STAT 453: Introduction to Deep Learning and Generative Models

Ben Lengerich

Lecture 10: Regularization

October 6, 2025

Logistics

- HW3 out
  - **Due next Friday (October 17) night**
- Projects
  - **Due next Friday (October 17) night**
  - Discussion board on Canvas to help you find teammates
- Midterm Exam
  - In-class **Wednesday, October 22**

Project

- **Proposal** (5%)
- **Midway Report** (5%)
- **Presentation** (5%)
- **Report** (15%)
- **Collaboration:** Teams of up to four students are allowed.
- **Honors Optional Component:** Individual extension to your project. Email me!
- **Details / formatting:**
  - https://adaptinfer.org/dgm-fall-2025/project/

Project Proposal

- **Due:** Friday, October 17, 2025, at 11:59 PM via Canvas.
  - Only one submission needed per group.
- **Content**:
  - Project title and team member list.
  - Problem statement and motivation (½ page).
  - Literature review of at least four relevant papers (~1 page).
  - Description of dataset(s) and planned activities.
- **Expected length:** ~2 pages, make it easy for us to read!
  - Use the ICML Style template.
- **Grading**:
  - 40%: Clear and concise description of the project.
  - 40%: Quality of literature survey.
  - 10%: Feasibility and detail of activity plan.
  - 10%: Writing quality.

Some course projects from prior years

- Spring 2023
  - Breast Cancer detection using ultrasound imaging
  - LSTM music generation
  - Predicting stock prices using LSTMs
  - Creating Surrealism Artworks with DCGAN
  - Exploring the Impact of Activation Functions and Normalization Techniques
- Spring 2024
  - Gradio framework with self-supervised learning
  - Song generation
  - Diagnosis of Chest X-ray Images
  - Audio-to-video image animation
  - Movie recommendations
  - Sentiment analysis using BERT

More project ideas for you:

- Computer Vision
  - Super-Resolution with Autoencoders: reconstruct high-res images from low-res inputs using convolutional autoencoders.
  - Diffusion Models for Handwritten Digits: implement a simple diffusion model to generate MNIST digits step-by-step
  - Skin Lesion Classification using CNNs: detect melanoma vs. benign moles using public dermoscopy datasets (e.g., ISIC)
- Language and Sequential Data
  - LSTM-based Weather Forecasting: predict daily temperature sequences from historical data.
  - Emotion Recognition in Tweets: classify emotional tone using BERT or DistilBERT.
  - Music Generation with Transformers: Extend LSTM-based music generation to Transformer-based models.

More project ideas for you:

- Generative AI
  - Style Transfer for Artwork: Transfer Van Gogh's style to photos using a convolutional neural style transfer model.
  - GAN-based Face Aging: Train a conditional GAN to transform faces to older or younger versions
  - Latent Space Arithmetic with DCGAN: show how semantic directions (smile, pose, etc.) can be captured in a GAN's latent space
- Multimodal / Applied AI
  - Image Captioning Model: combine a CNN encoder with an RNN or Transformer decoder to caption images.
  - Radiology Report Generation: match X-ray images with their diagnostic text.

More project ideas for you:

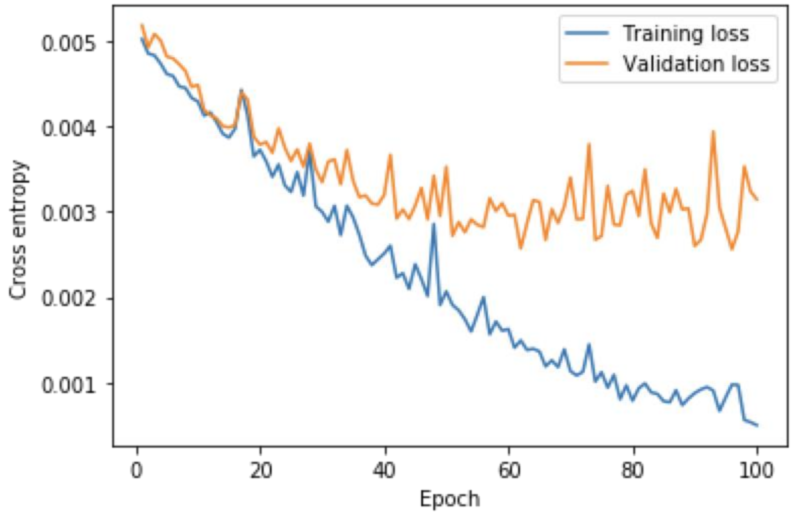
- Explainability and Fairness
  - Visualize Attention in Transformers
  - Explaining Image Classifiers: Use Grad-CAM or integrated gradients to interpret CNN prediction. Look at adversarial examples.
  - Bias Detection in Text Models: measure and visualize gender or racial bias in pretrained embeddings.
  - Calibration and Confidence in Deep Models: evaluate whether model probabilities reflect true accuracy. Does learning to abstain from prediction fix calibration?

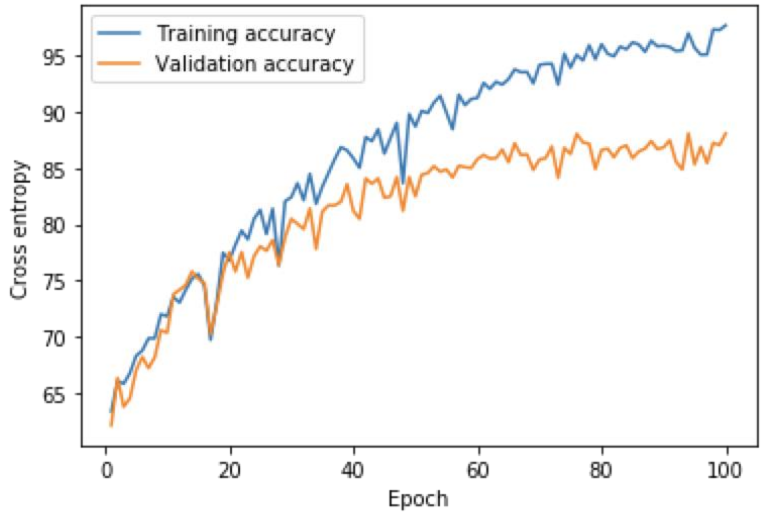
Questions about logistics?

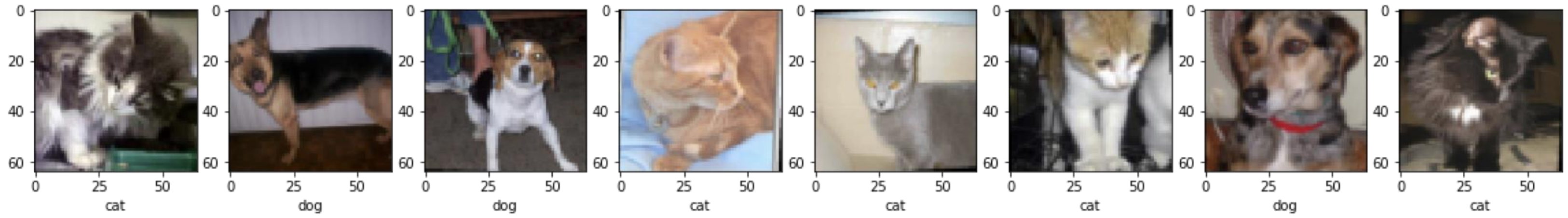
Last Time: Multilayer Perceptrons & Backpropagation

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

VGG16 CNN for Kaggle's Cats & Dogs Images





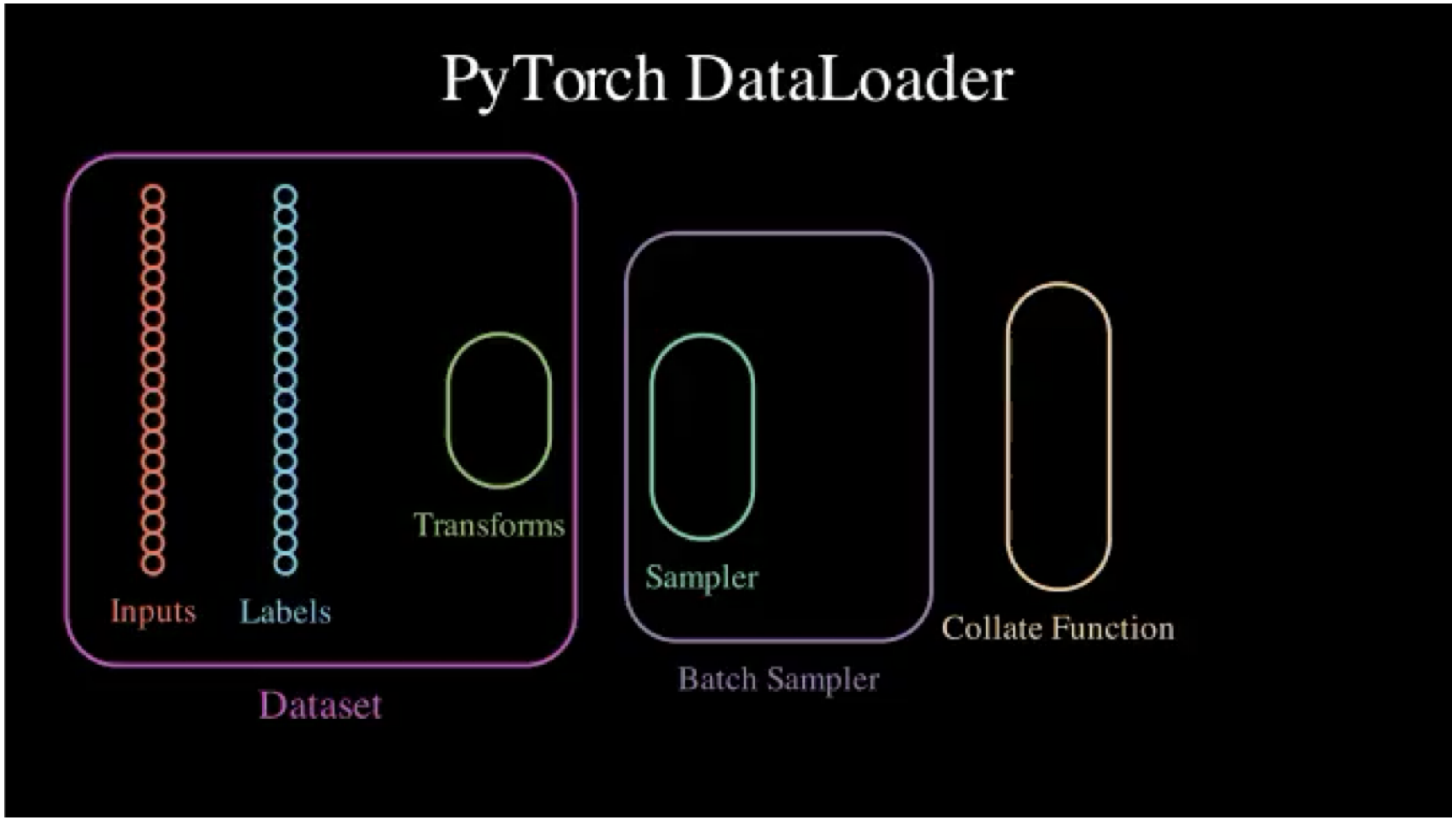



```
model.eval()
with torch.set_grad_enabled(False): # save memory during inference
    test_acc, test_loss = compute_accuracy_and_loss(model, test_loader, DEVICE)
    print(f'Test accuracy: {test_acc:.2f}%')
```

```
Test accuracy: 88.28%
```

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

Loading Data



https://x.com/_ScottCondron/status/1363494433715552259

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

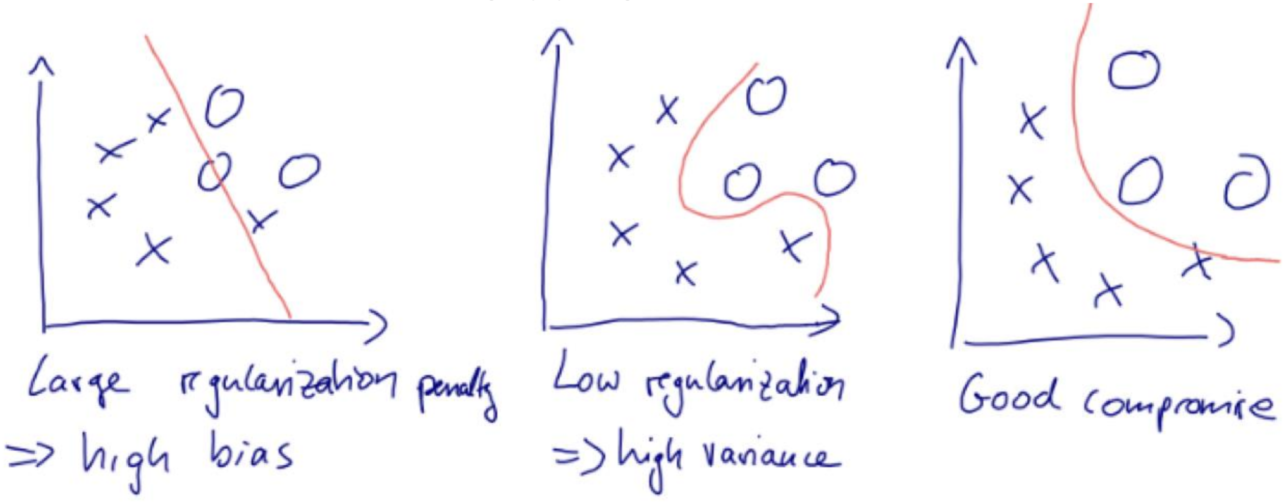
mnist_test
mnist_train
mnist_valid
custom-dataloader-example.ipynb
mnist_test.csv
mnist_train.csv
mnist_valid.csv

```python
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]
```

Where we are…

- Good news: We can solve non-linear problems!
- Bad news: Our multilayer neural networks have lots of parameters and it's easy to overfit the data…

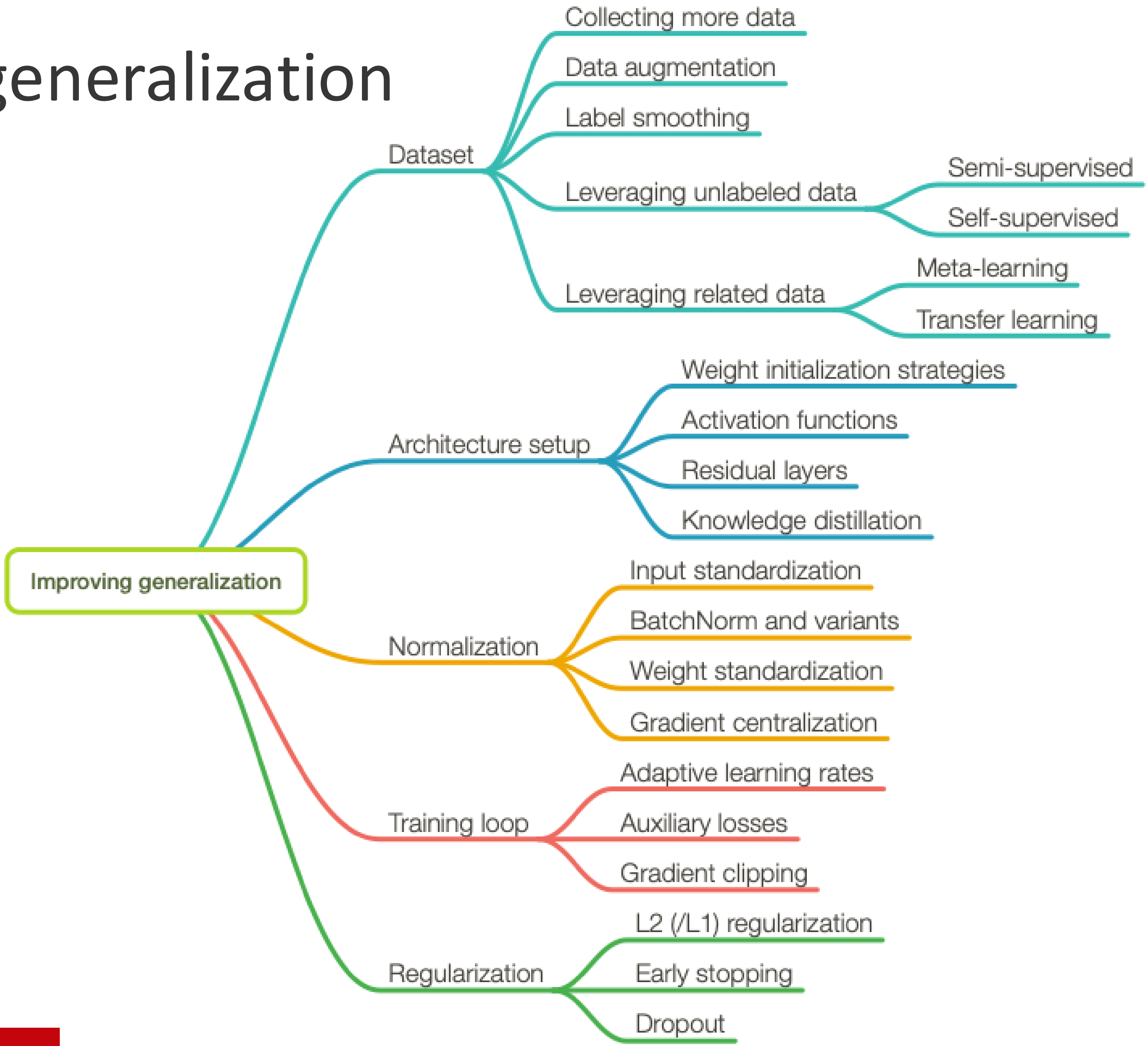
Next time:



Today: Regularization

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

Many ways to improve generalization



Today: Regularization

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

General Strategies to Avoid Overfitting

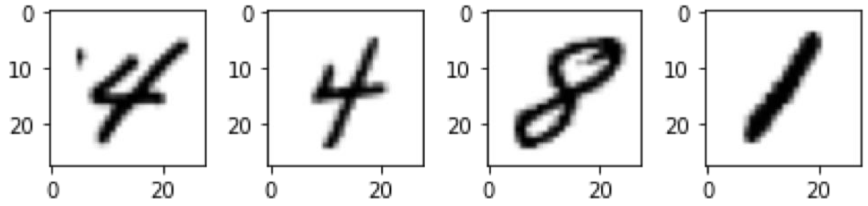
- Collecting more data, especially high-quality data, is best & always recommended
  - Alternatively: semi-supervised learning, transfer learning, and self-supervised learning
- Data augmentation is helpful
  - Usually requires prior knowledge about data or tasks
- Reducing model capacity can help

Data Augmentation

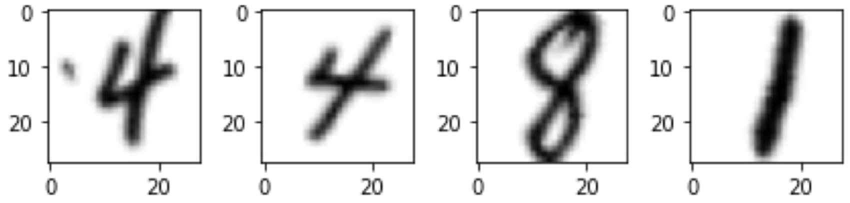
- **Key Idea:** If we know the label shouldn't depend on a transformation h(x), then we can generate new training data $h(x^i), y^i$
- But we must already know something that our outcome doesn't depend on
- Example: image classification
  - rotation, zooming, sepia filter, etc.

Data Augmentation in PyTorch via TorchVision

Original



Randomly Augmented



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

```python
# 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

Today: Regularization

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

Reduce Network's Capacity

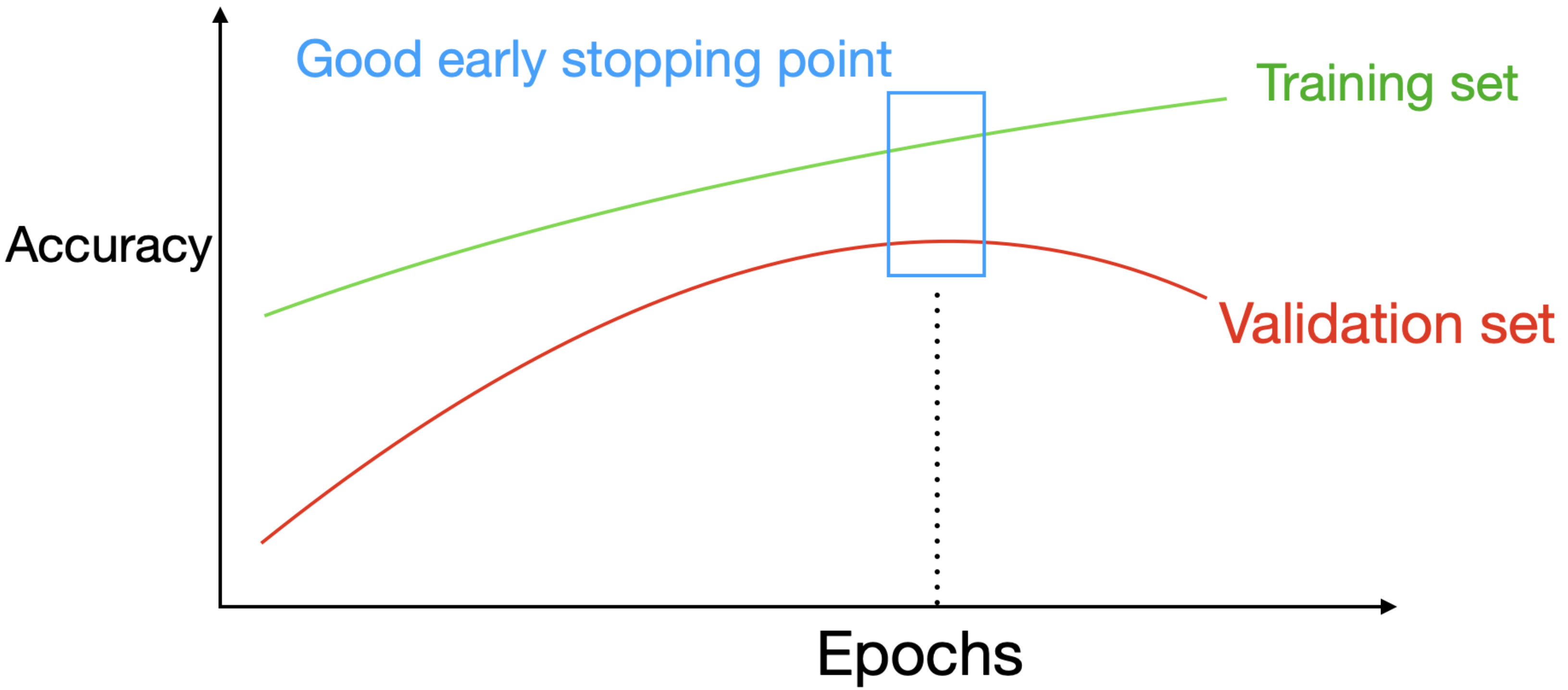
- **Key Idea:** The simplest model that matches the outputs should generalize the best
- Choose a smaller architecture: fewer hidden layers & units, add dropout, use ReLU + L1 penalty to prune dead activations,e tc.
- Enforce smaller weights: Early stopping, L2 norm penalty
- Add noise: Dropout
- Note: With recent LLMs and foundation models, it's possible to use a large pretrained model and perform efficient **fine-tuning** (updating small number of parameters of a large model)

Early Stopping

- **Step 1:** Split your dataset into 3 parts (as always)
  - Use test set only once at the end
  - Use validation accuracy for tuning

Early Stopping

- **Step 2:** Stop training early
  - Reduce overfitting by observing the training/validation accuracy gap during training and then stop at the "right" point



Today: Regularization

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

Recall from prior discussions…

- L1-regularization → LASSO regression
- L2-regularization → Ridge regression

L2 Regularization for Linear Models

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

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

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

and λ is a hyperparameter

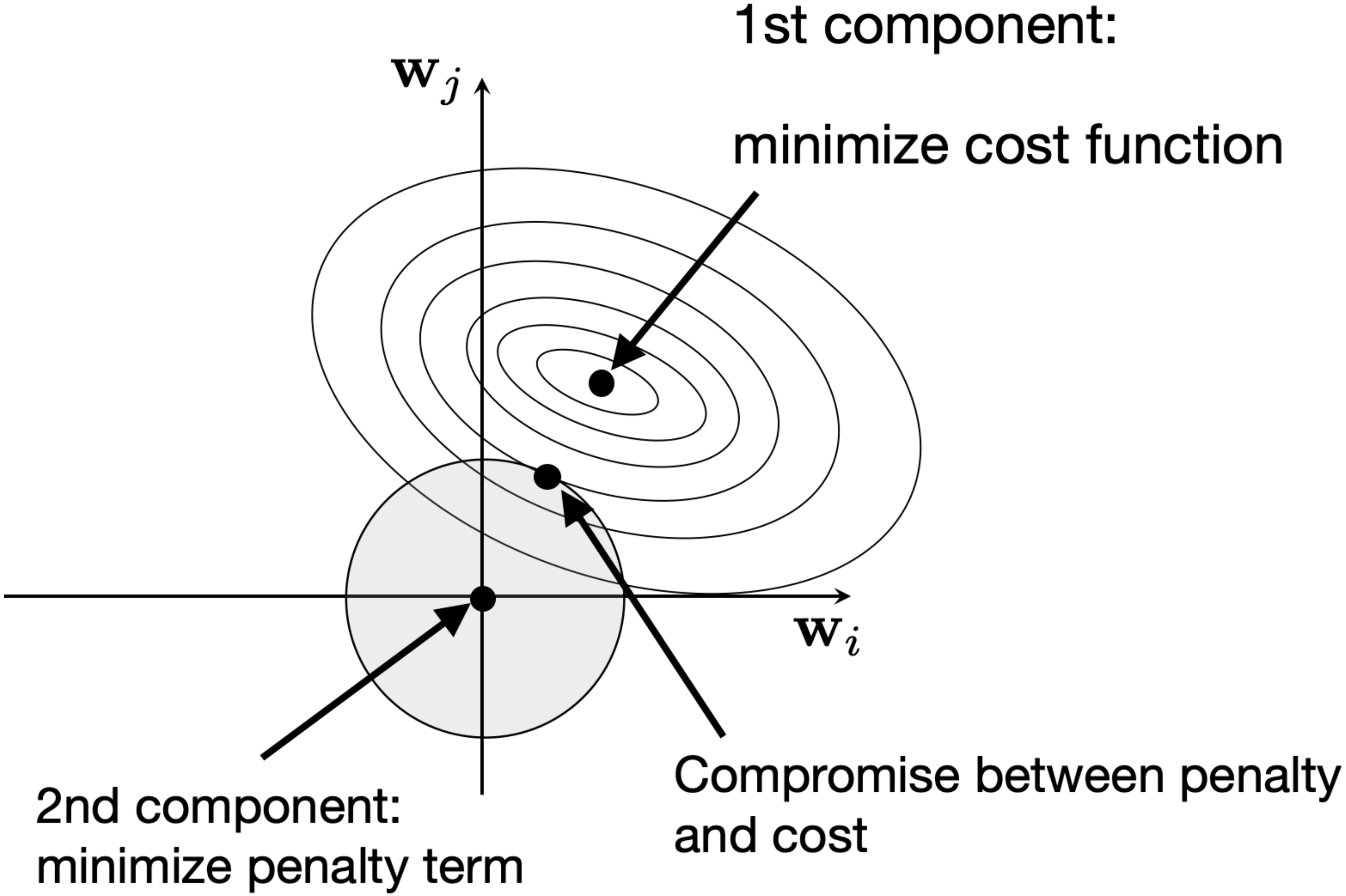
L1 Regularization for Linear Models

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

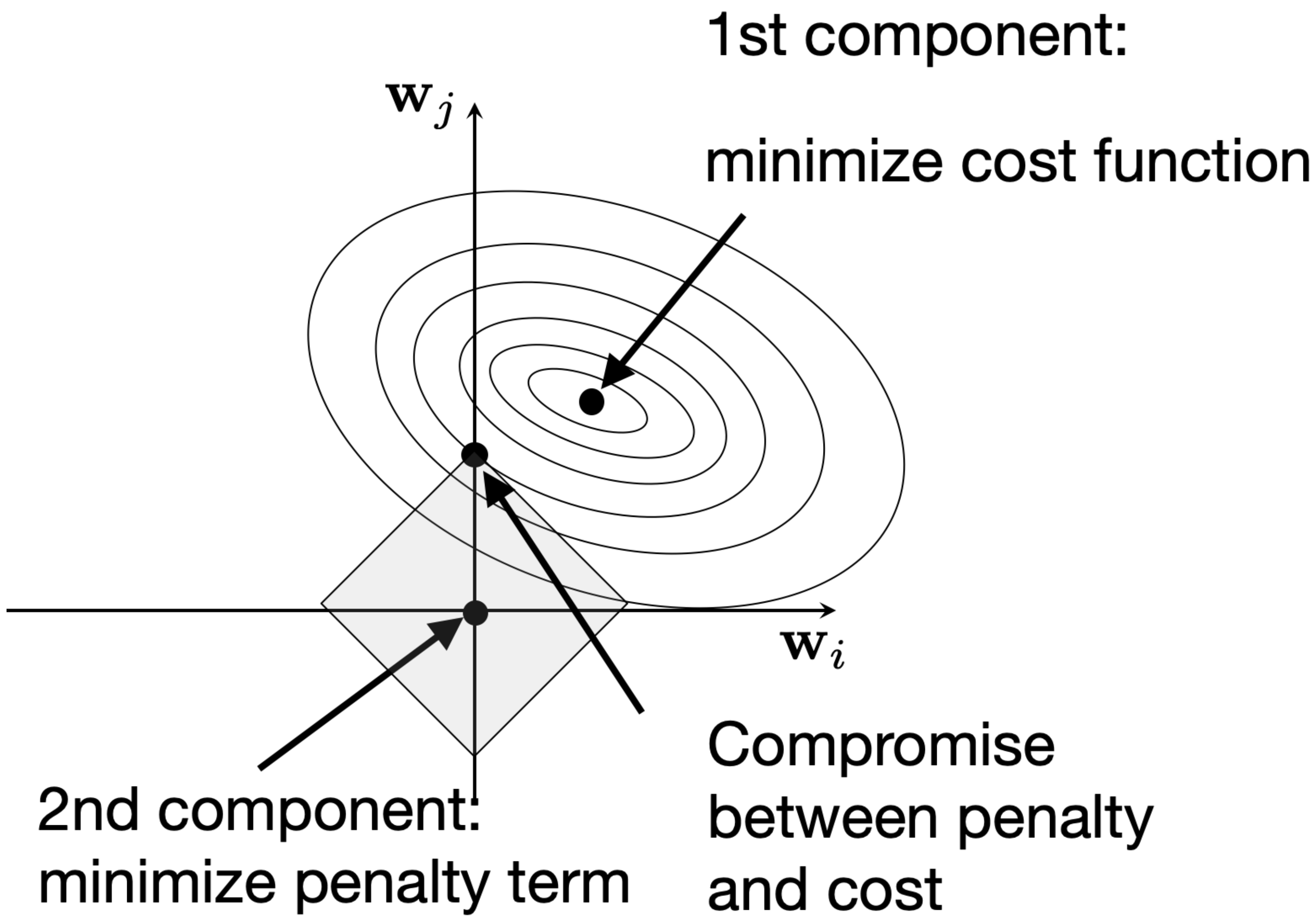
where: $\sum_{j}|w_j| = ||\mathbf{w}||_1$

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

Geometric Interpretation of L2 regularization

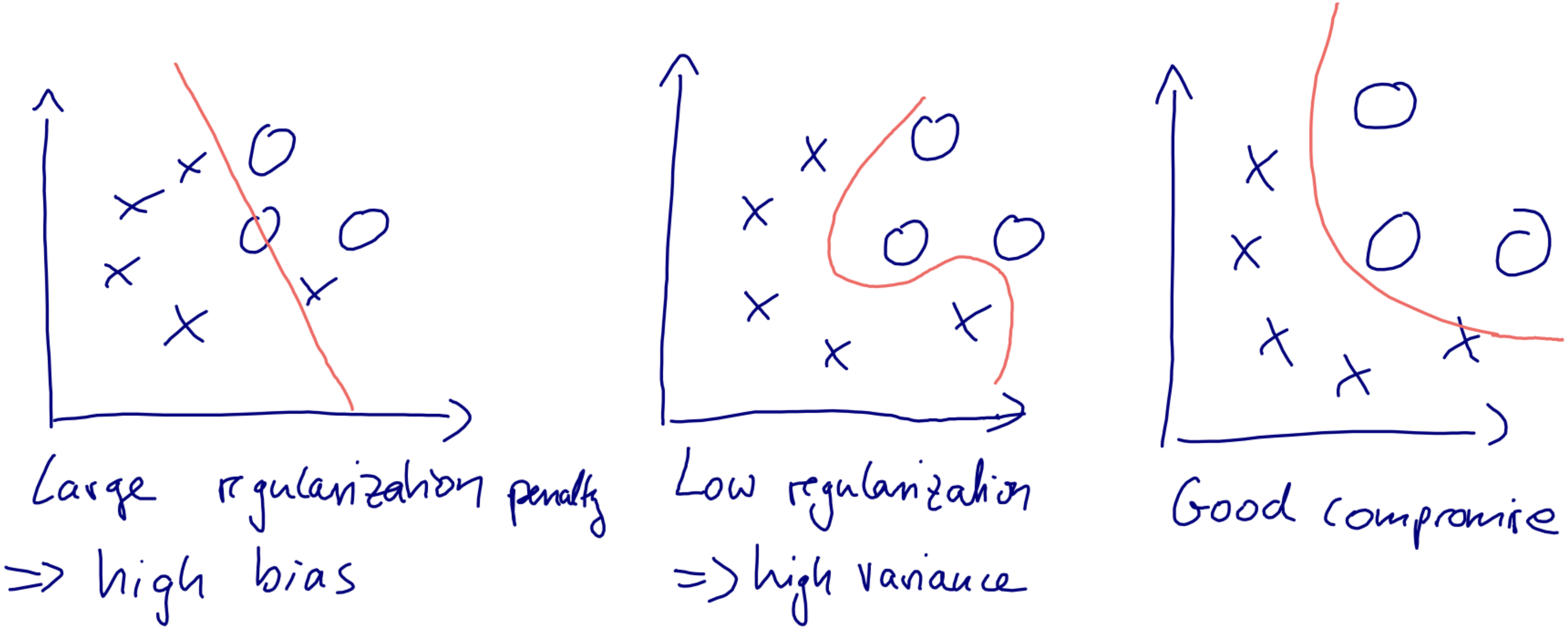


Geometric Interpretation of L1 regularization



Effect of Regularization on Decision Boundary

Assume a nonlinear model



L2 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):

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

L2 regularization for Multilayer Neural Networks

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:

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

L2 regularization for Neural Networks in PyTorch

Manually:

```
# 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()
```

L2 regularization for Neural Networks 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()
```

Today: Regularization

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**

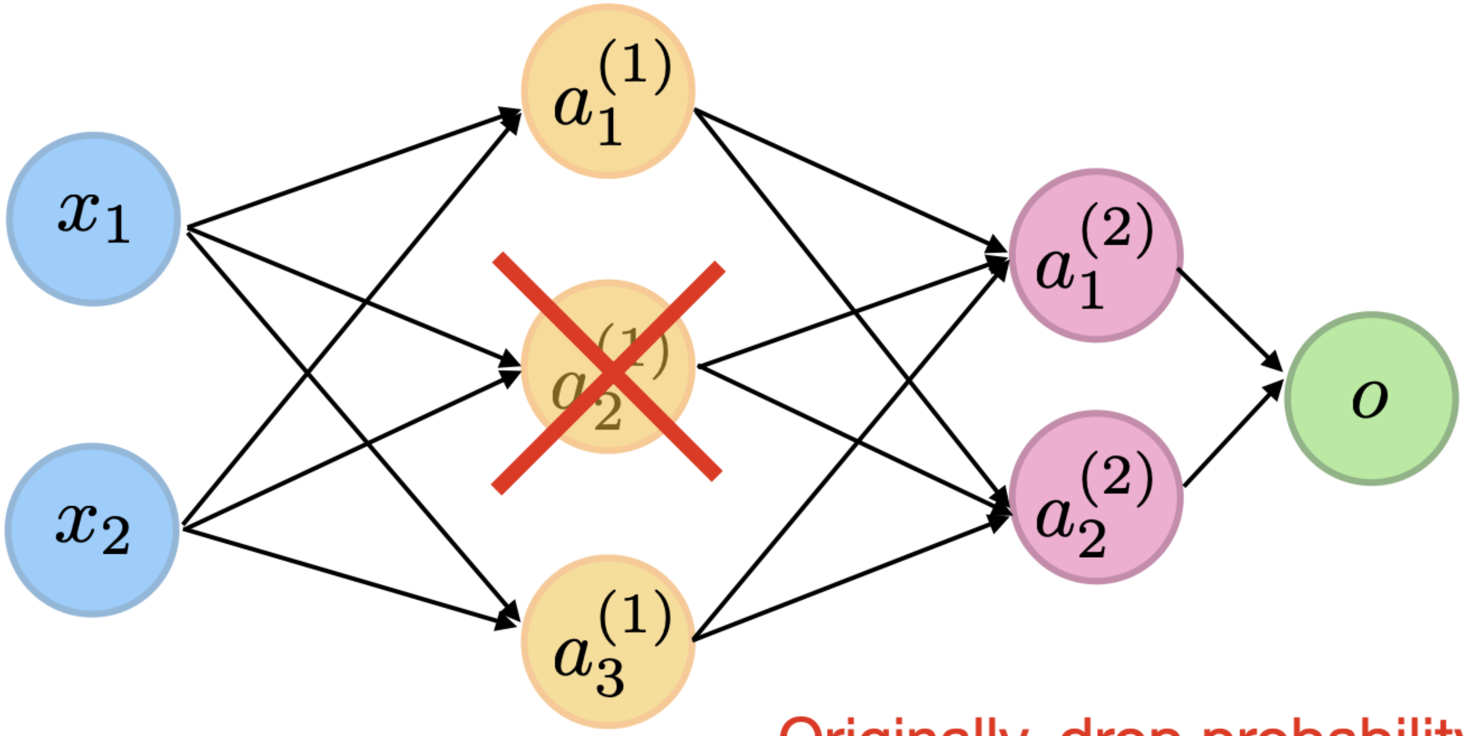
Dropout

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



Originally, drop probability 0.5

(but 0.2-0.8 also common now)

Dropout

- How do we drop node activations practically / efficiently?

<u>Bernoulli Sampling (during training):</u>

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

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

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

Dropout in PyTorch

```python
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
```

Why does Dropout work?

- **Co-Adaptation Interpretation**
  - 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)

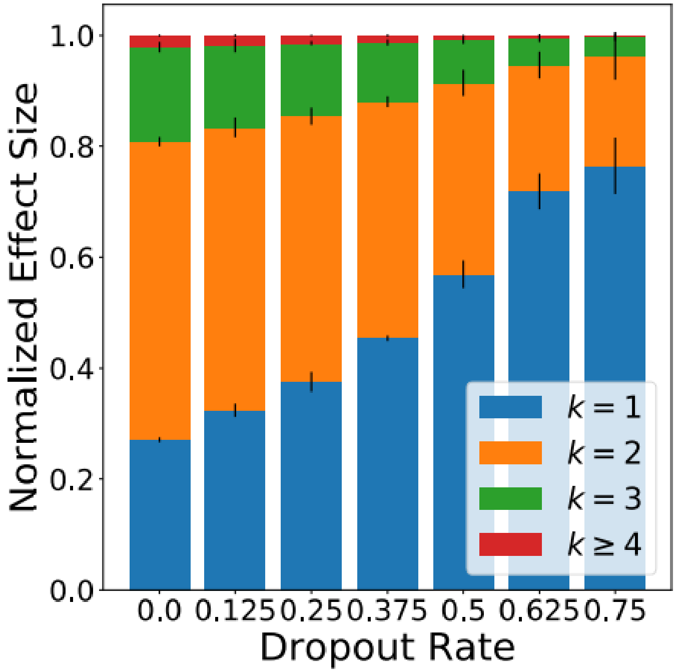
Why does Dropout work?

- **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)

If you are interested in more details, see FS 2019 ML class (L07):
https://github.com/rasbt/stat479-machine-learning-fs19/blob/master/07_ensembles/07-ensembles__notes.pdf

Why does Dropout work?

- **Interaction Effect Interpretation**
  - For p input variables there are $\binom{p}{k}$ selections of order-k interactions.
    - Grows as p^k for the first few orders.
  - The probability that an order-k interaction survives Dropout at rate r is $(1-r)^k$.
    - Decays exponentially with k.
  - These exponential rates cancel out.
  - This anti-interaction effect regularization happens at every layer.



Lengerich et al. *Dropout as a Regularizer of Interaction Effects.* AISTATS 2022

Dropout in PyTorch

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

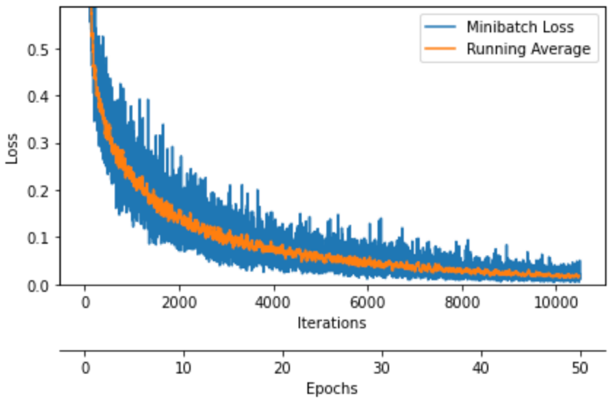
```
for epoch in range(NUM_EPOCHS):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

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

        ### FORWARD AND BACK PROP
        logits = 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))
```

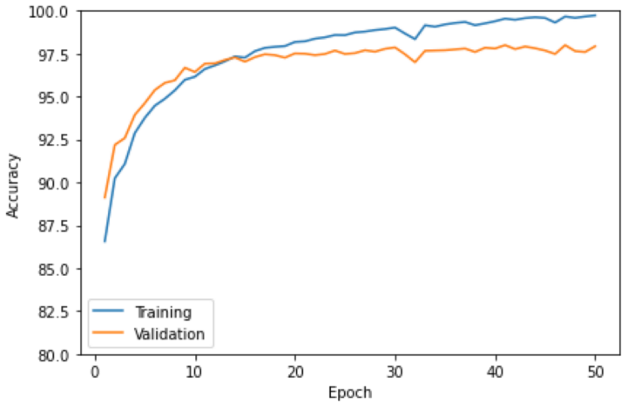
Dropout in PyTorch: Inverted Dropout

- Most frameworks (incl. PyTorch) actually implement **inverted** dropout
  - Here, the activation values are scaled by the factor 1/(1-p) during training instead of scaling the activations during "inference”
  - Helpful for models that will be used many times in “test” time

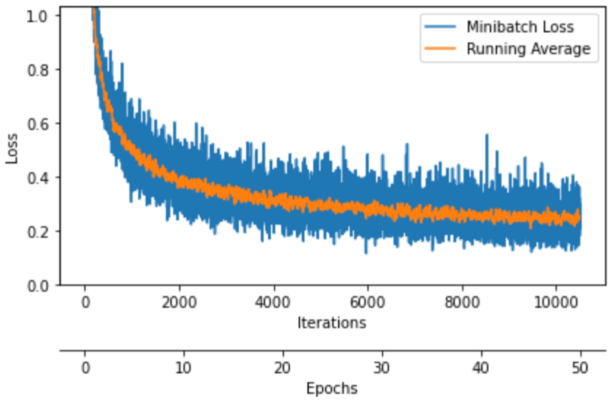
Dropout in PyTorch

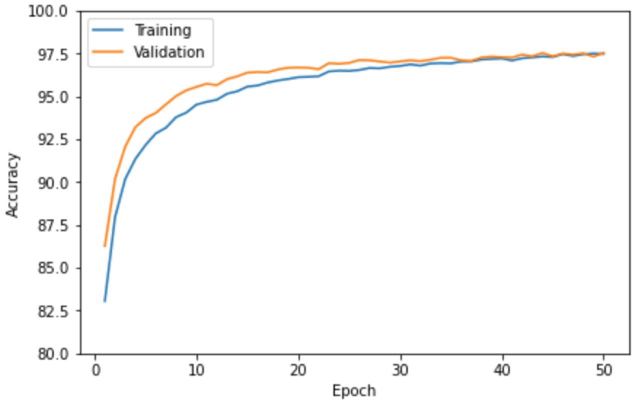
Without dropout:





With 50% dropout:





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

Questions?