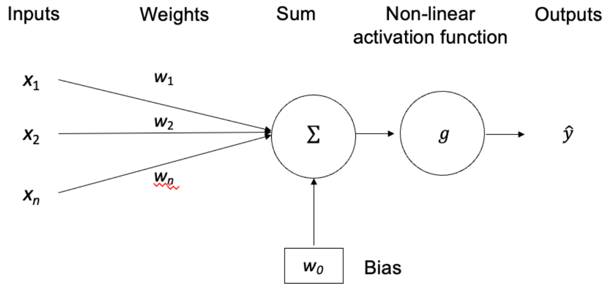
STATS 507
Data Analysis in Python

Week12-2: Stochastic Gradient Descent, DataLoader, Sequential Modeling

Dr. Xian Zhang

Recap: Core Foundation Review

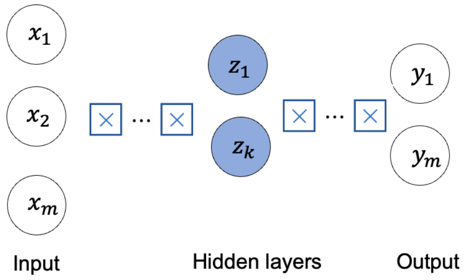
The perceptron



$$\hat{y} = g(w_0 + \sum_{i=1}^{n} x_i w_i) = g(z)$$

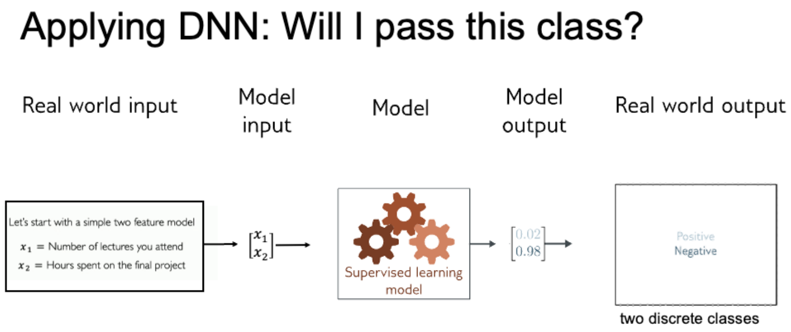
- Structural building blocks
- Numerical operator
- Nonlinear activation functions

Neural Networks



- Stacking Perceptions to form neural networks (MLP)
- Optimization through backpropagation

Training in Practice



- Adaptive learning rate
- Batching
- Regularization

Recap our first DNN: Will I pass this class?

Real world input	Model input	Model	Model output	Real world output

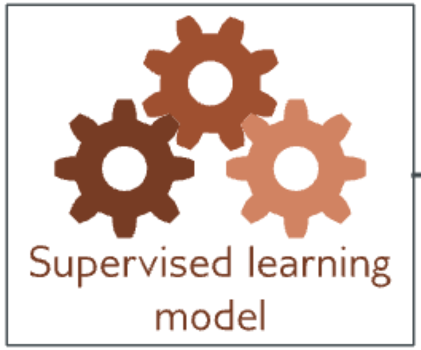
A two layer MLP

Let's start with a simple two feature model

x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

→ $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ →

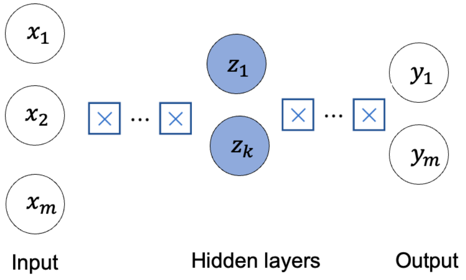


→ $\begin{bmatrix} 0.02 \\ 0.98 \end{bmatrix}$ →

Positive

Negative

two discrete classes



Solutions for a simple DNN

Step1: Define the model

A two layer MLP with linear layer

```
# Define a custom neural network class
class MultilayerPerceptron(nn.Module):
    def __init__(self, num_features, hidden_size1, hidden_size2, num_classes):
        super(MultilayerPerceptron, self).__init__()

        # 1st hidden layer
        self.linear_1 = nn.Linear(num_features, hidden_size1)
        # 2nd linear layer
        self.linear_2 = nn.Linear(hidden_size1, hidden_size2)x
        # output layer
        self.linear_out = nn.Linear(hidden_size2, num_classes)

    def forward(self, x):
        x = F.relu(self.linear_1(x))
        x = F.relu(self.linear_2(x))
        logits = self.linear_out(x)
        probas = torch.sigmoid(logits)
        return logits, probas
```

Define model parameters that will be instantiated when created an object of this class

Define how and in what order the model parameters should be used in the forward pass

Step 2: Creation

Create a model, define loss function and an optimization method

```
# Create the neural network model
model = MultilayerPerceptron(num_features, hidden_size1, hidden_size2, num_classes)


# Define loss function (binary cross-entropy) and optimizer (SGD)
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Instantiate model
(creates the model parameters)

Define loss function
Define an optimization method

Step 3: Training

A two layer MLP with linear layer

```
# Training loop
for epoch in range(epochs):
    # Forward pass
    logits, outputs = model(X)

    # Compute loss
    loss = criterion(outputs, y)

    # Backpropagation and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
```

Run for a specified number of epochs

This will run the `forward()` method

Set the gradient to zero
(could be non-zero from a previous forward pass)

Compute the gradients, the backward is automatically constructed by "autograd" based on the forward() method and the loss function

Use the gradients to update the weights according to the optimization method (defined on the previous slide) E.g., for SGD, `w := w + learning_rate × gradient`

Using sequential to stack layers

```python
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
```

```python
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
```

Much more compact and clear, but "forward" may be harder to debug if there are errors (we cannot simply add breakpoints or insert "print" statements

1. Stochastic with mini-batches

2. Regulation

Recall: training with gradient descent

Essentially, finding parameters that minimize the loss:

$$J(\boldsymbol{W}) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f\left(x^i; \boldsymbol{W}\right), y^i)$$

Also known as:

- objective function
- cost function
- empirical risk

Also formatted as:

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f\left(x^i; \boldsymbol{W}\right), y^i)$$

Using gradient descent to updating weights by:

- Computer gradient: $\boxed{\frac{\partial L}{\partial \boldsymbol{W}}}$
- Update weights: $\boldsymbol{W} \longleftarrow \boldsymbol{W} - \alpha \frac{\partial L}{\partial \boldsymbol{W}}$

Gradient descent

Full Batch

- Batch size = N
- 1 update per epoch
- Use all the data

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial L_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

- Very slow updates
- Most stable
- Can stuck in local minima

Mini-batch GD

- Batch size = bs
- N/32 updates per epoch
- Use sub dataset

$$\frac{\partial L}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{i=1}^{B}\frac{\partial L_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

- More accurate estimation of gradient
- Smoother convergence
- Allow for larger learning rates

Stochastic gradient descent

- Batch size = 1
- N updates per epoch
- Use single sample

$$\frac{\partial L_i}{\partial \boldsymbol{W}}$$

- Most efficient
- Very noisy updates
- Unstable

Loading and Batching Data in PyTorch

```
from torch.utils.data import TensorDataset, DataLoader

np.random.seed(0)
X = np.random.rand(100, 2)
y = (X[:, 0] + X[:, 1] > 1).astype(int)

# Convert data to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).unsqueeze(1)

dataset = TensorDataset(X_tensor, y_tensor)
batch_size = 32
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

- Load data in batches
- Handle data shuffling
- Provides iteration over the dataset

Common usage:

```
for batch_idx, (data, labels) in enumerate(dataloader):
    # data: tensor of shape (batch_size, *features)
    # labels: tensor of shape (batch_size)
    # Your training code here
```

Reference: https://pytorch.org/docs/stable/data.html

Loading and Batching Data in PyTorch

```python
print("Check for attributes of dataloader")
for attr, value in dataloader.__dict__.items():
    print(f"{attr}: {value}")
```

```
Check for attributes of dataloader
dataset: <torch.utils.data.dataset.TensorDataset object at 0x174fb5580>
num_workers: 0
prefetch_factor: None
pin_memory: False
pin_memory_device:
timeout: 0
worker_init_fn: None
_DataLoader__multiprocessing_context: None
_dataset_kind: 0
batch_size: 32
drop_last: False
sampler: <torch.utils.data.sampler.RandomSampler object at 0x16ab5e720>
batch_sampler: <torch.utils.data.sampler.BatchSampler object at 0x175723020>
generator: None
collate_fn: <function default_collate at 0x163613b00>
persistent_workers: False
_DataLoader__initialized: True
_IterableDataset_len_called: None
_iterator: None
```

Reference: https://pytorch.org/docs/stable/data.html

In-class practice

Use Mini-batch gradient descent for our simple DNN

In-class practice

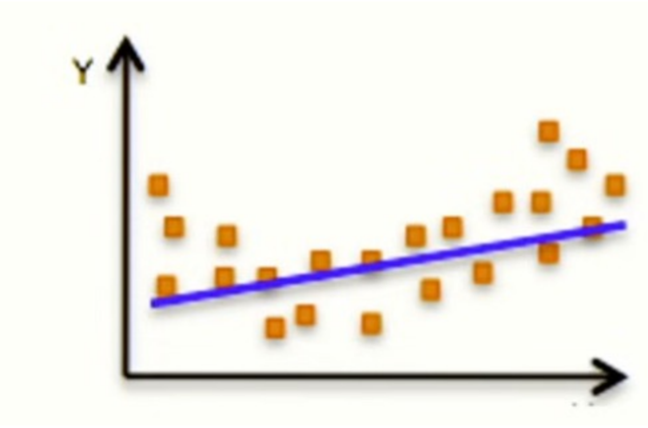
Find the mean brightness of CIFAR-10 images

1. Stochastic with mini-batches

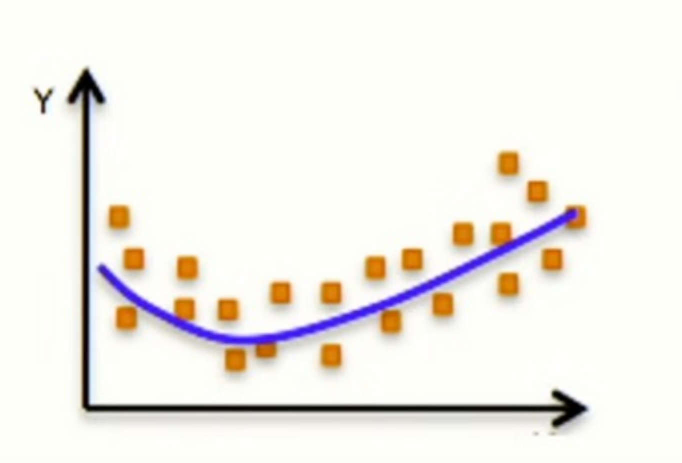
2. Regulation

The problem of overfitting

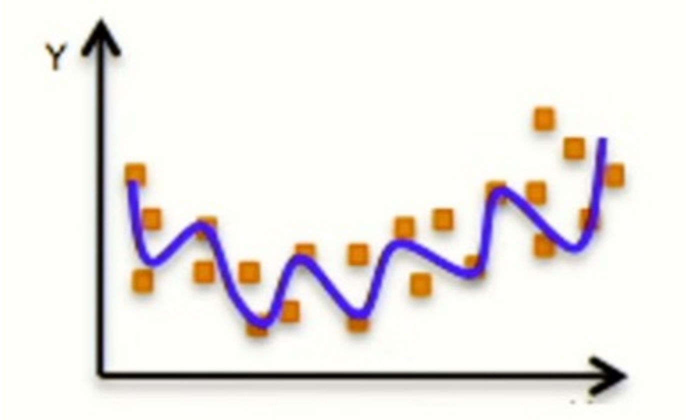
Model is doing really well on training data, but very badly on test data



Underfitting



Ideal fit



Overfitting

Model does not have the capacity to fully learn the data

Too complex, extra parameters, does not generalize well

Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

To reduce overfitting…

Regularization: technique that constrains our optimization problem to discourage **complex** models

To improve generalization of our model on **unseen** data

Dropout

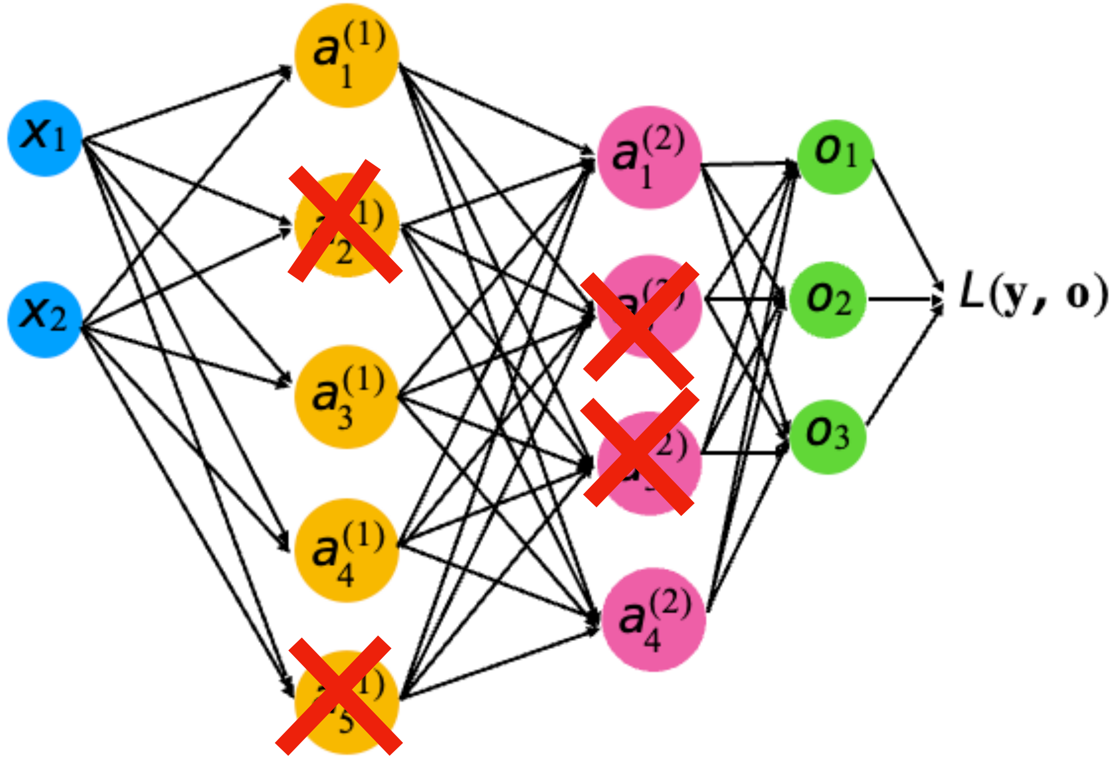
Early stopping

Collecting more data

Regulation I: dropout

During training, **randomly** set some activations to 0

- Originally, drop probability 0.5 (but 0.2 -0.5 also common now)
- Force network not to rely on very node.



Why Dropout Work?

- 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

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(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
```

Regulation 2: early stopping

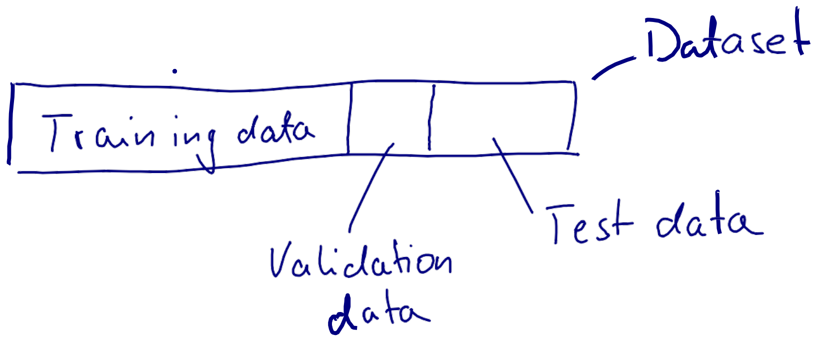
Slide Credit: Alexander Amini

Modified from MIT open course: 6.S191 and Nvidia blog

Early stopping in practice

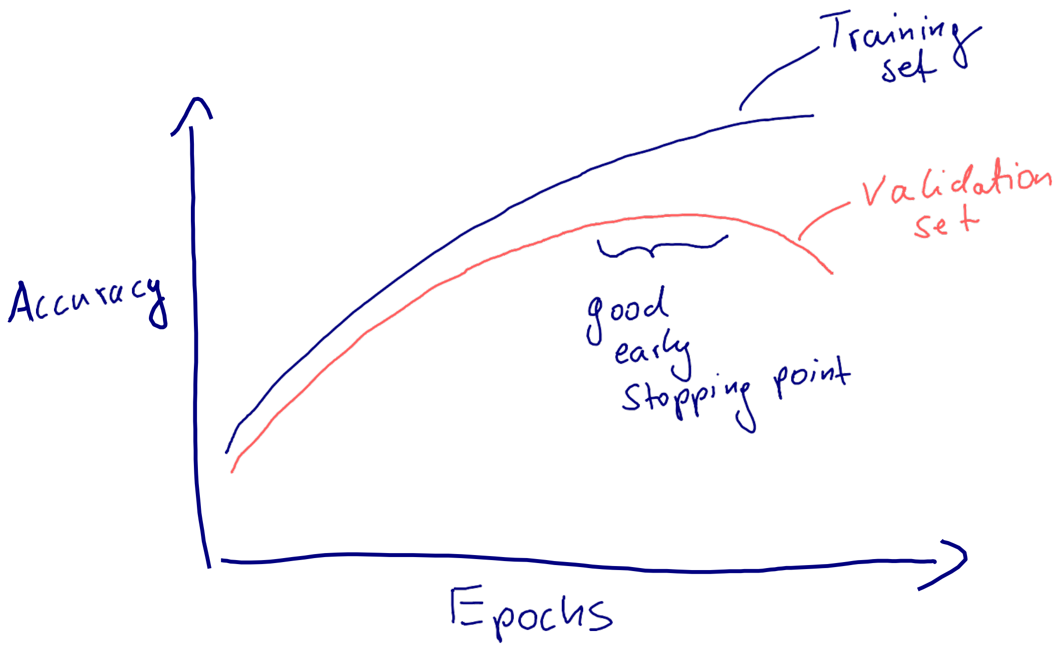
Step 1: Split your dataset into 3 parts

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



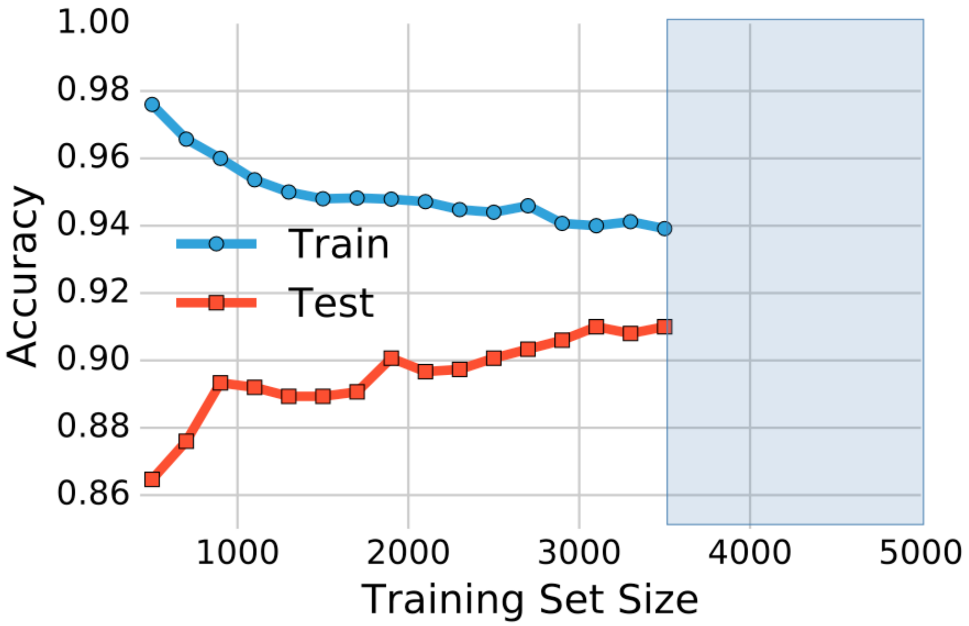
Step 2: Early stopping

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



Regulation 3: Add more data

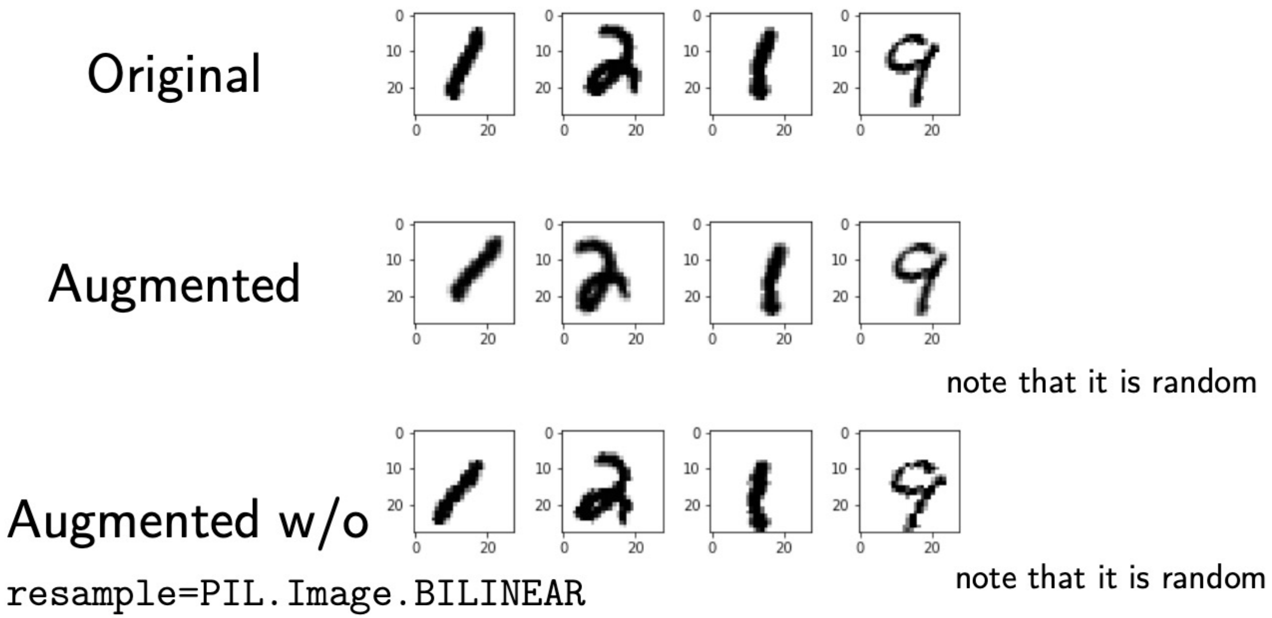
Best way to reduce overfitting is collecting more data.



Softmax on MNIST subset (kept test set size constant)

Regulation 3: Add more data

- Collecting more data is always helpful
- If not possible, **data augmentation** can be helpful (e.g., for images: random rotation, crop, translation ...)



Other things

HW7 due this week.

HW8 out.

Final project guideline out (**start early**)

Coming next:

Deep Sequential model (RNN, LSTM, Transformer…)