S&DS 689 Homework 3

Jonas Katona

December 21, 2023

1 Problem 1

1.1 Introduction

One of the earliest pioneering CNN architectures was LeNet-5, first proposed by Lecun, Bottou, Bengio, and Haffner in 1998 and used in online handwriting recognition. In particular, the authors apply LeNet-5 in the recognition of handwritten digits, which is exactly what this problem is asking us to do. As borrowed from the Wikipedia page for LeNet, the architecture for LeNet-5 can be shown in Figure 1. We implement this architecture for our problem using PyTorch with GPU acceleration in the script `1.py` (Section 1.3.1).

We would be concerned with splitting the data into training and test datasets, but this has already been done for us in the MNIST database. The script `1.py` works as follows. At first, we download the MNIST databases for training and testing. Since the output of the network should be a 10-dimensional vector of 1s and 0s, we convert the numerical categories in the dataset to unit vectors that represent them. For instance, the digit 5 would become $(0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$. Then taking the hints in the problem, we normalize the pixel values by dividing them by 255 in all cases and reshape each sample of rank 1 (a 1D array in 784 dimensions) to a sample of rank 2 (a 2D array/image in 28×28 dimensions). This allows us to use convolutional layers and actually create a CNN. From here, we implement the LeNet-5 architecture and use the Adam optimizer to train it on the MNIST training dataset with a learning rate of $\lambda = 10^{-3}$, 50 epochs, and splitting the dataset into 100 batches. For each epoch, we print both the cross-entropy loss and the training error (fraction of misidentified digits in the training dataset). Finally, after finishing the 50 epochs, we test our trained network on the MNIST test dataset and return the testing error (fraction of misidentified digits in the testing dataset).

There are a few differences between our implementation of LeNet-5 and that which Lecun et al. proposed in 1998. In particular, we replaced all Gaussian activation functions with ReLU activation functions and added batch normalization between the convolution and ReLU layers. However, other than that, our implementation stays faithful to the original; we even introduce the same stride and padding.

`1.sh` (Section 1.3.2) runs `1.py` and pipes the output into `1.txt`. Afterwards, it extracts the cross-entropy losses and training errors as functions of epoch and pipes these into `1_plot.py` (Section 1.3.3) to generate the plots in Figure 2. Hence, to retrieve the final testing error/accuracy, we must look at the last line of `1.txt` directly.

1.2 Results

Figure 2 shows the training error and cross-entropy loss as a function of epoch for the training of our implementation of LeNet-5. In particular, the training error goes to zero after our

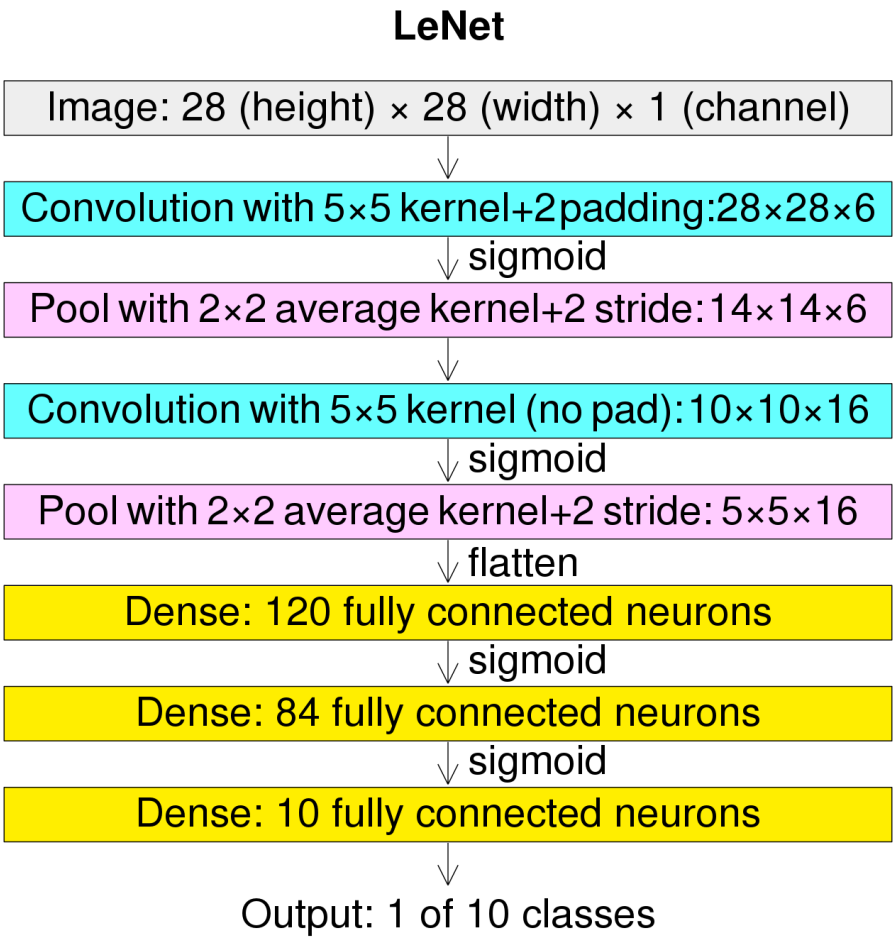

Figure 1: A diagram describing/showing the network architecture for LeNet-5, as taken from the LeNet Wikipedia page.

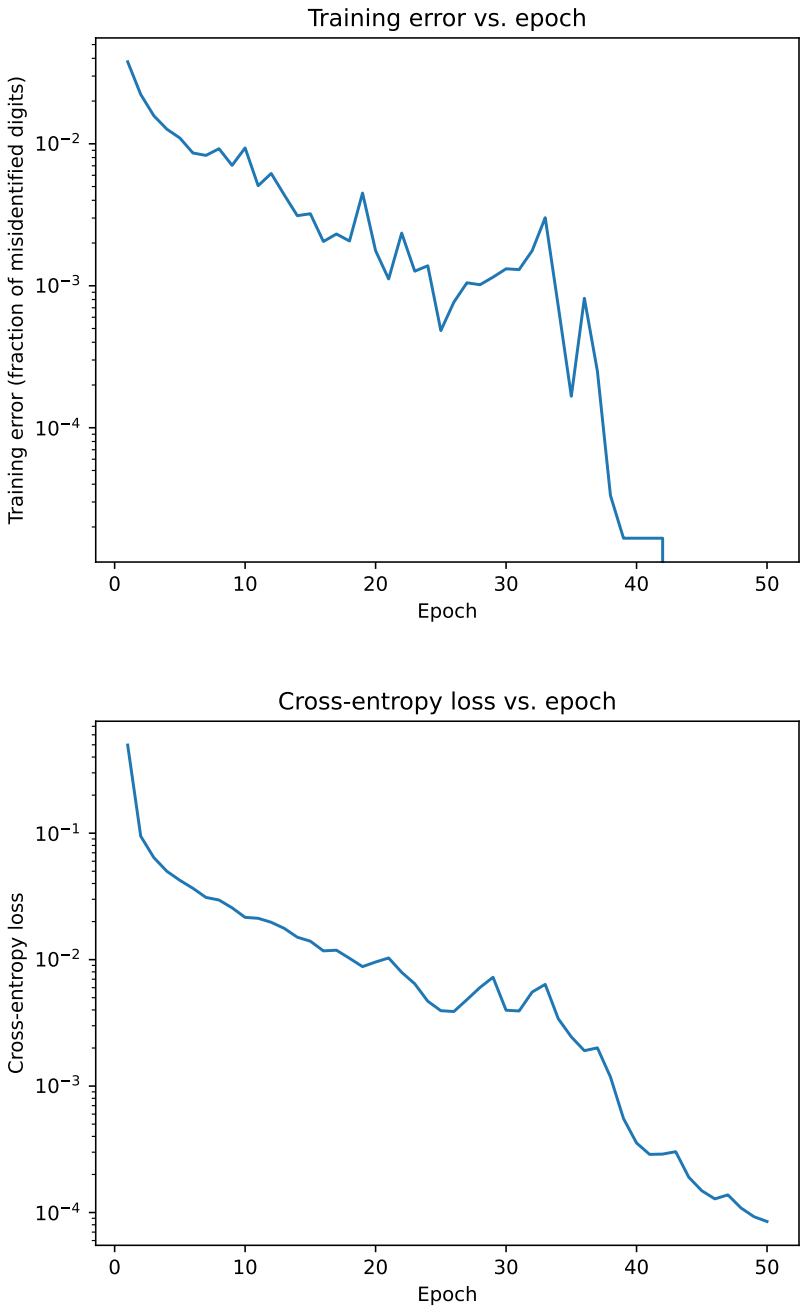

Figure 2: Training errors (fraction of misidentified digits for training) and cross-entropy losses as a function of epoch in the training of the LeNet-5 architecture implemented for Problem 1.

training passes 42 epochs, demonstrating how our trained LeNet-5 architecture learns the MNIST training dataset *exactly* and after a quite small number of epochs. Of course, the cross-entropy loss is still nonzero in this case. This is because our network ultimately outputs a vector in 10 dimensions which need not consist of 0s and 1s, but still classifies each digit correctly since the argmax across the coordinates of that vector matches with the correct digit. And ending with a cross-entropy loss of around 10^{-4} shows superb predictions for training.

Originally, I did not implement `1.py` using GPU acceleration and `1.py` took over 5 minutes to run (maybe even 10 minutes if I recall). However, once I was able to get PyTorch to recognize my Nvidia GPU with CUDA installed, `1.py` finished running in about one minute. This was a noticeable improvement.

By reading the last line of `1.txt`, we can also retrieve the final testing error (fraction of misidentified digits in the testing dataset): 0.90%. Equivalently, we have a final testing accuracy of 99.10%, which is well above 98%. I did not need to do anything too laborious to eventually achieve this; LeNet-5 is a well-known CNN architecture which is quite accurate in recognizing handwritten digits, can be trained quickly, and made huge waves in the learning community when it was first introduced in 1998 for exactly these reasons.

1.3 Code

1.3.1 1.py

`1.py` trains and tests our implementation of the LeNet-5 architecture (i.e., with a few changes from the original proposed by Lecun et al. in 1998) on the MNIST digit dataset using PyTorch with GPU acceleration via CUDA. The hyperparameters for training are described in Section 1.1. The script returns the cross-entropy losses and testing errors for each epoch and then the training error at the very end.

```
# download digits datasets

# the CSVs are in the following format:
# -first column is the digit in question
# -rest of the columns contain the image data
# -rows are samples

import pandas as pd
import numpy as np
import torch

if torch.cuda.is_available():
  print("CUDA available. Using GPU acceleration.")
  device = "cuda"
else:
  print("CUDA is NOT available. Using CPU for training.")
  device = "cpu"

# load digit dataset for TRAINING
# load directly as array of ints, skip header
train_data = pd.read_csv('mnist_train.csv', skiprows=1, header=None, dtype=int).values
# load labels as a list of unit vectors
```

```
# (0 if wrong label, 1 if correct label, for each sample)
train_labels = torch.tensor(train_data[:,0], device=device)
train_labels_tensor = torch.tensor(np.arange(10) == train_data[:,0,None],
                                   dtype=torch.float32, device=device)
n_train = train_data[:,0].size # number of samples in training
N = int(np.sqrt(train_data[0].size - 1)) # length of 2D image
# reshape each 1D array of sample data of size 784 to a 2D image of size (28, 28, 1)
train_data = np.expand_dims(train_data[:,1:].reshape(n_train, N, N), axis=1)
# normalize train_data, convert to PyTorch tensor
train_data = torch.tensor(train_data / np.max(train_data),
                          dtype=torch.float32, device=device)

# load digit dataset for TESTING
# load directly as array of ints, skip header
test_data = pd.read_csv('mnist_test.csv', skiprows=1, header=None, dtype=int).values
n_test = test_data[:,0].size
test_labels = torch.tensor(test_data[:,0], device=device)
test_data = np.expand_dims(test_data[:,1:].reshape(n_test, N, N), axis=1)
# normalize train_data, convert to PyTorch tensor
test_data = torch.tensor(test_data / np.max(test_data),
                         dtype=torch.float32, device=device)

# train neural network
import torch.nn as nn
import torch.optim as optim

# hyperparameters
loss_fn = nn.CrossEntropyLoss()  # use cross entropy loss
n_epochs = 50 # number of epochs
lr = 0.001 # learning rate for Adam optimizer
batches = 1e2 # number of batches

# use Sequential PyTorch class to construct LeNet-5 architecture
# with some differences:
# -replaced Gaussian filter with ReLU
# -added batch normalization between convolution and ReLU layers

class CNN(nn.Module):
    def __init__(self, channels, classes):
        super(CNN, self).__init__() # initialize parent constructor

    # conv => batchnorm => ReLU => pooling (first)
        self.layer1 = nn.Sequential(
            nn.Conv2d(channels, 6, kernel_size=(5, 5), padding=2),
            nn.BatchNorm2d(6),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2))

# conv => batchnorm => ReLU => pooling (second)
```

```
        self.layer2 = nn.Sequential(
            nn.Conv2d(6, 16, kernel_size=(5, 5), padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=(2, 2), stride=2))

        # series of FC NNs
        self.layer3 = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(),
            nn.Linear(120, 84),
            nn.ReLU(),
            nn.Linear(84, classes)
        )

    def forward(self, x):
        output = self.layer1(x)
        output = self.layer2(output)
        output = torch.flatten(output,start_dim=1)
        output = self.layer3(output)
        return output

model = CNN(channels=1, classes=10).to(device) # define network
# use Adam optimizer with learning rate lr
optimizer = optim.Adam(model.parameters(), lr=lr)
batch_size = int(np.floor(n_train / batches)) # batch size
from random import shuffle
for epoch in (np.arange(n_epochs) + 1):
    inds = list(range(n_train))
    shuffle(inds)
    running_loss = 0
    for i in np.arange(batches): # loop through batches
        # randomly sample indices for each batch without replacement
        batch_ind = inds[-batch_size:]
        del inds[-batch_size:]
        pred = model(train_data[[batch_ind]]) # forward pass
        loss = loss_fn(pred, train_labels_tensor[[batch_ind]]) # compute loss
        running_loss += loss * len(batch_ind)
        optimizer.zero_grad() # zero out gradient
        loss.backward() # backpropagation
        optimizer.step() # adjust parameters
    print(f'Cross-entropy loss for epoch {epoch}: {running_loss / n_train}')
    predicted_labels = torch.argmax(model(train_data), dim=1)
    frac_incorrect = torch.count_nonzero(predicted_labels - train_labels) / n_train
    print(f'Training error for epoch {epoch}: {frac_incorrect}')

# test network on test dataset, get predicted labels
predicted_labels = torch.argmax(model(test_data), dim=1)
# test fraction of predicted labels correct
```

```
frac_incorrect = torch.count_nonzero(predicted_labels - test_labels) / n_test
print(f'Testing error: {frac_incorrect}')
```

1.3.2 1.sh

`1.sh` runs `1.py`, takes the cross-entropy losses and training errors as functions of epoch for the LeNet-5 architecture implemented in Problem 1, and then processes and pipes this data into `1_plot.py` to save the plots shown in Figure 2.

```
# runs 1.py and saves outputs in 1.txt (for viewing later)
# Then, processes the output and pipes it into 1_plot.py for plotting
# the cross-entropy loss and training error as a function of epoch

cd /c/Users/jonas/Documents/SciML_class/HW3
python3 1.py > 1.txt
# extract cross-entropy loss as a function of epoch
grep -i Cross-entropy 1.txt | awk '{print $NF}' > tmp.txt
# pipe into 1_plot.py contains three columns:
# 1) cross-entropy loss as a function of epoch
# 2) training error as a function of epoch
paste tmp.txt <(grep -i Training 1.txt | awk '{print $NF}') | python3 1_plot.py
```

1.3.3 1_plot.py

`1_plot.py` generates and saves the two plots shown in Figure 2 using the outputs from `1.py`.

```
import sys
import numpy as np

data = np.loadtxt(sys.stdin, dtype=np.float64)
epochs = np.arange(len(data)) + 1

import matplotlib.pyplot as plt
plt.plot(epochs,data[:,0])
plt.title('Cross-entropy loss vs. epoch')
plt.xlabel('Epoch')
plt.ylabel('Cross-entropy loss')
plt.yscale('log')
plt.savefig('1_loss.pdf')
plt.close()

plt.plot(epochs,data[:,1])
plt.title('Training error vs. epoch')
plt.xlabel('Epoch')
plt.ylabel('Training error (fraction of misidentified digits)')
plt.yscale('log')
plt.savefig('1_error.pdf')
plt.close()
```

2 Problem 2

2.1 Introduction

For this problem, we implement vanilla LSTM architectures (i.e., using only one LSTM layer) with the following widths: 10, 20, 30, 40, and 50, and a lookback period of 1. Hence, the only network hyperparameter we tune is the width. This is done in TensorFlow/Keras, and since I could not get TensorFlow to recognize my GPU for some reason (this was not a problem in PyTorch), I ran the training and testing on my CPU. Running this across all widths took around three minutes in total, so it really did not pose an issue that I was forced to use my CPU.

The training and testing was done in `2.py` (Section 2.3.1). The training data was taken across 201 uniformly distributed samples on $[0, 10]$ (i.e., with a spacing of $\Delta x = 0.05$) for the function $f(x) = \cos(2\pi x)$ with mean-zero, standard deviation 0.02 Gaussian noise added. Testing was done on 41 uniformly distributed samples in $[10, 12]$ (so again with $\Delta x = 0.05$) for the same f. Since I did the training and testing on the CPU, I had no qualms with using the native NumPy libraries for defining the training and test datasets, although I would use the TensorFlow implementations if running these on my GPU. We used an MSE loss and optimized our network parameters using the Adam optimizer with a learning rate of $\lambda = 10^{-3}$ over 100 epochs with no batch training. The output is the MSE losses as functions of epoch, the L^2 relative error for testing, and plots of the training/testing data vs. the prediction via our LSTM, the latter of which are saved as the plots shown in Figure 3 rather than displayed immediately as figures.

To run `2.py` over multiple network widths, `2.py` runs with a network width piped in as an input directly from the command line. This allows us to use `2.sh` (Section 2.3.2) to run `2.py` over the network widths 10, 20, 30, 40, and 50, and save the MSE losses vs. epoch in each case. Then, after accumulating the list of MSE losses vs. epoch in the file `tmp.txt`, these can be piped into `2_plot.py` (Section 2.3.3) to generate the plots shown in Figure 4.

Initially, I took the hint and tried implementing an LSTM with a lookback period of 10, as suggested in the hint, but I could not get the testing error to fall below around 3%. I tried varying the width of the LSTM layer from 5 to 100, adding an extra LSTM layer, linear feedforward layer at the end, nesting LSTM and linear feedforward layers, and/or adding ReLU layers, etc. to no avail. However, by using a lookback period of only 1 (i.e., only use the previous timestep to predict the next), I was able to get an L^2 relative error for testing as low as 0.0058% for a vanilla LSTM with a width of only 40. This demonstrates the significance of overfitting when training LSTMs. In particular, while increasing the lookback period might help in terms of gauging the predictive complexity of the LSTM, doing so also decreases the number of available samples for training, thereby increasing the degree of overfitting.

2.2 Results

Even for a network width of only 10, qualitatively speaking, Figure 3 shows that the LSTM prediction is quite accurate and mostly fails where the curve turns back around near the edges of the range. However, the orange curve in Figure 3 (the testing data, which is only visible where the prediction does not match with it) gradually becomes obscured as the network width increases, thereby showing the increasing accuracy of the LSTM prediction.

More concretely, we may look at the MSE losses vs. epoch and L^2 relative error for testing shown in Figure 4. These plots show that we only get an L^2 relative error below 2% once the network width is at 20 or greater (although we did not test widths greater than

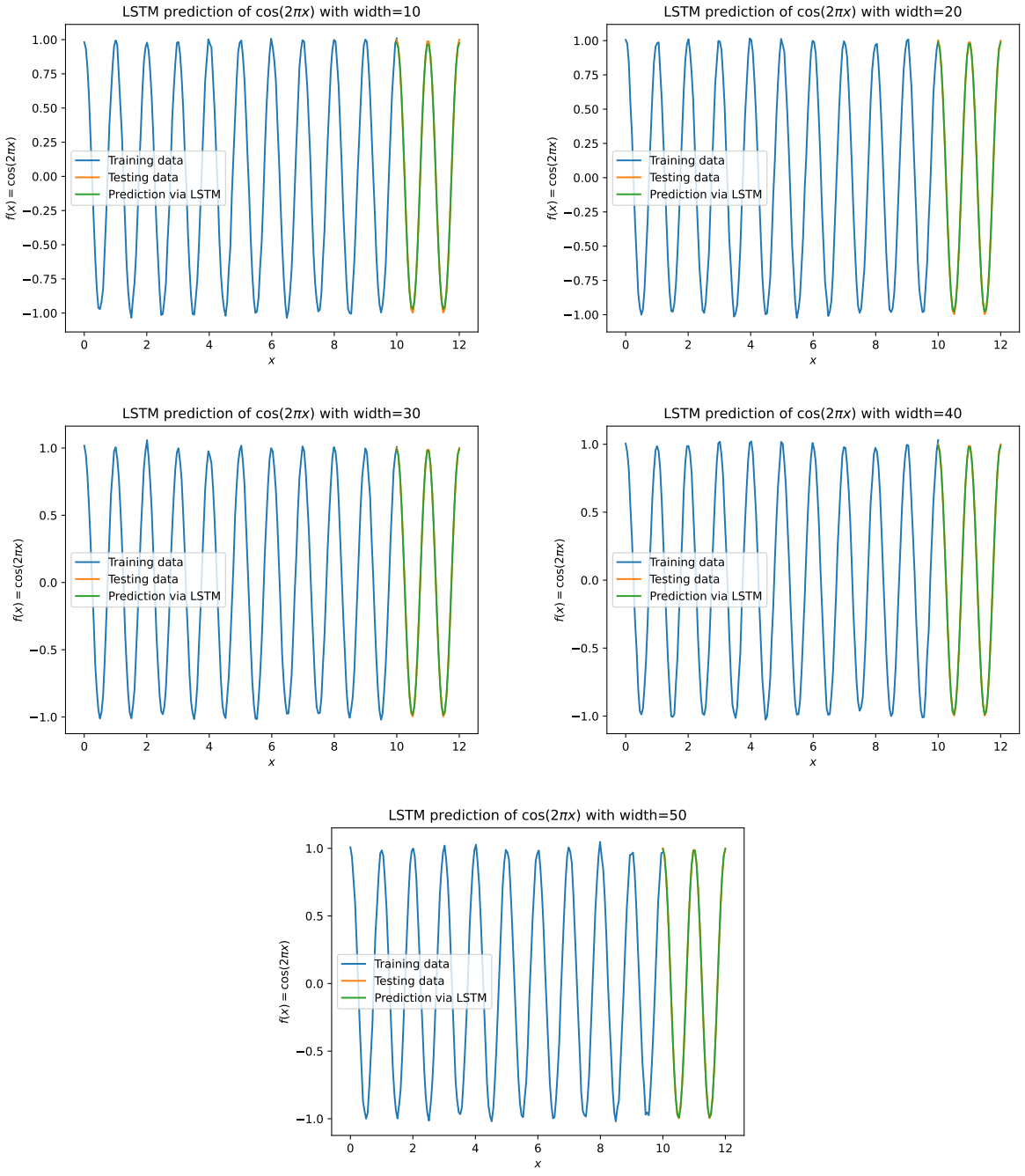

Figure 3: Plots for the training data, testing data, and LSTM prediction across multiple network widths (10, 20, 30, 40, and 50) for the vanilla LSTM architecture implemented in Problem 2 for learning $\cos(2\pi x)$ on $[0, 10]$ for predictions on $[10, 12]$.

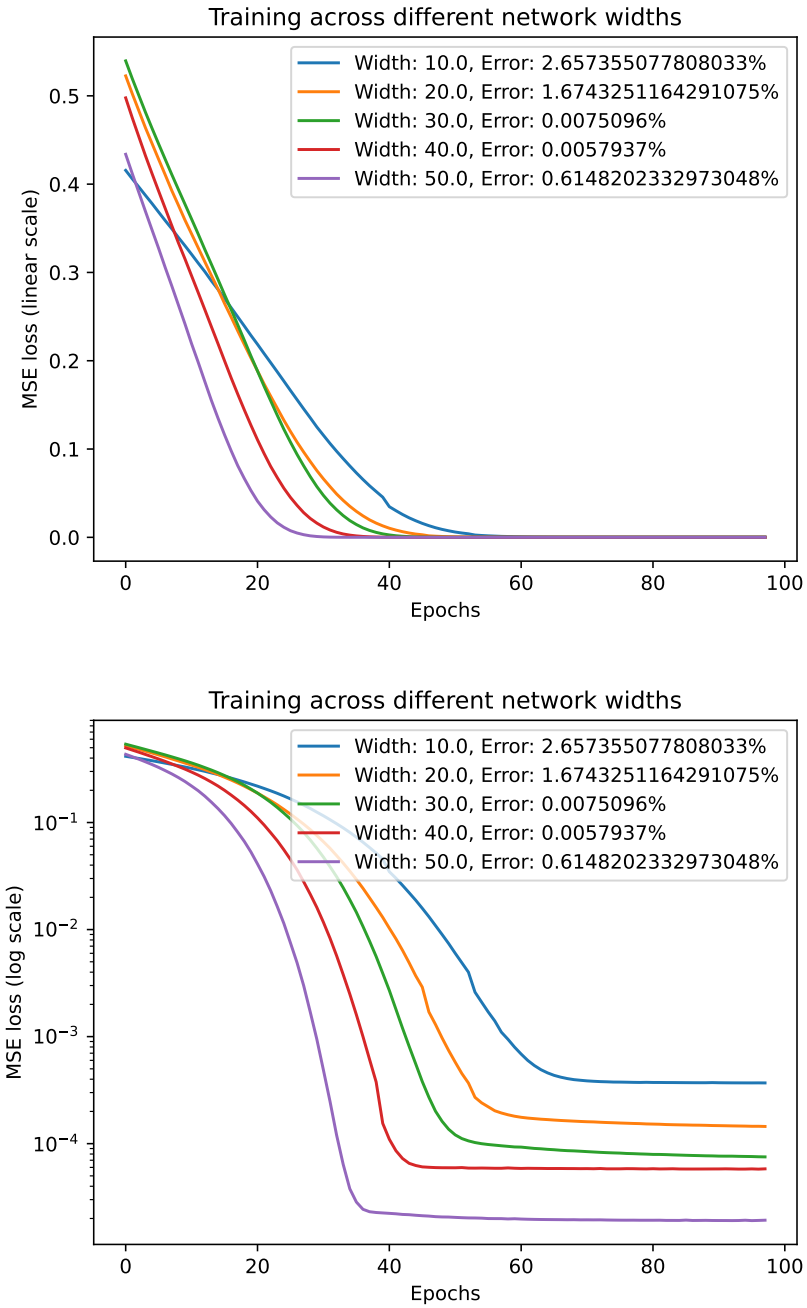

Figure 4: The MSE training loss vs. epoch across multiple network widths (10, 20, 30, 40, and 50) for the vanilla LSTM architecture implemented in Problem 2 for learning $\cos(2\pi x)$ on $[0, 10]$ for predictions on $[10, 12]$. The reported errors in the legend are the L^2 relative errors on the test data for the network with the given width.

50). In particular, the L^2 relative error quickly drops by over two orders of magnitude once the width increases past 20 to 30, but then saturates and does not decrease significantly as the width goes from 30 to 40. The effects of overfitting can be seen slightly as the network width increases from 40 to 50, since the L^2 relative error increases again (although still stays below 1%) despite the clear decrease in MSE loss for training. Yet again, this shows another example of how the effects of overfitting can be quite detectable for LSTMs, and how one ultimately needs to increase the number of samples to boost the accuracy of LSTM predictions rather than add to the model complexity.

2.3 Code

2.3.1 2.py

`2.py` trains and tests our implementation of the vanilla LSTM architecture on learning $\cos(2\pi x)$; the training and testing domains and training hyperparameters are defined in Section 2.1. The script prints the MSE losses for each epoch and the L^2 relative error for testing, and then saves a plot of the testing/training data and the LSTM prediction vs. x on $[0, 12]$. All network hyperparameters are defined therein aside from the network width, which is taken as command line input; see `2.sh` as described in Section 2.3.2 for why this is relevant.

```
a = 0 # start of training domain
b = 10 # end of training domain/start of testing domain
c = 12 # end of testing domain
# (training is on [a,b], testing is on [b,c])
dx = 0.05 # interval size
mu = 0 # mean of Gaussian noise
sigma = 0.02 # standard deviation of Gaussian noise
n_epochs = 100 # number of epochs for training
w = int(input()) # width of hidden dense layer (taken as command line input)
lr = 0.001 # learning rate

# generate training dataset
import numpy as np
from keras import Sequential
from keras.layers import LSTM, Dense
import matplotlib.pyplot as plt

x_train = np.linspace(a, b, int((b-a) / dx))
y_train = np.cos(2*np.pi*x_train)
# add noise to training data
y_train += np.random.normal(loc=mu, scale=sigma, size=np.shape(y_train))
x_test = np.linspace(b, c, int((c-b) / dx))
y_test = np.cos(2*np.pi*x_test)

input_train = y_train.reshape((len(y_train),1,1))
input_test = y_test.reshape((len(y_test),1,1))

from keras.optimizers import Adam
optimizer = Adam(learning_rate=lr)
```

```
model = Sequential()
model.add(LSTM(w, input_shape = (1, 1)))
model.add(Dense(1))
model.compile(optimizer=optimizer, loss='MSE')

h = model.fit(input_train, y_train, epochs=n_epochs, verbose=1)
y_pred = model.predict(input_test).reshape(1,-1)

# define L2 relative error
def L2_rel_error(approx, actual):
    top = np.sum(np.square(approx - actual))
    bottom = np.sum(np.square(actual))
    return np.sqrt(top / bottom)

# return L2 relative testing error
print(f'L2 relative testing error: {L2_rel_error(y_pred, y_test)}')

plt.title(f'LSTM prediction of $\cos(2\pi x)$ with width={w}')
plt.plot(x_train, y_train, label='Training data')
plt.plot(x_test, y_test, label='Testing data')
plt.plot(x_test, y_pred[0], label='Prediction via LSTM')
plt.xlabel('$x$')
plt.ylabel('$f(x)=\cos(2\pi x)$')
plt.legend()
plt.savefig(f'2_{w}.pdf')
plt.close()
```

2.3.2 2.sh

Using the fact that `2.py` (Section 2.3.1) takes in command line inputs for the network width, `2.sh` tests the LSTM training and testing across network widths 10, 20, 30, 40, and 50, and extracts the MSE losses as functions of epoch and L^2 relative testing errors for each case. After doing so, `2.sh` pipes these into `2_plot.py` (Section 2.3.3) which generates the plots in Figure 4.

```
# test different LSTM widths and plot the MSE loss vs. epoch
echo -n "" > tmp.txt # to pipe into plotting script
widths='10 20 30 40 50'
for w in ${widths}
do
    echo $w | python3 2.py > tmp0.txt # run with width w
    grep -i ms/step tmp0.txt | awk '{print $NF}' > tmp1.txt # get losses vs. epoch
    # add relative L2 error for testing
    tail -1 tmp0.txt | awk '{print $NF}' >> tmp1.txt
paste tmp.txt tmp1.txt > tmp0.txt # accumulate list of losses in tmp.txt
    cat tmp0.txt > tmp.txt
done
# add widths to end of tmp.txt then pipe into 2_plot.py for plotting
echo $widths >> tmp.txt
cat tmp.txt | python3 2_plot.py # pipe tmp.txt into 2_plot.py
```

2.3.3 2_plot.py

2_plot.py generates and saves the plots in Figure 4 with the help of 2.py (Section 2.3.1) and 2.sh (Section 2.3.2).

```
# plots MSE loss vs. epoch for different widths

import sys
import numpy as np
data = np.genfromtxt(sys.stdin, dtype=np.float64, invalid_raise = False)
widths = data[-1] # network widths
errors = data[-2] # L2 relative errors for testing
losses = data[:-2]

import matplotlib.pyplot as plt
plt.title('Training across different network widths')
plt.xlabel('Epochs')
plt.ylabel('MSE loss (linear scale)')
for i, w in enumerate(widths):
    plt.plot(np.arange(len(losses)), losses[:,i],
             label='Width: '+str(w)+', Error: '+str(100*errors[i])+'%')
plt.legend()
plt.savefig('2_linear.pdf')
plt.close()

plt.title('Training across different network widths')
plt.xlabel('Epochs')
plt.ylabel('MSE loss (log scale)')
for i, w in enumerate(widths):
    plt.plot(np.arange(len(losses)), losses[:,i],
             label='Width: '+str(w)+', Error: '+str(100*errors[i])+'%')
plt.legend()
plt.yscale('log')
plt.savefig('2_log.pdf')
plt.close()
```