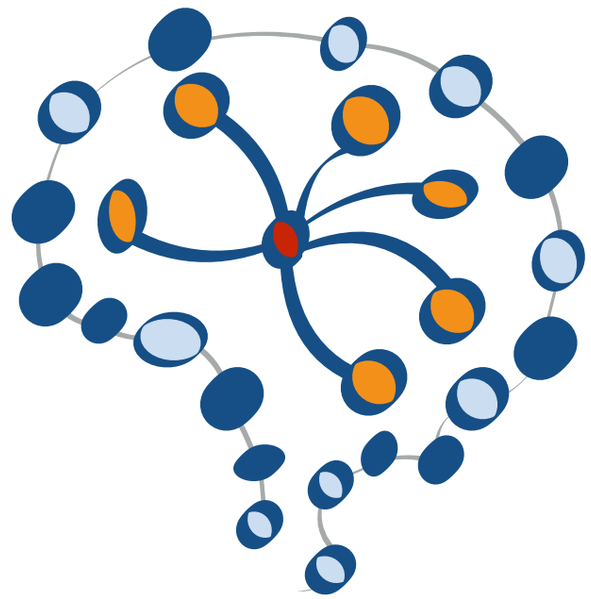
STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching

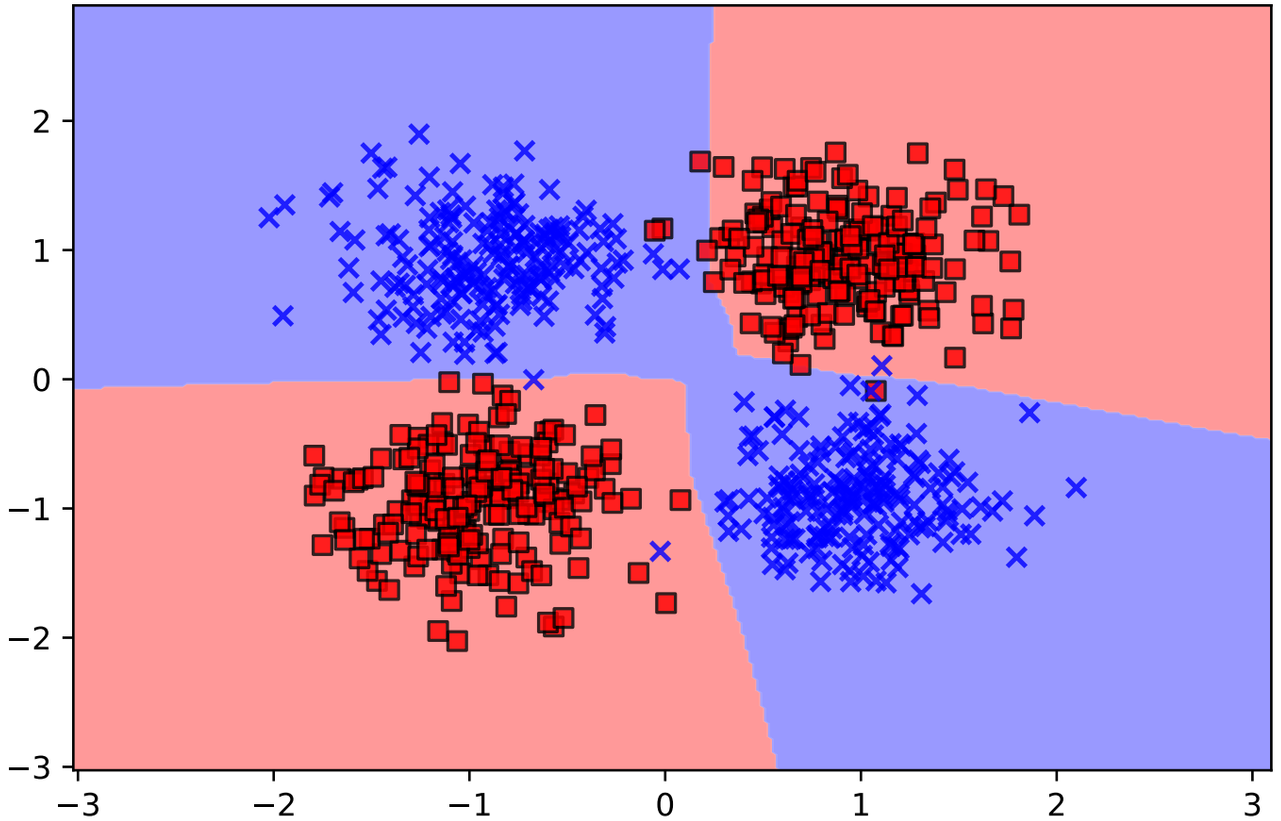


Lecture 09

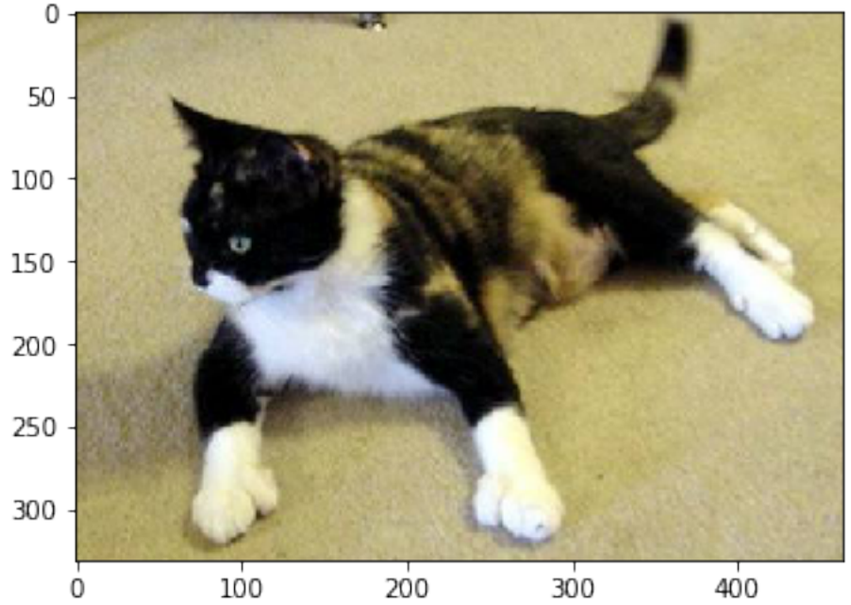
Multilayer Perceptrons

Today

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



... and talk about cats!



Lecture Overview

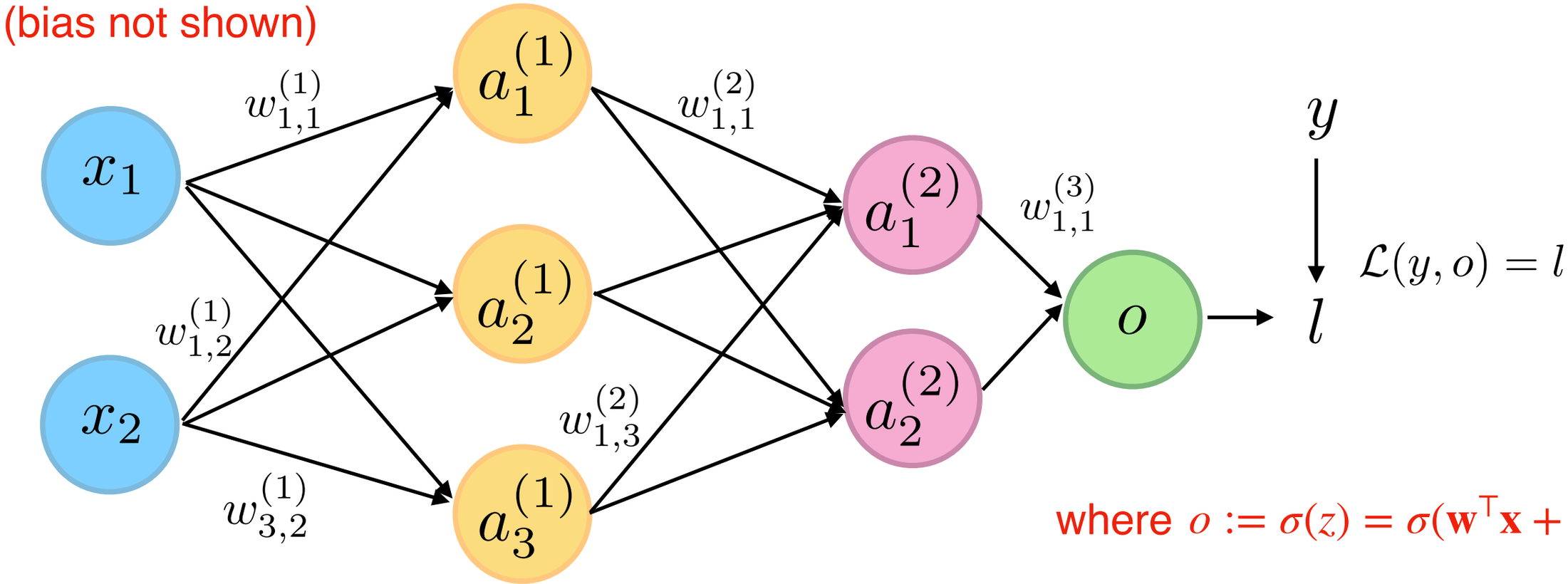
1. Multilayer Perceptron Architecture
2. Nonlinear Activation Functions
3. Multilayer Perceptron Code Examples
4. Overfitting and Underfitting
5. Cats & Dogs and Custom Data Loaders

Fully-connected feedforward neural networks with one or more hidden layers

1. Multilayer Perceptron Architecture
2. Nonlinear Activation Functions
3. Multilayer Perceptron Code Examples
4. Overfitting and Underfitting
5. Cats & Dogs and Custom Data Loaders

Computation Graph with Multiple Fully-Connected Layers = Multilayer Perceptron

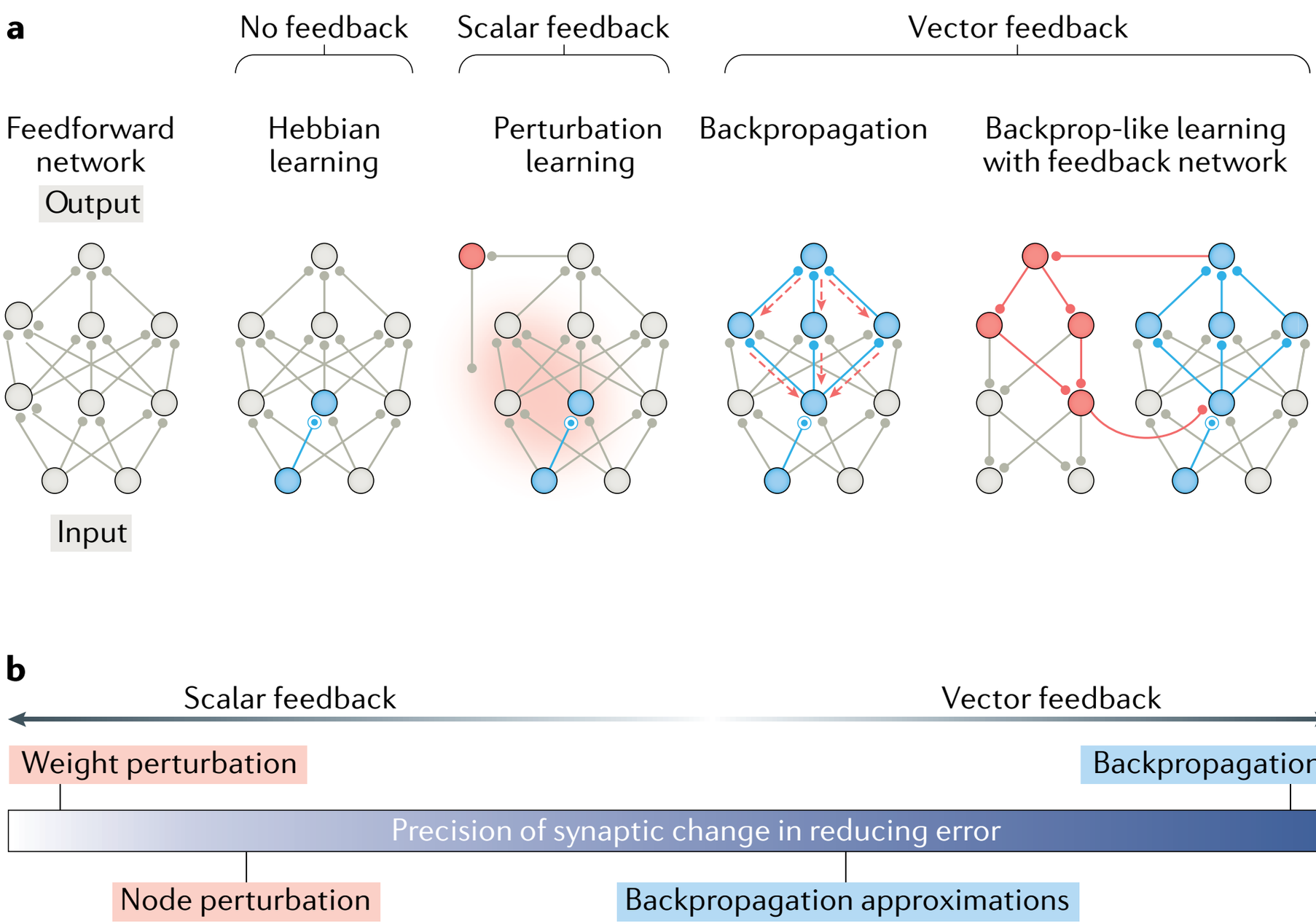
Nothing new, really



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

(Assume network for binary classification)

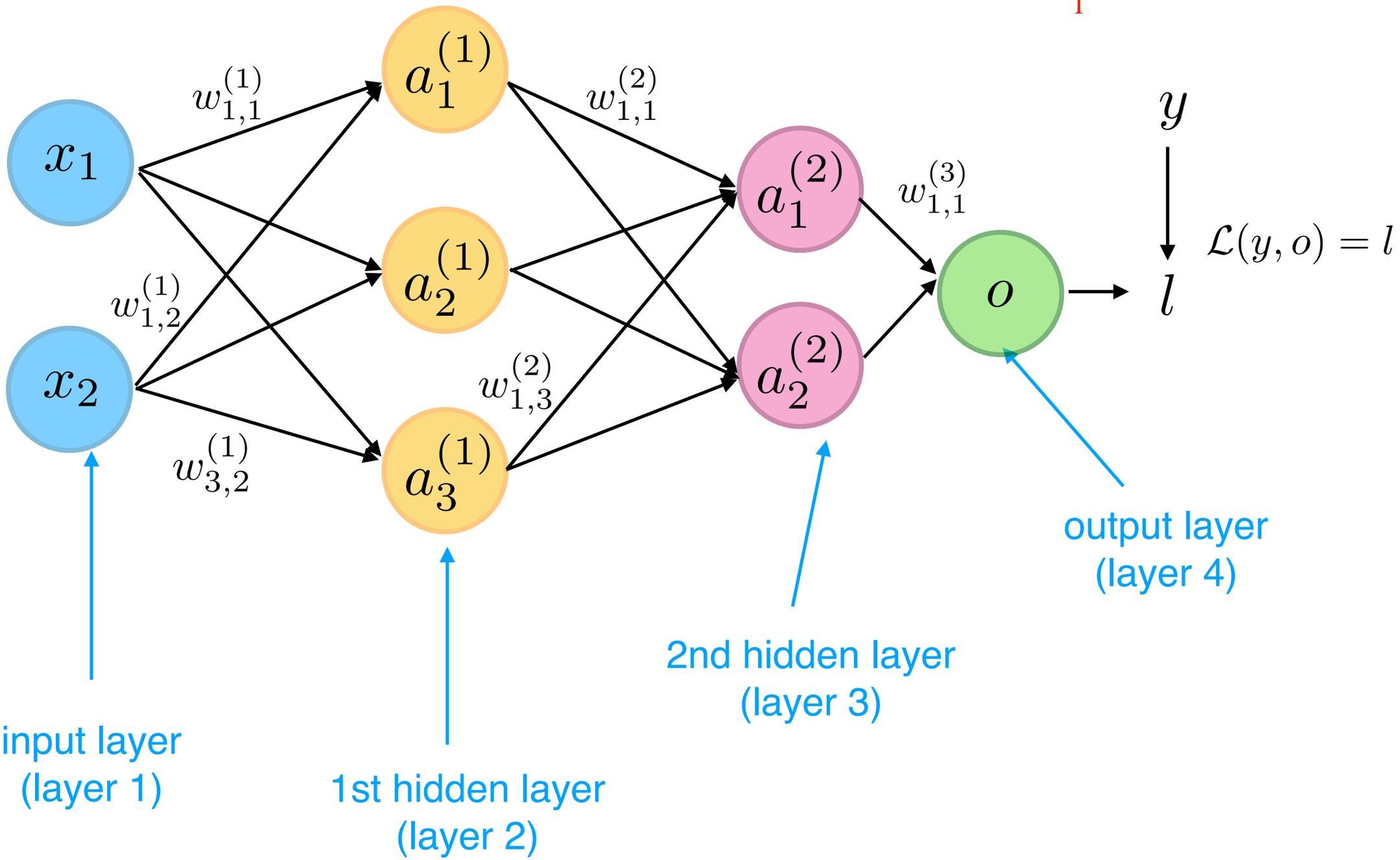
$$\frac{\partial l}{\partial w_{1,1}^{(1)}} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1^{(2)}} \cdot \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \cdot \frac{\partial a_1^{(1)}}{\partial w_{1,1}^{(1)}} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2^{(2)}}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



Lillicrap, T. P., Santoro, A., Marris, L., & Akerman, C. (n.d.). Backpropagation and the brain. *Nature Reviews Neuroscience*, 1–12. https://doi.org/10.1038/s41583-020-0277-3

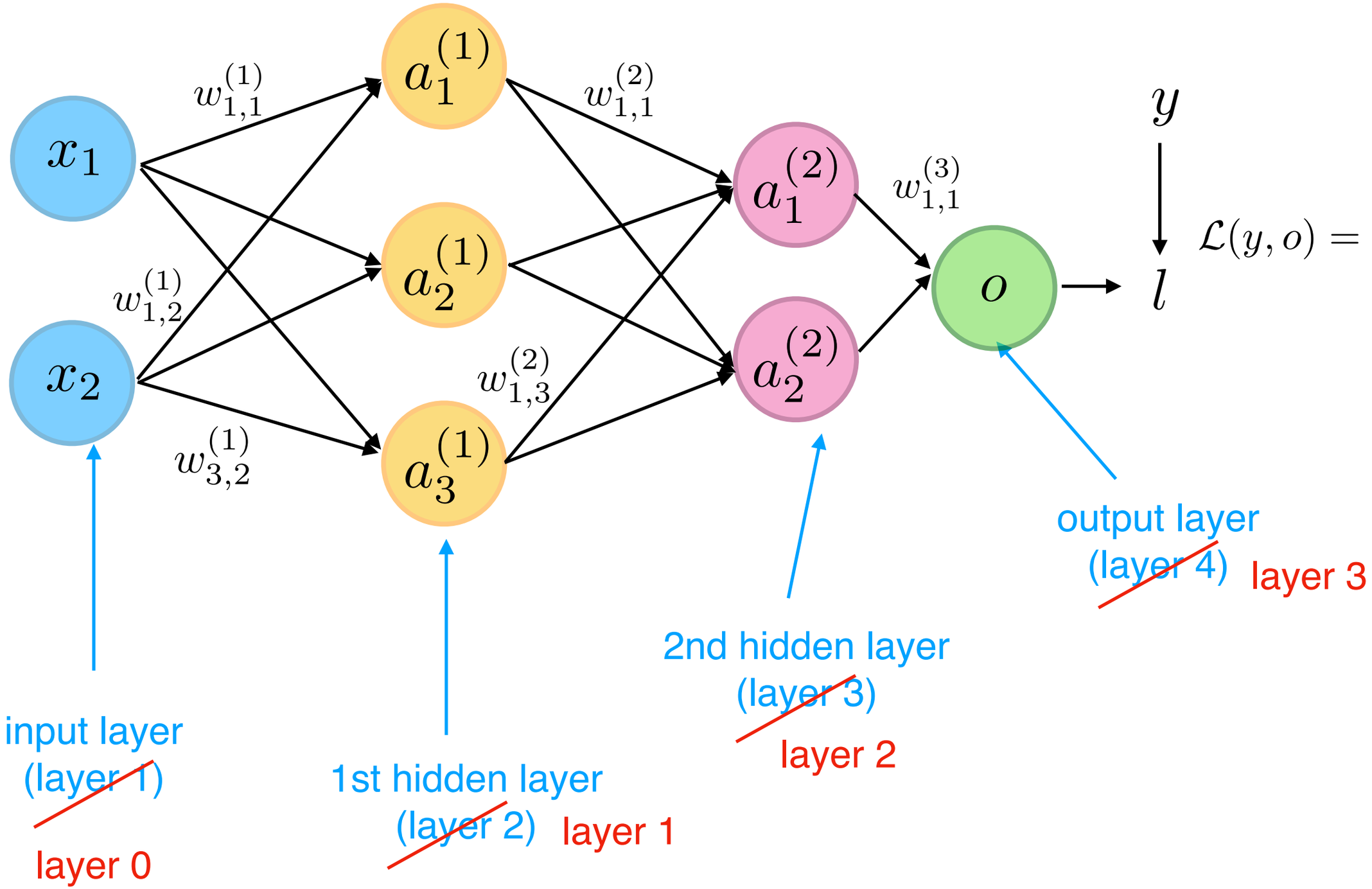
Computation Graph with Multiple Fully-Connected Layers = Multilayer Perceptron

here, we could also write o as $a_1^{(3)}$

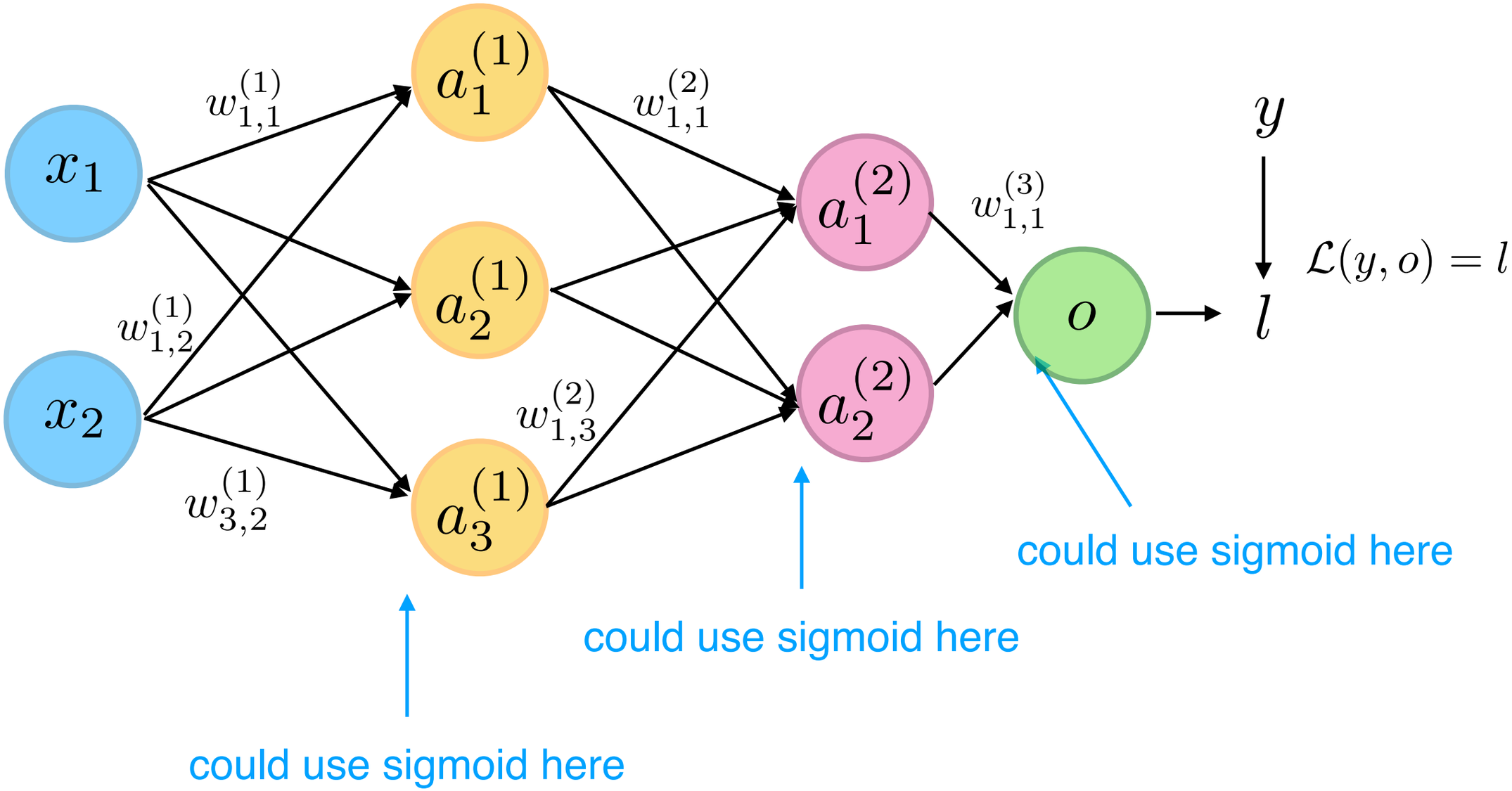


Computation Graph with Multiple Fully-Connected Layers = Multilayer Perceptron

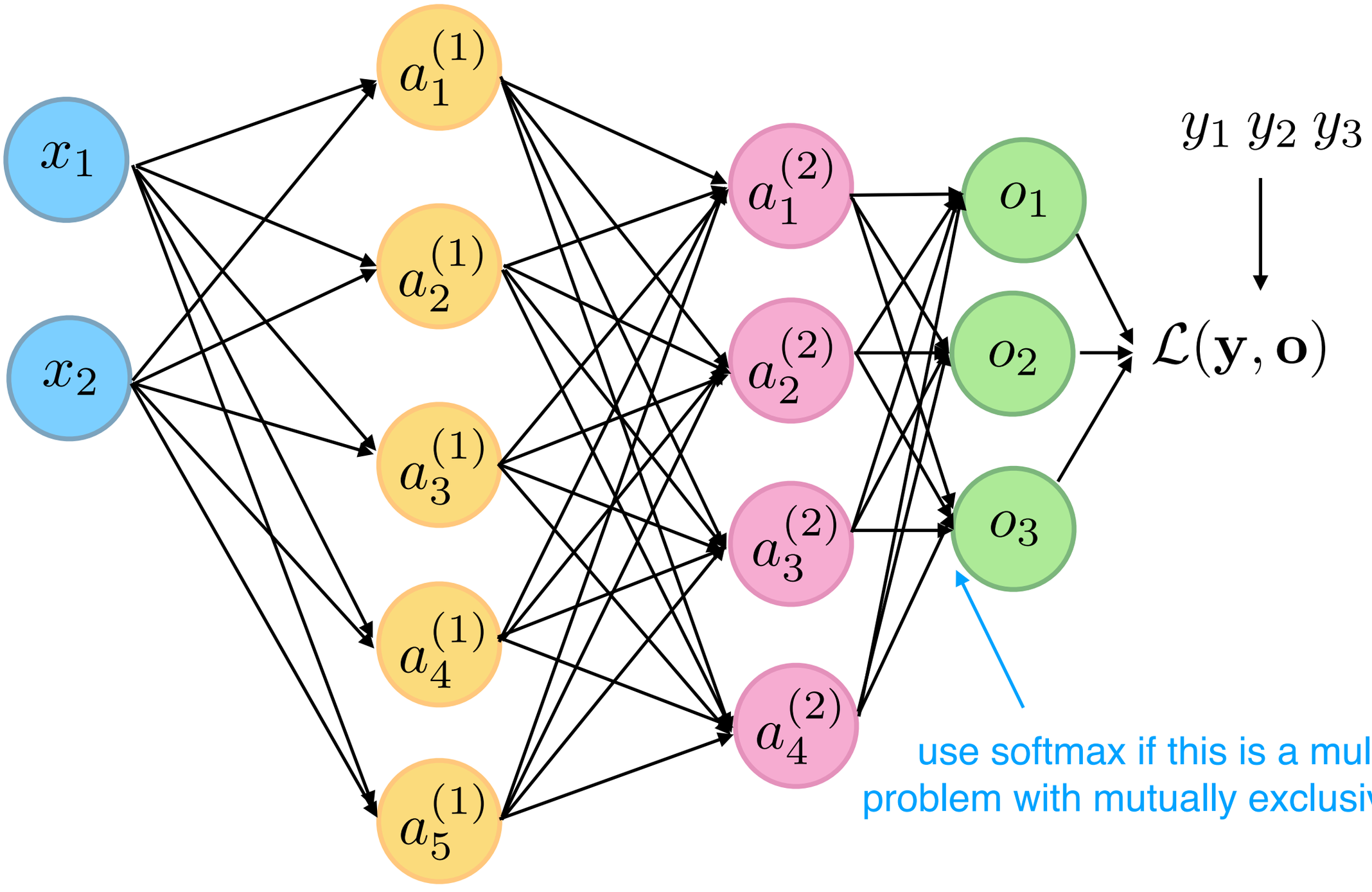
A more common counting/naming scheme, because then a perceptron/Adaline/ logistic regression model can be called a "1-layer neural network"



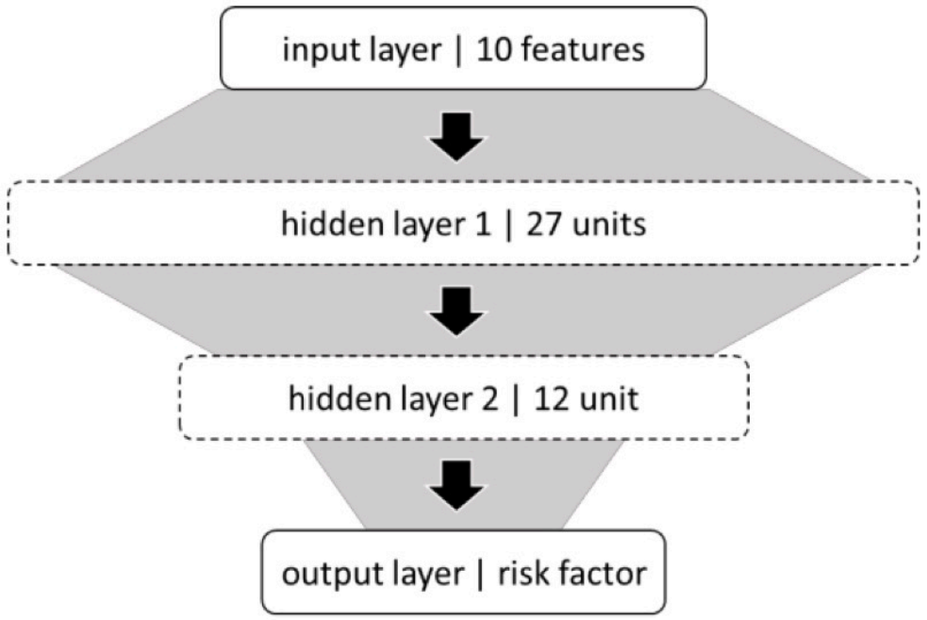
Computation Graph with Multiple Fully-Connected Layers = Multilayer Perceptron



Computation Graph with Multiple Fully-Connected Layers = Multilayer Perceptron



Is It Deep Learning?



Supplementary Figure 2. Illustration of the feed forward neural network architecture of the proposed deep survival model.

Note That the Loss is Not Convex Anymore

- Linear regression, Adaline, Logistic Regression, and Softmax Regression have convex loss functions

- 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

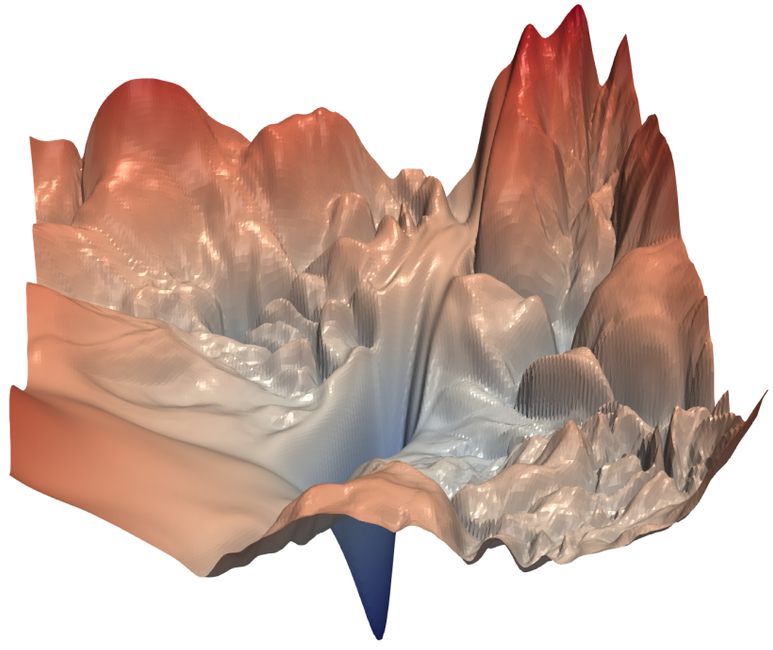


Image Source: Li, H., Xu, Z., Taylor, G., Studer, C. and Goldstein, T., 2018. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems* (pp. 6391-6401).

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 usually better than sigmoid

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

Complex, nonlinear decision boundaries with hidden layer and nonlinear activations

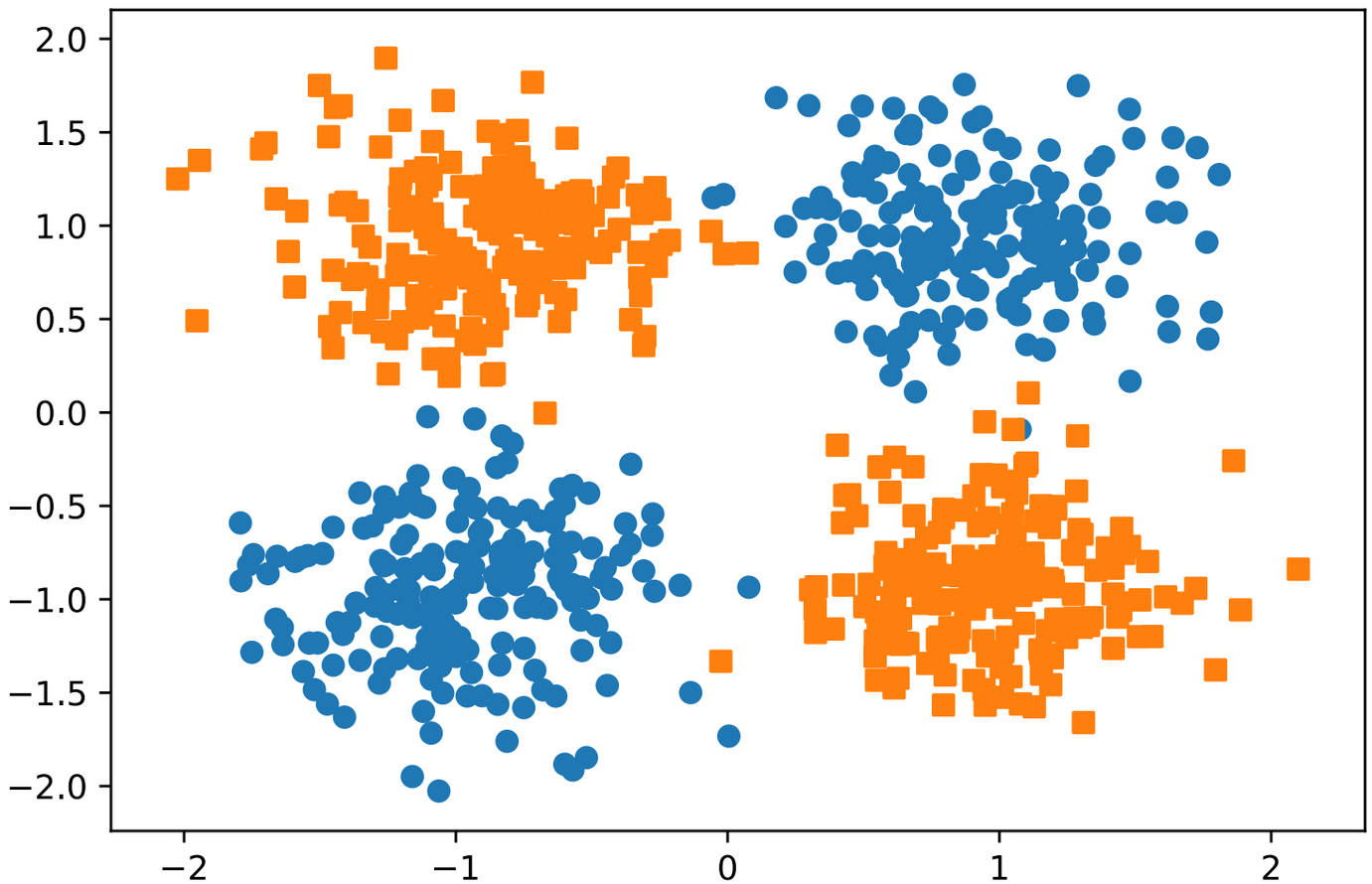
1. Multilayer Perceptron Architecture
2. **Nonlinear Activation Functions**
3. Multilayer Perceptron Code Examples
4. Overfitting and Underfitting
5. Cats & Dogs and Custom Data Loaders

PyTorch Recap

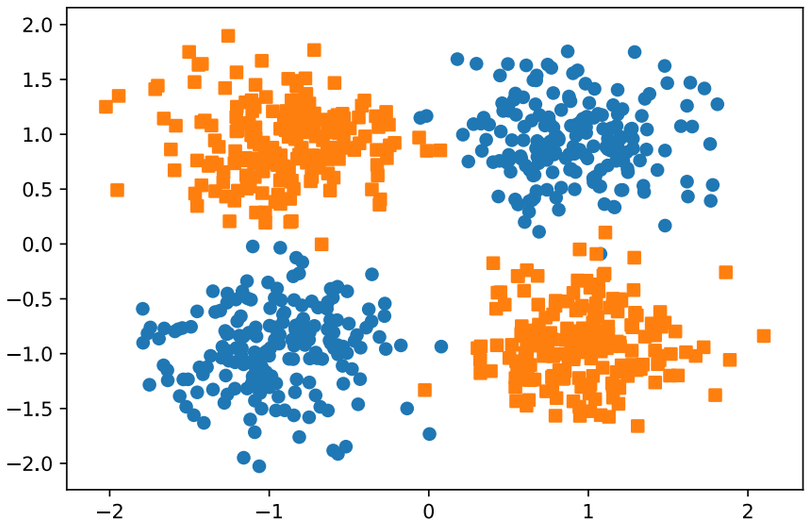
```
import torch.nn.functional as F


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
```

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

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

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

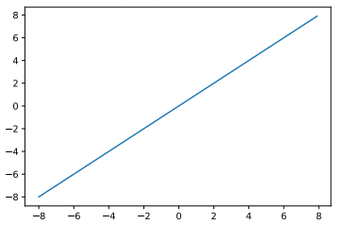
Solving the XOR Problem with Non-Linear Activations

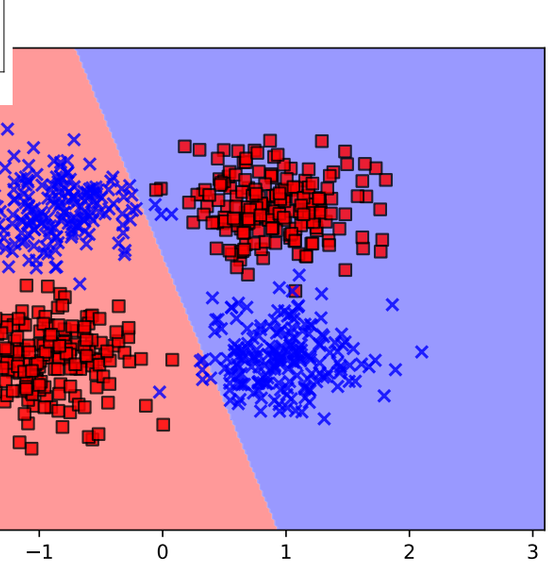


Solving the XOR Problem with Non-Linear Activations



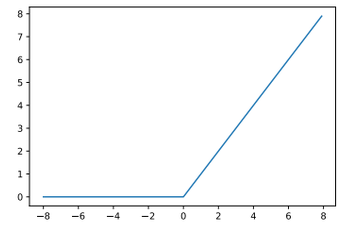
$\sigma(z) = z$

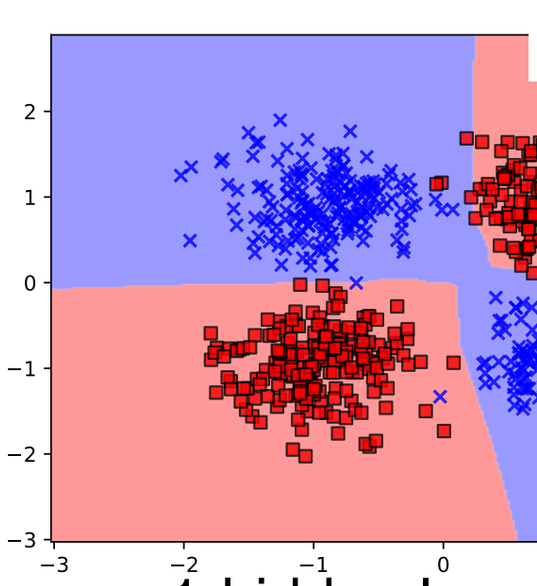




1-hidden layer MLP
with linear activation function

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



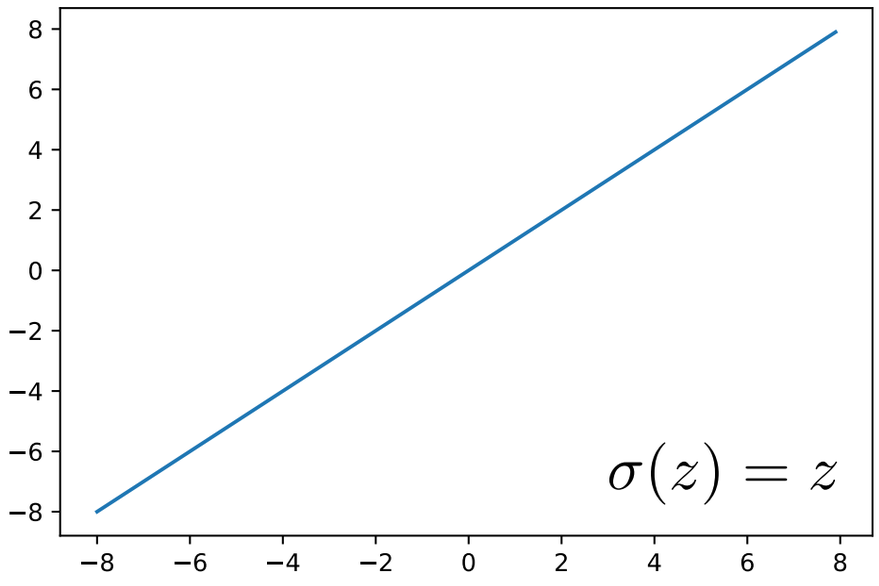


1-hidden layer MLP
with non-linear activation function (ReLU)

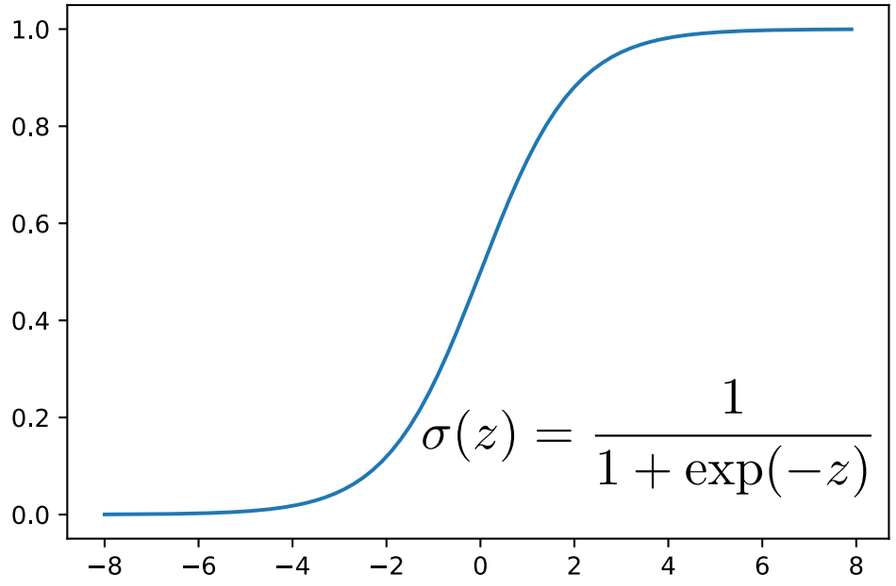
https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L09/code/xor-problem.ipynb

A Selection of Common Activation Functions (1)

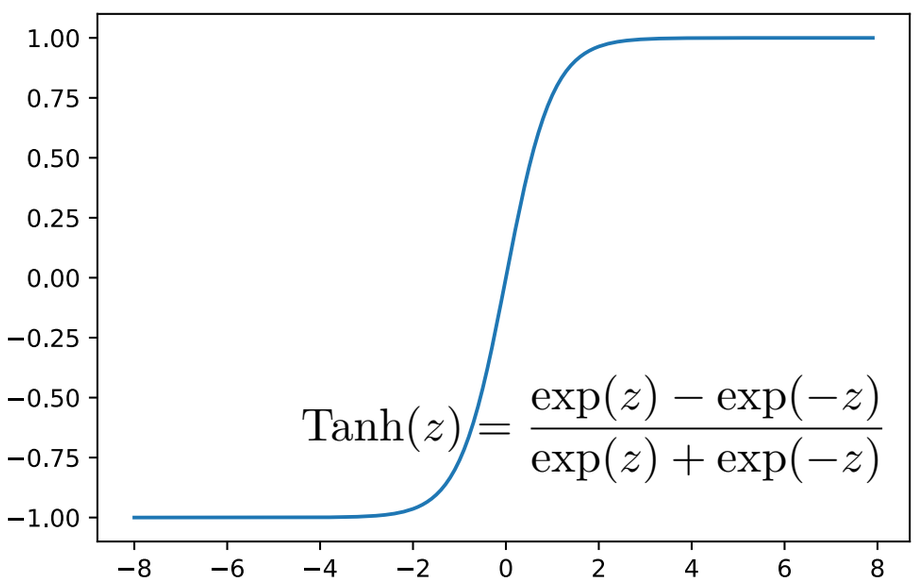
Identity



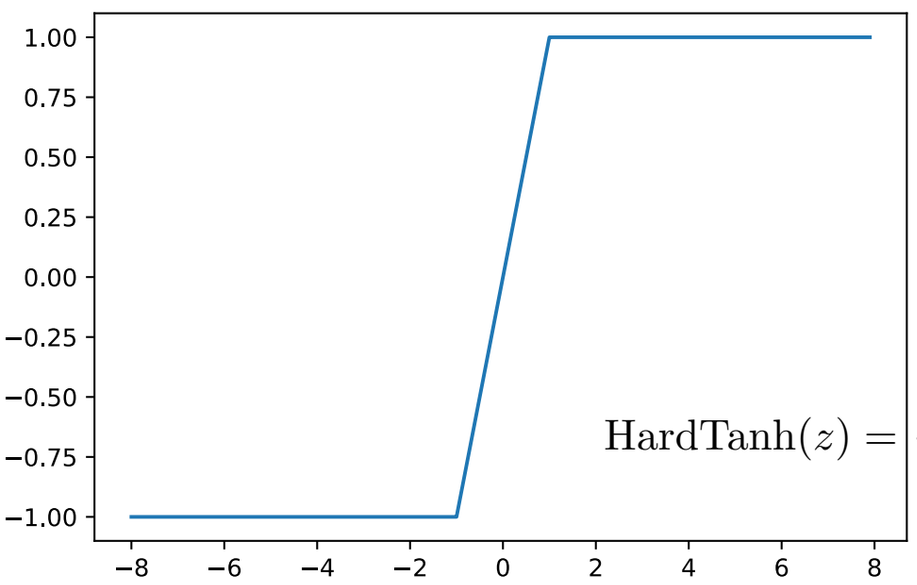
(Logistic) Sigmoid



Tanh ("tanH")



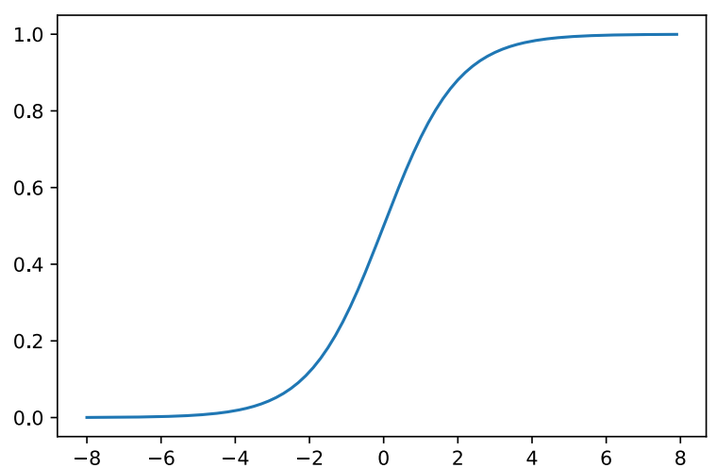
Hard Tanh



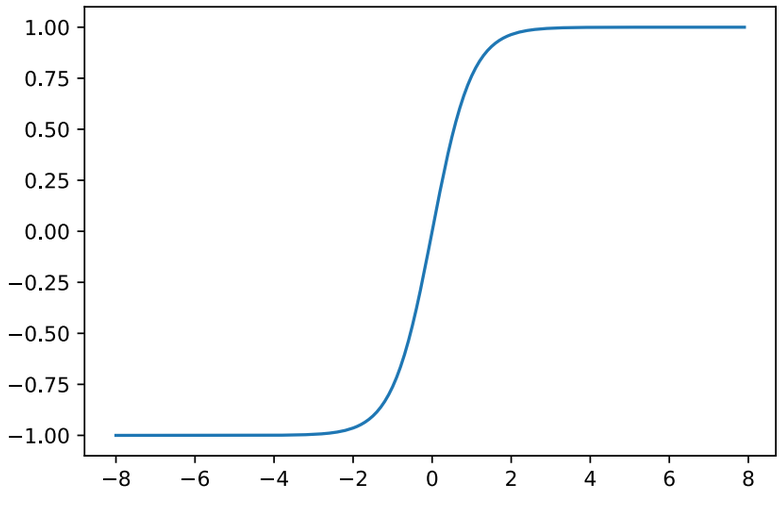
A Selection of Common Activation Functions (1)

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

(Logistic) Sigmoid



Tanh ("tanH")



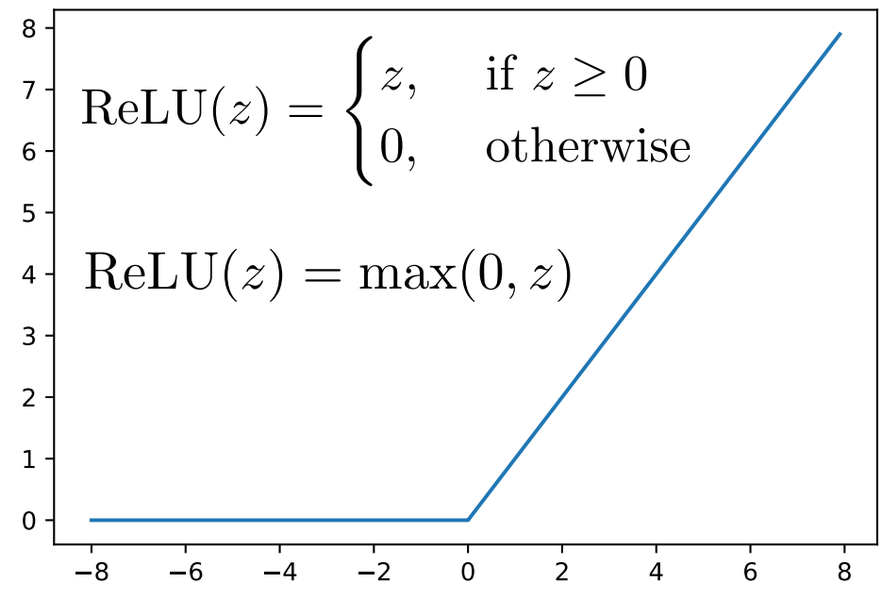
Also simple derivative:

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

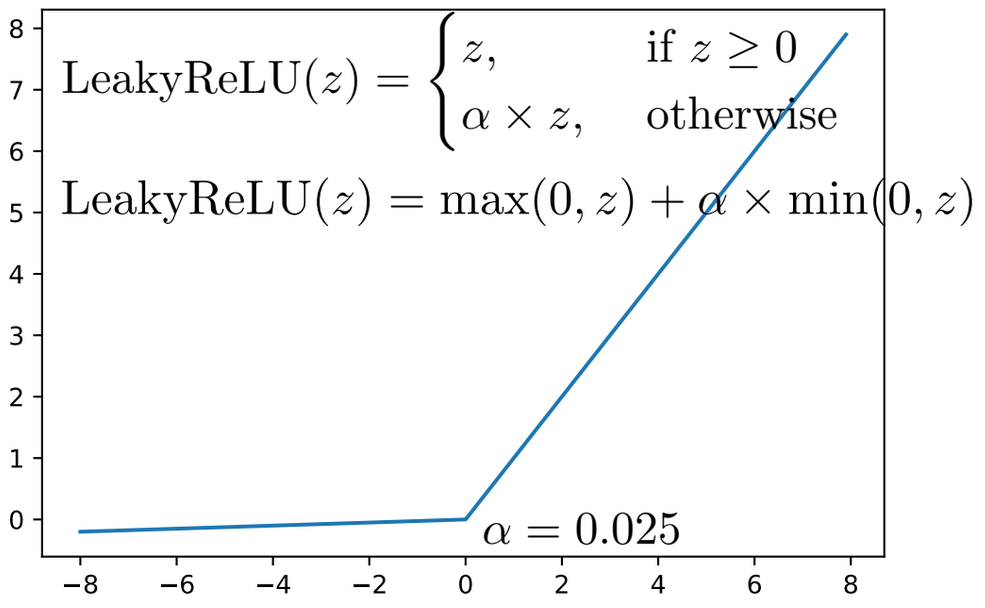
Additional tip: Also good to normalize inputs to mean zero and use random weight initialization with avg. weight centered at zero

A Selection of Common Activation Functions (2)

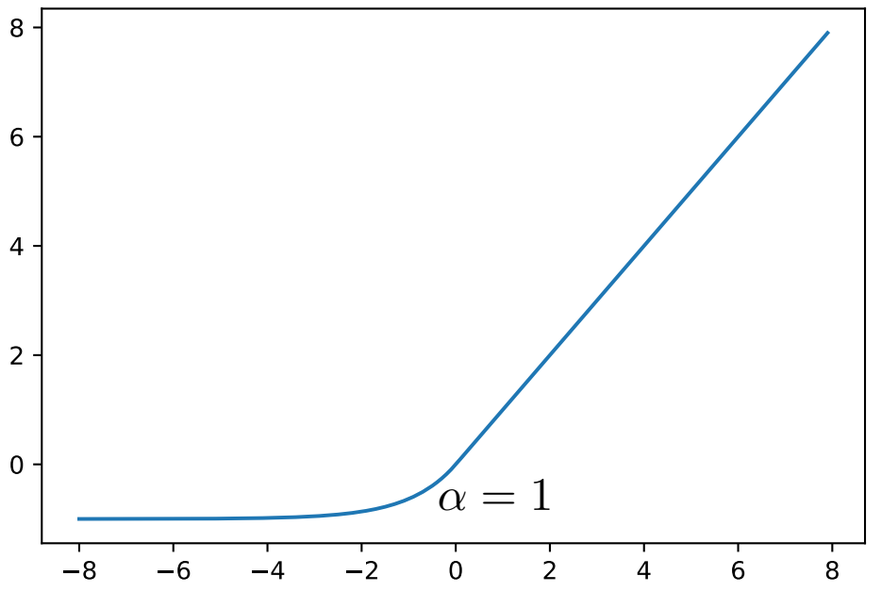
ReLU (Rectified Linear Unit)



Leaky ReLU



ELU (Exponential Linear Unit)



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

PReLU (Parameterized Rectified Linear Unit)

here, alpha is a trainable parameter

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

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

arXiv.org > cs > arXiv:2006.14536

Computer Science > Machine Learning

[Submitted on 25 Jun 2020]

Smooth Adversarial Training

Cihang Xie, Mingxing Tan, Boqing Gong, Alan Yuille, Quoc V. Le

https://arxiv.org/abs/2006.14536

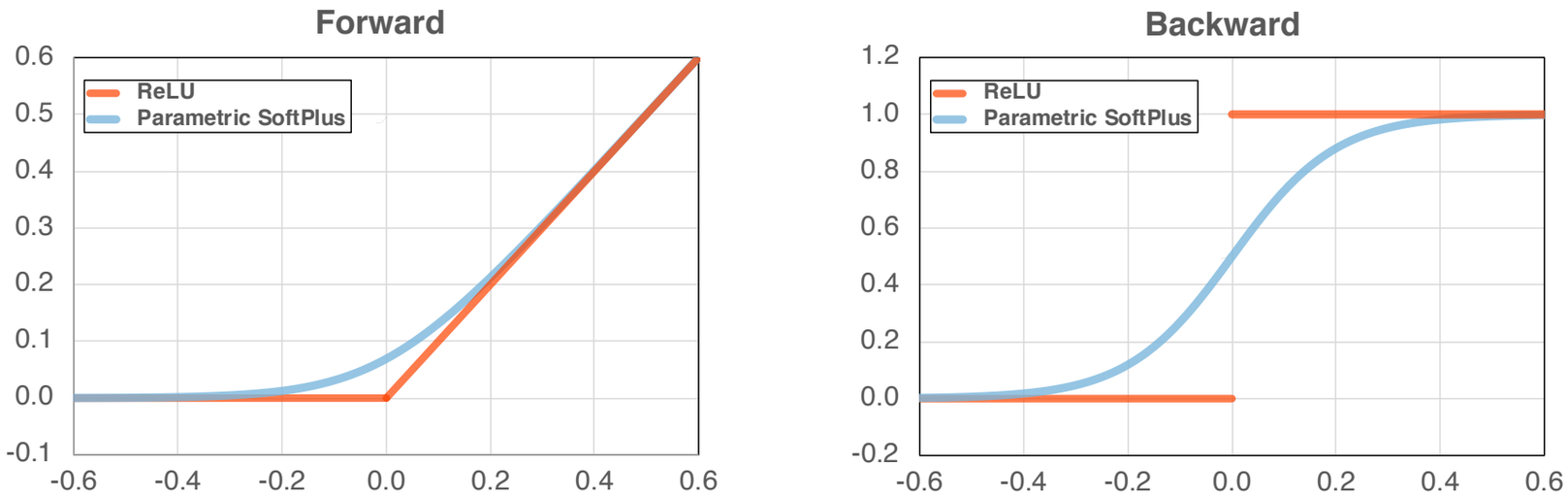


Figure 1: Left Panel: ReLU and Parametric SoftPlus. Right Panel: the first derivatives for ReLU and Parametric SoftPlus. Compared to ReLU, Parametric Softplus is smooth with continuous derivatives.

It is commonly believed that networks cannot be both accurate and robust, that gaining robustness means losing accuracy. [...] Our key observation is that the widely-used ReLU activation function significantly weakens adversarial training due to its non-smooth nature. Hence we propose *smooth adversarial training (SAT)*, in which we replace ReLU with its smooth approximations to strengthen adversarial training.

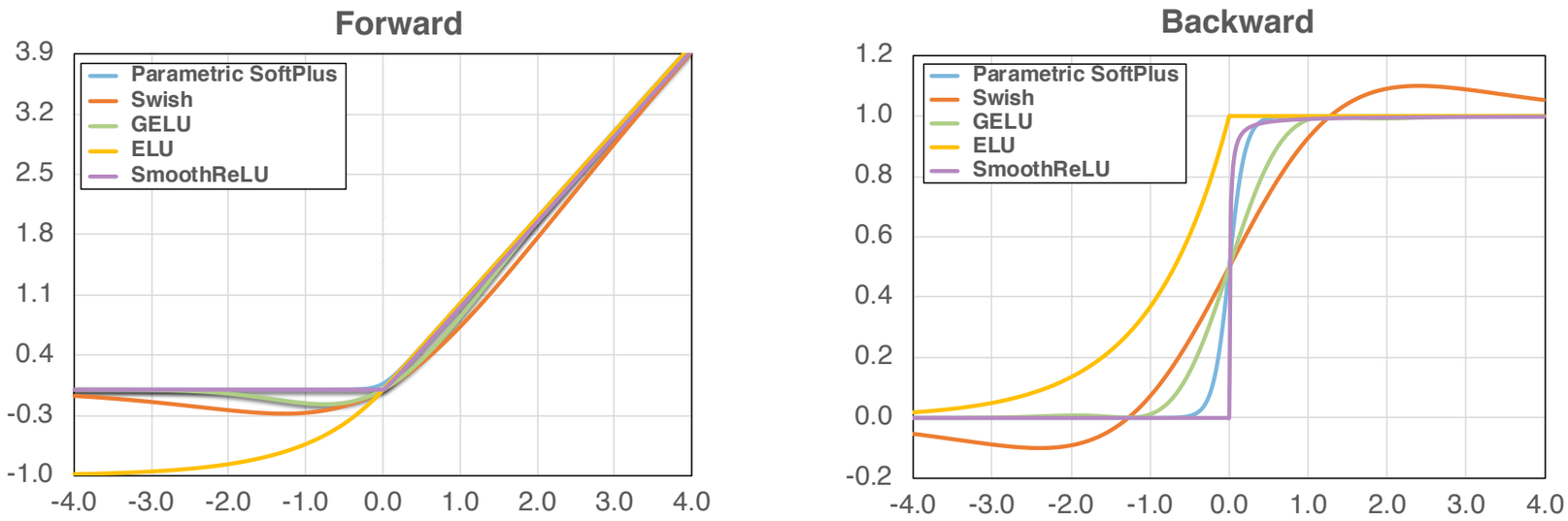


Figure 2: Visualizations of smooth activation functions and their derivatives.

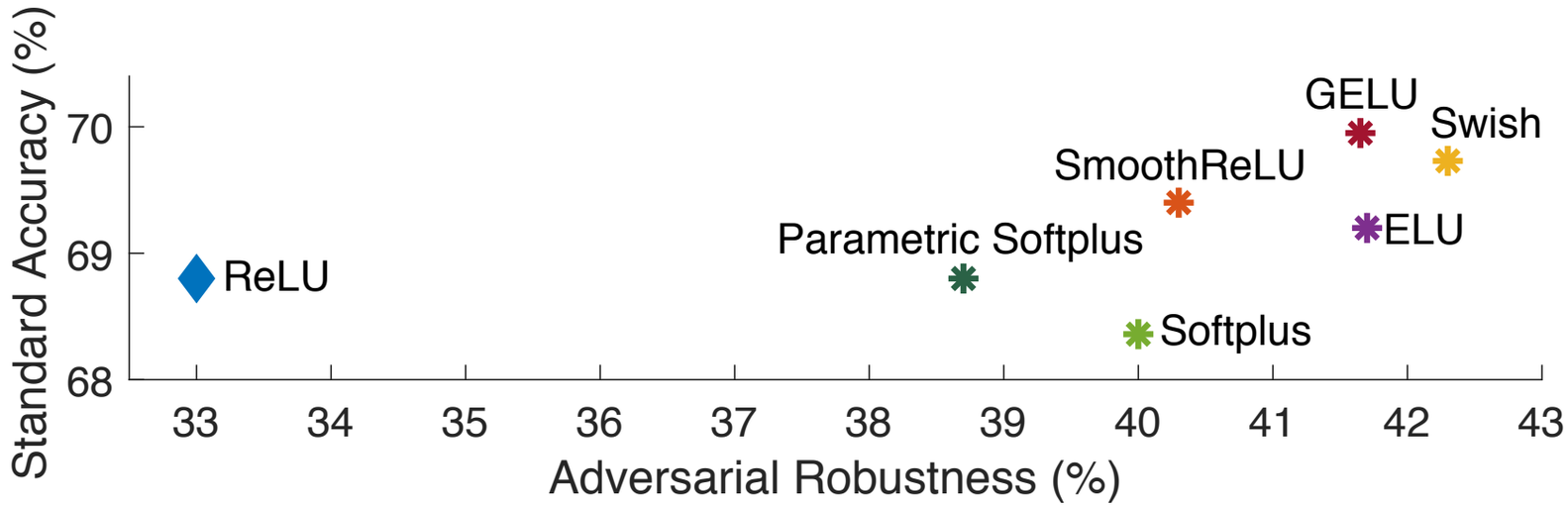
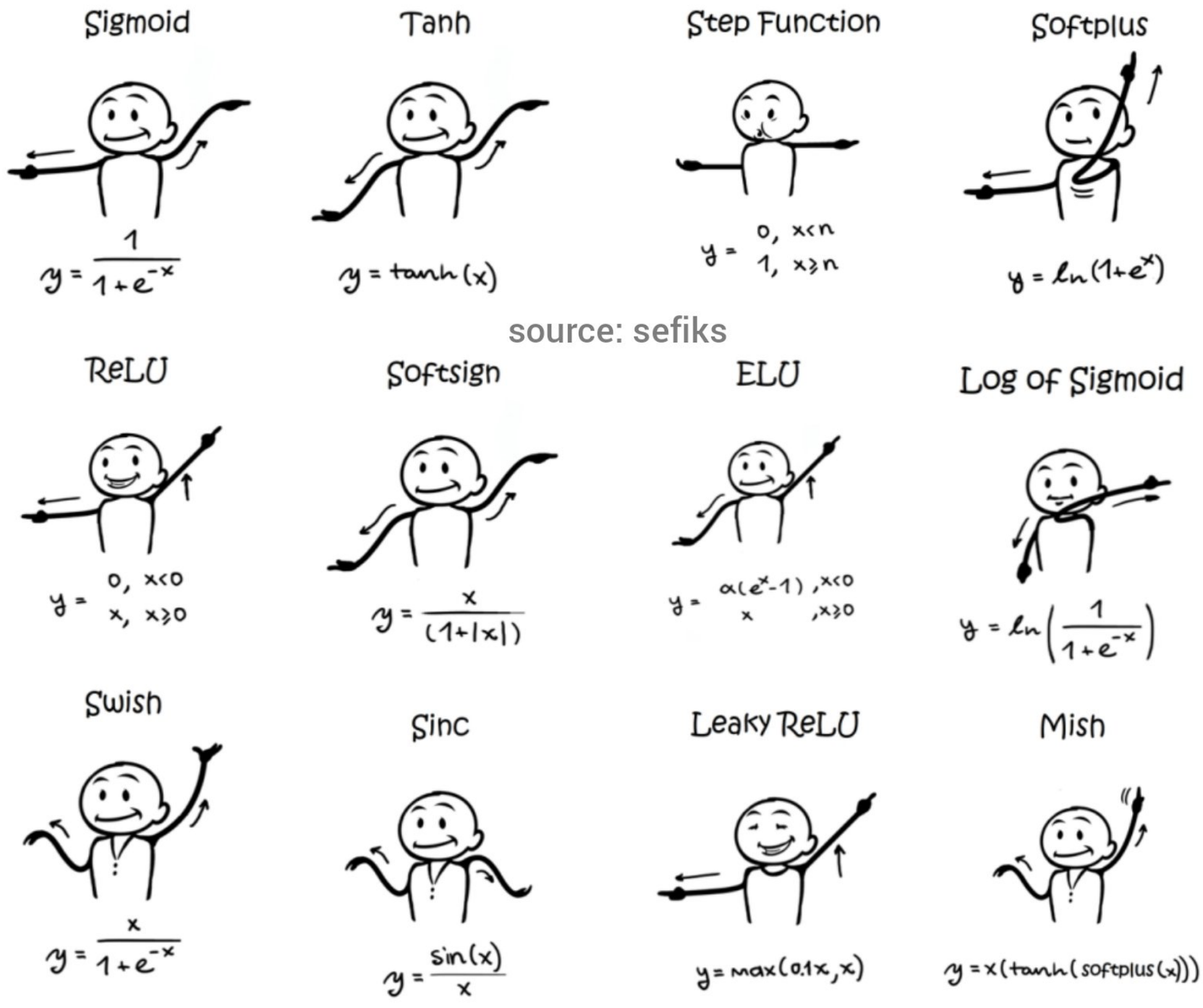


Figure 3: Smooth activation functions improve adversarial training. Compared to ReLU, all smooth activation functions significantly boost robustness, while keeping accuracy almost the same.

https://arxiv.org/abs/2006.14536

Dance Moves of Deep Learning Activation Functions



source: sefiks

https://twitter.com/TheInsaneApp/status/1366324846976659461?s=20

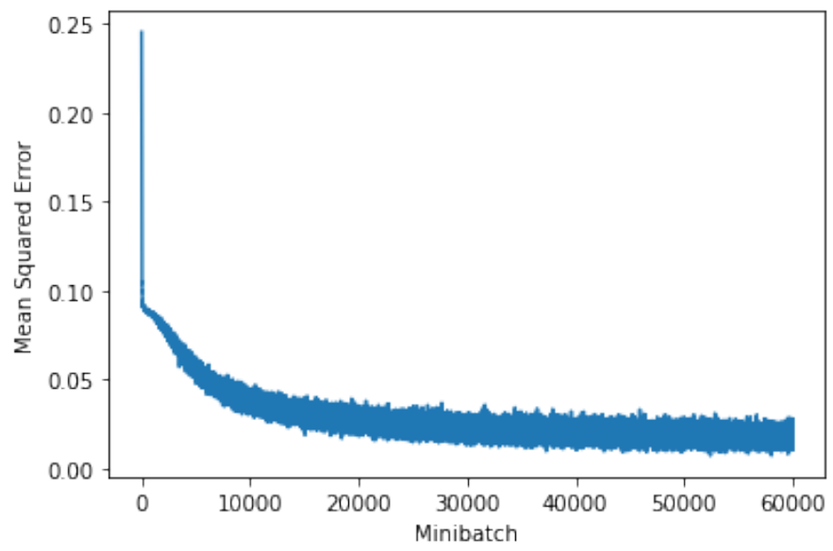
Implementing Multilayer Perceptrons in PyTorch

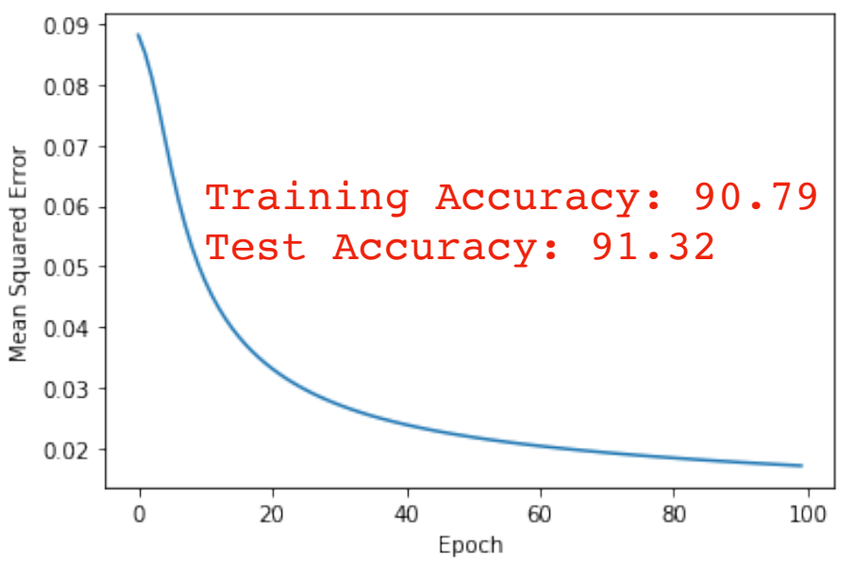
Multilayer Perceptron with Sigmoid Activation and MSE Loss

VS

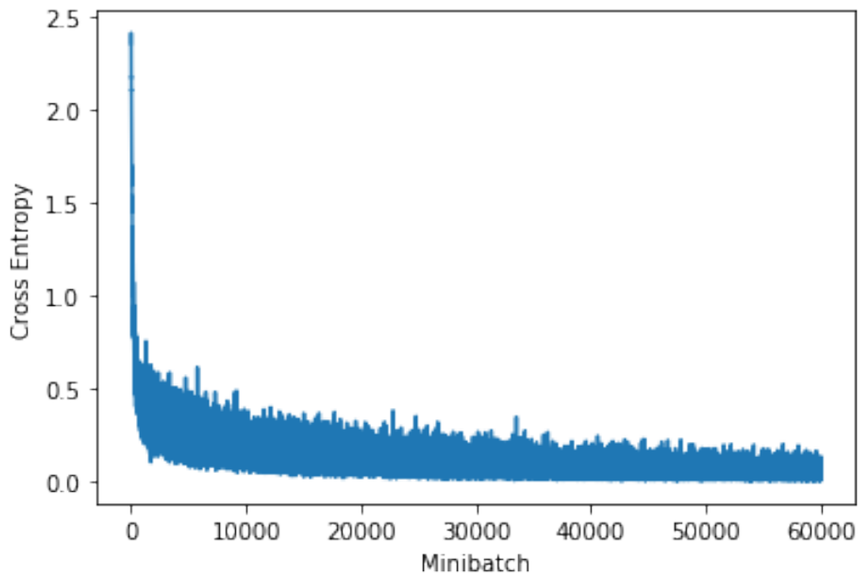
Multilayer Perceptron with Softmax Activation and Cross Entropy Loss

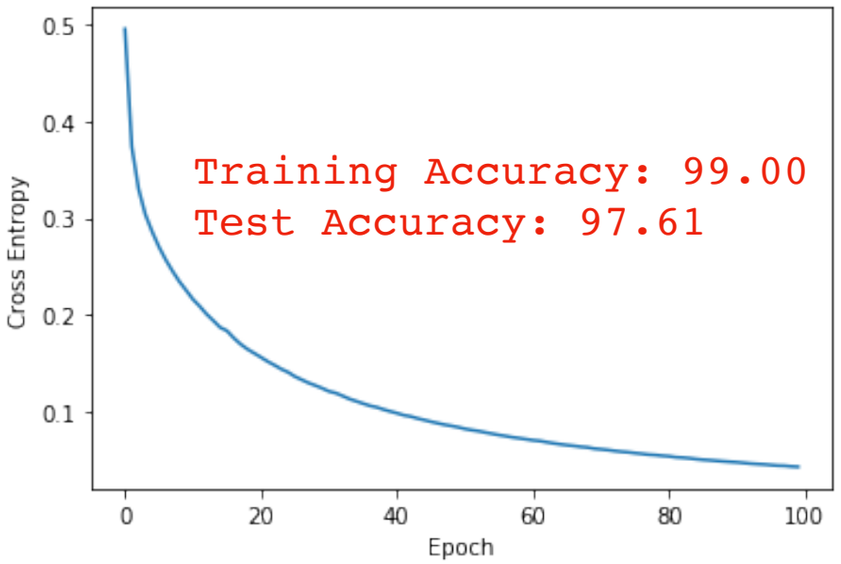
https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L09/code/mlp-pytorch_sigmoid-mse.ipynb





https://github.com/rasbt/stat453-deep-learning-ss21/blob/main/L09/code/mlp-pytorch_softmax-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")

Wide 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?

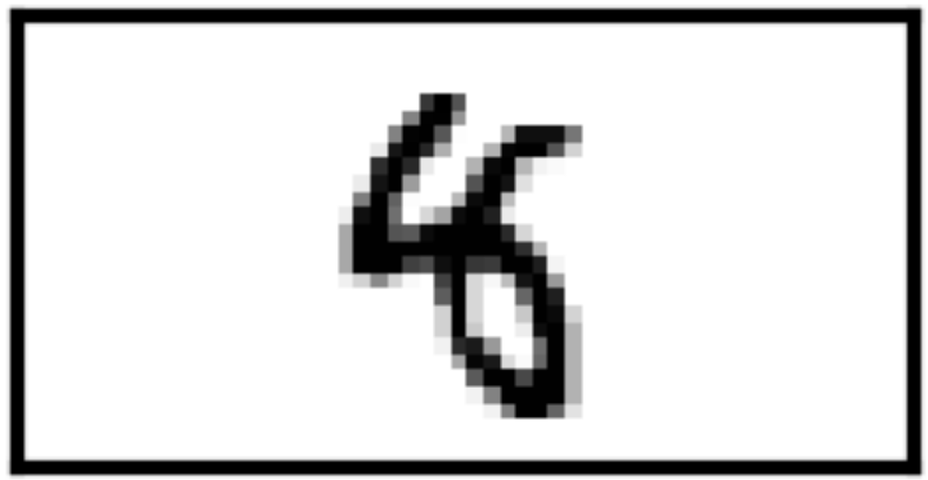
[1] Balázs Csanád Csáji (2001) Approximation with Artificial Neural Networks; Faculty of Sciences; Eötvös Loránd University, Hungary
[2] Cybenko, G. (1989) "Approximations by superpositions of sigmoidal functions", Mathematics of Control, Signals, and Systems, 2(4), 303–314. doi:10.1007/BF02551274
[3] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural networks, *2*(5), 359-366.

Wide vs Deep Architectures (Breadth vs Depth)

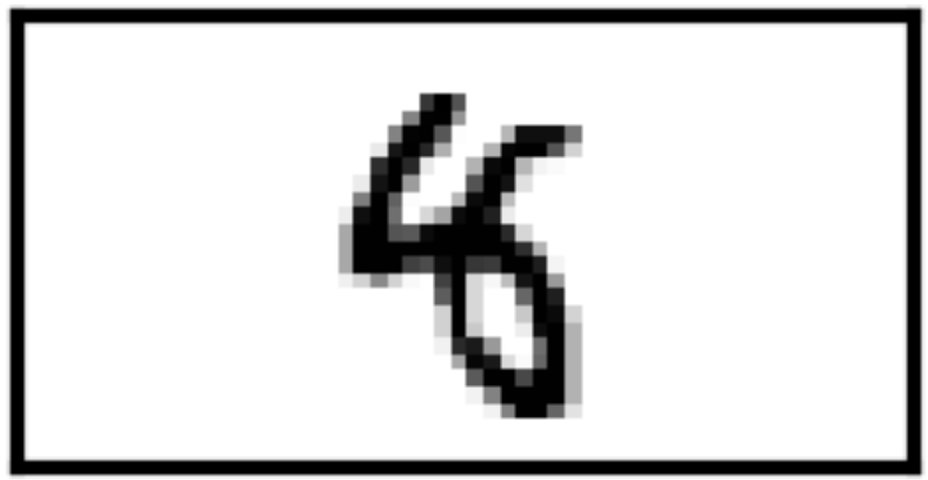
- Can achieve the same expressiveness with more layers but fewer parameters (combinatorics);
  fewer parameters => less overfitting
- Also, having more layers provides some form of regularization: later layers are constrained on the behavior of earlier layers
- However, more layers => vanishing/exploding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

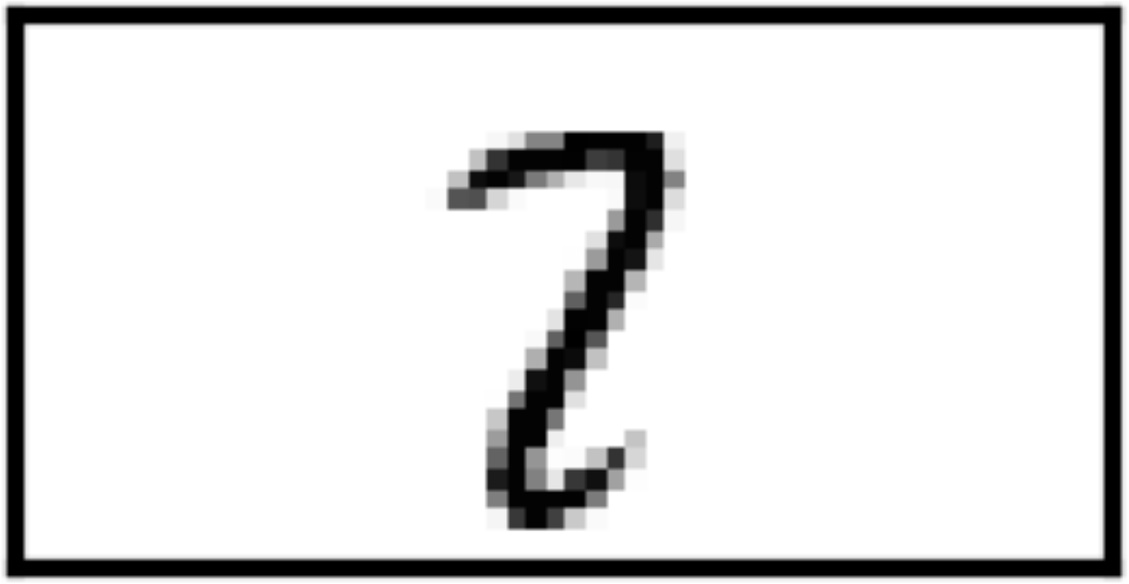
The problems with models that are too simple and models that fit the training data "too well"

1. Multilayer Perceptron Architecture
2. Nonlinear Activation Functions
3. Multilayer Perceptron Code Examples
4. **Overfitting and Underfitting**
5. Cats & Dogs and Custom Data Loaders

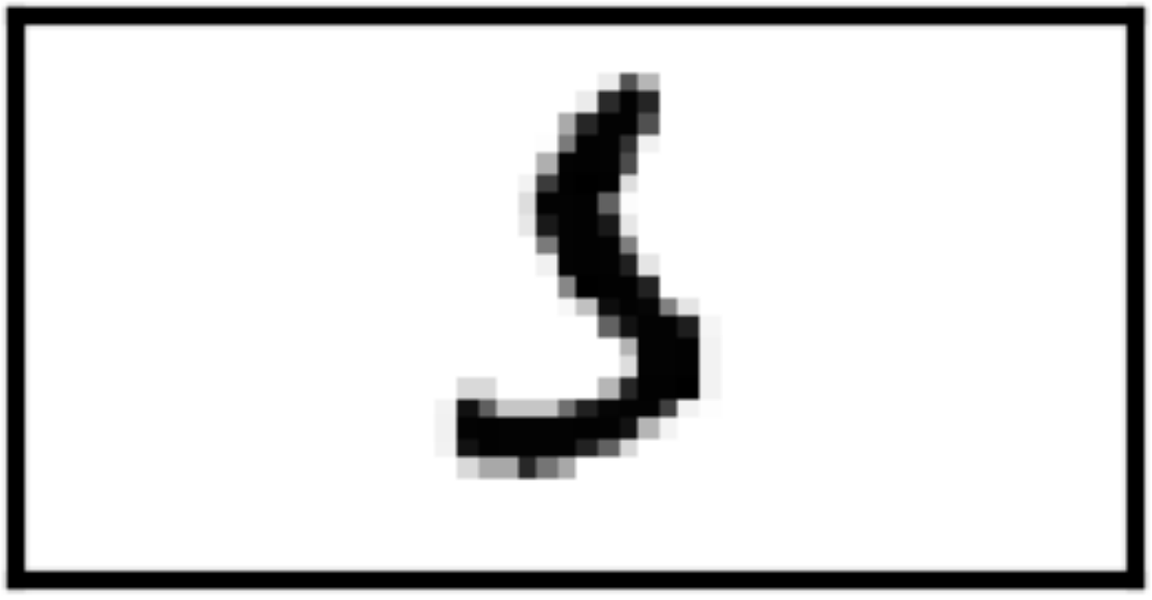


6) t: 8 p: 4

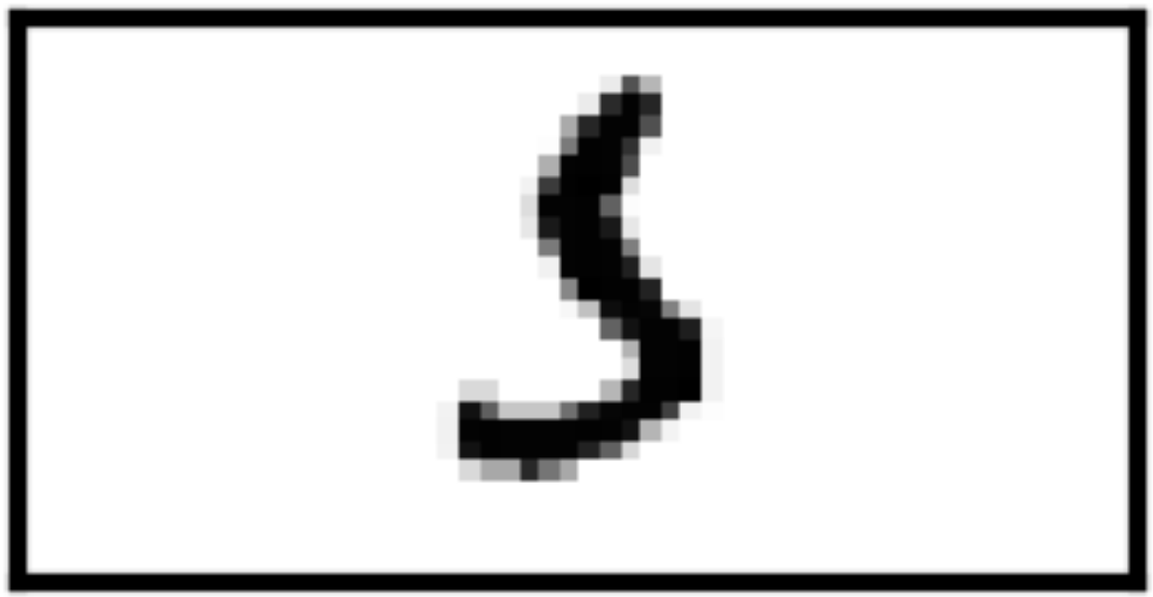




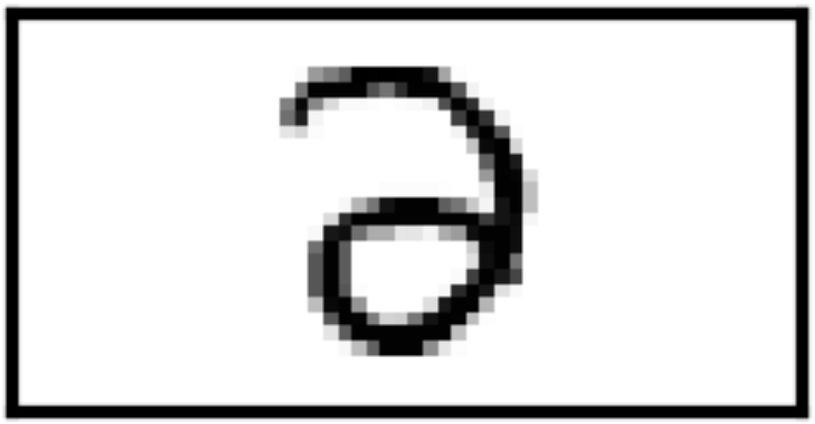
8) t: 2 p: 7



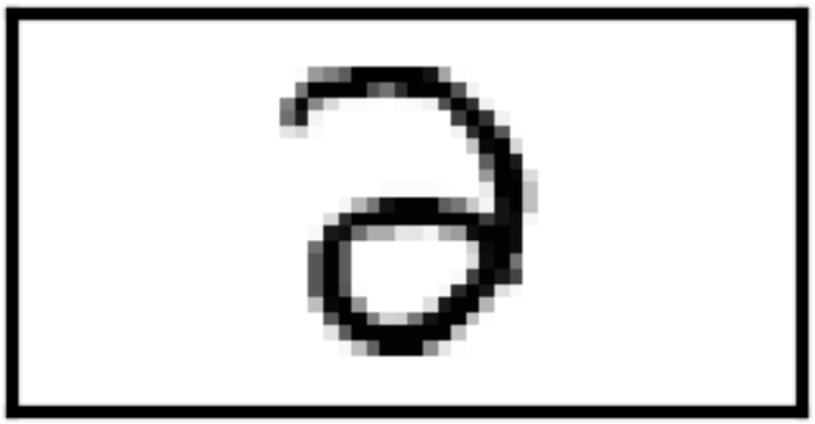
10) t: 5 p: 3



15) t: 6 p: 0

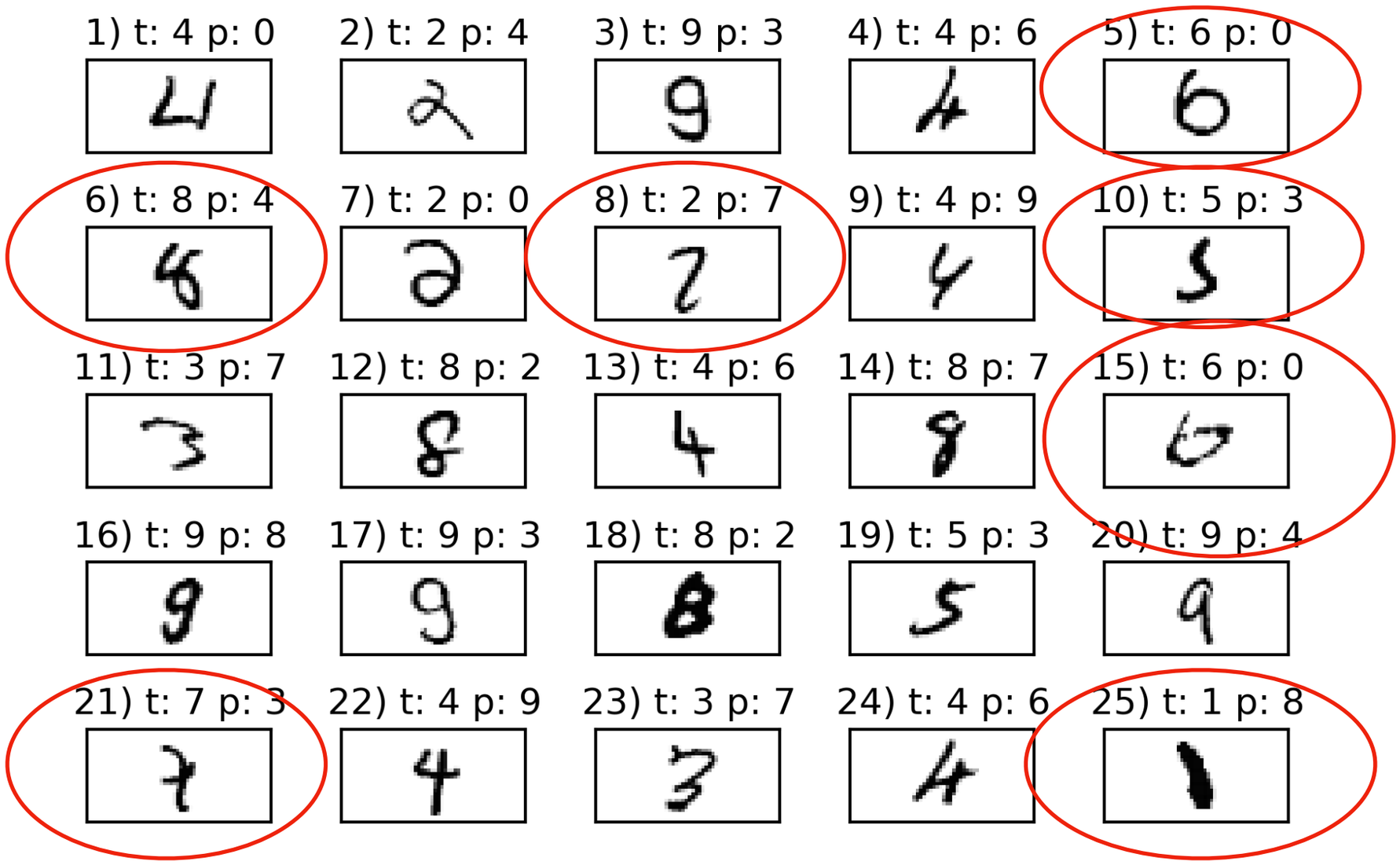


7) t: 2 p: 0



8) t: 2 p: 7

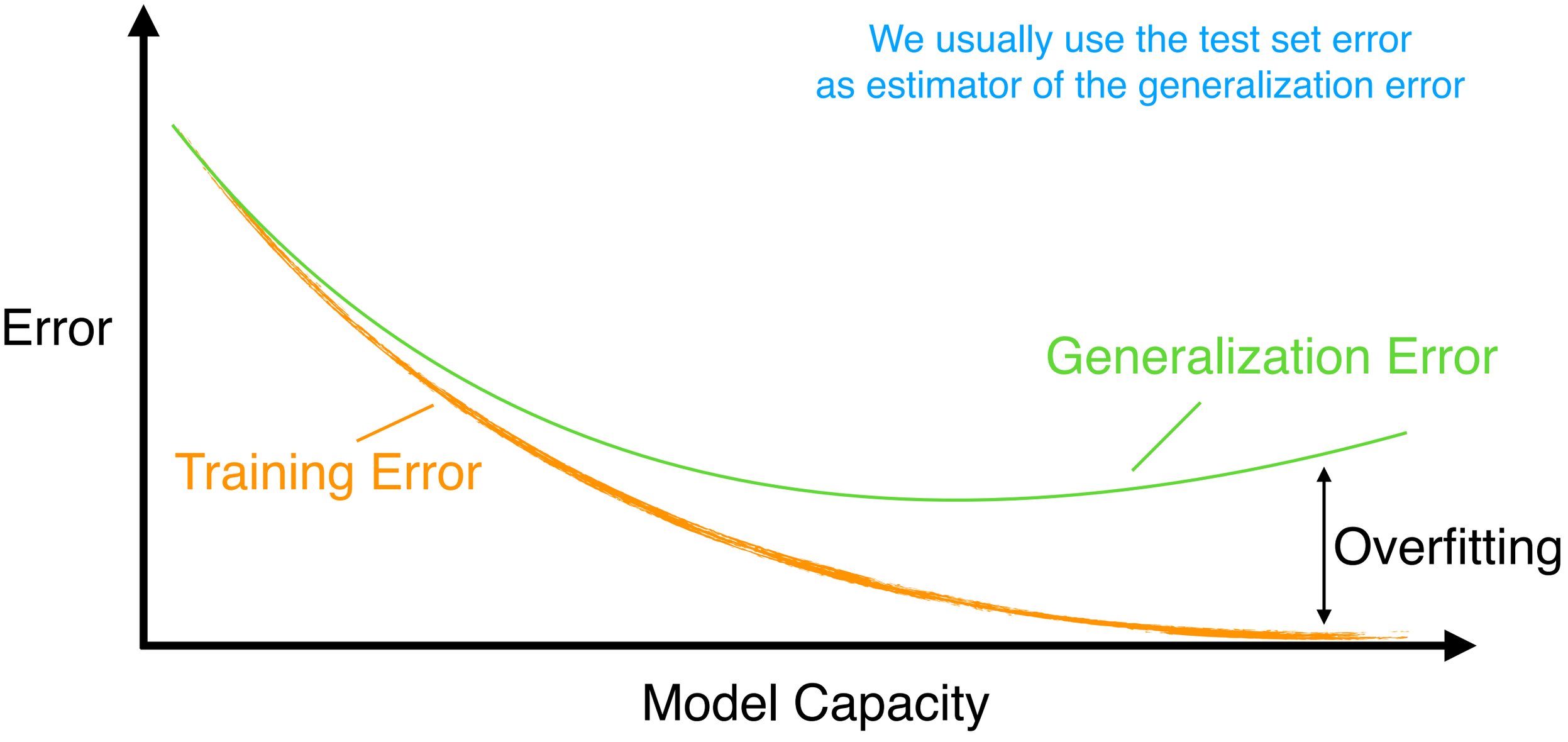
Recommended Practice: Looking at Some Failure Cases



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

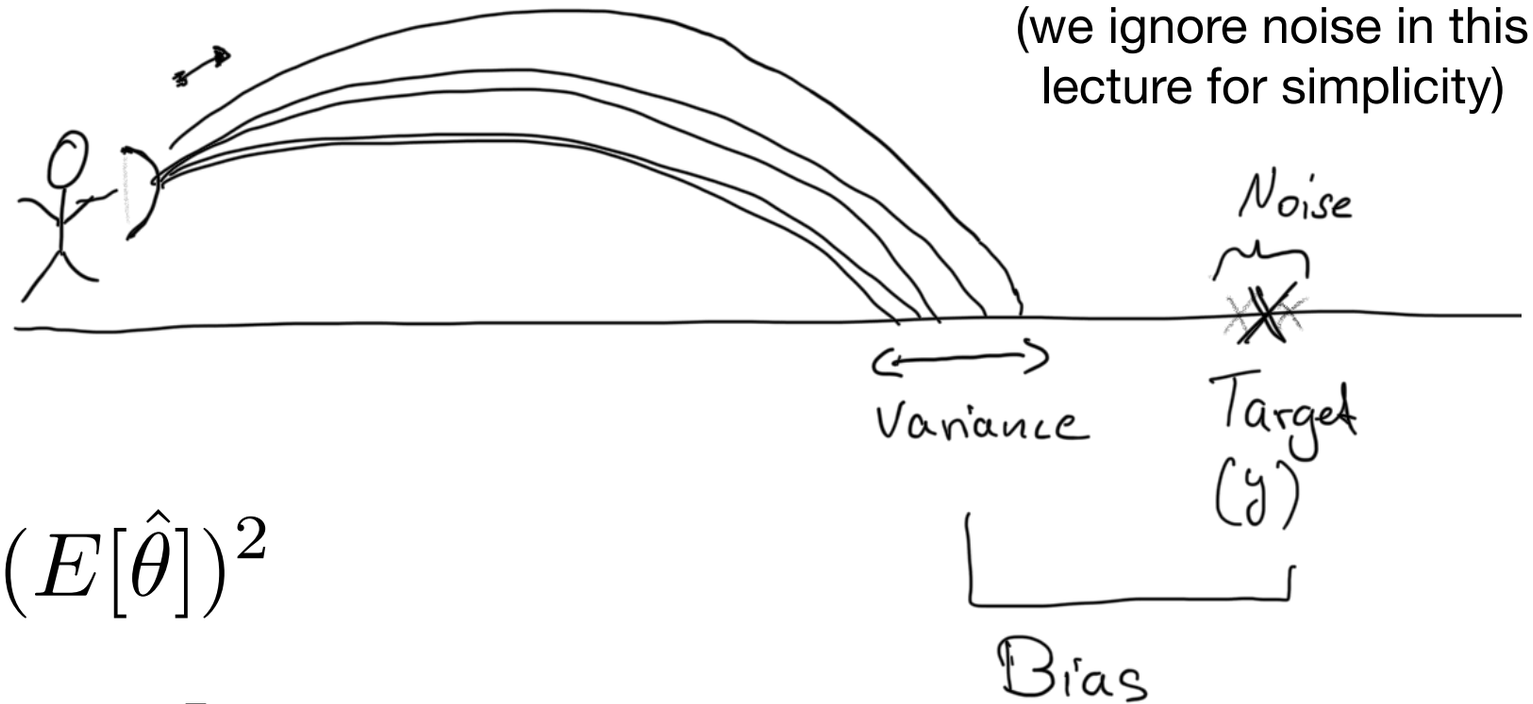
General Definition:

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

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

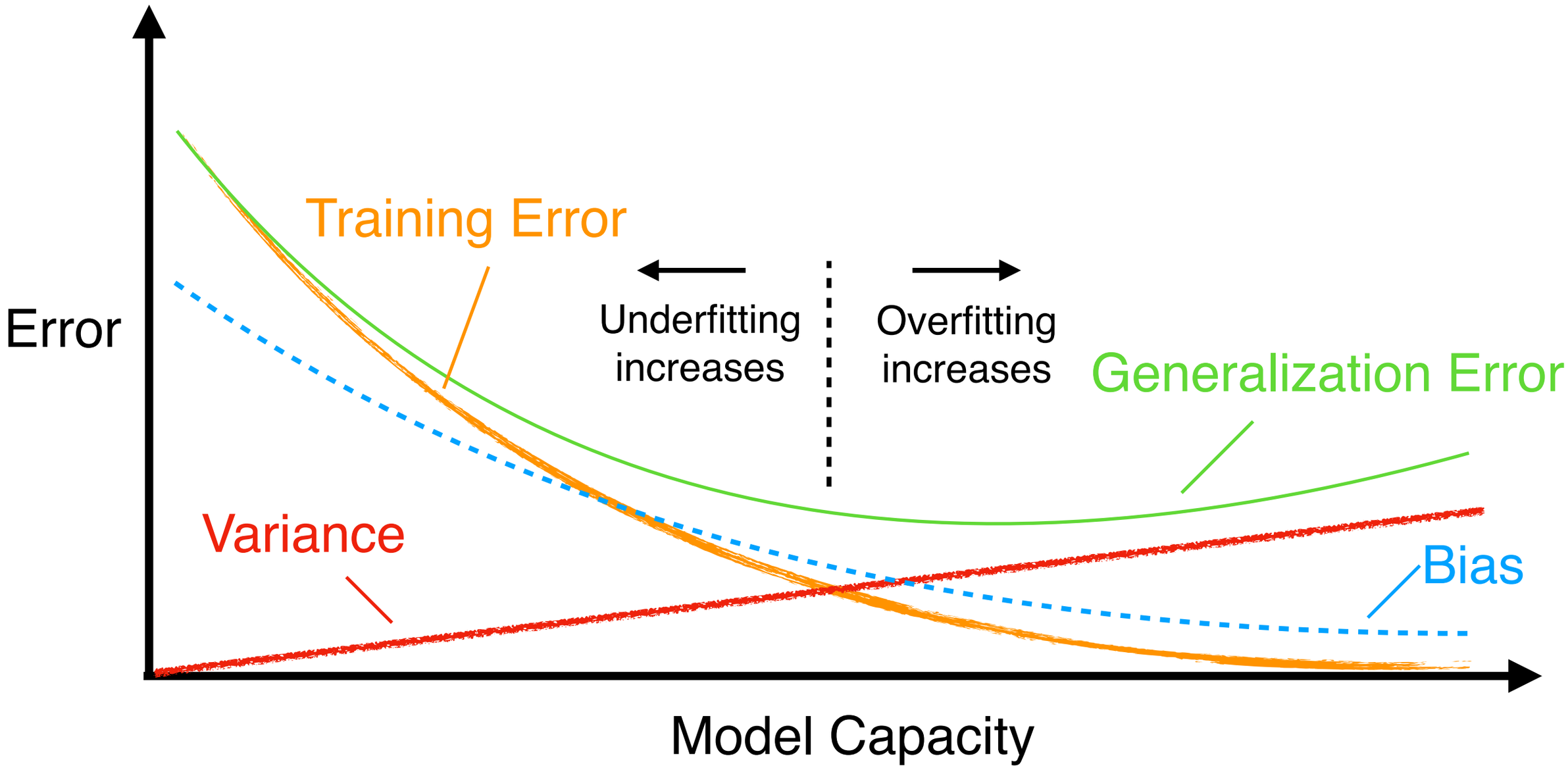
$$\text{Var}_\theta[\hat{\theta}] = E\left[(E[\hat{\theta}] - \hat{\theta})^2\right]$$

Intuition:



(we ignore noise in this lecture for simplicity)

Bias & Variance vs Overfitting & Underfitting



capacity: abstract concept meaning roughly the number of parameters of the model times how efficiently the parameters are used

Further Reading Material

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

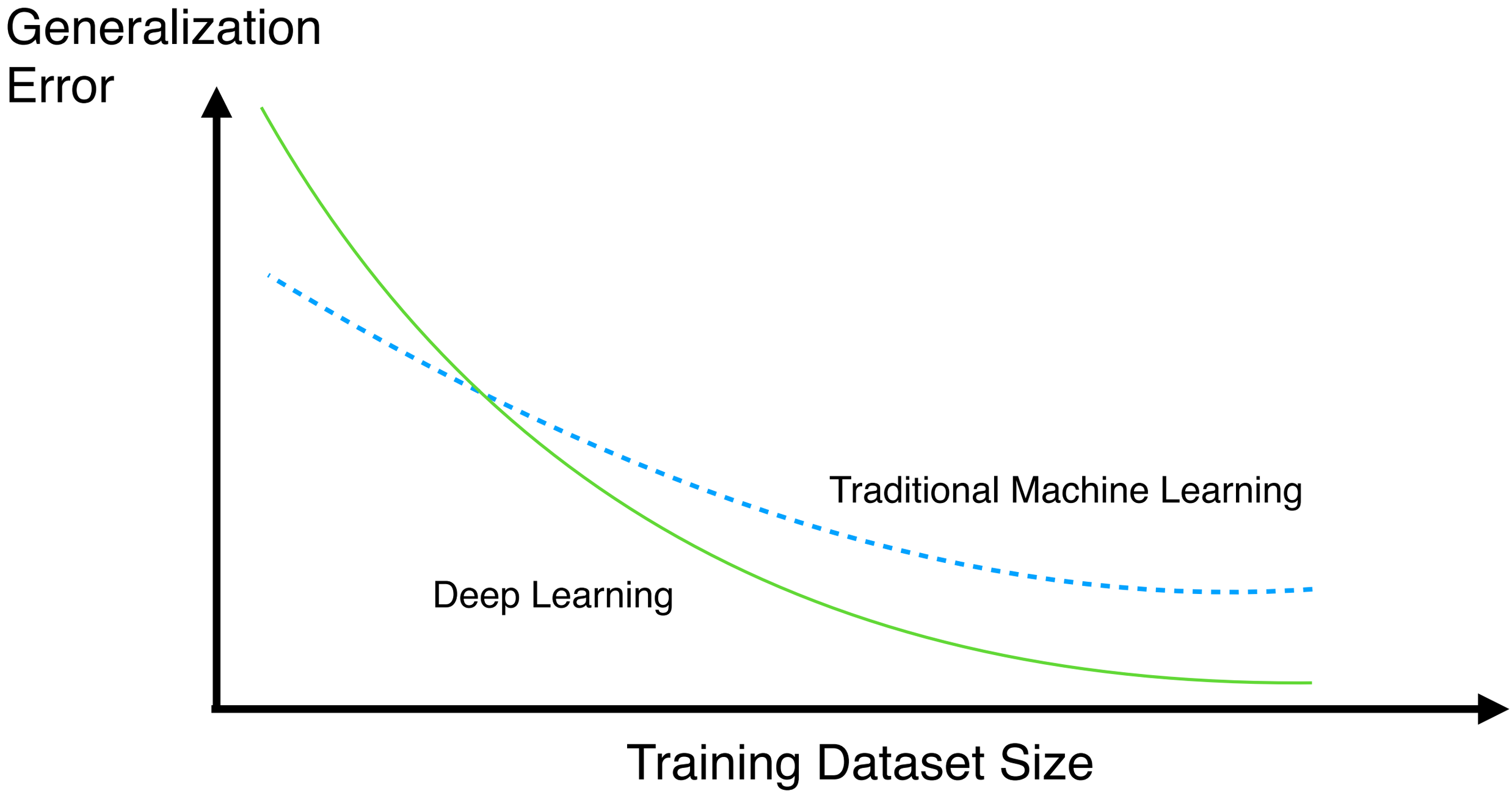
- Also, train/valid/test splits are usually sufficient for training & estimating the generalization performance in deep learning

- However, if you are interested, we covered these topics in the model evaluation lectures in my STAT 451 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



arXiv.org > cs > arXiv:1912.02292

Computer Science > Machine Learning

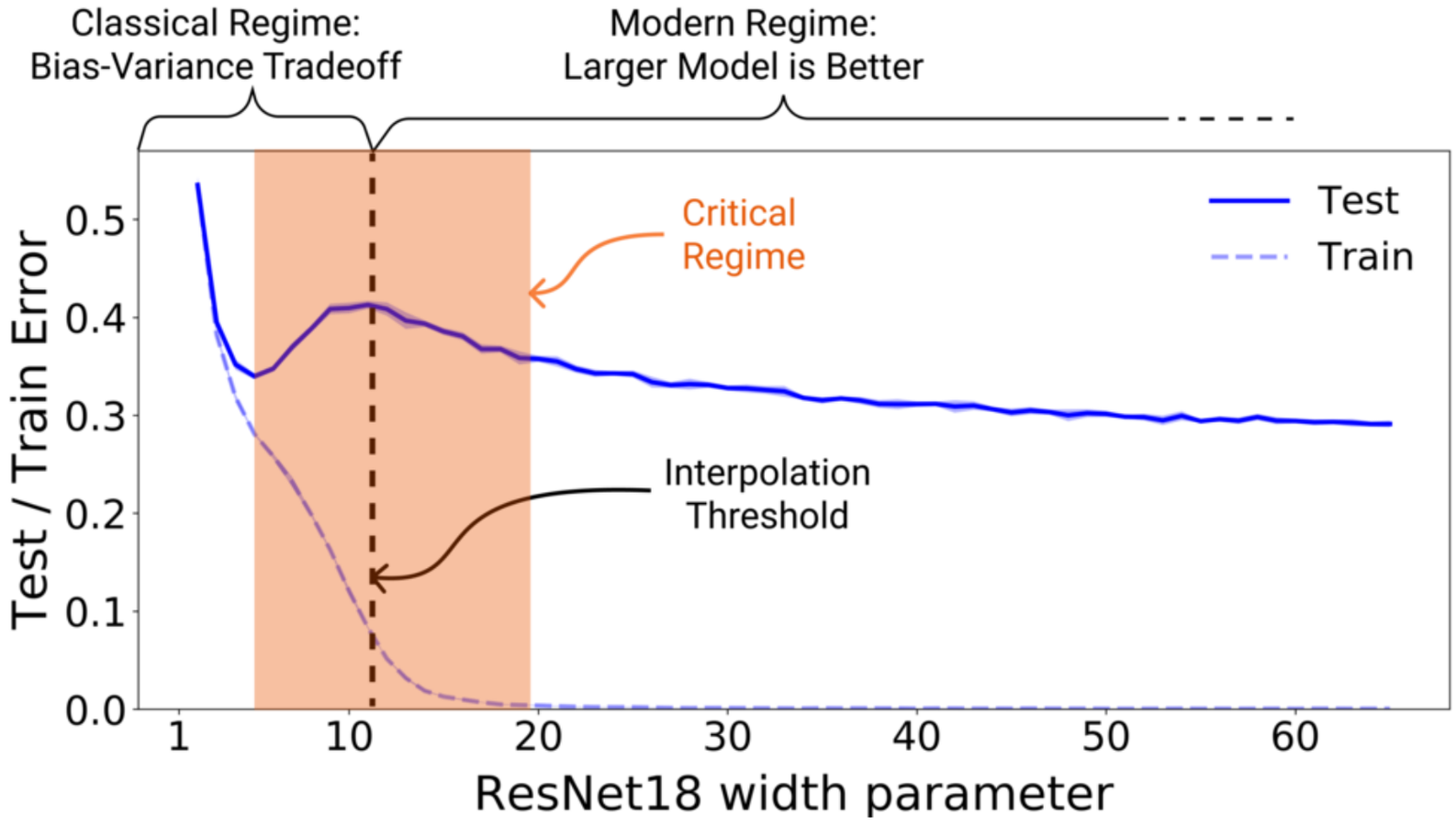
[Submitted on 4 Dec 2019]

Deep Double Descent: Where Bigger Models and More Data Hurt

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, Ilya Sutskever

https://arxiv.org/abs/1912.02292

Architectures: CNNs (standard & ResNet) and transformers trained with cross-entropy loss



arXiv.org > cs > arXiv:1912.02292

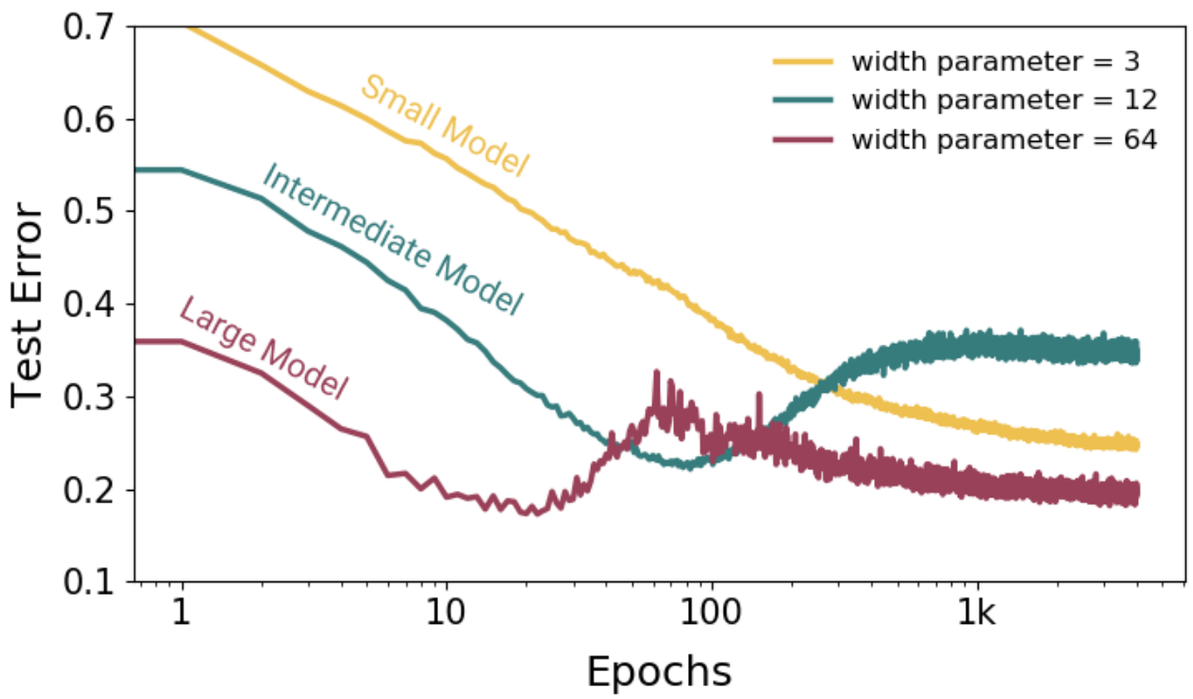
Computer Science > Machine Learning

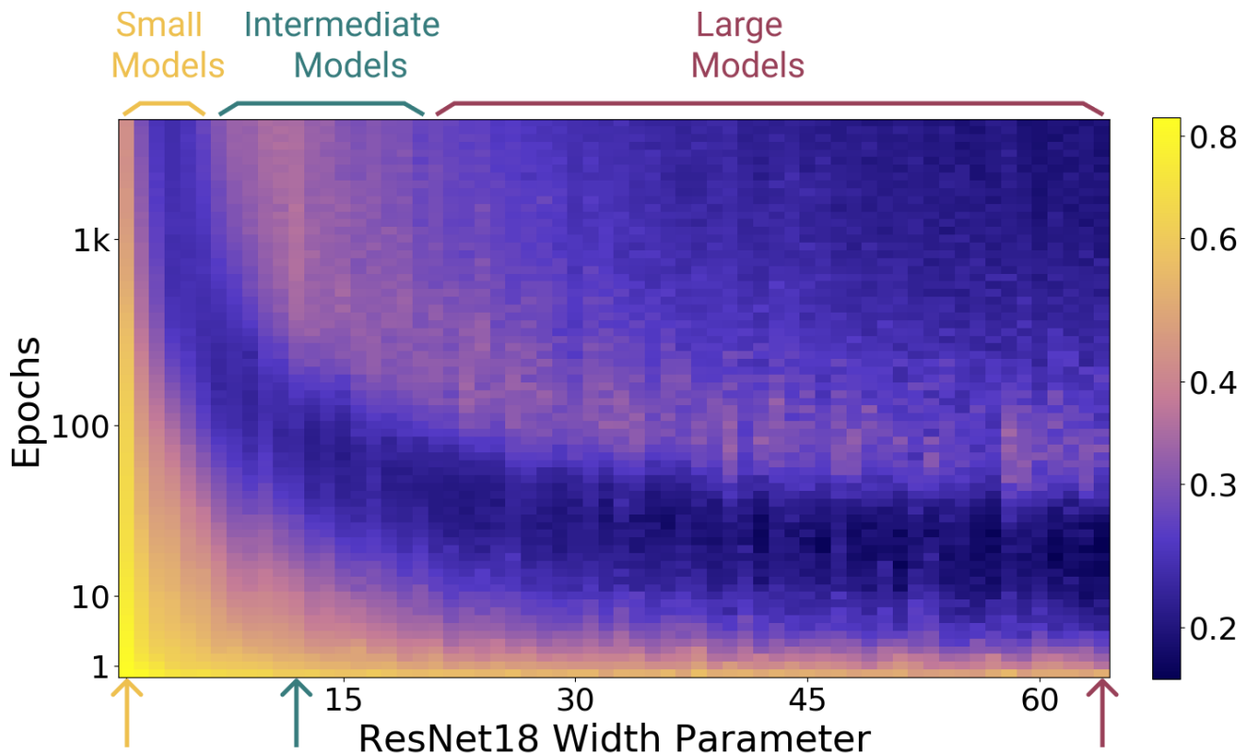
[Submitted on 4 Dec 2019]

Deep Double Descent: Where Bigger Models and More Data Hurt

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, Ilya Sutskever

https://arxiv.org/abs/1912.02292





Thoughts:

- at critical region, only one model fits the data well and is very sensitive to noise
- overparametrized models: many fit the data well, SGD finds one that memorizes the training set but also performs well on the test set

arXiv.org > cs > arXiv:1912.02292

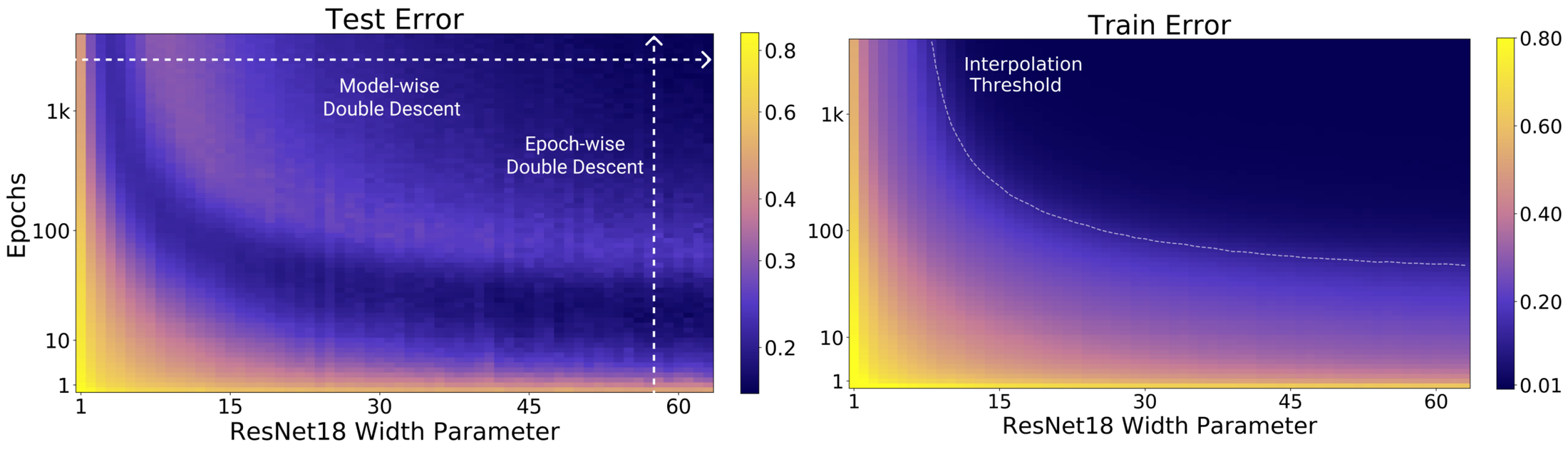
Computer Science > Machine Learning

[Submitted on 4 Dec 2019]

Deep Double Descent: Where Bigger Models and More Data Hurt

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, Ilya Sutskever

https://arxiv.org/abs/1912.02292



arXiv.org > cs > arXiv:1912.02292

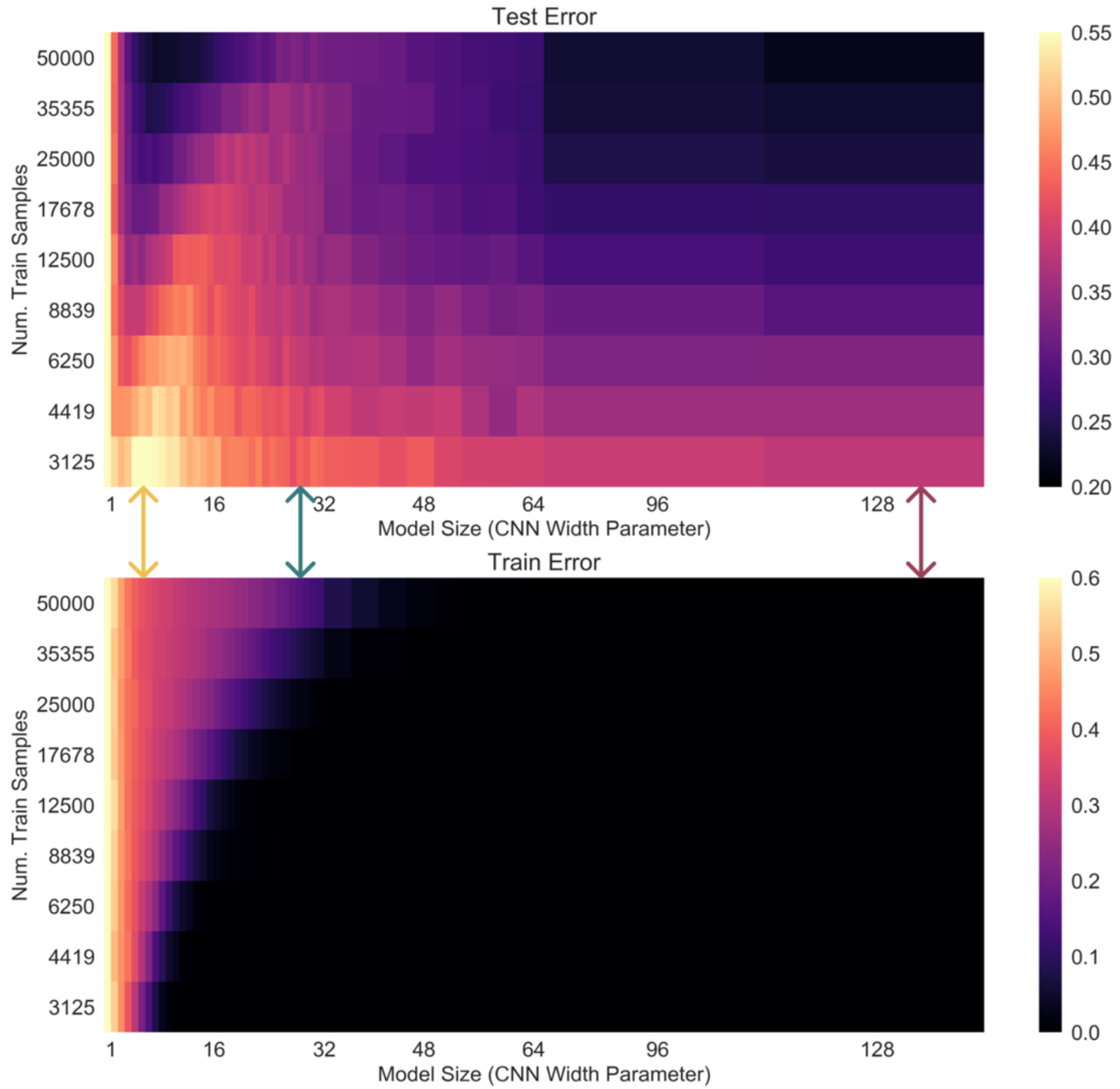
Computer Science > Machine Learning

[Submitted on 4 Dec 2019]

Deep Double Descent: Where Bigger Models and More Data Hurt

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, Ilya Sutskever

https://arxiv.org/abs/1912.02292

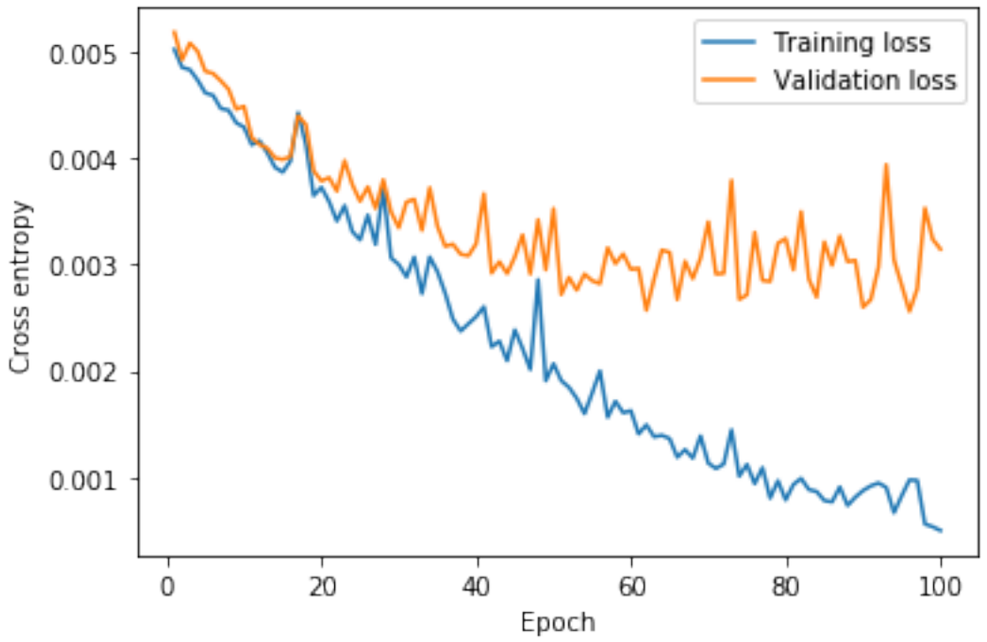


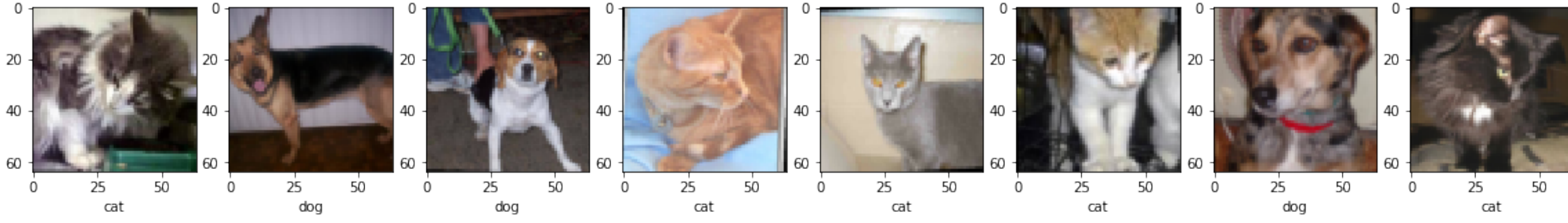
Training multilayer perceptrons with your own datasets

1. Multilayer Perceptron Architecture
2. Nonlinear Activation Functions
3. Multilayer Perceptron Code Examples
4. Overfitting and Underfitting
5. **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%
```

https://github.com/rasbt/deeplearning-models/blob/master/pytorch_ipynb/cnn/cnn-vgg16-cats-dogs.ipynb

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
...
```

```
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}%')
```

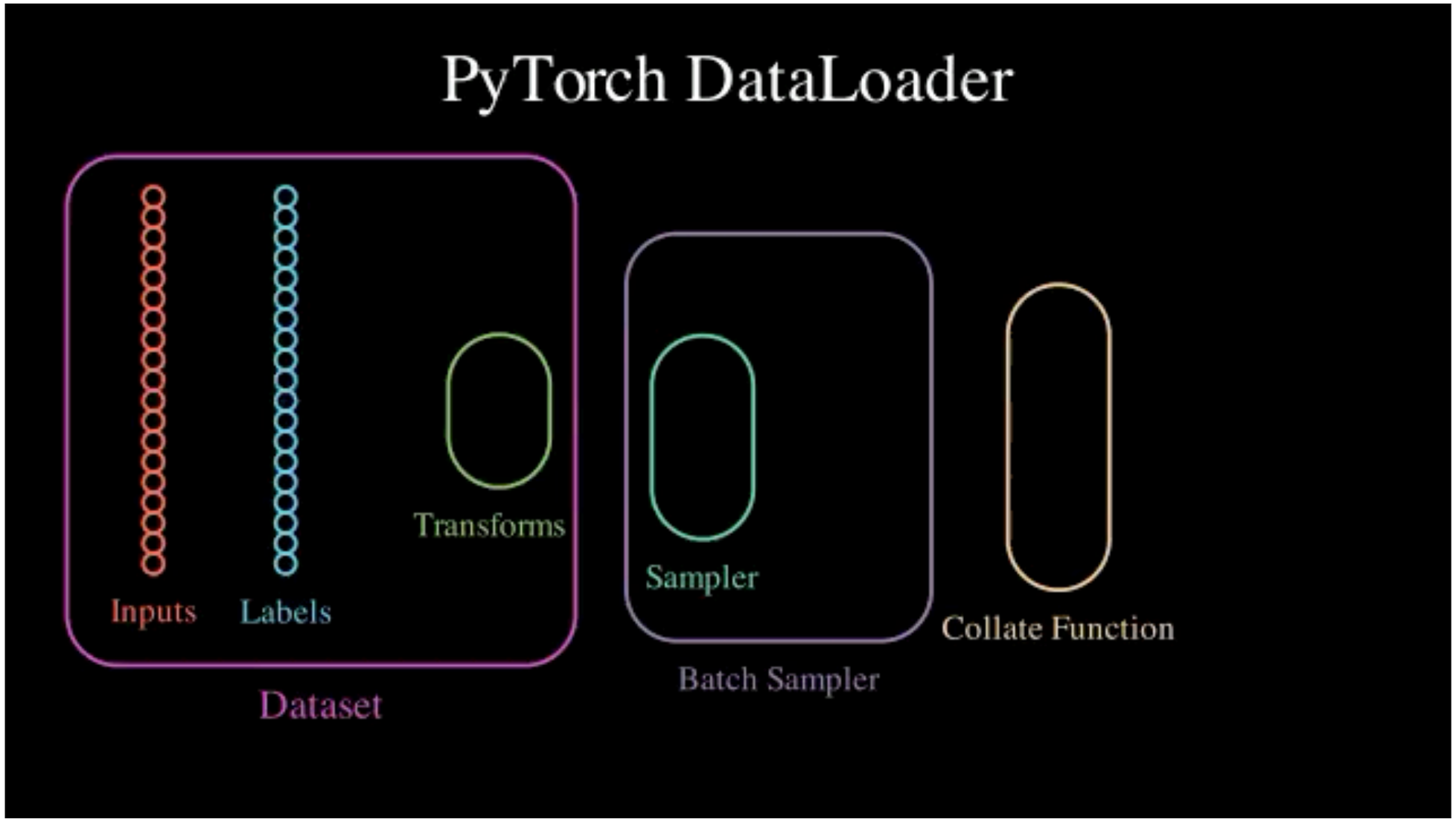
```
Test accuracy: 88.28%
```

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)
- ...

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

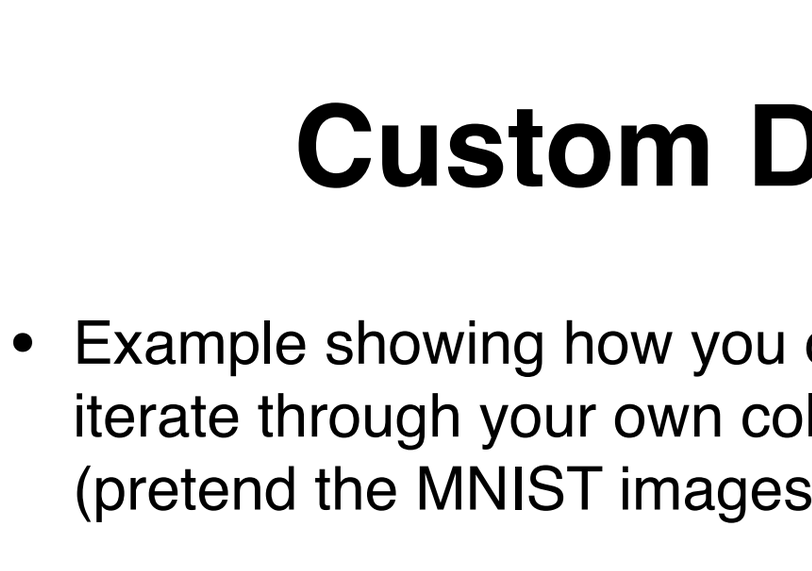


https://twitter.com/_ScottCondron/status/1363494433715552259?s=20

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/main/L09/code/custom-dataloader




```
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]
```

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