Homework #4 - A

Spring 2020, CSE 446/546: Machine Learning
Richy Yun
Due: Thursday 5/28/2020 11:59 PM

Conceptual questions

A.1

a. *[2 points]* True or False: Training deep neural networks requires minimizing a non-convex loss function, and therefore gradient descent might not reach the globally-optimal solution.

   True. Due to the nonlinearity of the activation functions and increased complexity due to multiple layers, deep neural networks have non-convex loss functions.

b. *[2 points]* True or False: It is a good practice to initialize all weights to zero when training a deep neural network.

   False. As the loss function of a deep neural network is non-convex, you want to initialize the weights with approximations to make sure you don't end up in a local minima far from the absolute minima.

c. *[2 points]* True or False: We use non-linear activation functions in a neural network's hidden layers so that the network learns non-linear decision boundaries.

   True. If the activation functions were linear the decision boundaries would be as well, but that doesn't necessarily best describe the data.

d. *[2 points]* True or False: Given a neural network, the time complexity of the backward pass step in the backpropagation algorithm can be prohibitively larger compared to the relatively low time complexity of the forward pass step.

   False. The backward and forward passes have the same complexity.

e. *[2 points]* True or False: Autoencoders, where the encoder and decoder functions are both neural networks with nonlinear activations, can capture more variance of the data in its encoded representation than PCA using the same number of dimensions.

   True. PCA is a linear transformation, so the nonlinearity of the autoencoder would allow for better fitting.

Think before you train

A.2

a. *[5 points]*
One statistical problem is feature selection. Crime rate is usually calculated with demographic information (poverty level, race, etc.). However, crime rate is strongly affected by various issues such as reporting rate and police response and police presence. Crime reporting rate has been shown to be much lower in black communities likely due to their mistrust of police [1]. There is also rampant manipulation of reports by the police [2]. Predictive policing can fall into feedback loops in which police are repeatedly sent to the same neighborhoods regardless of the crime rate [3]. Miscalculations of crime rate can lead to unfair evaluations of the neighborhood and its residents.

Another statistical problem is deciphering correlation vs causation. Continuing with the crime rate example, deciphering why crime rate is high in certain areas can be extremely misleading as we saw in our previous homework. Although some parameters may seem to explain difference in crime rate between neighborhoods well, it is often difficult to uncover the main cause of that parameter. This can lead to incorrect assumptions on how to lower crime and help the neighborhoods in need.

1. Desmond, M., Papachristos, A. V., & Kirk, D. S. (2016). Police Violence and Citizen Crime Reporting in the Black Community. American Sociological Review, 81(5), 857–876.

2. Eterno, J., Verma, A., & Silverman, E. (2014). Police Manipulations of Crime Reporting: Insiders' Revelations. Justice Quarterly, 33(5), 811-835.

3. Ensign, D., Friedler, S. A., Neville, S., Scheidegger, C., & Venkatasubramanian, S. (2017). Runaway feedback loops in predictive policing. arXiv preprint arXiv:1706.09847.

b. *[5 points]* For the first problem, either adding more features or using additional features to normalize the original metrics would better represent the landscape of a neighborhood. For the second problem, both better regularization of the features and further testing in an effort to better analyze and fully understand the results would be helpful in finding more effective solutions.

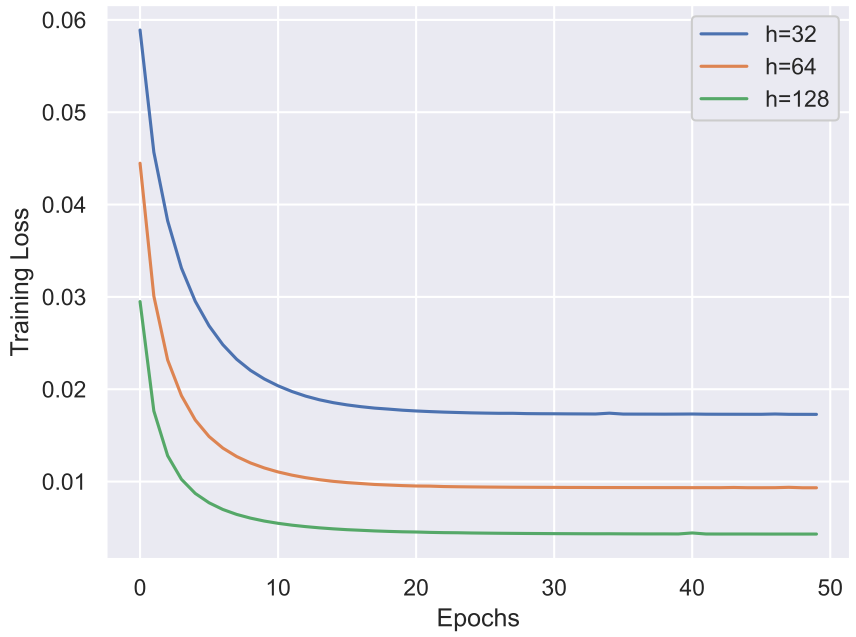
c. *[5 points]* One reason is simply the difference between the populations. Training a machine learning model on New York and applying it to Beijing will clearly have inconsistencies due to their possible differences in causes of crime. Applying a model trained on one dataset can be confidently applied to another dataset only if the training data was representative of the entire population. Another reason is over/underfitting. Overfitting the model on the training data will give wildly different accuracy on test data. The model must be properly generalized, which may be extremely difficult without an idea of the actual population.

Unsupervised Learning with Autoencoders

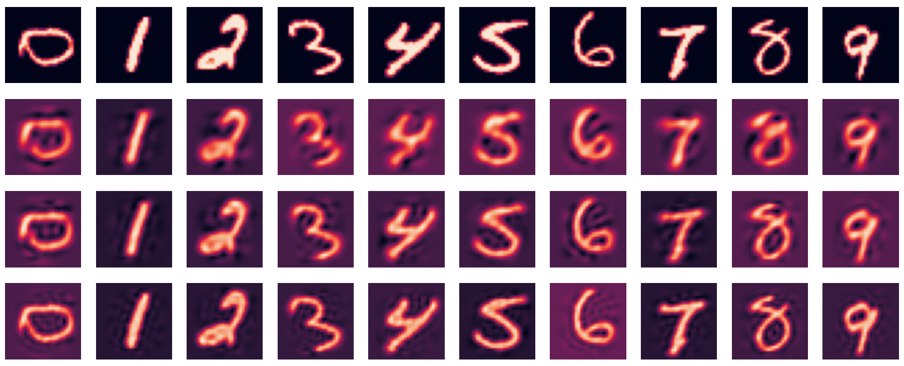
A.3

a. *[10 points]* After training on 50 epochs using minibatches of 1000 samples the final losses for each h is:

$$h_{32} = 0.0173$$
$$h_{64} = 0.0093$$
$$h_{128} = 0.0043$$

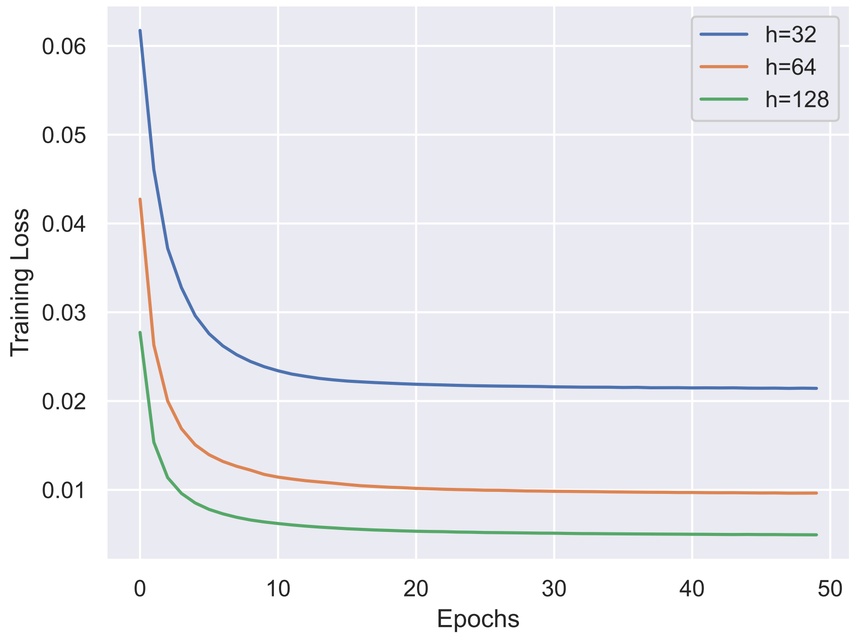


Reconstructions are shown below for each digit. The first row is the original and the next three rows are for $h = 32$, $h = 64$, and $h = 128$ respectively.

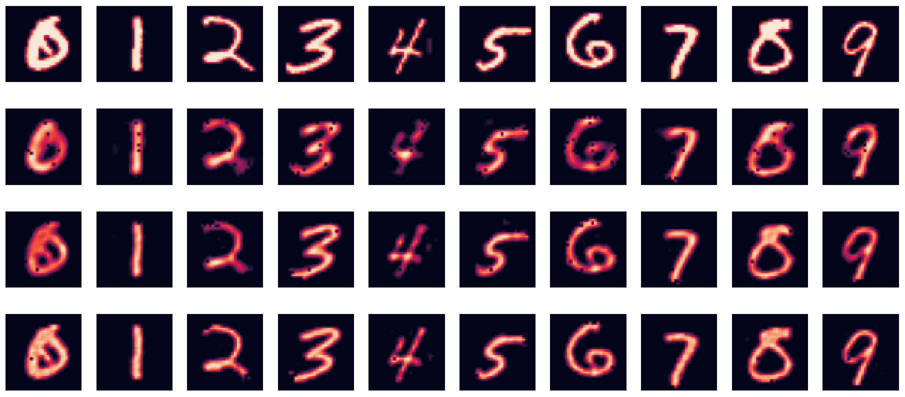


b. *[10 points]* torch.nn.ReLU() was used for the nonlinear activation functions. The final losses were:

$$h_{32} = 0.0214$$
$$h_{64} = 0.0096$$
$$h_{128} = 0.0049$$

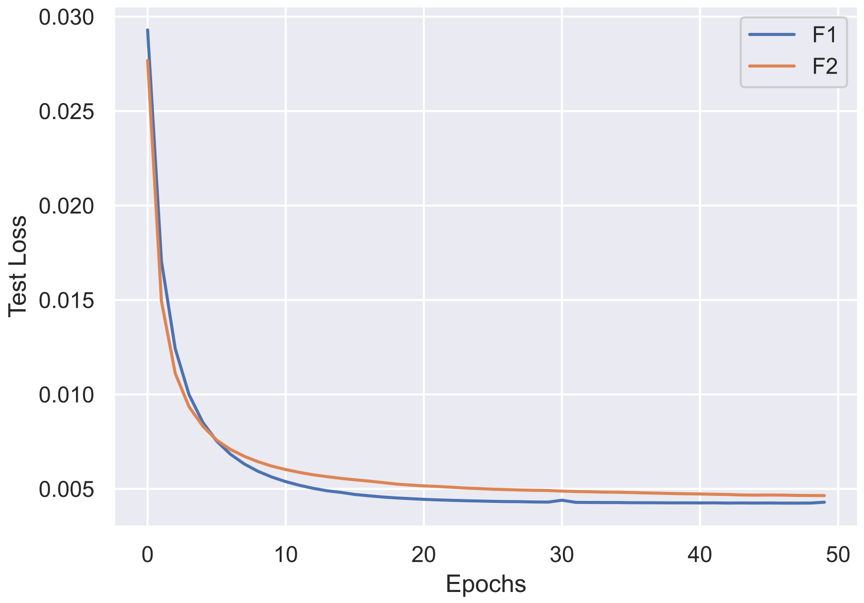


Reconstructions are shown below for each digit. The first row is the original and the next three rows are for $h = 32$, $h = 64$, and $h = 128$ respectively.



c. *[5 points]* The test errors were:

Model	Test Loss
$\mathcal{F}_1$	0.0043
$\mathcal{F}_2$	0.0046



d. *[5 points]* The train and test losses when performing PCA with 128 dimensions were 0.0043 and 0.0042 respectively. Simply looking at error all three methods seem comparable. However, looking at the reconstructions themselves, the PCA reconstruction is very similar to the $\mathcal{F}_1$ reconstruction due to their linearity. The reconstruction of $\mathcal{F}_2$ does a much better job distinguishing the background from the digit thanks to its nonlinearity.

Source Code A.3

```python
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
import torch
import seaborn as sns

sns.set()
# Load data
mndata = MNIST('../MNIST/data/')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())

# Rearrange data
X_train = X_train/255.0
X_train = torch.from_numpy(X_train)
X_train = X_train.type(torch.FloatTensor)
X_test = X_test/255.0
X_test = torch.from_numpy(X_test)
X_test = X_test.type(torch.FloatTensor)
```

```
Y_train = np.zeros((len(X_train), 10))
Y_train[range(0, len(X_train)), labels_train] = 1
Y_train = torch.from_numpy(Y_train)
Y_train = Y_train.type(torch.FloatTensor)
Y_test = np.zeros((len(X_test), 10))
Y_test[range(0, len(X_test)), labels_test] = 1
Y_test = torch.from_numpy(Y_test)
Y_test = Y_test.type(torch.FloatTensor)

# Initialize variables
d = 784
lr = 1e-3
epochs = 50
batch = 1000
loops = int(len(X_train)/batch)
H = [32, 64, 128]
TrainLoss = np.zeros((len(H), epochs))
TestLoss = np.zeros((len(H), epochs))

plotind = np.zeros(10)
for digit in range(10):         # indices of example digits, plot originals
    ind = np.nonzero(labels_train == digit)
    ind = np.squeeze(np.asarray(ind))
    plotind[digit] = np.random.choice(ind)
    x = torch.reshape(X_train[int(plotind[digit]), :], (28, 28))
    plt.subplot(len(H) + 1, 10, digit + 1)
    plt.imshow(x)
    plt.axis('off')

# Train autoencoder
for h in range(len(H)):
    # Define model
    model = torch.nn.Sequential(
        torch.nn.Linear(d, H[h]),
        torch.nn.ReLU(),
        torch.nn.Linear(H[h], d),
        torch.nn.ReLU()
    )
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    count = 0
    acc = 0
    #while acc < 0.9:
    for e in range(epochs):
        print(e)
        # mini-batch
        perm = torch.randperm(len(X_train))
        for b in range(loops):
            ind = perm[b * batch:((b + 1) * batch)]
            tempx = X_train[ind, :]
            x_hat = model(tempx)

            # cross entropy combines softmax calculation with NLLLoss
            loss = torch.nn.functional.mse_loss(x_hat, tempx)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        # Save losses
        x_hat = model(X_train)
        loss = torch.nn.functional.mse_loss(x_hat, X_train)
        TrainLoss[h, e] = loss.item()
        print(loss.item())

        x_hat = model(X_test)
        loss = torch.nn.functional.mse_loss(x_hat, X_test)
        TestLoss[h, e] = loss.item()

    # Plot reconstruction
    x_hat = model(X_train)
    for digit in range(10):
        x = torch.reshape(x_hat[int(plotind[digit]), :], (28, 28))
        plt.subplot(len(H)+1, 10, digit + 1 + 10*(h+1))
        plt.imshow(x.data.numpy())
        plt.axis('off')

# Plot losses
plt.figure()
for i in range(len(H)):
    plt.plot(TrainLoss[i, :])
plt.ylabel('Training Loss')
plt.xlabel('Epochs')
plt.legend(('h=32', 'h=64', 'h=128'))

plt.figure()
plt.plot(TestLoss[0, :])
plt.ylabel('Test Loss')
plt.xlabel('Epochs')
plt.legend(('F1', 'F2'))
```

Text classification on SST-2

A.6

Note: I made one adjustment to `train.py` in line 7: `from hw4_a3` → `from hw4_a6`.

a. *[10 points]* See code below.

b. *[15 points]* See code below.

c. *[5 points]* See code below.

d. *[15 points]* Using batch size of 32, training rate of 1e-4, and 16 epochs we have:

	Validation Loss	**Validation Accuracy**	**Training Loss**	**Training Accuracy**
RNN	0.7568	0.7661	0.1556	0.9480
LSTM	0.7312	0.8096	0.1289	0.9551
GRU	0.6474	0.8073	0.1283	0.9552

e. *[5 points]* Since we want to predict the tag of every word in a sequence, we would supposedly have labels for each word. The last fully connected layer only had an output of 1 node for our purposes of predicting positive/negative review, but will need to be changed to have an output of `embedding_dim` nodes that corresponds to each word's tag.

f. *[1 points]* "be tried as a war criminal" - still not sure how this could be a legitimate review even taken out of context.

Source Code hw4_a6.py

```
import torch
import torch.nn as nn


def collate_fn(batch):

    sentences, labels = zip(*batch)

    N = len(sentences)
    max_sequence_length = max([len(s) for s in sentences])
    PaddedBatch = torch.zeros((N, max_sequence_length), dtype=torch.int32)
    Labels = torch.zeros((N, 1), dtype=torch.int32)
    for s in range(N):
        n = len(sentences[s])
        PaddedBatch[s, 0:n] = sentences[s]
        Labels[s, 0] = labels[s]

    return (PaddedBatch, Labels)

class RNNBinaryClassificationModel(nn.Module):
    def __init__(self, embedding_matrix):
        super().__init__()

        vocab_size = embedding_matrix.shape[0]
        embedding_dim = embedding_matrix.shape[1]

        # Construct embedding layer and initialize with given embedding matrix.
        # Do not modify this code.
        self.embedding = nn.Embedding(num_embeddings=vocab_size,\
            embedding_dim=embedding_dim, padding_idx=0)
        self.embedding.weight.data = embedding_matrix

        # Variables
        input_size = embedding_dim
        hidden_size = 64

        # RNN
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)

        # LSTM
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

        # GRU
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)

        # Fully connected layer
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, inputs):

        inputs = inputs.type(torch.LongTensor)  # embedding requires long
        embeds = self.embedding(inputs)

        # # RNN
        # output, hn = self.rnn(embeds)

        # # LSTM
        # output, (hn, cn) = self.lstm(embeds)

        # RNN
        output, hn = self.gru(embeds)

        prediction = self.fc(hn.squeeze(0))

        return prediction

    def loss(self, logits, targets):

        binary = torch.sigmoid(logits)
        targets = targets.type(torch.FloatTensor)  # cross entropy requires float
        loss = nn.functional.binary_cross_entropy(binary, targets)

        return loss

    def accuracy(self, logits, targets):

        binary = torch.sigmoid(logits)
        prediction = torch.round(binary)

        accuracy = sum(prediction == targets).type(torch.FloatTensor)/len(targets)

        return accuracy
```

```
# Training parameters
TRAINING_BATCH_SIZE = 32
NUM_EPOCHS = 16
LEARNING_RATE = 1e-4

# Batch size for validation, this only affects performance.
VAL_BATCH_SIZE = 128
```