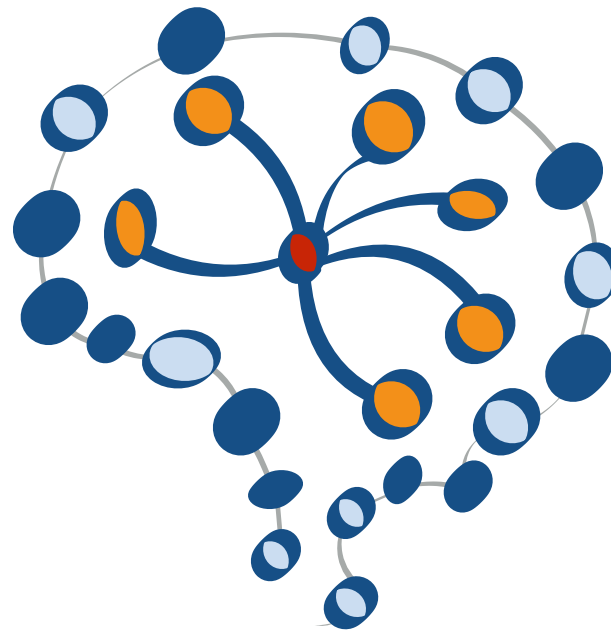


# STAT 453: Introduction to Deep Learning and Generative Models

Sebastian Raschka

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



## Lecture 10

# Regularization Methods for Neural Networks

# Goal: Reduce Overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions (as explained last lecture)

# Regularization

In the context of deep learning, regularization can be understood as the process of adding information / changing the objective function to prevent overfitting

# Regularization / Regularizing Effects

Goal: reduce overfitting

usually achieved by reducing model capacity and/or reduction of the variance of the predictions (as explained last lecture)

## Common Regularization Techniques for DNNs:

- Early stopping
- $L_1/L_2$  regularization (norm penalties)
- Dropout

# Regularization (mathematics)

From Wikipedia, the free encyclopedia



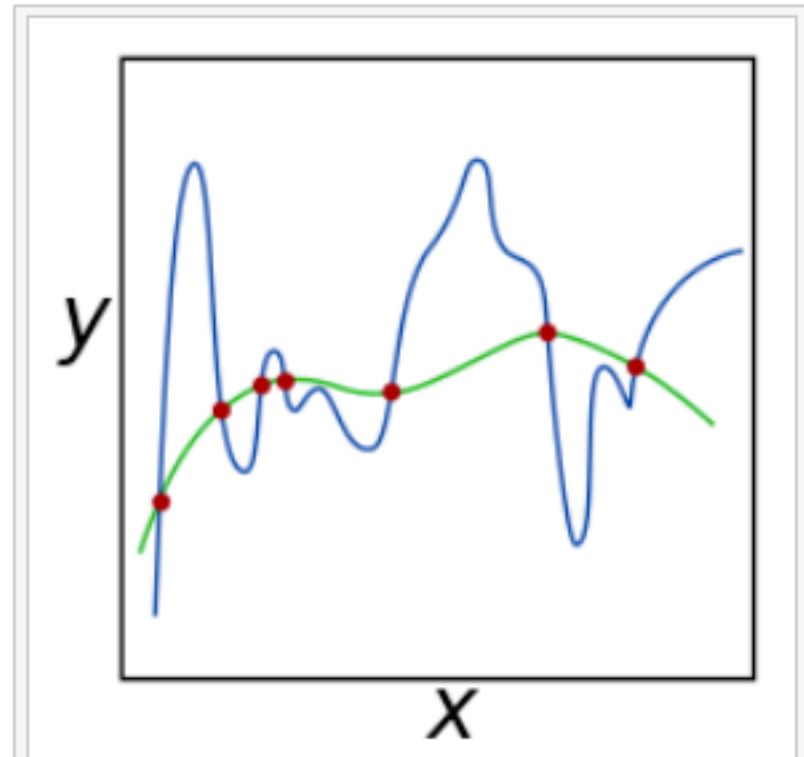
This article **only describes one highly specialized aspect of its associated subject**. Please help [improve this article](#) by adding more general information. The [talk page](#) may contain suggestions. *(November 2020)*

In [mathematics](#), [statistics](#), [finance](#),<sup>[1]</sup> [computer science](#), particularly in [machine learning](#) and [inverse problems](#), **regularization** is the process of adding information in order to solve an [ill-posed problem](#) or to prevent [overfitting](#).<sup>[2]</sup>

Regularization applies to objective functions in ill-posed optimization problems. The regularization term, or penalty, imposes a cost on the optimization function for overfitting the function or to find an optimal solution.

In [machine learning](#), [regularization is any modification](#) one makes to a learning algorithm that is intended to reduce its generalization error but not its training error<sup>[3]</sup>

**Contents** [\[hide\]](#)



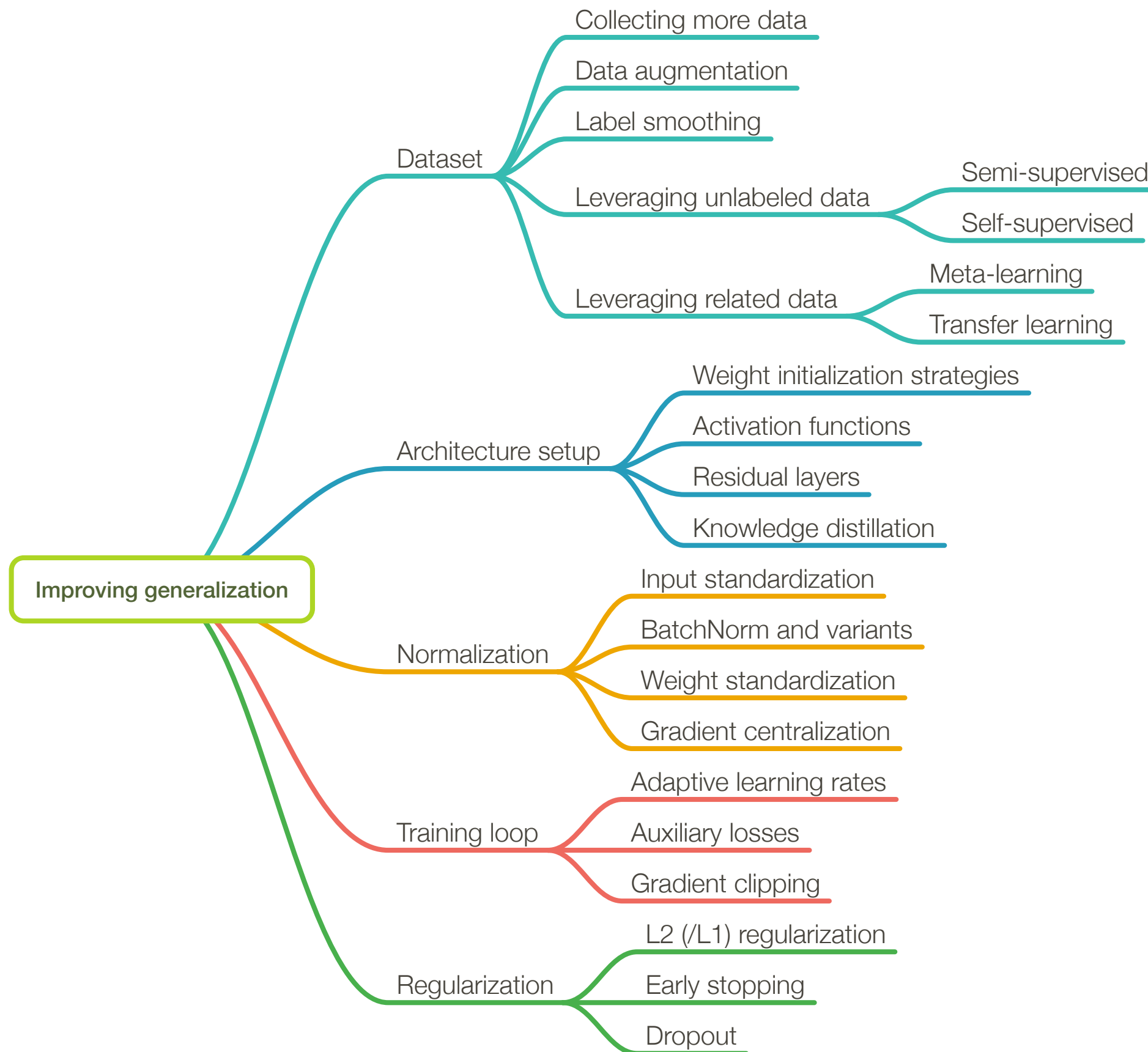
The green and blue functions both incur zero loss on the given data points. A learned

# Lecture Overview

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

# An Overview of Techniques for ...

- 1. Improving generalization performance**
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

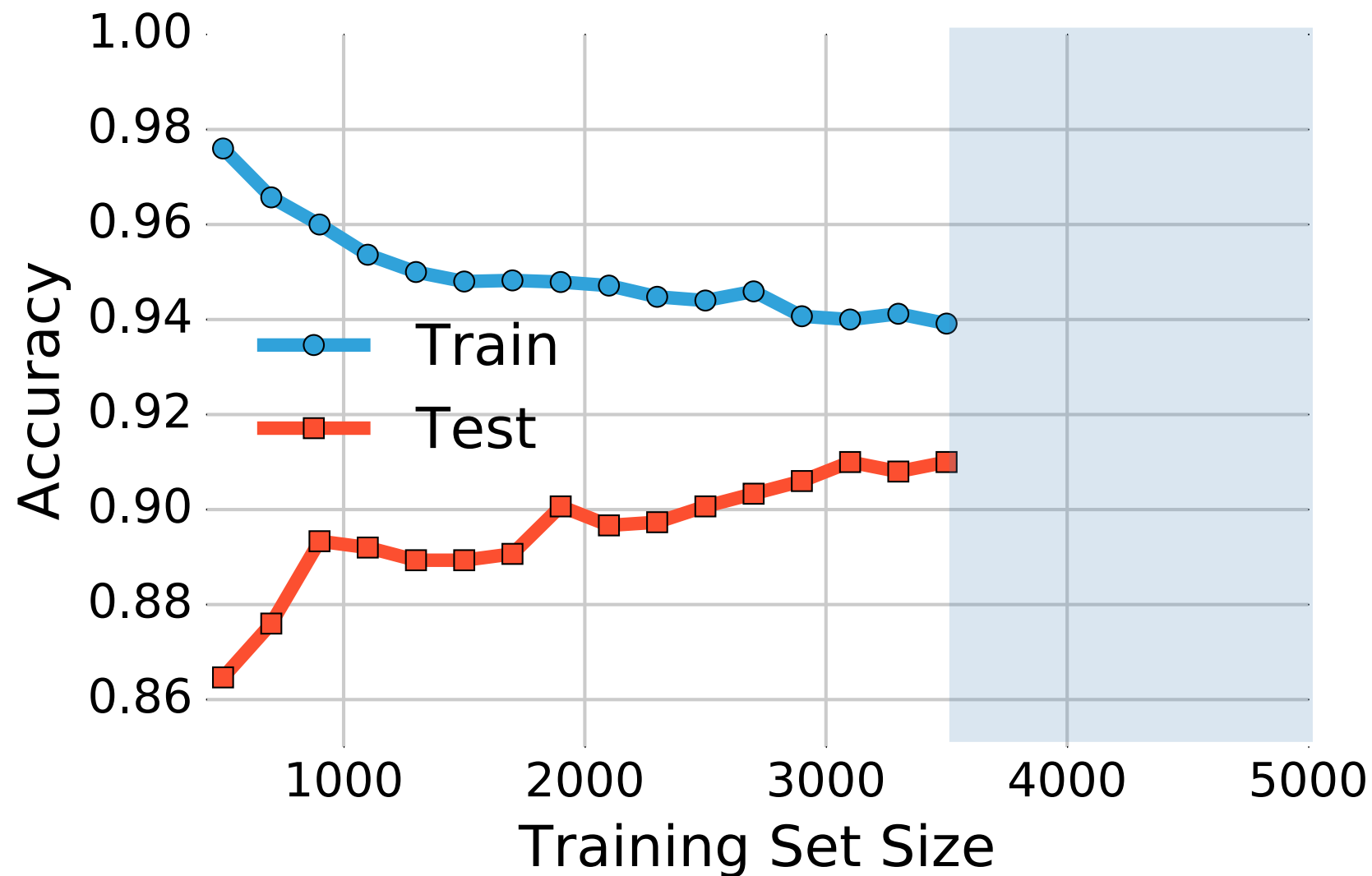




# First step to improve performance: Focusing on the dataset itself

1. Improving generalization performance
- 2. Avoiding overfitting with (1) more data and (2) data augmentation**
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

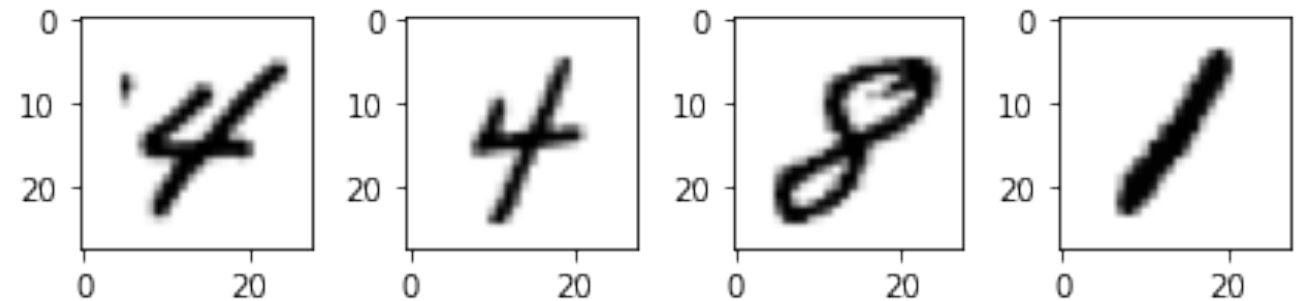
# Often, the Best Way to Reduce Overfitting is Collecting More Data



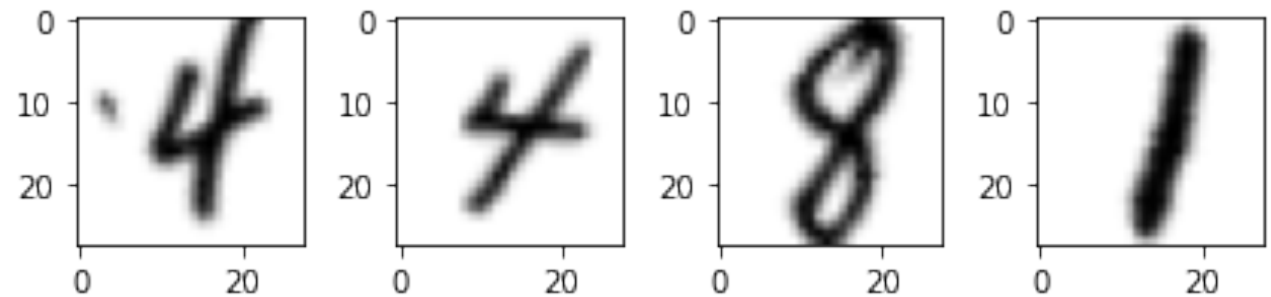
Softmax on MNIST subset (test set size is kept constant)

# Data Augmentation in PyTorch via TorchVision

Original



Randomly Augmented



<https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L10/code/data-augmentation.ipynb>

```

# Note transforms.ToTensor() scales input images
# to 0-1 range

training_transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomRotation(degrees=30, interpolation=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.CenterCrop(size=(28, 28)),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
])

# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html

train_dataset = datasets.MNIST(root='data',
                                train=True,
                                transform=training_transforms,
                                download=True)

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

```

<https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L10/code/data-augmentation.ipynb>

```

# Note transforms.ToTensor() scales input images
# to 0-1 range

training_transforms = torchvision.transforms.Compose([
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomRotation(degrees=30, interpolation=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
    # normalize does (x_i - mean) / std
    # if images are [0, 1], they will be [-1, 1] afterwards
])

test_transforms = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Resize(size=(32, 32)),
    torchvision.transforms.CenterCrop(size=(28, 28)),
    torchvision.transforms.Normalize(mean=(0.5,), std=(0.5,)),
])

# for more see
# https://pytorch.org/docs/stable/torchvision/transforms.html

train_dataset = datasets.MNIST(root='data',
                                train=True,
                                transform=training_transforms,
                                download=True)

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

```

Use (0.5, 0.5, 0.5) for RGB images

<https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L10/code/data-augmentation.ipynb>

# Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible

## => Reducing Network's Capacity by Other Means

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
- 3. Reducing network capacity & early stopping**
4. Adding norm penalties to the loss: L1 & L2 regularization
5. Dropout

# Early Stopping

Step 1: Split your dataset into 3 parts (always recommended)

- use test set only once at the end (for unbiased estimate of generalization performance)
- use validation accuracy for tuning (always recommended)

## Dataset



Training  
dataset

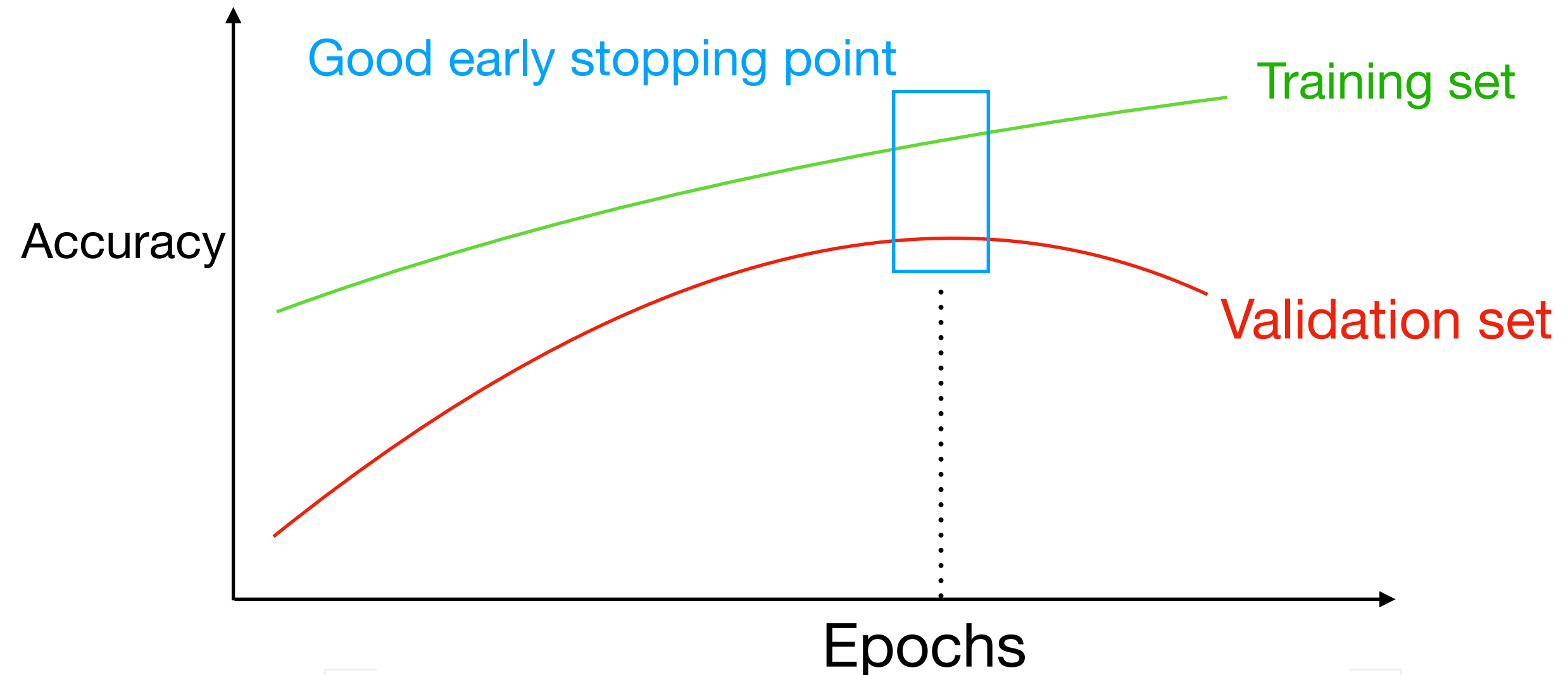
Validation  
dataset

Test  
dataset

# Early Stopping

## Step 2: Early stopping (not very common anymore)

- reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point





# Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible

## Adding a Penalty Against Complexity

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. **Adding norm penalties to the loss: L1 & L2 regularization**
5. Dropout

# $L_1/L_2$ Regularization

As I am sure you already know it from various statistics classes, we will keep it short:

- $L_1$ -regularization  $\Rightarrow$  LASSO regression
- $L_2$ -regularization  $\Rightarrow$  Ridge regression (Thikonov regularization)

Basically, a "weight shrinkage" or a "penalty against complexity"

# L<sub>2</sub> Regularization for Linear Models (e.g., Logistic Regression)

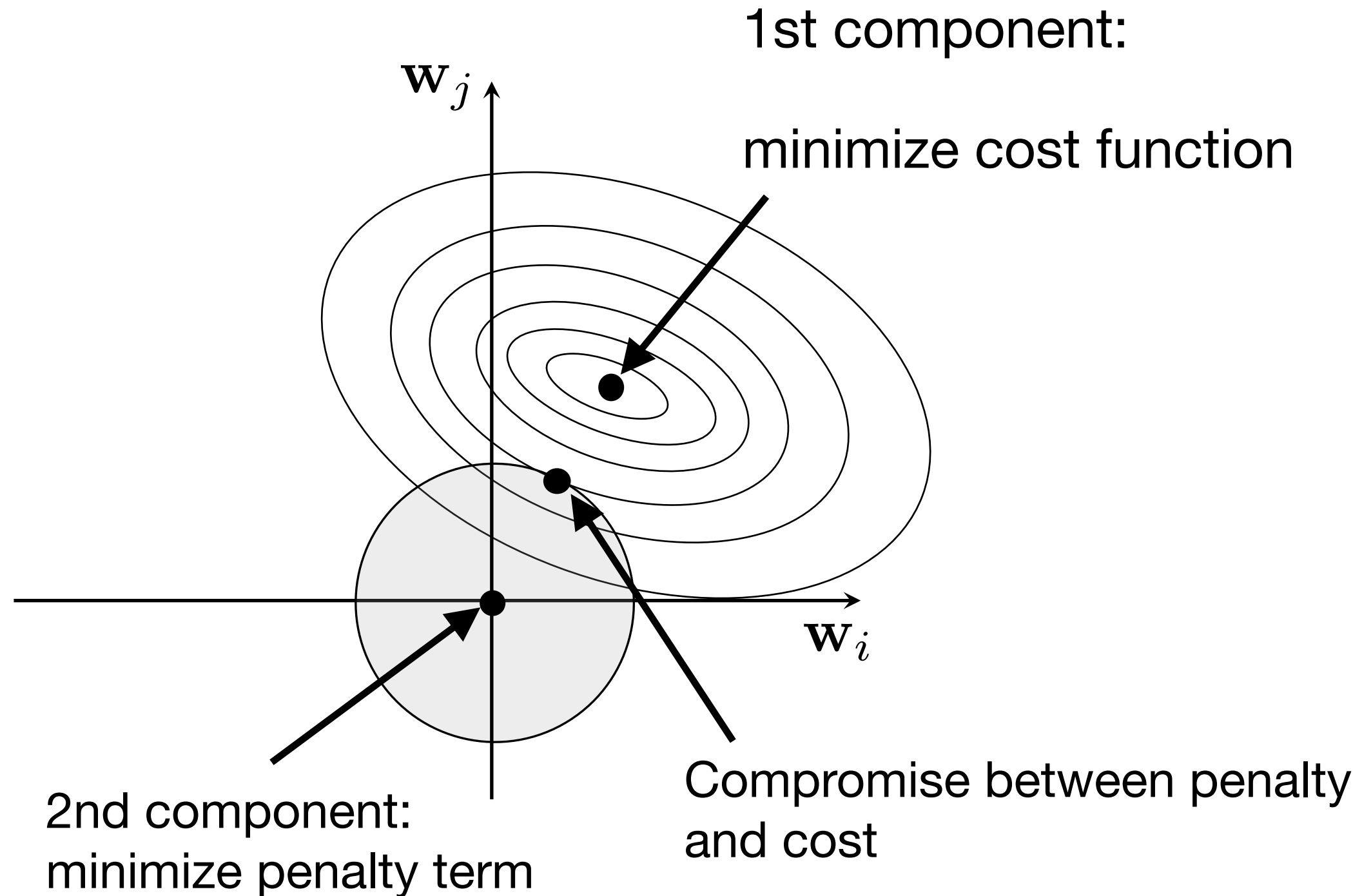
$$\text{Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]})$$

$$\text{L2-Regularized-Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_j w_j^2$$

where:  $\sum_j w_j^2 = \|\mathbf{w}\|_2^2$

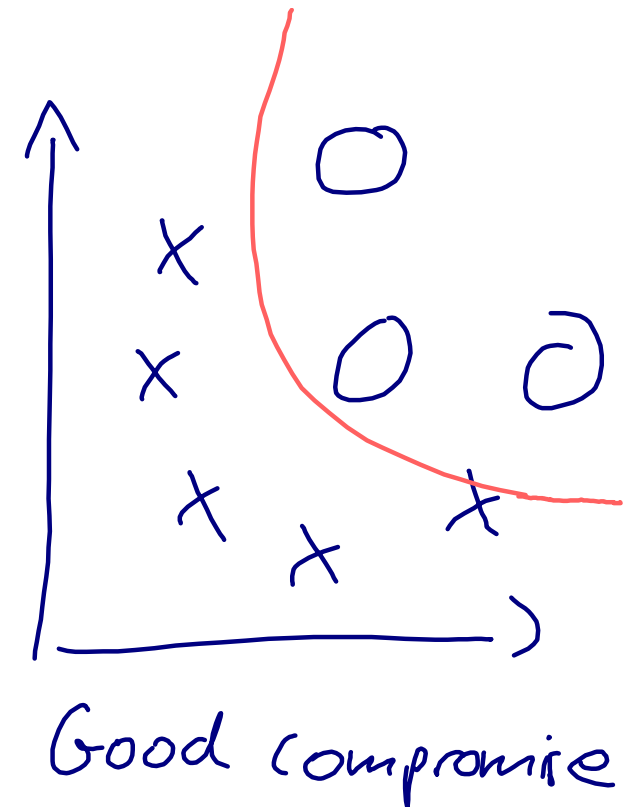
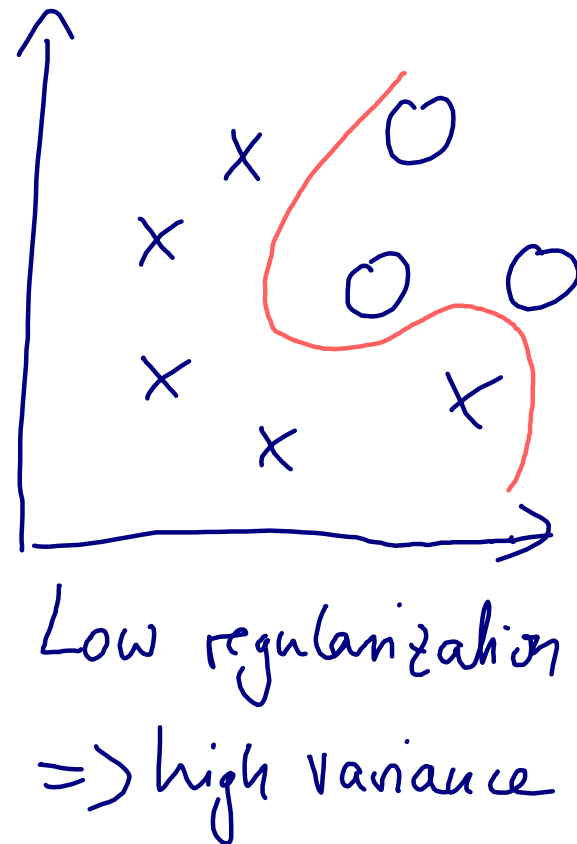
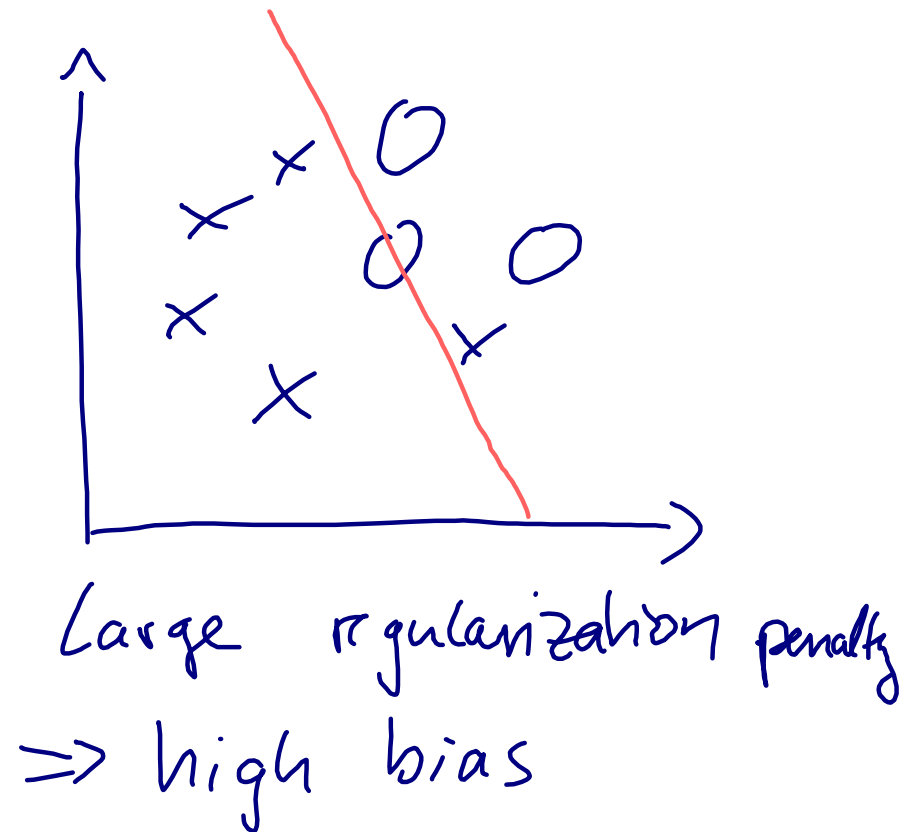
and  $\lambda$  is a hyperparameter

# Geometric Interpretation of L<sub>2</sub> Regularization



# Effect of Norm Penalties on the Decision Boundary

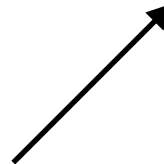
Assume a nonlinear model



# L<sub>2</sub> Regularization for Multilayer Neural Networks

$$\text{L2-Regularized-Cost}_{\mathbf{w}, \mathbf{b}} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n} \sum_{l=1}^L ||\mathbf{w}^{(l)}||_F^2$$

sum over layers



where  $||\mathbf{w}^{(l)}||_F^2$  is the Frobenius norm (squared):

$$||\mathbf{w}^{(l)}||_F^2 = \sum_i \sum_j (w_{i,j}^{(l)})^2$$

# L<sub>2</sub> Regularization for Neural Nets

Regular gradient descent update:

$$w_{i,j} := w_{i,j} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}}$$

Gradient descent update with L2 regularization:

$$w_{i,j} := w_{i,j} - \eta \left( \frac{\partial \mathcal{L}}{\partial w_{i,j}} + \frac{2\lambda}{n} w_{i,j} \right)$$

# L<sub>2</sub> Regularization for Neural Nets in PyTorch

```
# regularize loss
L2 = 0.
for name, p in model.named_parameters():
    if 'weight' in name:
        L2 = L2 + (p**2).sum()

cost = cost + 2./targets.size(0) * LAMBDA * L2

optimizer.zero_grad()
cost.backward()
```



# L<sub>2</sub> Regularization for Logistic Regression in PyTorch

## Automatically:

```
#####  
## Apply L2 regularization  
optimizer = torch.optim.SGD(model.parameters(),  
                             lr=0.1,  
                             weight_decay=LAMBDA)  
#-----
```

```
for epoch in range(num_epochs):
```

```
    ##### Compute outputs #####  
    out = model(X_train_tensor)
```

```
    ##### Compute gradients #####  
    cost = F.binary_cross_entropy(out, y_train_tensor)  
    optimizer.zero_grad()  
    cost.backward()
```

# Dropout

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization

## **5. Dropout**

### **5.1 The Main Concept Behind Dropout**

5.2 Dropout: Co-Adaptation Interpretation

5.3 Dropout: Ensemble Method Interpretation

5.4 Dropout in PyTorch

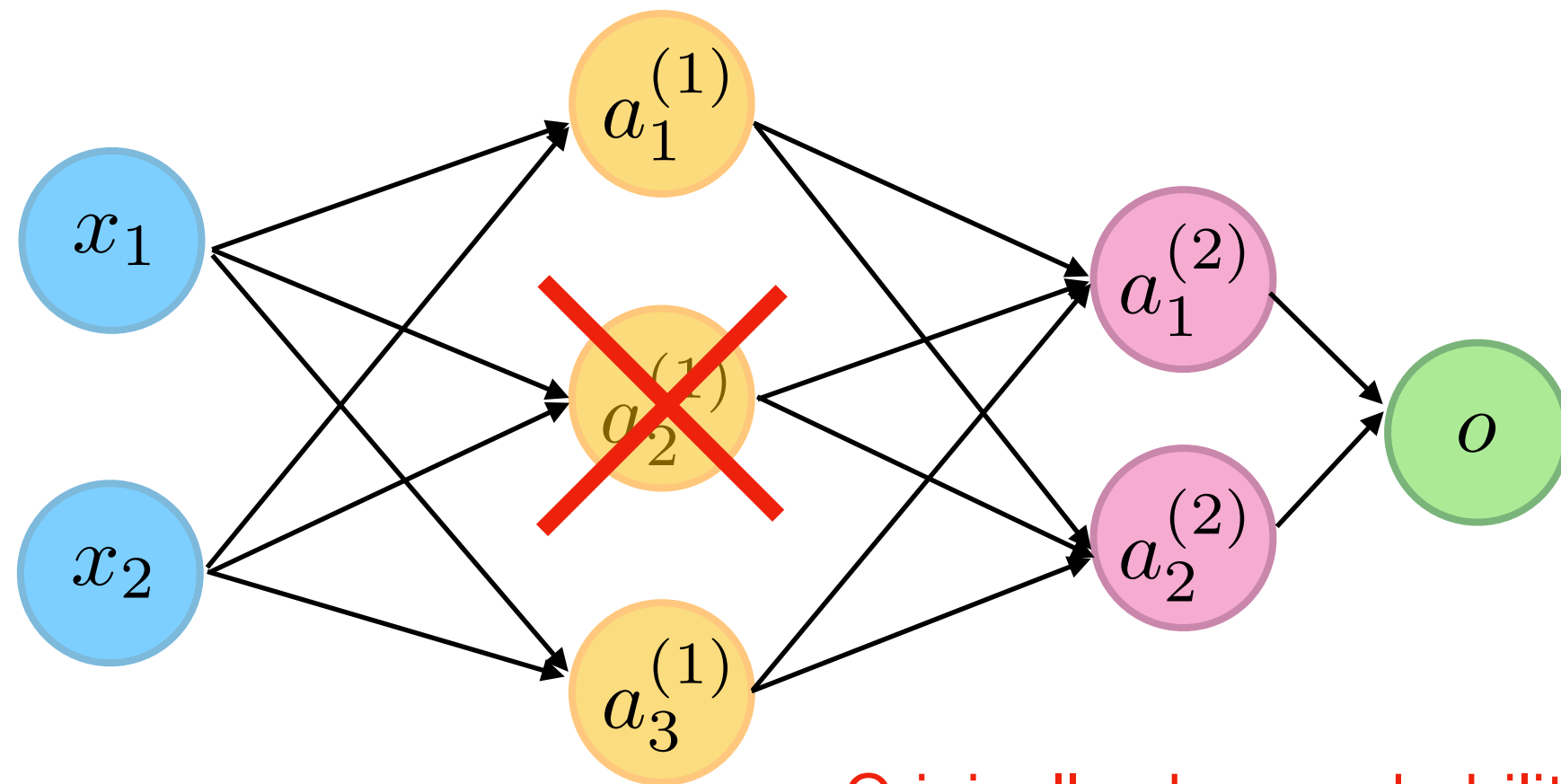
# Dropout

## Original research articles:

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929-1958.

# Dropout in a Nutshell: Dropping Nodes



Originally, drop probability 0.5

(but 0.2-0.8 also common now)

# Dropout in a Nutshell: Dropping Nodes

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- $p :=$  drop probability
- $\mathbf{v} :=$  random sample from uniform distribution in range  $[0, 1]$
- $\forall i \in \mathbf{v} : v_i := 0$  if  $v_i < p$  else 1
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$  *( $p \times 100\%$  of the activations  $\mathbf{a}$  will be zeroed)*

# Dropout in a Nutshell: Dropping Nodes

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- $p :=$  drop probability
- $\mathbf{v} :=$  random sample from uniform distribution in range  $[0, 1]$
- $\forall i \in \mathbf{v} : v_i := 0$  if  $v_i < p$  else 1
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$  *( $p \times 100\%$  of the activations  $\mathbf{a}$  will be zeroed)*

Then, after training when making predictions (during "inference")

scale activations via  $\mathbf{a} := \mathbf{a} \odot (1 - p)$

Q for you: Why is this required?

# Dropout

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization

## **5. Dropout**

5.1 The Main Concept Behind Dropout

### **5.2 Dropout: Co-Adaptation Interpretation**

5.3 Dropout: Ensemble Method Interpretation

5.4 Dropout in PyTorch

# Dropout: Co-Adaptation Interpretation

Why does Dropout work well?

- Network will learn not to rely on particular connections too heavily
- Thus, will consider more connections (because it cannot rely on individual ones)
- The weight values will be more spread-out (may lead to smaller weights like with L2 norm)
- Side note: You can certainly use different dropout probabilities in different layers (assigning them proportional to the number of units in a layer is not a bad idea, for example)



# Dropout

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization

## **5. Dropout**

5.1 The Main Concept Behind Dropout

5.2 Dropout: Co-Adaptation Interpretation

**5.3 Dropout: Ensemble Method Interpretation**

5.4 Dropout in PyTorch

# Dropout: Ensemble Method Interpretation

- In dropout, we have a "different model" for each minibatch
- Via the minibatch iterations, we essentially sample over  $M=2^h$  models, where  $h$  is the number of hidden units
- Restriction is that we have weight sharing over these models, which can be seen as a form of regularization
- During "inference" we can then average over all these models (but this is very expensive)

# Dropout: Ensemble Method Interpretation

- During "inference" we can then average over all these models (but this is very expensive)

This is basically just averaging log likelihoods (this is for one particular class):

$$p_{\text{Ensemble}} = \left[ \prod_{j=1}^M p^{\{i\}} \right]^{1/M} = \exp \left[ 1/M \sum_{j=1}^M \log(p^{\{i\}}) \right]$$

(you may know this as the "geometric mean" from other classes)

For multiple classes, we need to normalize so that the probas

sum  
to 1:

$$p_{\text{Ensemble}, j} = \frac{p_{\text{Ensemble}, j}}{\sum_{j=1}^k p_{\text{Ensemble}, j}}$$

# Dropout: Ensemble Method Interpretation

- During "inference" we can then average over all these models (but this is very expensive)
- However, using the last model after training and scaling the predictions by a factor  $1-p$  approximates the geometric mean and is much cheaper (actually, it's exactly the geometric mean if we have a linear model)

# Dropout

1. Improving generalization performance
2. Avoiding overfitting with (1) more data and (2) data augmentation
3. Reducing network capacity & early stopping
4. Adding norm penalties to the loss: L1 & L2 regularization

## **5. Dropout**

5.1 The Main Concept Behind Dropout

5.2 Dropout: Co-Adaptation Interpretation

5.3 Dropout: Ensemble Method Interpretation

**5.4 Dropout in PyTorch**

# Inverted Dropout

- Most frameworks implement inverted dropout
- Here, the activation values are scaled by the factor  $(1-p)$  during training instead of scaling the activations during "inference"
- I believe Google started this trend (because it's computationally cheaper in the long run if you use your model a lot after training)
- PyTorch's Dropout implementation is also inverted Dropout

# Dropout in PyTorch

```
class MultilayerPerceptron(torch.nn.Module):  
  
    def __init__(self, num_features, num_classes, drop_proba,  
                  num_hidden_1, num_hidden_2):  
        super().__init__()  
  
        self.my_network = torch.nn.Sequential(  
            # 1st hidden layer  
            torch.nn.Flatten(),  
            torch.nn.Linear(num_features, num_hidden_1),  
            torch.nn.ReLU(),  
            torch.nn.Dropout(drop_proba),  
            # 2nd hidden layer  
            torch.nn.Linear(num_hidden_1, num_hidden_2),  
            torch.nn.ReLU(),  
            torch.nn.Dropout(drop_proba),  
            # output layer  
            torch.nn.Linear(num_hidden_2, num_classes)  
        )  
  
    def forward(self, x):  
        logits = self.my_network(x)  
        return logits
```

# Dropout in PyTorch

Here, it is very important that you use `model.train()` and `model.eval()`!

```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(DEVICE)

        ### FORWARD AND BACK PROP
        logits = model(features)

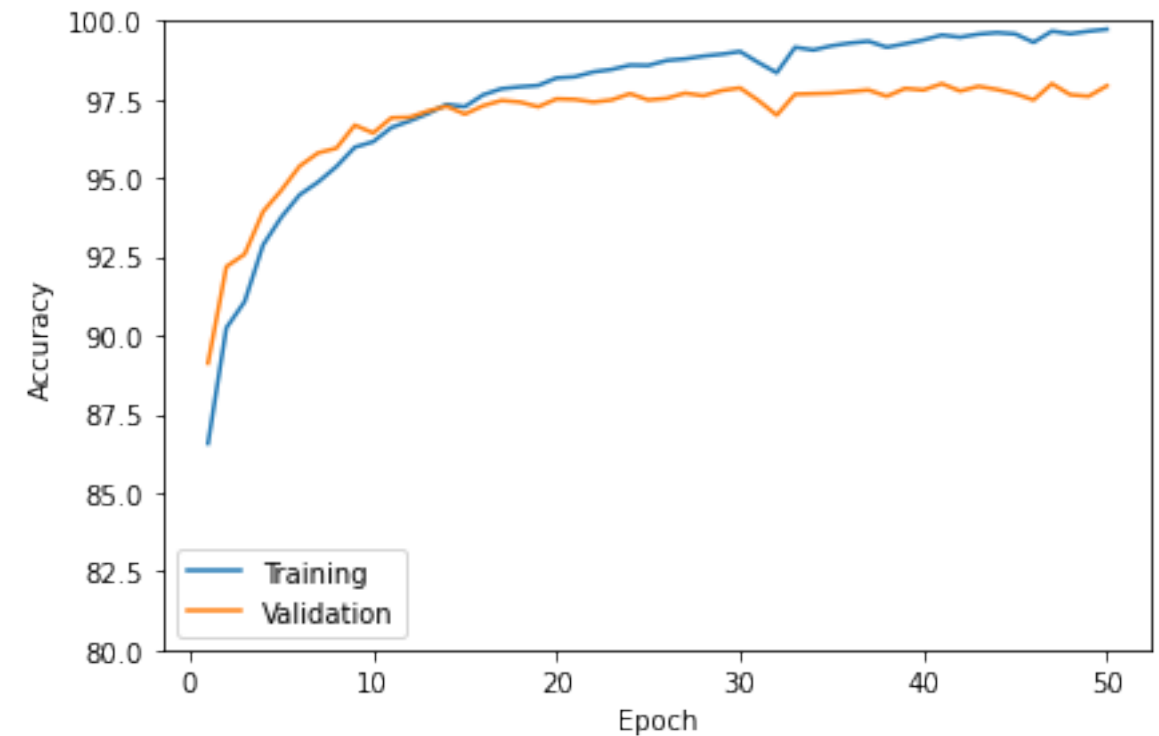
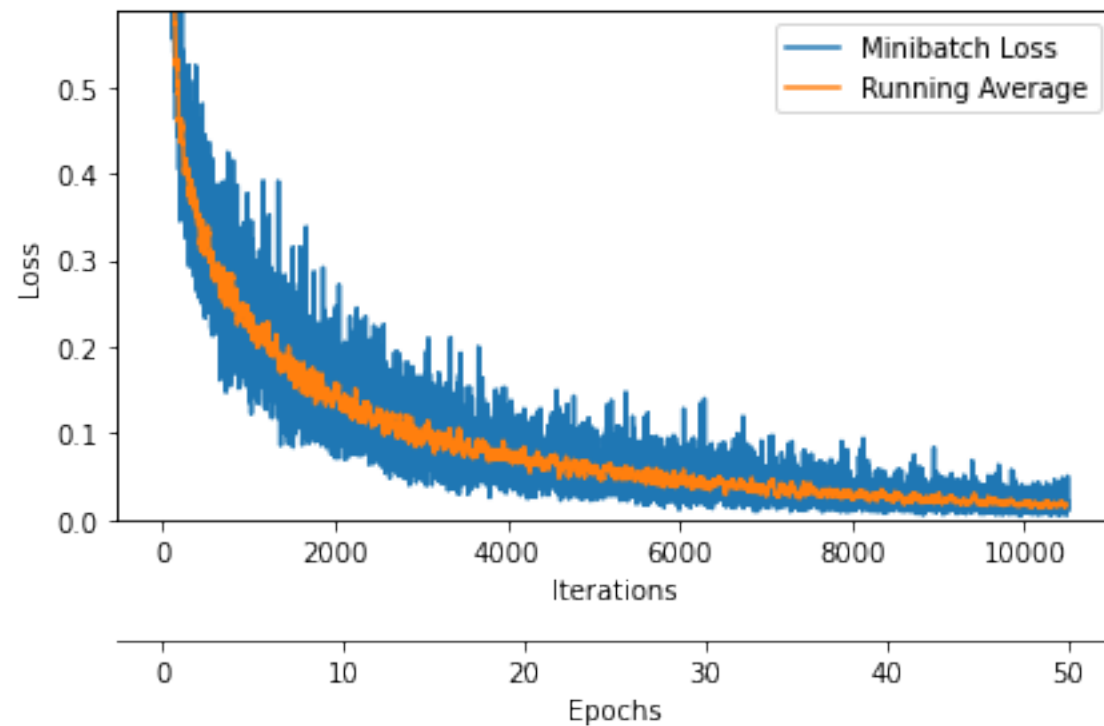
        cost = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        cost.backward()
        minibatch_cost.append(cost)
        ### UPDATE MODEL PARAMETERS
        optimizer.step()

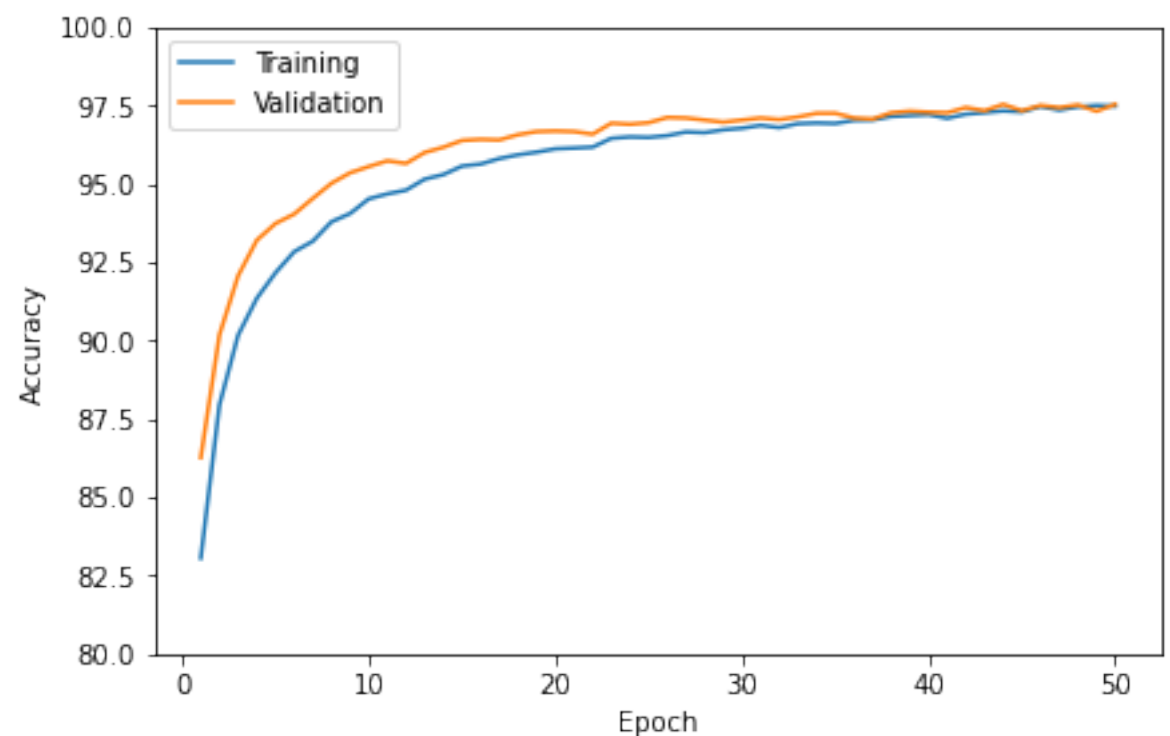
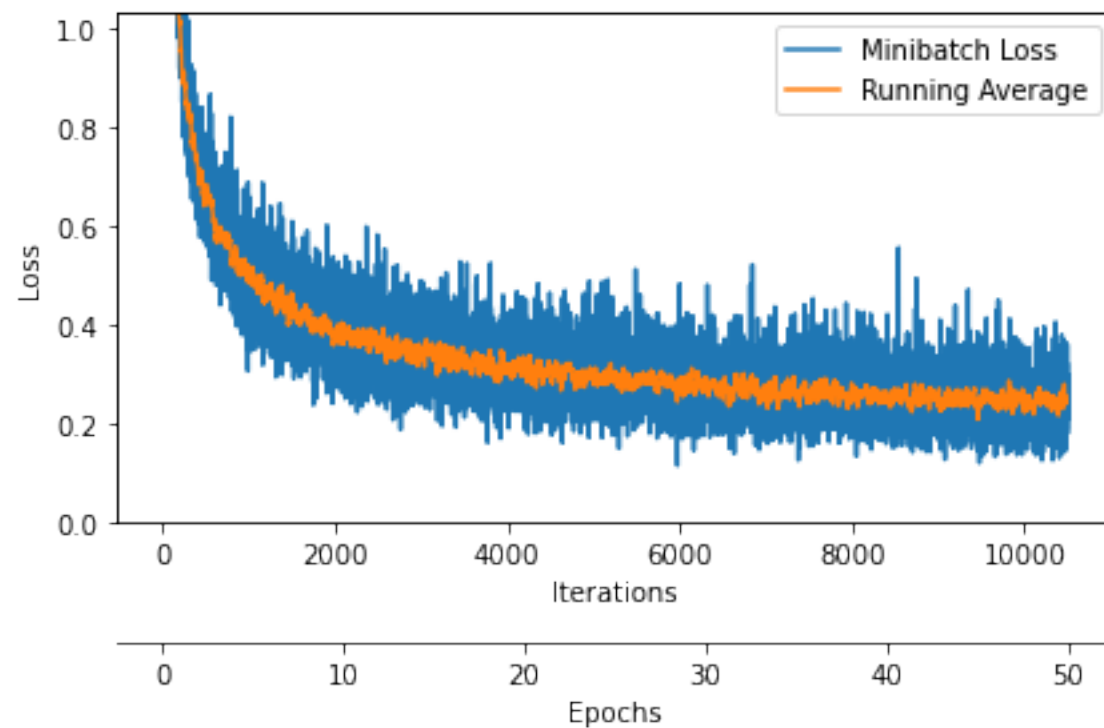
    model.eval()
    with torch.no_grad():
        cost = compute_loss(model, train_loader)
        epoch_cost.append(cost)
        print('Epoch: %03d/%03d Train Cost: %.4f' % (
            epoch+1, NUM_EPOCHS, cost))
        print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))
```



# Without dropout:



# With 50% dropout:

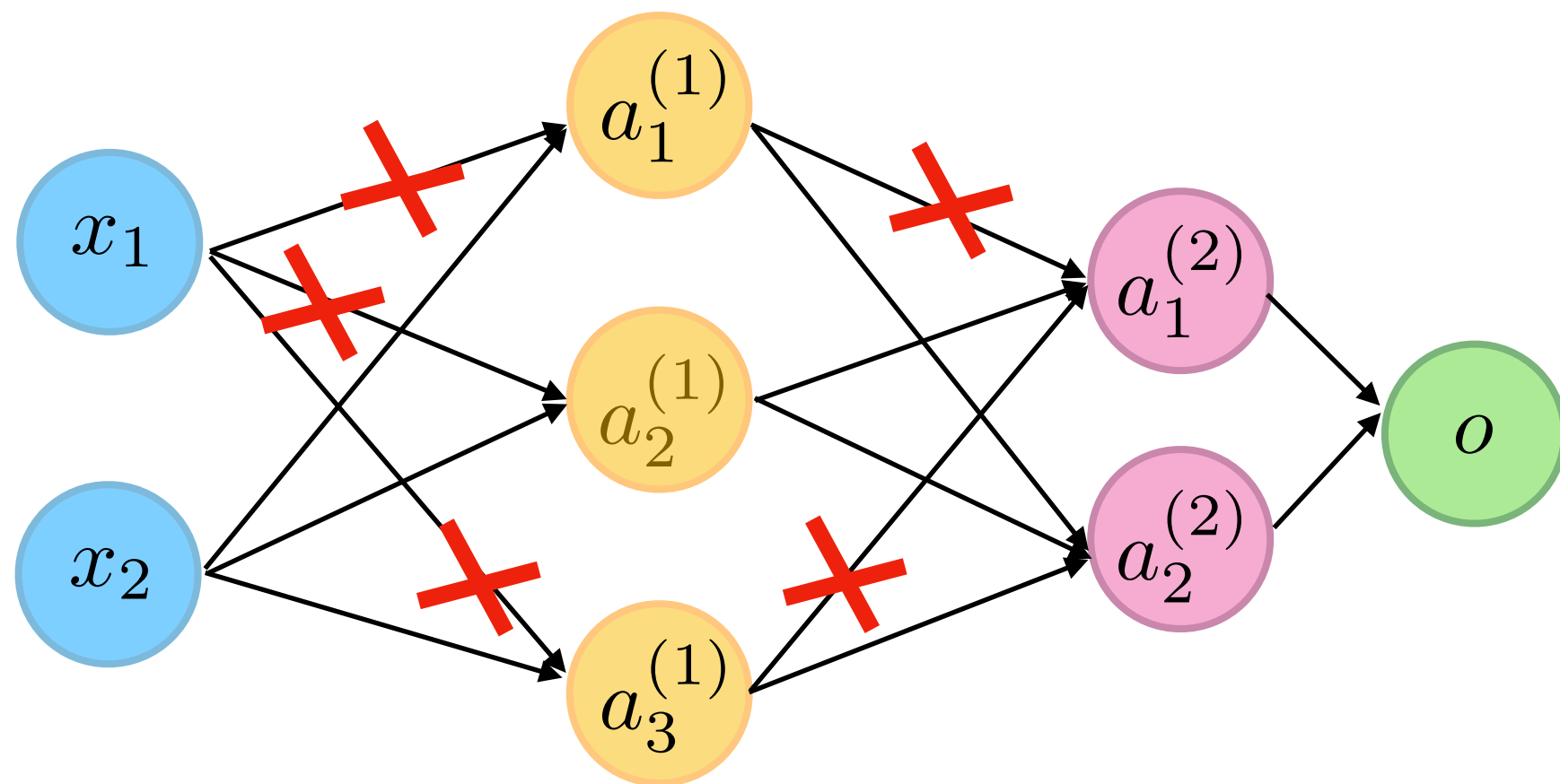


<https://github.com/rasbt/stat453-deep-learning-ss21/blob/master/L10/code/dropout.ipynb>

# Dropout: More Practical Tips

- Don't use Dropout if your model does not overfit
- However, in that case above, it is then recommended to increase the capacity to make it overfit, and then use dropout to be able to use a larger capacity model (but make it not overfit)

# DropConnect: Randomly Dropping Weights



# DropConnect

- Generalization of Dropout
- More "possibilities"
- Less popular & doesn't work so well in practice

## Original research article:

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., & Fergus, R. (2013, February). Regularization of neural networks using DropConnect. In *International conference on machine learning* (pp. 1058-1066).

# Recommended Reading Assignment

- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1), 1929-1958.  
<http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>

