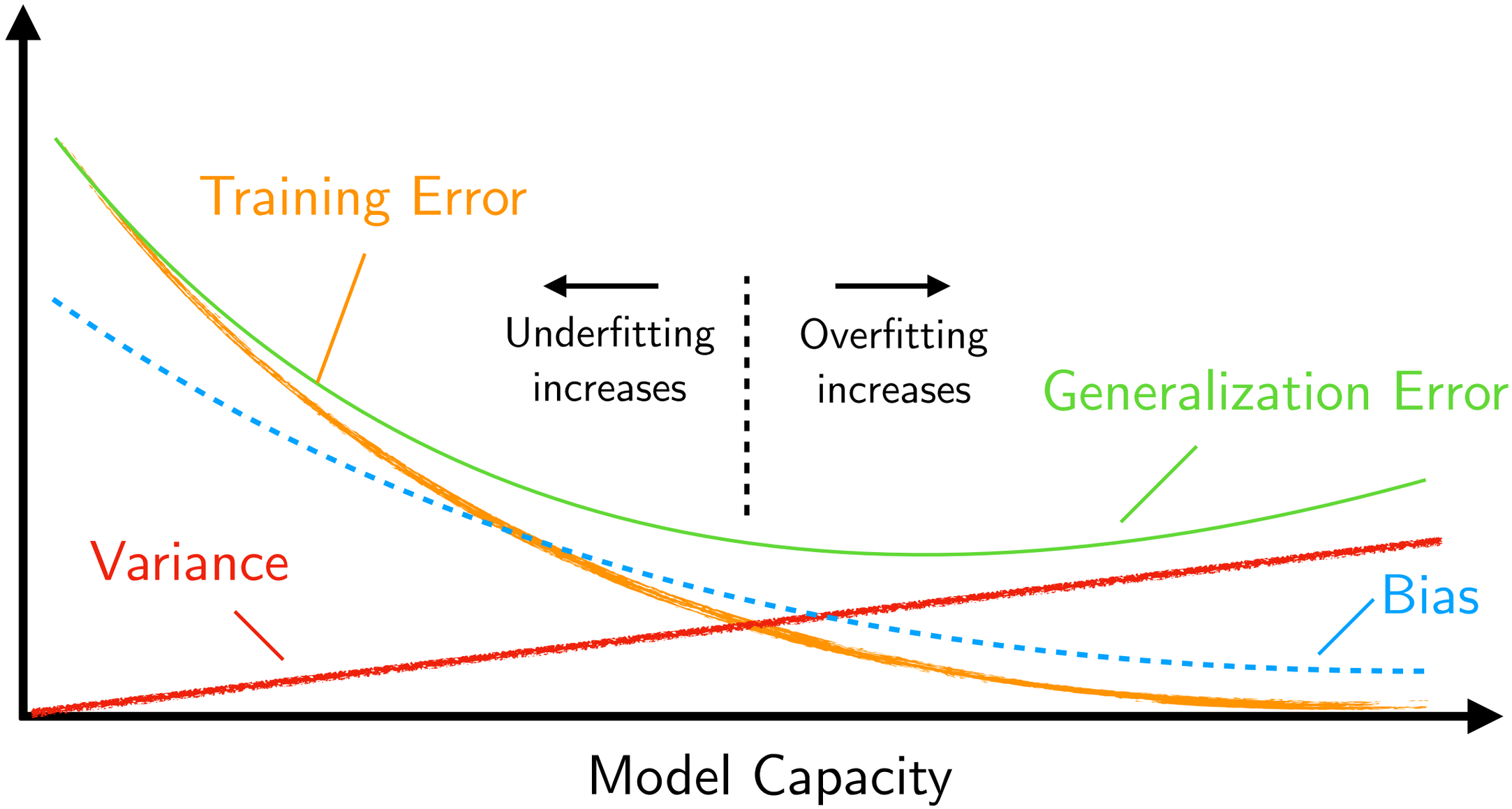
$$\text{Var}_\theta[\hat{\theta}] = E\left[\hat{\theta}^2\right] - (E[\hat{\theta}])^2$$

$$\text{Var}_\theta[\hat{\theta}] = E\left[(E[\hat{\theta}] - \hat{\theta})^2\right]$$

Intuition:



Bias & Variance vs Overfitting & Underfitting



Further Reading Material

- A more detailed understanding of bias and variance is not mandatory for this class

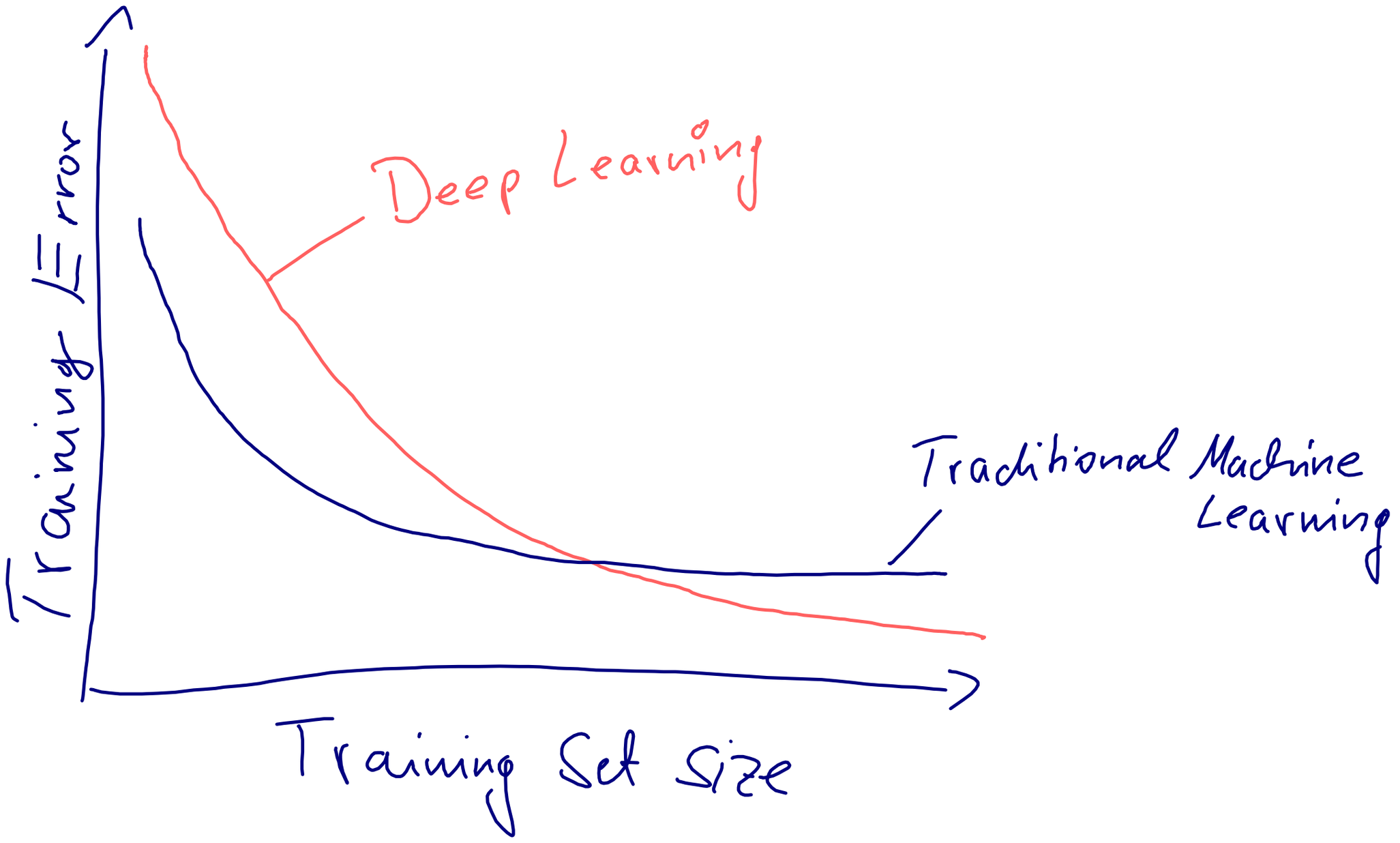
- Also, train/valid/test splits are usually sufficient for training & estimating the generalization performance in deep learning

- However, if you are interested, we covered these topics in the model evaluation lectures in my STAT 479 class. If you are interested, you can find a compilation of the lecture material here:

  Raschka, S. (2018). Model evaluation, model selection, and algorithm selection in machine learning. *arXiv preprint arXiv:1811.12808*.

  https://arxiv.org/pdf/1811.12808.pdf

Deep Learning Works Best with Large Datasets



Bias & Variance vs Overfitting & Underfitting

When reading DL resources, you'll notice many researchers use *bias* and *variance* to describe *underfitting* and *overfitting* (they are related but not the same!)

Multilayer Perceptron Architecture

Nonlinear Activation Functions

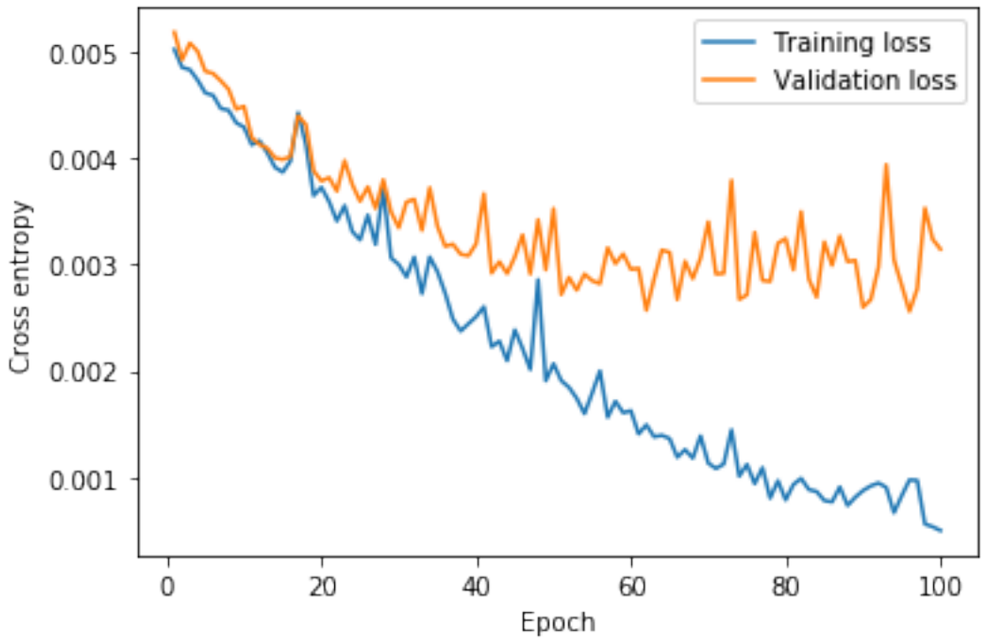
Multilayer Perceptron Code Examples

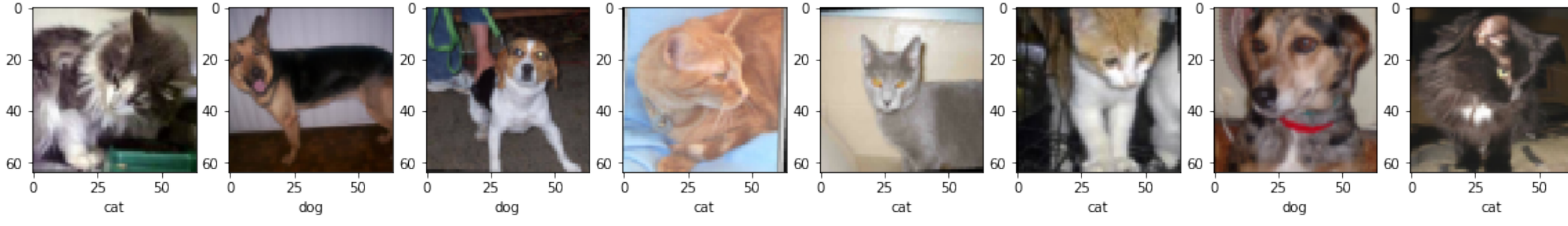
Overfitting and Underfitting

Cats & Dogs and Custom Data Loaders

VGG16 Convolutional Neural Network for Kaggle's Cats and Dogs Images

A "real world" example






```
model.eval()
with torch.set_grad_enabled(False): # save memory during inference
    test_acc, test_loss = compute_accuracy_and_loss(model, test_loader, DEVICE)
    print(f'Test accuracy: {test_acc:.2f}%')
```

```
Test accuracy: 88.28%
```

Training/Validation/Test splits

Ratio depends on the dataset size, but a 80/5/15 split is usually a good idea

- Training set is used for training, it is not necessary to plot the training accuracy during training but it can be useful
- Validation set accuracy provides a rough estimate of the generalization performance (it can be optimistically biased if you design the network to do well on the validation set ("information leakage")
- Test set should only be used once to get an unbiased estimate of the generalization performance

Training/Validation/Test splits

```
Epoch: 001/100 | Batch 000/156 | Cost: 1136.9125
Epoch: 001/100 | Batch 120/156 | Cost: 0.6327
Epoch: 001/100 Train Acc.: 63.35% | Validation Acc.: 62.12%
Time elapsed: 3.09 min
Epoch: 002/100 | Batch 000/156 | Cost: 0.6675
Epoch: 002/100 | Batch 120/156 | Cost: 0.6640
Epoch: 002/100 Train Acc.: 66.05% | Validation Acc.: 66.32%
Time elapsed: 6.15 min
Epoch: 003/100 | Batch 000/156 | Cost: 0.6137
Epoch: 003/100 | Batch 120/156 | Cost: 0.6311
Epoch: 003/100 Train Acc.: 65.82% | Validation Acc.: 63.76%
Time elapsed: 9.21 min
Epoch: 004/100 | Batch 000/156 | Cost: 0.5993
Epoch: 004/100 | Batch 120/156 | Cost: 0.5832
Epoch: 004/100 Train Acc.: 66.75% | Validation Acc.: 64.52%
Time elapsed: 12.27 min
Epoch: 005/100 | Batch 000/156 | Cost: 0.5918
Epoch: 005/100 | Batch 120/156 | Cost: 0.5747
Epoch: 005/100 Train Acc.: 68.29% | Validation Acc.: 67.00%
Time elapsed: 15.33 min
...
```

Parameters vs Hyperparameters

Parameters

- weights (weight parameters)
- biases (bias units)

Hyperparameters

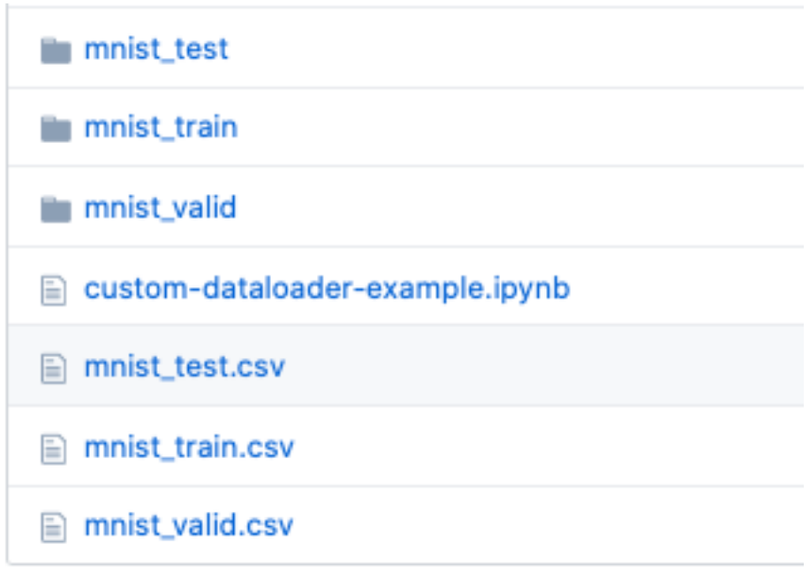
- minibatch size
- data normalization schemes
- number of epochs
- number of hidden layers
- number of hidden units
- learning rates
- (random seed, why?)
- loss function
- various weights (weighting terms)
- activation function types
- regularization schemes (more later)
- weight initialization schemes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

(Mostly no scientific explanation, mostly engineering;
need to try many things -> "graduate student descent")

Custom DataLoader Classes ...

- Example showing how you can create your own data loader to efficiently iterate through your own collection of images
  (pretend the MNIST images there are some custom image collection)

  https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/custom-dataloader/custom-dataloader-example.ipynb




```
import torch
from PIL import Image
from torch.utils.data import Dataset
import os


class MyDataset(Dataset):

    def __init__(self, csv_path, img_dir, transform=None):

        df = pd.read_csv(csv_path)
        self.img_dir = img_dir
        self.img_names = df['File Name']
        self.y = df['Class Label']
        self.transform = transform

    def __getitem__(self, index):
        img = Image.open(os.path.join(self.img_dir,
                                      self.img_names[index]))

        if self.transform is not None:
            img = self.transform(img)

        label = self.y[index]
        return img, label

    def __len__(self):
        return self.y.shape[0]
```

DataLoader with Train/Validation/Test splits

https://github.com/rasbt/stat453-deep-learning-ss20/blob/master/L08-mlp/code/mnist-validation-split.ipynb

```
# however, we can now choose a different transform method
valid_dataset = datasets.MNIST(root='data',
                               train=True,
                               transform=valid_transform,
                               download=False)

test_dataset = datasets.MNIST(root='data',
                              train=False,
                              transform=valid_transform,
                              download=False)

train_loader = DataLoader(train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=4,
                          sampler=train_sampler)

valid_loader = DataLoader(valid_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=4,
                          sampler=valid_sampler)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         num_workers=4,
                         shuffle=False)
```

Reading Assignments

Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning, *pp. 1-15*

https://arxiv.org/pdf/1811.12808.pdf

Additional Reading Material

http://www.deeplearningbook.org/contents/mlp.html

Good news: We can solve non-linear problems now! Yay! :)

Bad news: Our multilayer neural nets have a lot of parameters now, and it's easy to overfit the the data! :(

Next Lecture

Regularization for deep neural networks to prevent overfitting