

Deep Neural Networks (DNNs)

CSCI-P556 Applied Machine Learning

Lecture 15

D.S. Williamson

Agenda and Learning Outcomes

Today's Topics

- **Topics:**
 - DNN Training and Testing Pipeline example
- **Announcements:**
 - Use of libraries for assignments.
 - Homework 2 - “ignore words in testing set, that do not exist in training” (e.g. assume conditional probability of 1)

Digit Recognition Example

DNN Training and Testing Pipeline

- Suppose you are a Data Scientist at the post office. Your boss wants you to build a classification model that converts handwritten digits to their digital form
- **Data has been collected**, which contains several handwritten examples of each of the digits (e.g. 0 through 9)
 - MNIST dataset will be used (classic baseline classification dataset)
 - Contains 70,000 images of digits handwritten by high school students and employees of the US Census Bureau
 - Each image has a label with the digit it represents
- **How can this be done using a deep neural network?**

Collect the Data

- Scikit-Learn has helper functions to download popular datasets, which includes MNIST
- The code below gets the MNIST data

- Grabs MNIST dataset and returns a dictionary
- The dictionary has different keys
 - One for the actual images
 - One for the labels
 - And other descriptors

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1) # Returns a dictionary with the data
mnist.keys()
```

```
dict_keys(['data', 'target', 'frame', 'categories', 'feature_names', 'target_names', 'DESCR', 'details', 'url'])
```

Get the features and labels

- Get the input data and labels, using the MNIST dictionary keys
- Label is stored as a character, hence needs to be converted
- Now can see the dimensions of the data and labels

What is
happening
here?

```
import numpy as np
X, y = mnist["data"], mnist["target"]
y = y.astype(int) # Convert y to list of integers

X = ((X/255.) - .5)*2

print(X.shape, y.shape)
print(min(y), max(y))
print(type(X), type(y))
print(np.min(np.min(X)), np.max(np.max(X)))

(70000, 784) (70000,)
0 9
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
-1.0 1.0
```


Visualize the data

Plot an example from each class

- Get image from each class, then reshape the image to its proper dimensions
- Use *imshow()* to plot the image
- Note that the “images” have been flattened (vectorized) into 784-dimensional vectors. Hence, they need to be reshaped to 28 x 28 images

```
import matplotlib as mpl
import matplotlib.pyplot as plt

fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True)
ax = ax.flatten()
for i in range(10):
    matchlist = [i for i, x in enumerate(y == i) if x]
    img = X[matchlist[0]].reshape(28,28)
    ax[i].imshow(img, cmap='Greys')

ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```



Data Splitting

Divide data into training and testing sets

- MNIST has already been divided into training and testing sets.
 - The first 60000 images are for training
 - The last 10000 images are for testing

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]  
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(60000, 784) (10000, 784) (60000,) (10000,)
```

Further Splits

Divide Training data into training and development sets

- **K-fold cross validation is not often performed with DNNs.**
 - Training process is long, since they usually require large datasets
 - Performing k-fold validation would increase this time by a factor of k
- However, training data is still further divided into a training and development set
 - Helps to determine if model is overfitting to training data
 - Helps with model and hyper parameter selection (more on this later)

```
from sklearn.model_selection import StratifiedShuffleSplit
import collections

split = StratifiedShuffleSplit(n_splits=1, test_size= 0.1, random_state = 42)
for train_index, val_index in split.split(X_train,y_train):
    X_train_strat = X[train_index,:]
    y_train_strat = y[train_index]

    X_dev_strat = X[val_index,:]
    y_dev_strat = y[val_index]

print(X_train_strat.shape, y_train_strat.shape, X_dev_strat.shape, y_dev_strat.shape)
```

(54000, 784) (54000,) (6000, 784) (6000,)

```
New Train: [(0, 0.1), (1, 0.11), (2, 0.1), (3, 0.1), (4, 0.1), (5, 0.09), (6, 0.1), (7, 0.1), (8, 0.1), (9, 0.1)]
Dev:       [(0, 0.1), (1, 0.11), (2, 0.1), (3, 0.1), (4, 0.1), (5, 0.09), (6, 0.1), (7, 0.1), (8, 0.1), (9, 0.1)]
Orig Train: [(0, 0.1), (1, 0.11), (2, 0.1), (3, 0.1), (4, 0.1), (5, 0.09), (6, 0.1), (7, 0.1), (8, 0.1), (9, 0.1)]
```


Questions to consider

DNN Training

- Now that you have your data, let's train a DNN classifier.
- What do I need to consider (e.g. design decisions)?:
 - How many layers does my DNN need?
 - How many neurons are needed in each layer?
 - What type of activation function in each layer?
 - How are my network weights initialized?
 - How do I measure performance?
 - ...

Import Libraries

- We'll be using PyTorch for our DNN implementation. Relatively easy to use, but it does have its fair share of “implementation” issues (e.g. datatypes and use of several (unnecessary) functions)
 - May also use TensorFlow or Keras if you'd like. This is really a personal decision or preference

```
# Import key libraries
import torch
import torch.nn as nn
import torch.nn.functional as Func
from torch.autograd import Variable
import torch.optim as optim
import torch.utils.data as data
import random
from scipy.io import savemat
import os
from os import path
from sklearn.preprocessing import normalize
from torch.nn.utils import clip_grad_norm_
import torch.nn.parallel.data_parallel as data_parallel
from sklearn.metrics import confusion_matrix
```

Define DNN Components

- DNNs are defined and categorized by their components
- Three main components are:
 - **Architecture** (e.g. fully connected, number of layers, units, type of connections,...)
 - **Activation function**
 - **Learning rule (algorithm, optimization approach)**
- This is both a strength and drawback: there are many components to select and adjust! **How do you know what to select?**
 - **Generally**, cannot search through all possible combinations (e.g. Grid Search, too time consuming)
 - Scikit-Learn's *GridSearchCV* performs cross-validation using user-specified range of values for hyper parameters. It finds the best option in the range (see Chapter 2 of HOML).
 - May try Randomized Search cross-validation instead or use an external tool. Randomized Search evaluates a given number of random combinations, where random values are selected for hyper parameters during each iteration.
 - Need reasonable (literature based) options to start.

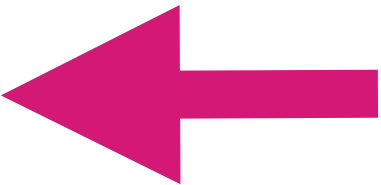
Define Hyper-parameters

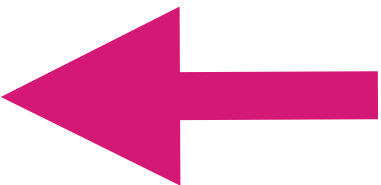
It's all Empirical. A Theory for doing this doesn't exist!

- The number of neurons (units) in the input and output layers depend on the problem
- The number of hidden layers and the number of units in these layers is a design decision
 - May start with one hidden layer, train/test performance. Then add one layer at a time until performance gets worse
 - May use the same number of units in each layer.
 - May “funnel” number of units down from high to lower values for each layer

```
# Define key hyperparameters
class hyperparam:
    num_hid1_units = 300
    num_hid2_units = 100
    num_classes    = 10
    input_dim      = 28*28

    lr              = 0.01 # Learning rate
    num_epochs     = 50  # Number of epochs
    bs             = 50  # Mini-batch size
```

- 
- Three layer DNN (two hidden layers)
 - 300 units in first hidden layer. 100 units in 2nd hidden layer

- 
- Parameters for Mini-batch Learning (e.g. backpropagation and gradient descent)

Data Loader for Mini-Batch Processing

Generally, you're working with **LARGE** datasets with DNNs

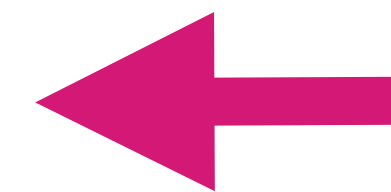
- *DataLoader* is an efficient way to perform Mini-batch learning. It effectively manages batches and getting the feature sample and label for each batch
 - This will change depending on the dataset
 - You may need to do extra processing. May also read data from file (e.g. **.npz*).

```
class myDataset(data.Dataset):
    def __init__(self, input_data, labels):
        # Initialize variables
        self.feats = input_data
        self.labels = labels

    def __len__(self):
        return len(self.labels) # returns the number of samples

    def __getitem__(self, index): # Returns sample data and label
        X = self.feats[index,:]
        y = self.labels[index]

        return X, y
```



- Need to define a class for this
- `__init__()` is called when an instance of this class is *created*
- `__getitem__()` returns the feature-label pair for the given instance

Data Loader for Mini-Batch Processing

Generally, you're working with **LARGE** datasets with DNNs

- Need to create DataLoader instances for each dataset (training, development/validation, testing)

```
params = {'batch_size': hyperparam.bs,
          'shuffle': True,
          'num_workers': 6,
          'drop_last': False,
          'pin_memory': True}

training_set = myDataset(X_train_strat, y_train_strat)
dev_set      = myDataset(X_dev_strat, y_dev_strat)
test_set     = myDataset(X_test, y_test)

training_gen = data.DataLoader(training_set, **params)
dev_gen      = data.DataLoader(dev_set, **params)
testing_gen  = data.DataLoader(test_set, **params)
```



Check documentation of DataLoader for description of all possible attributes

- 
- First use Dataset class
 - Then create DataLoader instances

Define the Neural Network

```
class DNN(nn.Module):
    def __init__(self):
        super(DNN, self).__init__()

        self.nhid1_units = hyperparam.num_hid1_units
        self.nhid2_units = hyperparam.num_hid2_units
        self.nout_units = hyperparam.num_classes
        self.in_dim = hyperparam.input_dim

        self.fc1 = nn.Linear(self.in_dim, self.nhid1_units)
        self.fc2 = nn.Linear(self.nhid1_units, self.nhid2_units)
        self.fc3 = nn.Linear(self.nhid2_units, self.nout_units)

        # Initial values for the network
        stddev1 = 2/np.sqrt(self.in_dim + self.nhid1_units)
        nn.init.normal_(self.fc1.weight, std = stddev1)
        nn.init.zeros_(self.fc1.bias)

        stddev2 = 2/np.sqrt(self.nhid1_units + self.nhid2_units)
        nn.init.normal_(self.fc2.weight, std = stddev2)
        nn.init.zeros_(self.fc2.bias)

        stddev3 = 2/np.sqrt(self.nhid2_units + self.nout_units)
        nn.init.normal_(self.fc3.weight, std = stddev3)
        nn.init.zeros_(self.fc3.bias)

    def forward(self, sig):
        #sig = sig.unsqueeze_(1)
        hid1 = self.fc1(sig)
        hid1 = Func.relu(hid1)

        hid2 = self.fc2(hid1)
        hid2 = Func.relu(hid2)
        out = self.fc3(hid2)

        return out
```

- Need to extend `Torch.nn.Module` to have all necessary functionality
- Then need to override the `__init__()` function

- The `__init__()` function is where you define the basic architecture (e.g. types of layers, units per layer, number of layers, needed functions, ...)
- You also determine weight/bias initialization here (more on this on Tuesday)

- Define the forward pass of the DNN
- Use desired activation function in each layer

Define the Neural Network

- Once the DNN architecture is defined, create an instance of it.
- Then define the optimizer (e.g. Stochastic Gradient Descent, ADAM, ADAGRAD, ...) (more on these later)
- Also define the loss function
 - We are using Cross Entropy, since we are performing multi-class classification

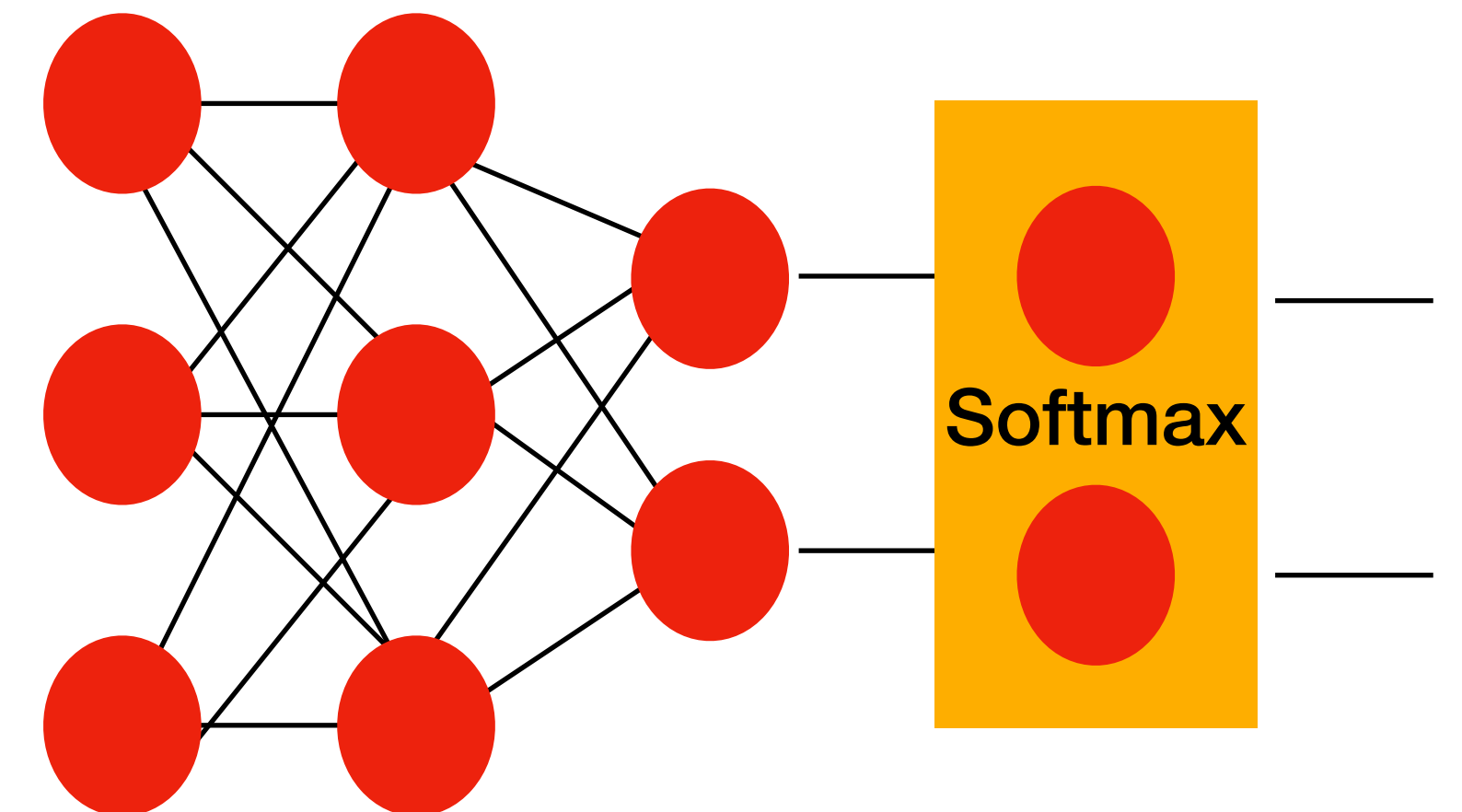
```
# Create DNN and define loss function  
dnn_model = DNN()  
optimizer = torch.optim.SGD(dnn_model.parameters(), lr=hyperparam.lr)  
loss = nn.CrossEntropyLoss()
```

DNNs as Multi-Class Classifiers

Softmax layer

- For binary classification, we can merely threshold the output neuron, as was shown previously for Linear Regression
- For multi-class classification (when the classes are exclusive), each neuron in the output layer corresponds to a single class.
 - The output layer is then modified to replace the individual activation functions with a shared **soft-max function**
 - Outputs represent estimated probability of the corresponding class
 - The activation potential, v_k , is still computed first, where it represents the “score” for each class.
 - Hence class is represented by a neuron with own weight vector
 - The neuron with the highest score is selected as the class (e.g. argmax)

$$\hat{y}_i = p_{y_i} = \phi_k(v) = \frac{e^{v_k}}{\sum_{j=1}^K e^{v_j}}$$



Cross Entropy Loss to train network

$$L_{ce} = -\frac{1}{m} \sum_{i=1}^m y_i \log(p_{y_i})$$

DNN Training

Training prediction and evaluation

- We can now begin the training process, since everything has been defined and initialized
- We need to do a few other things:
 - Keep track of training losses
 - Iterate over each epoch
 - Perform mini-batch gradient descent
 - Update the weights

```
# Train the DNN
tr_avgLoss_list = []
tr_accuracy_list = []
dev_avgLoss_list = []
dev_accuracy_list = []

# Loop over epochs
for epoch in range(hyperparam.num_epochs):
    tr_num_correct = 0
    tr_num_samples = 0
    tr_total_loss = 0.0

    dev_num_correct = 0
    dev_num_samples = 0
    dev_total_loss = 0.0

    # Training
    dnn_model.train(True)

    with torch.set_grad_enabled(True):
        for local_batch, local_labels in training_gen:
            optimizer.zero_grad()

            local_batch = local_batch.float()
            local_labels = local_labels.float()
            local_batch, local_labels = Variable(local_batch), Variable(local_labels)

            # Model computations
            out1 = dnn_model(local_batch)

            #CrossEntropy loss calculation
            pLoss = loss(out1, local_labels.long())
            tr_total_loss += pLoss*hyperparam.bs #Correct for average based on batch size

            # Backpropagation
            pLoss.backward() #gradient calculation
            optimizer.step() #weight update

            sel_class = torch.argmax(out1, dim=1)

            tr_num_correct += sel_class.eq(local_labels).sum().item()
            tr_num_samples += hyperparam.bs

    tr_avgLoss = tr_total_loss/len(training_gen.dataset)
    tr_avgLoss_list.append(tr_avgLoss)

    tr_accuracy = tr_num_correct/tr_num_samples
    tr_accuracy_list.append(tr_accuracy)
```


DNN Training

Validation Loss for Performance Evaluation

- We need to always be aware of the potential to overfit
- Hence, we should evaluate the model as it trains, using the validation/development data
- DO NOT update network based on this
 - Can use this for early stopping and model selection (more on this next)

```
# Validation
with torch.set_grad_enabled(False):
    dnn_model.eval()

    for local_batch, local_labels in dev_gen:

        local_batch = local_batch.float()
        local_labels = local_labels.float()
        local_batch, local_labels = Variable(local_batch), Variable(local_labels)

        # Model computations
        out1 = dnn_model(local_batch)

        #CrossEntropy loss calculation
        pLoss = loss(out1, local_labels.long())
        dev_total_loss += pLoss*hyperparam.bs #Correct for average based on batch size

        sel_class = torch.argmax(out1, dim=1)

        dev_num_correct += sel_class.eq(local_labels).sum().item()
        #print(correction)
        dev_num_samples += hyperparam.bs

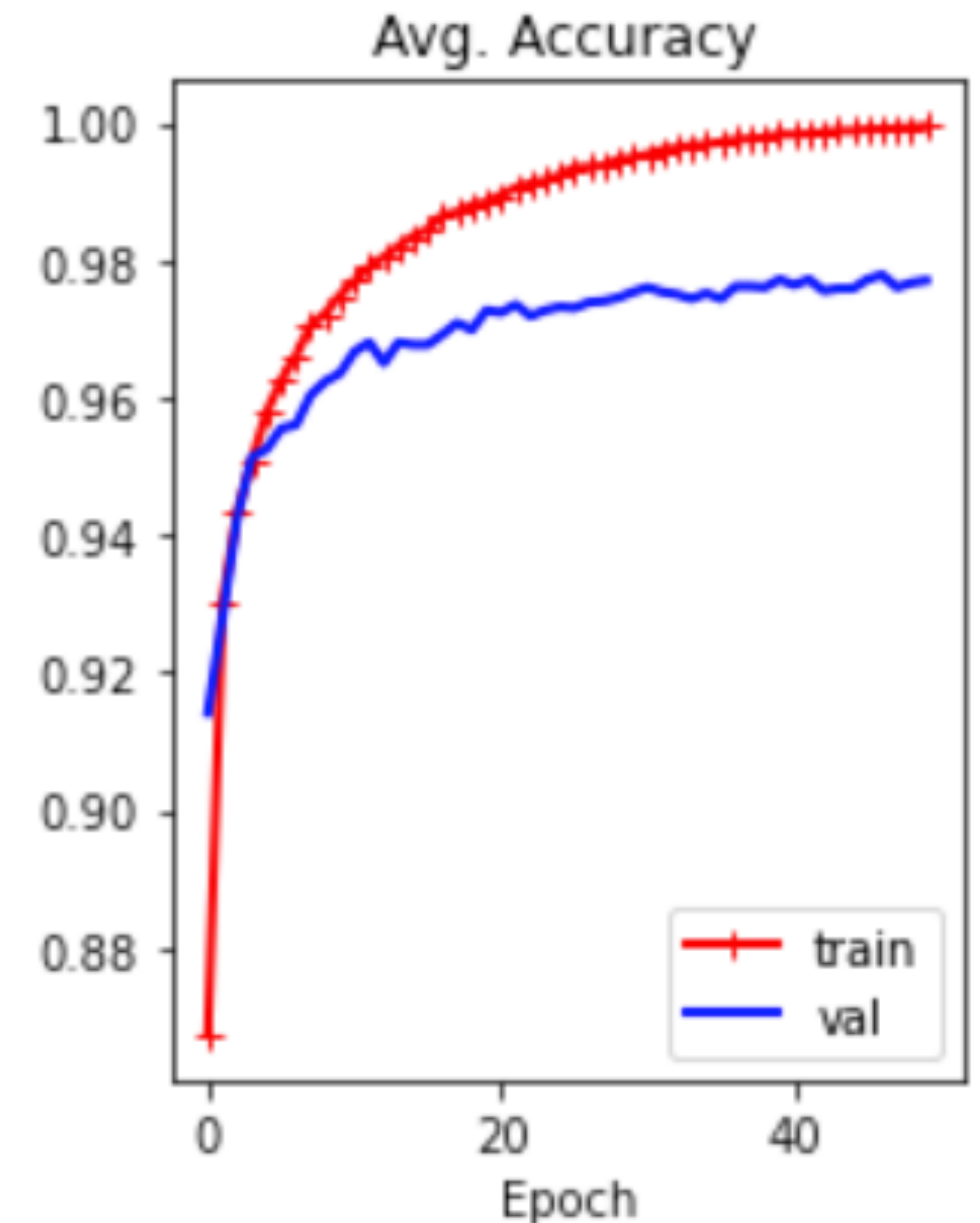
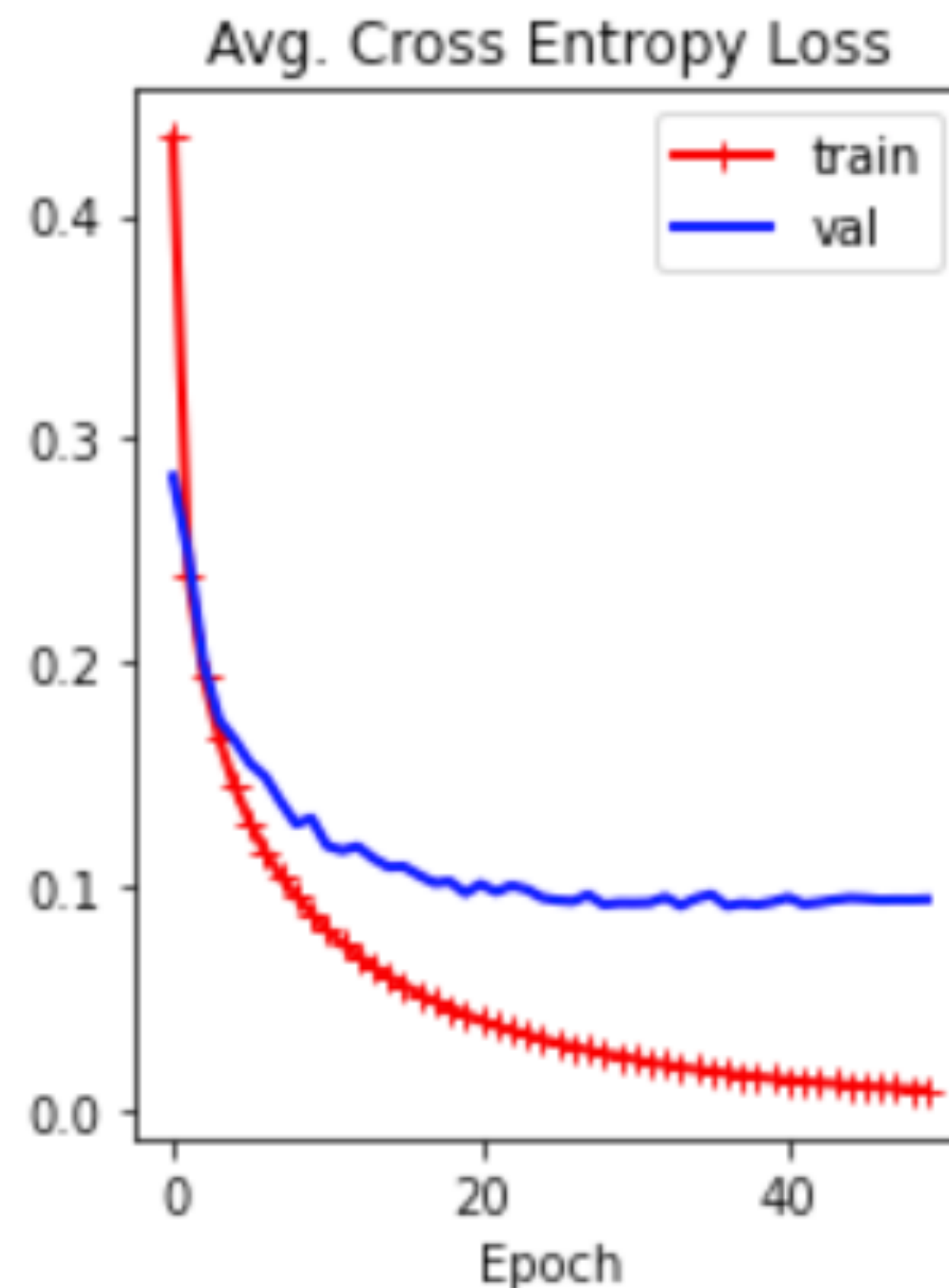
    dev_avgLoss = dev_total_loss/len(dev_gen.dataset)
    dev_avgLoss_list.append(dev_avgLoss)

    dev_accuracy = dev_num_correct/dev_num_samples
    dev_accuracy_list.append(dev_accuracy)
```

Evaluating Training and Validation Errors

Learning Curves

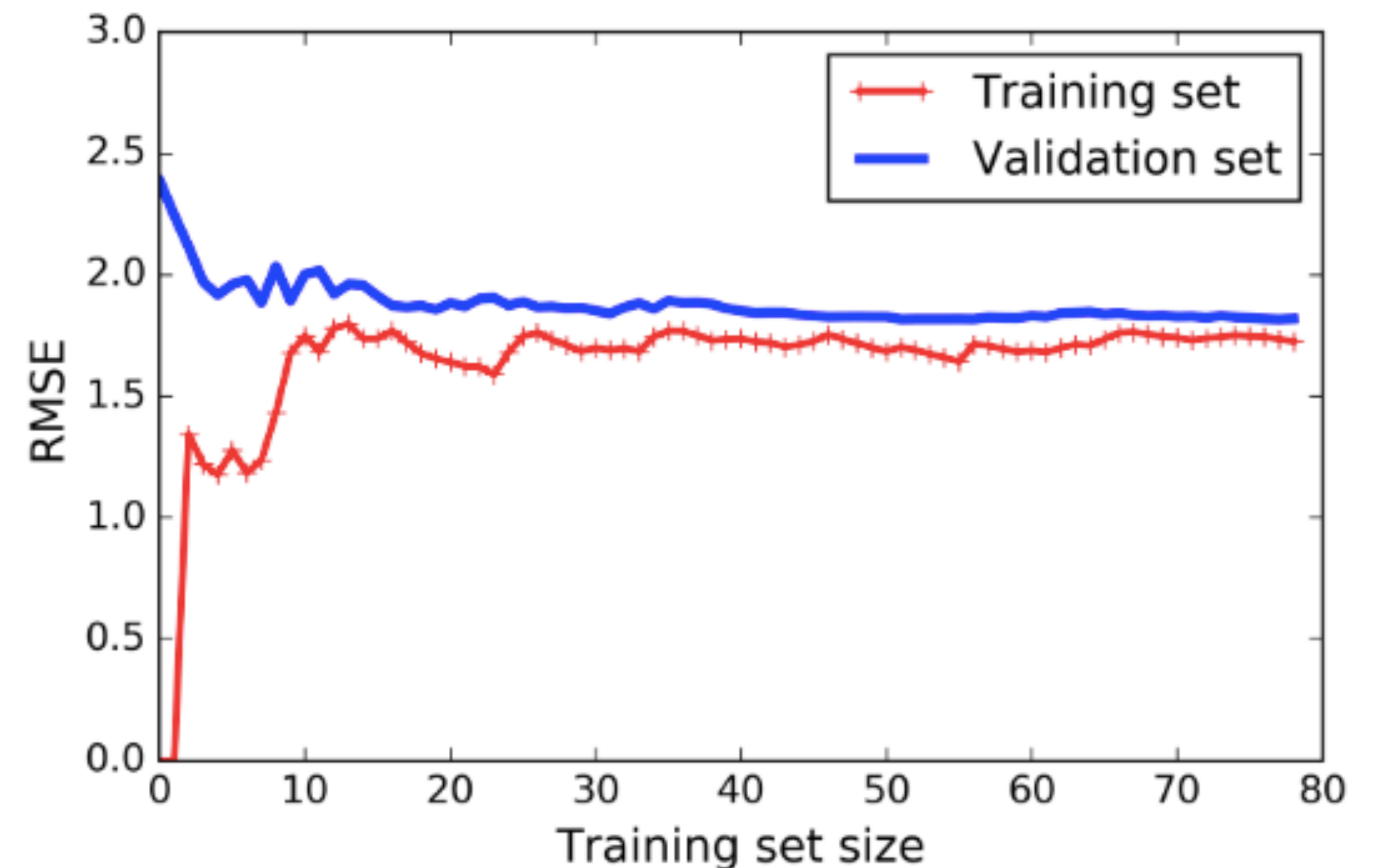
- Performance improves for each iteration
- Works better on training data, than validation/development data
- Possibly could run this for more epochs



Determining Fit from Training/Val Errors

General way to determine if over- or under-fitting the data

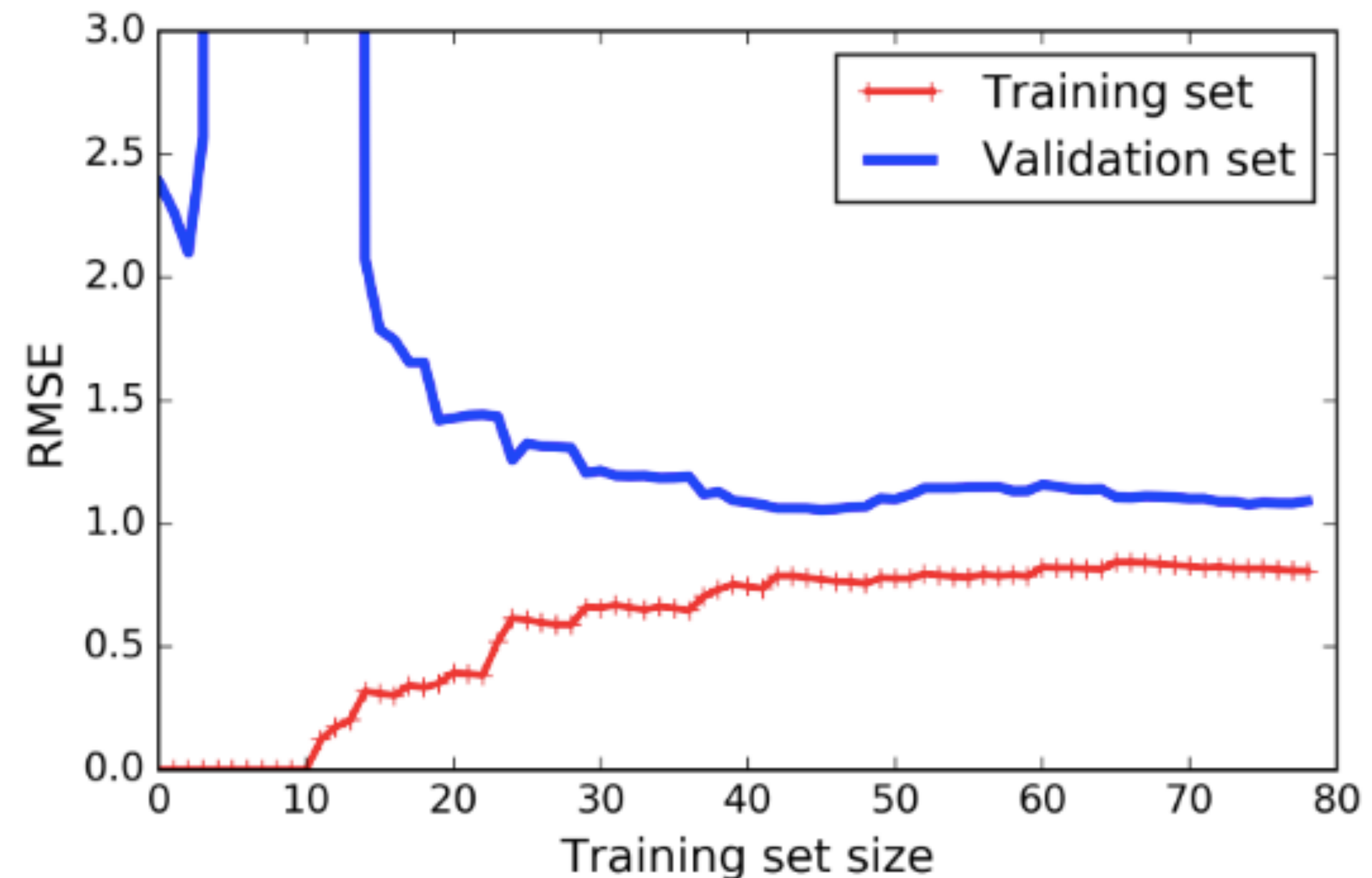
- Learning curves allow us to evaluate the model's performance on the training and validation sets as a function of iteration (or a different hyper parameter).
- For the training data, the performance gets worse as the training set size increases, then it plateaus
- For the validation data, it doesn't generalize when the training set size is small, but it slightly improves as size increases (though not much)
- This is an example of **underfitting**. The curves plateau, and they are close and high in value
 - Adding more training data doesn't help!
 - You need a more complex and better model



Determining Fit from Training/Val Errors

General way to determine if over- or under-fitting the data

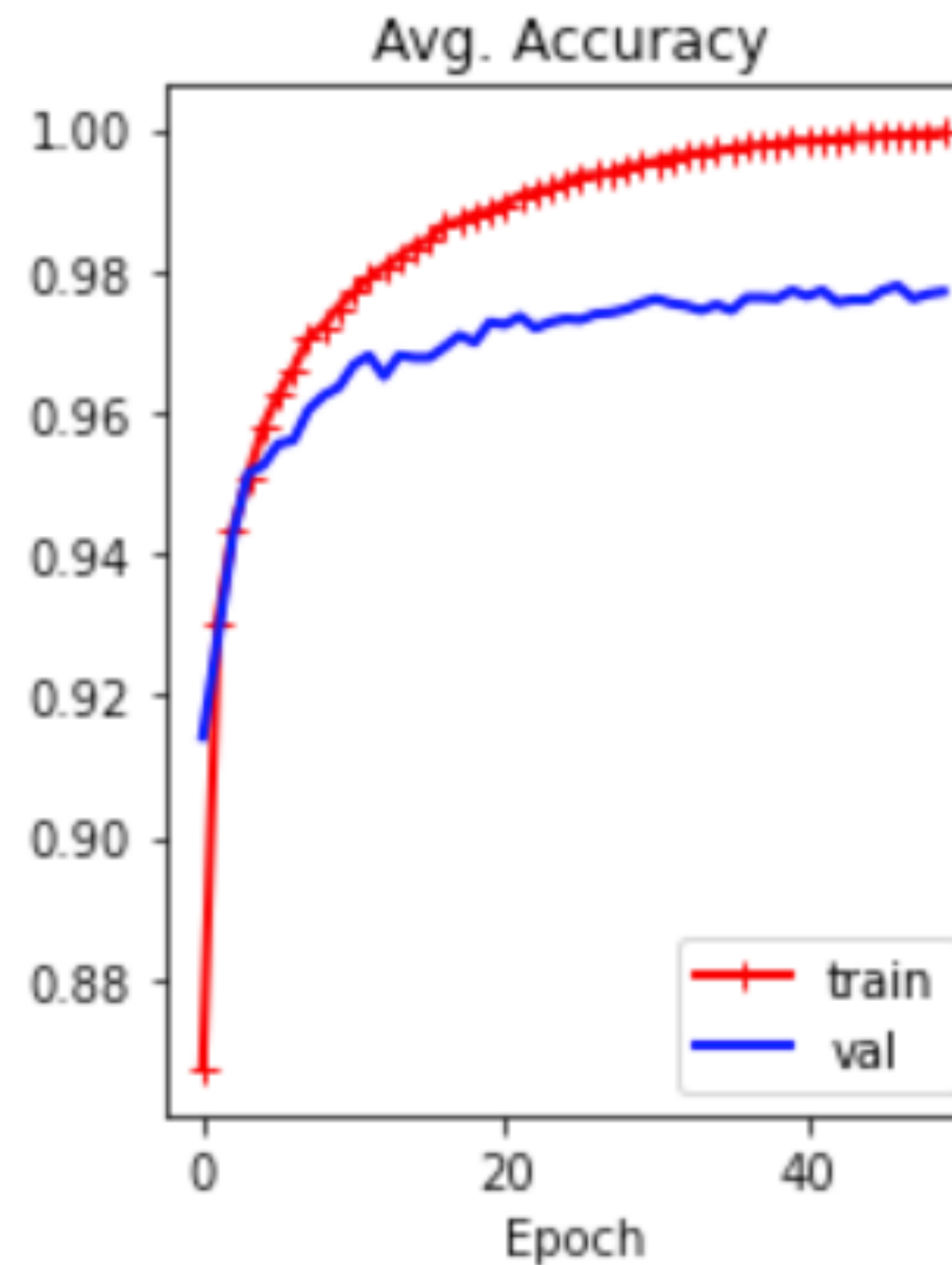
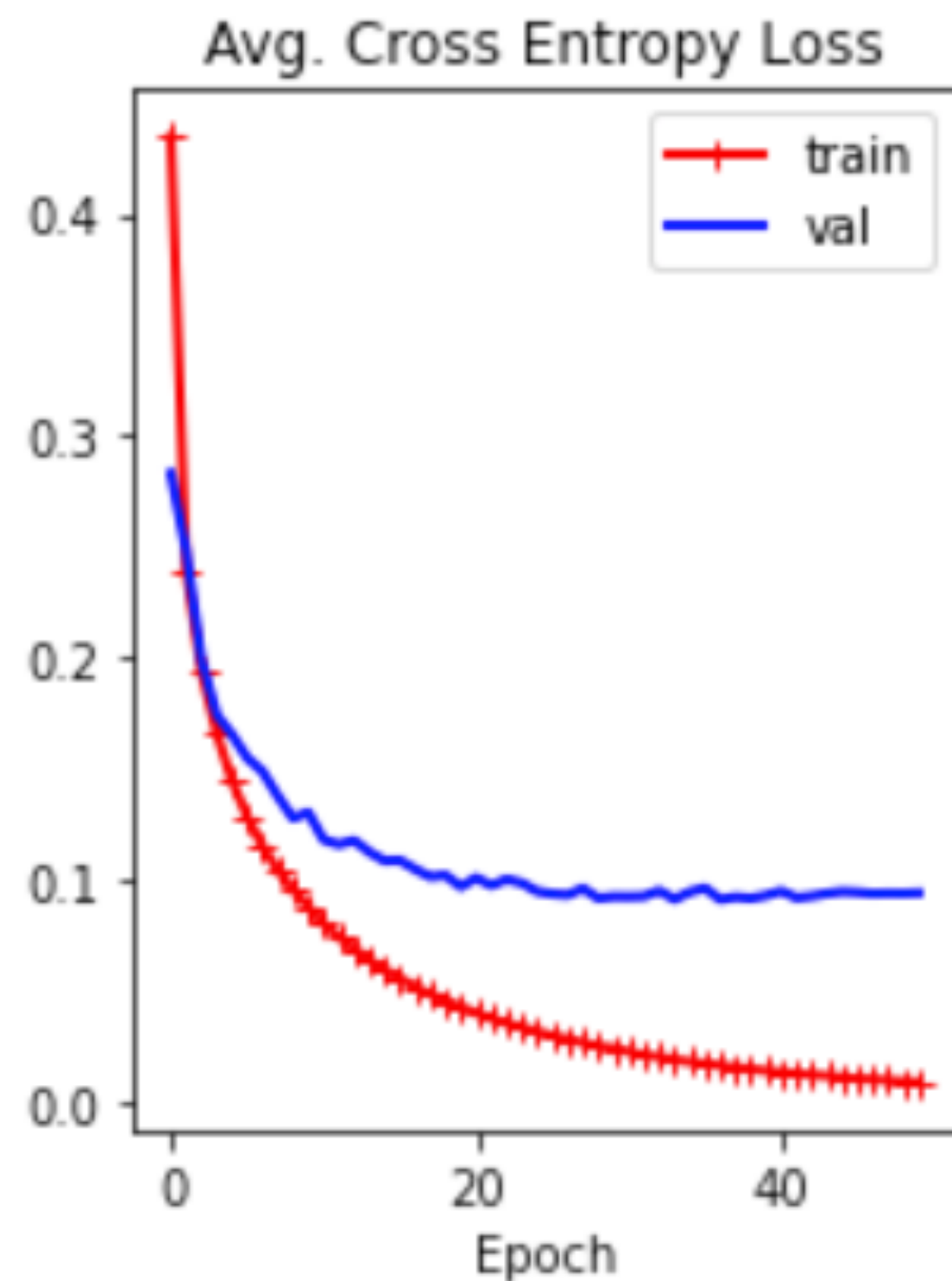
- Let's suppose we increase the complexity of the model, and train this model using different training set sizes.
- These curves are similar but different from the more simpler model's curves
- The training curve is better and lower than the validation curve, with a noticeable gap between the two
 - This is a sign that the model is **overfitting**.
 - Increasing the data set size may help avoid this, until the validation error reaches the training error



Evaluating Training and Validation Errors

Learning Curves

- Is this model overfitting or underfitting?



The Bias-Variance Tradeoff

Generalization error

- A model's generalization error is based on:
 - **Bias**: wrong assumptions in the model. Leads to underfitting when is high.
 - **Variance**: the model is overly sensitive to the data, leading to overfitting.
 - **Irreducible error**: the data is noisy. Need to get newer data or clean existing data in this case
- The ***tradeoff***:
 - Increasing complexity typically **increases** variance and **reduces** bias (hence overfitting)
 - Reducing a model's complexity **increases** its bias and **reduces** variance (hence underfitting)
- Regularization is subsequently used to reduce overfitting (more next week)

Next Class

**Neural Networks IV: DNN Regularization,
Optimization and Initialization**

Read Chapter 11 from HOML