Lecture 10

Regularization

STAT 479: Deep Learning, Spring 2019

Sebastian Raschka

http://stat.wisc.edu/~sraschka/teaching/stat479-ss2019/

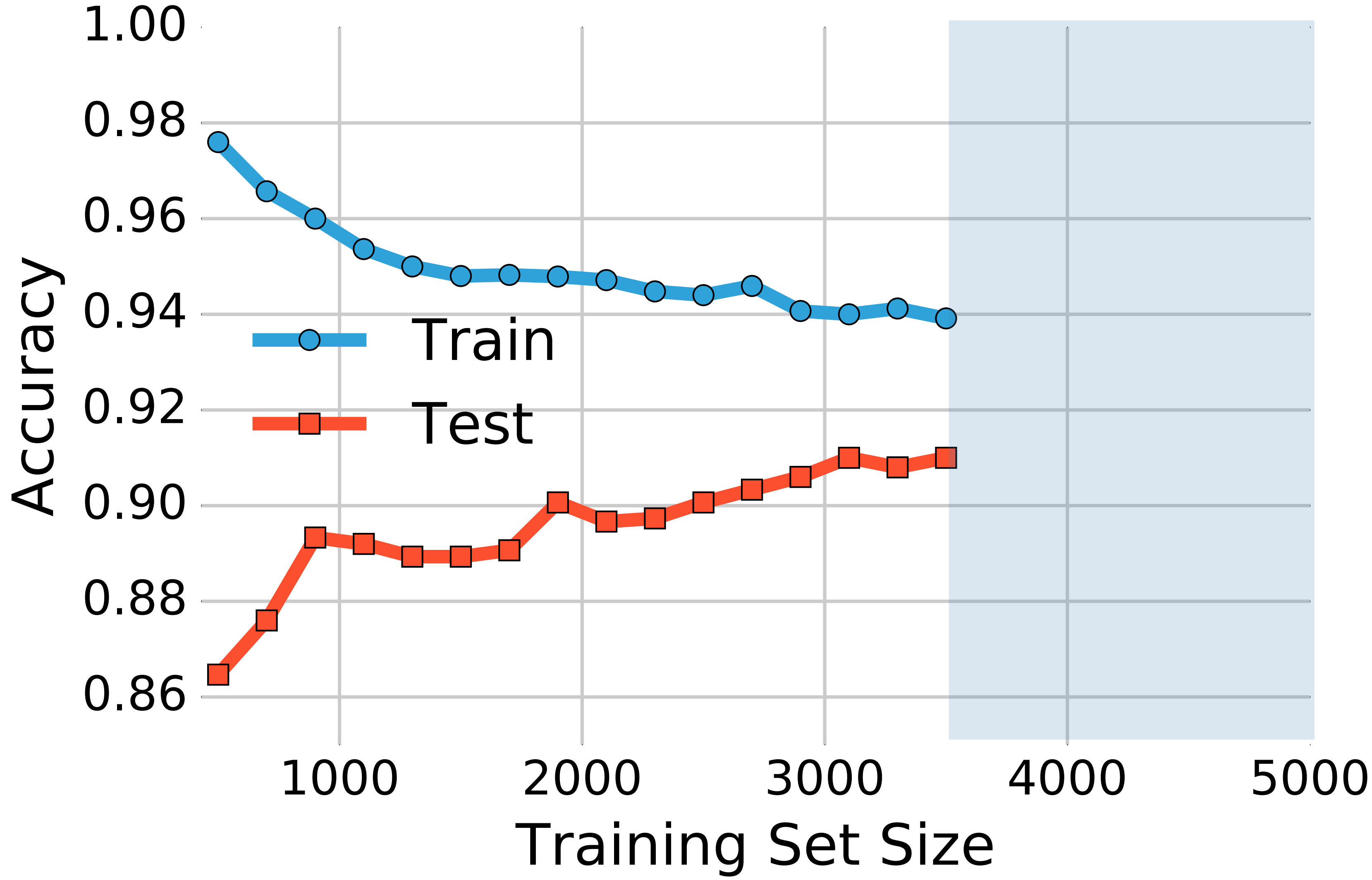
Overview: Regularization / Regularizing Effects

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

Goal: reduce overfitting

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

Best Way to Reduce Overfitting is Collecting More Data



Softmax on MNIST subset (kept test set size constant)

Best Way to Reduce Overfitting is Collecting More Data

- Collecting more data is always recommended
- If not possible, data augmentation is also helpful (e.g., for images: random rotation, crop, translation ...) -- actually, this is always recommended (and easy to do)
- Additionally, reducing the capacity (e.g., regularization) helps

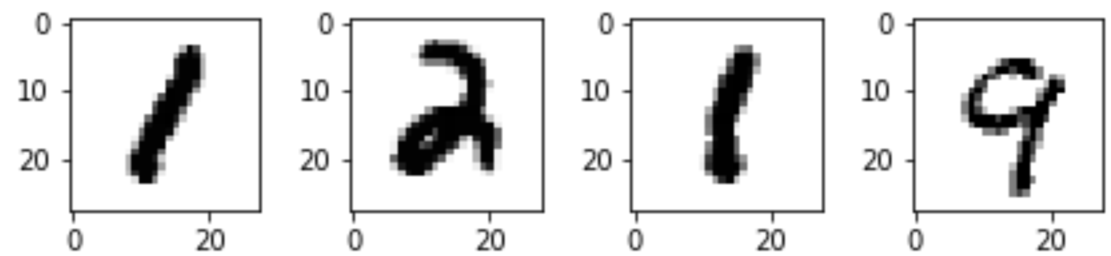
(In statistics, I notice the tendency to come up with more and more complex modeling techniques, based on heavy and unrealistic assumptions, whereas usually the data amount and quality is the real bottleneck ... e.g., a Bayes Optimal Classifier is not really more useful than logistic regression if the data is no good => "garbage in garbage out" principle)

Data Augmentation in PyTorch via TorchVision

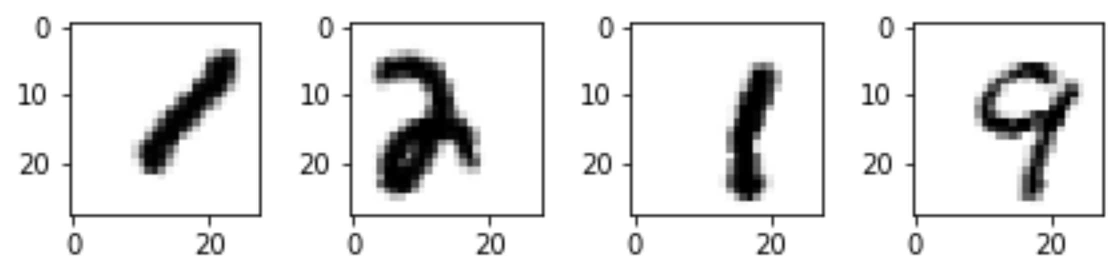
```
training_transforms = torchvision.transforms.Compose([
    #torchvision.transforms.RandomRotation(degrees=20),
    #torchvision.transforms.Resize(size=(34, 34)),
    #torchvision.transforms.RandomCrop(size=(28, 28)),
    torchvision.transforms.RandomAffine(degrees=(-20, 20), translate=(0.15, 0.15),
                                        resample=PIL.Image.BILINEAR),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 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.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 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)
```

Data Augmentation in PyTorch via TorchVision

Original



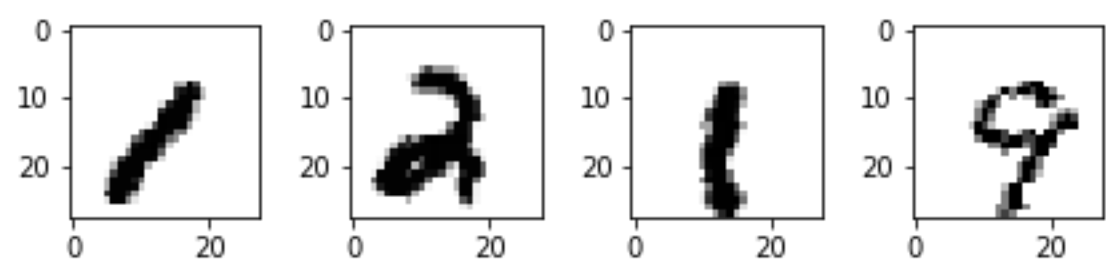
Augmented



note that it is random

Augmented w/o

`resample=PIL.Image.BILINEAR`



note that it is random

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L10_regularization/code/data-augmentation.ipynb

Now: Other Ways for Dealing with Overfitting
if Collecting More Data is not Feasible
=> Reducing Network's Capacity by Other Means

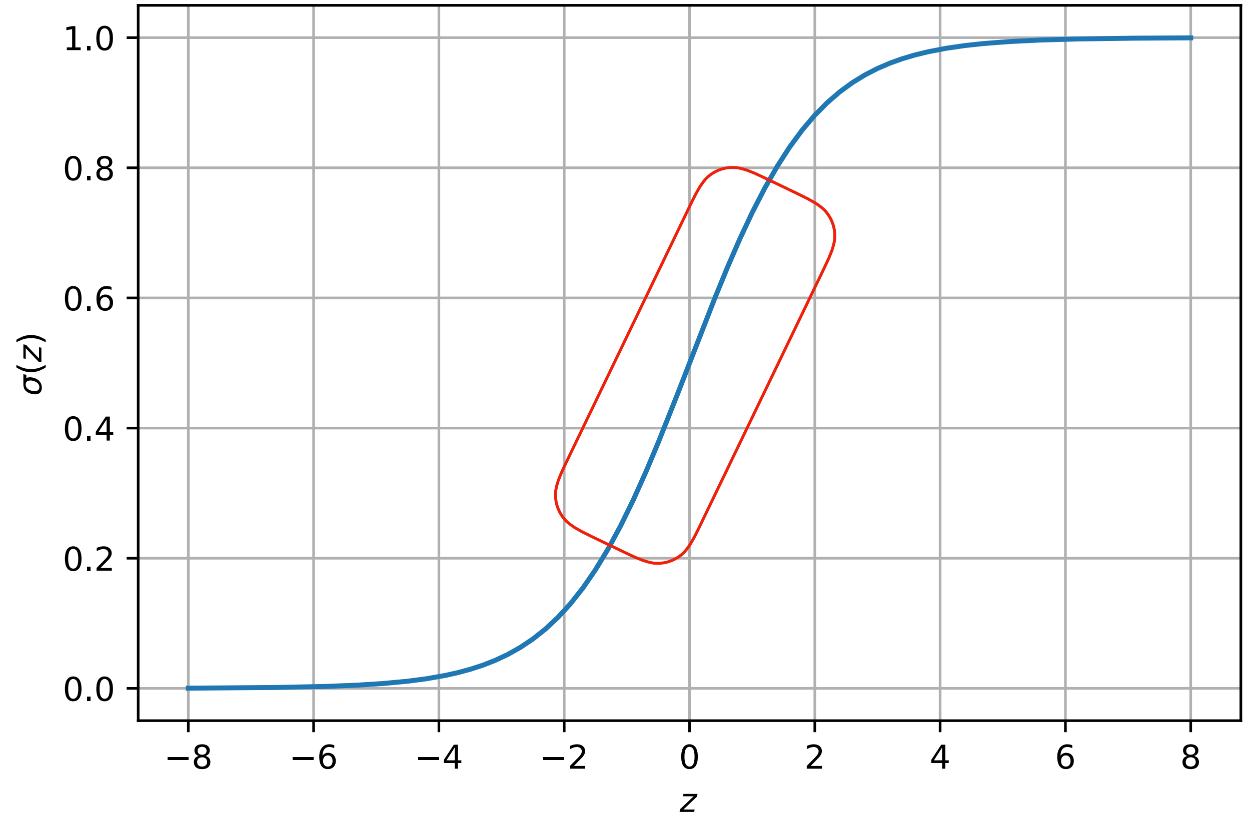
Now: Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible => Reducing Network's Capacity by Other Means

- smaller architecture: fewer hidden layers & units, dropout, (dead ReLUs, L1 norm penalty)
- smaller weights: Early stopping, norm penalties
- adding noise: Dropout

Now: Other Ways for Dealing with Overfitting if Collecting More Data is not Feasible => Reducing Network's Capacity by Other Means

- smaller architecture: fewer hidden layers & units, dropout, (dead ReLUs, L1 norm penalty)
- smaller weights: Early stopping, norm penalties
- adding noise: Dropout

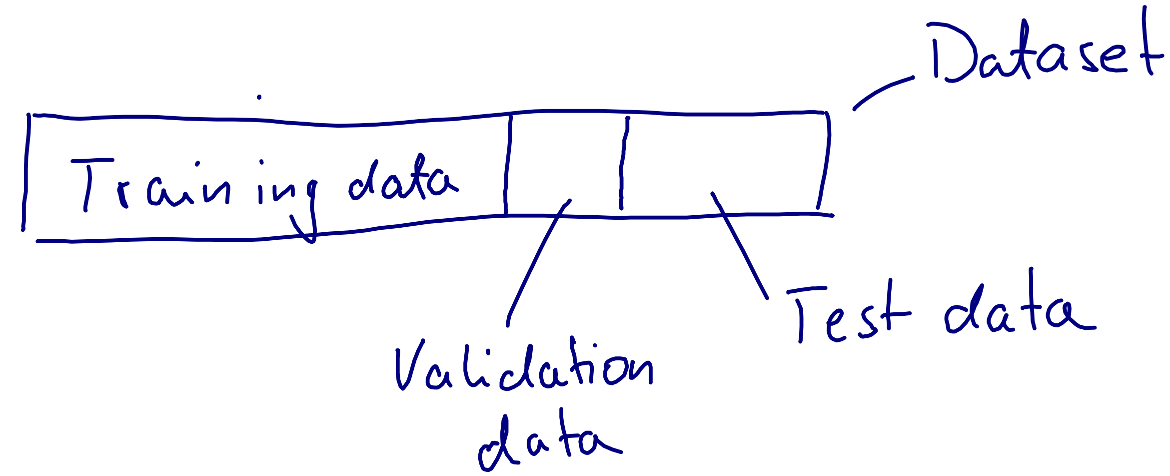
Consider extreme case
(and think of what that leads to,
in context of last lecture)



Early Stopping

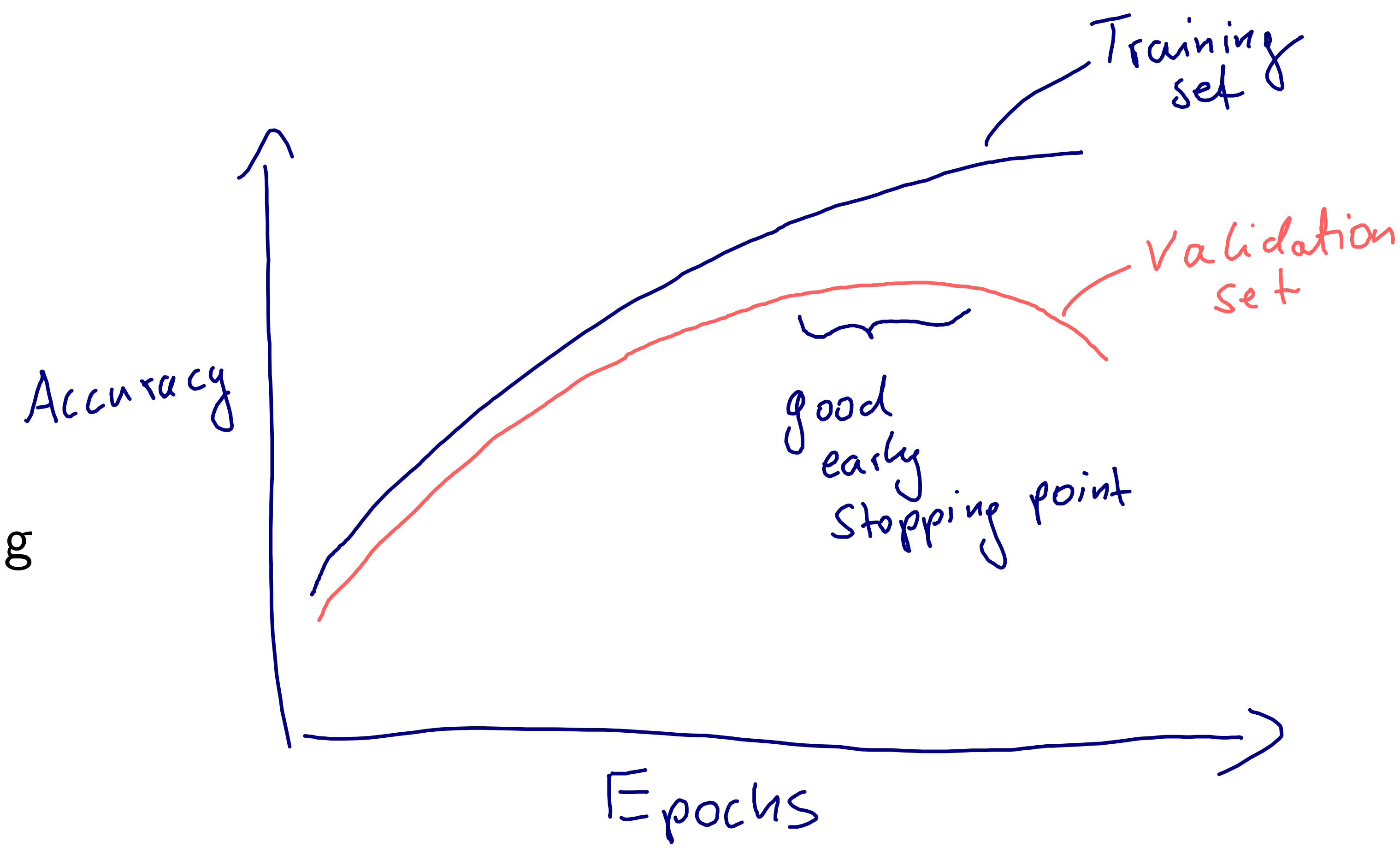
Step 1: Split your dataset into 3 parts (always recommended)

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



Step 2: Early stopping (not very common anymore)

- reduce overfitting by observing the training/validation accuracy gap during training



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 => LASSO regression
- L_2-regularization => Ridge regression (Thikonov regularization)

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

L_1/L_2 Regularization

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

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

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

and λ is a hyperparameter

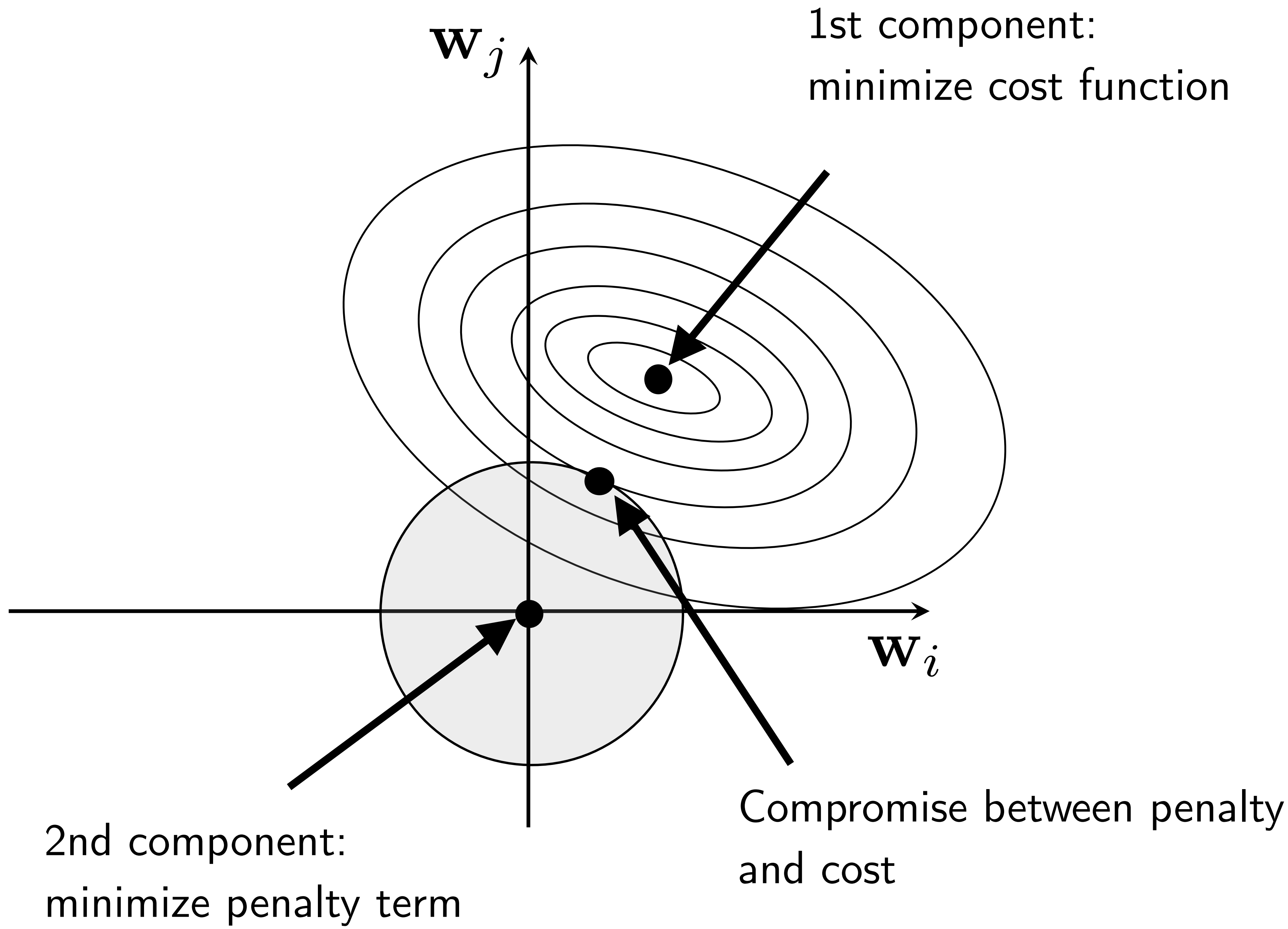
L_1/L_2 Regularization

$$\text{L1-Regularized-Cost}_{\mathbf{w},\mathbf{b}} = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(y^{[i]}, \hat{y}^{[i]}) + \frac{\lambda}{n}\sum_{j}|w_j|$$

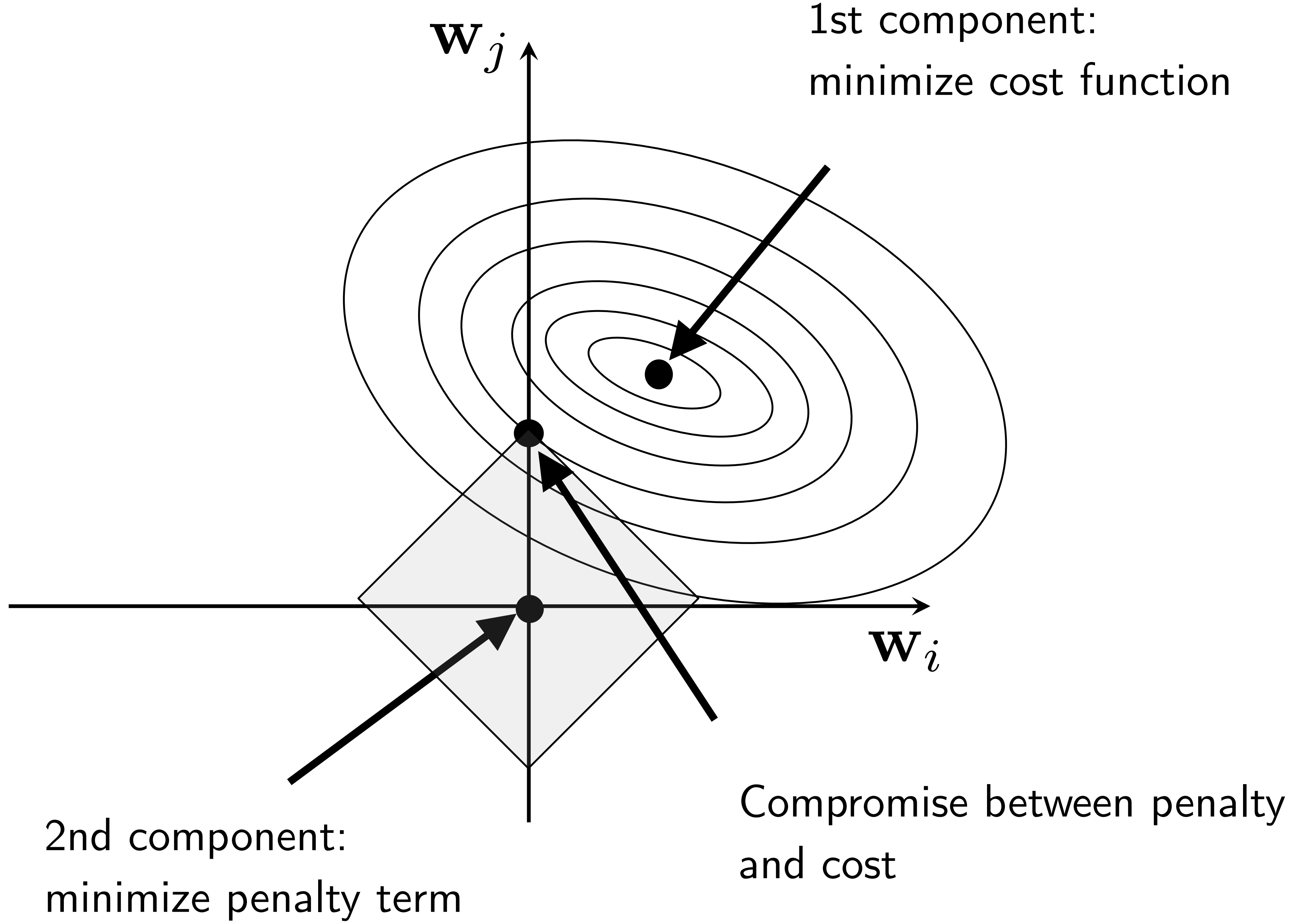
where: $\sum_{j}|w_j| = ||\mathbf{w}||_1$

- L1-regularization encourages sparsity (which may be useful)
- However, usually L1 regularization does not work well in practice and is very rarely used
- Also, it's not smooth and harder to optimize

Geometric Interpretation of L_2 Regularization



Geometric Interpretation of L_2 Regularization



L_2 Regularization for Neural Nets

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

sum over layers

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

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

L_2 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_2 Regularization for Logistic Regression in PyTorch

Manually:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)

for epoch in range(num_epochs):

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

    #### Compute gradients ####

    #########################################################
    ## Apply L2 regularization (weight decay)
    cost = F.binary_cross_entropy(out, y_train_tensor, reduction='sum')
    cost = cost + 0.5 * LAMBDA * torch.mm(model.linear.weight,
                                          model.linear.weight.t())

    # note that PyTorch also regularizes the bias, hence, if we want
    # to reproduce the behavior of SGD's "weight_decay" param, we have to add
    # the bias term as well:
    cost = cost + 0.5 * LAMBDA * model.linear.bias**2
    #--------------------------------------------------------

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

(Note that I am using 0.5 here because PyTorch does it; Could be considered "convenient " as the exponent "2" cancels in the derivative. This implementation exactly matches the one on the next slide)

L2-log-reg.ipynb

L_2 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, reduction='sum')
    optimizer.zero_grad()
    cost.backward()
```

L2-log-reg.ipynb

L_2 Regularization for Neural Nets in PyTorch

- For all layers, same as before ("automatic approach" via **`weight_decay`**)
- Or, manually:

```
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)
        targets = targets.to(DEVICE)

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

        cost = F.cross_entropy(logits, targets)

        # regularize loss
        L2 = 0.
        for p in model.parameters():
            L2 = L2 + (p**2).sum()
        cost = cost + 2./targets.size(0) * LAMBDA * L2

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

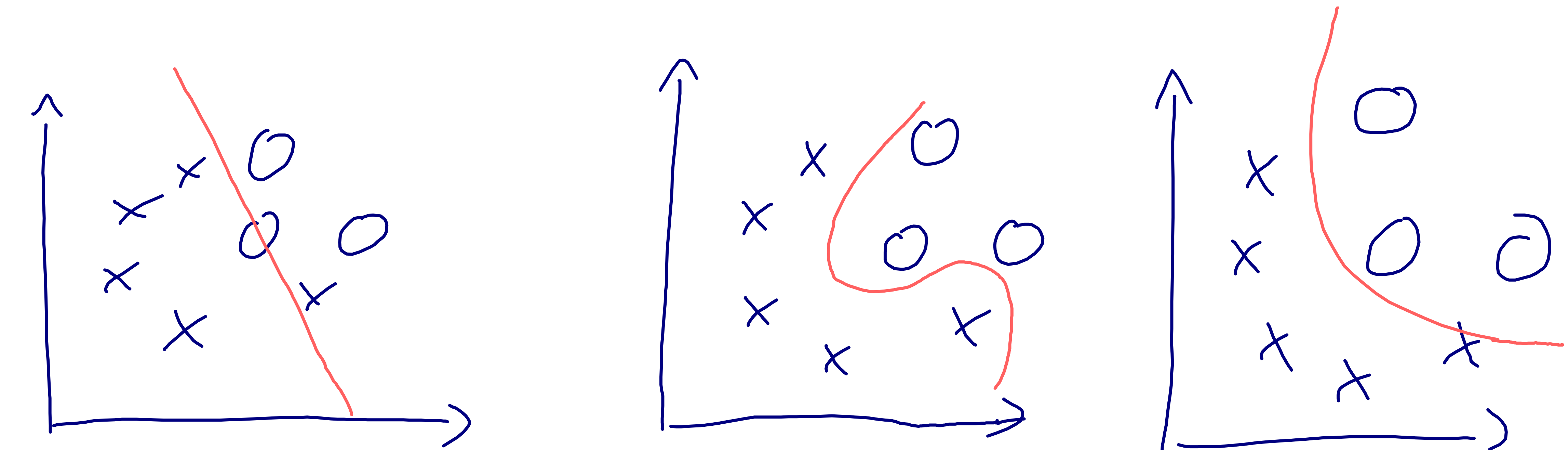
L_2 Regularization for Neural Nets in PyTorch

- Or, if you only want to regularize the weights, not the biases:

```
# 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()
```

Effect of Norm Penalties on the Decision Boundary

Assume a nonlinear model



Large regularization penalty
=> high bias

Low regularization
=> high variance

Good compromise

Dropout

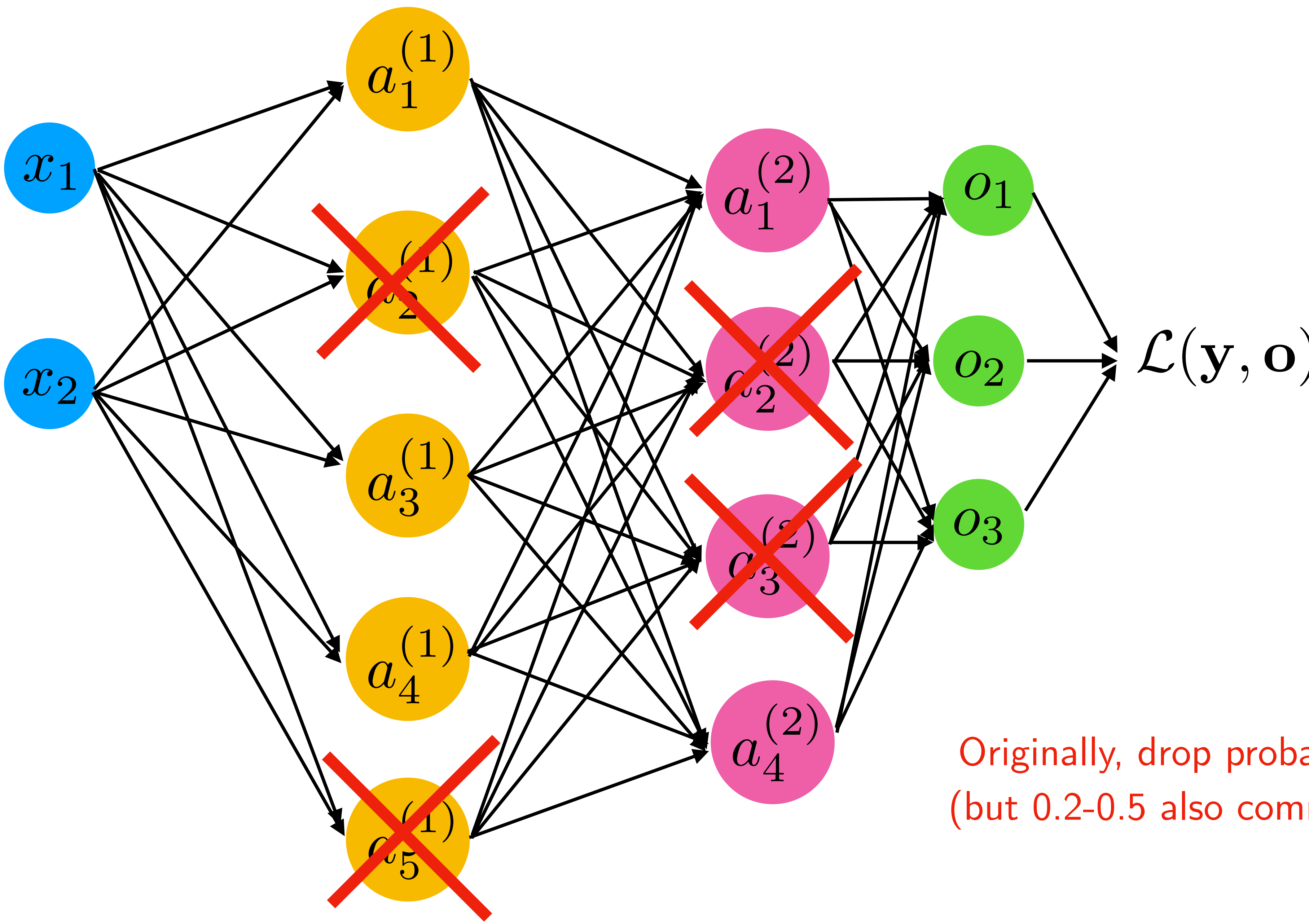
Dropout

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 0.2-0.5 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]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i > p \text{ else } 0$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$

Then, after training to make predictions (DL jargon: "inference")

$\mathbf{a} := \mathbf{a}/(1-p)$

Dropout in a Nutshell: Dropping Nodes

How do we drop the nodes practically/efficiently?

Bernoulli Sampling (during training):

- p := drop probability
- **v** := random sample from uniform distribution in range [0, 1]
- $\forall i \in \mathbf{v} : v_i := 0 \text{ if } v_i > p \text{ else } 0$
- $\mathbf{a} := \mathbf{a} \odot \mathbf{v}$

Then, after training to make predictions (DL jargon: "inference")

$\mathbf{a} := \mathbf{a}/(1-p)$ Q for you: Why is this required?

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: Ensemble Method Interpretation

Model Averaging (Ensembling)

If you are interested in more details, see FS 2018 ML class (L07):
https://github.com/rasbt/stat479-machine-learning-fs18/blob/master/07_ensembles/07_ensembles_notes.pdf

Dropout: Ensemble Method Interpretation

- In DL, we typically don't do regular ensembling (majority vote over a large number of networks, bagging, etc.) because it is very expensive to fit neural nets
- However, we know that the squared error for a prediction by a randomly selected model is larger than the squared error using an ensemble prediction (here, average over class probabilities)

$$E[(y - \hat{y}^{\{i\}})^2] = (y - E[\hat{y}^{\{i\}}])^2 + (\hat{y}^{\{i\}} - E[\hat{y}^{\{i\}}])^2$$

(expectation is over models i)

If you are interested in more details and where this comes from, see FS 2018 ML class (L08):

https://github.com/rasbt/stat479-machine-learning-fs18/blob/master/08_eval-intro/08_eval-intro_notes.pdf

Dropout: Ensemble Method Interpretation

- Now, 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:

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

(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/(1-p) approximates the geometric mean and is much cheaper
(actually, it's exactly the geometric mean if we have a linear model)

Inverted Dropout

- Most Frameworks implement inverted dropout
- Here, the activation values are scaled by the factor 1/(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

Here, is 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, probas = 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))
```

Dropout in PyTorch ([more] Object-Oriented API)

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

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Dropout(drop_proba),
            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
```

Dropout in PyTorch (Functional API)

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

    def __init__(self, num_features, num_classes, drop_proba,
                 num_hidden_1, num_hidden_2):
        super(MultilayerPerceptron, self).__init__()

        self.drop_proba = drop_proba
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)

        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)

        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 = F.dropout(out, p=self.drop_proba, training=self.training)
        out = self.linear_2(out)
        out = F.relu(out)
        out = F.dropout(out, p=self.drop_proba, training=self.training)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

Dropout in PyTorch (Functional API)

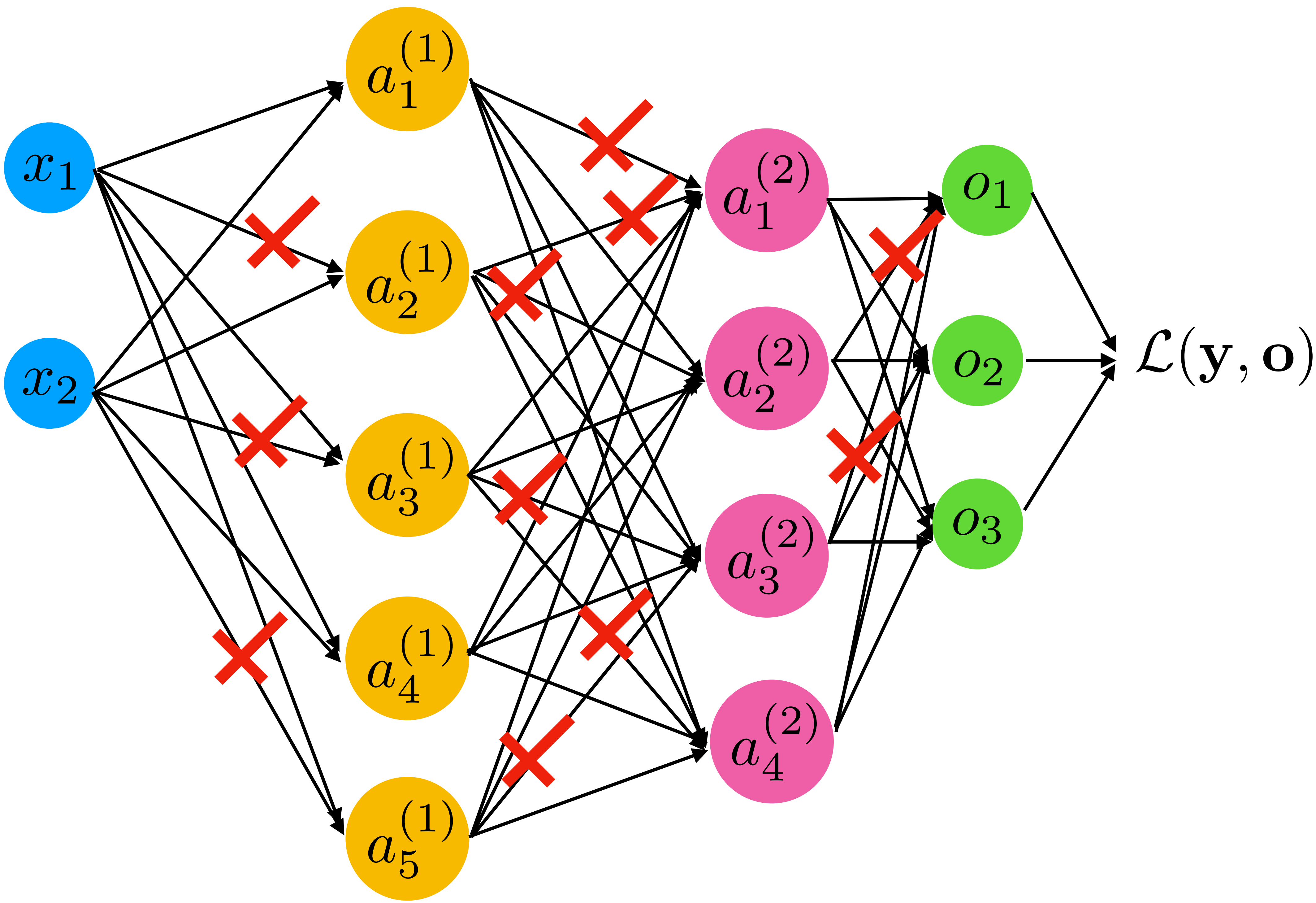
Example implementation of the 3 previous slides:

https://github.com/rasbt/stat479-deep-learning-ss19/tree/master/L10_regularization/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).

Reading Assignments (today optional)

- 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