Homework #4

CSE 446: Machine Learning
Eric Boris: 1976637

Conceptual Questions

Problem 1

a. **True**. There are d columns in X and $k \leq d$, there are enough columns to project onto a k dimensional subspace using PCA.

b. **False**. The columns of V are equal to the eigenvalues of $X^T X$.

c. **False**. This would create meaningless clusters because it will result in single data point clusters with zero distance from the center of each cluster.

d. **False**. Consider that $uIu^T = I$ and $vIv^T = I$ are distinct singular values of the identity matrix I where u and v are distinct orthogonal matricies.

e. **False**. Consider that an $n \times n$ matrix has exactly one eigenvalue.

f. **True**. Because the autoencoder has non-linear activation functions it can capture more variance than PCA's projection of the data onto a linear subspace.

Basics of SVD

Problem 2

a. Use the SVD of X to show that $\widehat{w} > \widehat{w}_R$.

$$
\begin{aligned}
\widehat{w} &> \widehat{w}_R \\
\min_w \|Xw - y\|_2^2 &> \min_w \|Xw - y\|_2^2 + \lambda \|w\|_2^2 \\
\left(X^T X\right)^{-1} X^T y &> \left(X^T X + \lambda I\right)^{-1} X^T y \\
\left(X^T X\right)^{-1} &> \left(X^T X + \lambda I\right)^{-1} \\
\left(V \Sigma^T U^T U \Sigma V^T\right)^{-1} &> \left(V \Sigma^T U^T U \Sigma V^T + \lambda I\right)^{-1} && \text{Use SVD} \\
V D^{-1} V^T &> V \left(D + \lambda I\right)^{-1} V^T && \text{V is unitary and let } \Sigma^2 = D \text{ s.t. } D \in \mathbb{R}_+^d
\end{aligned}
$$

Thus, since $\lambda > 0$, $\widehat{w} > \widehat{w}_R$.

b. Show that $UU^T = U^T U = I$.
U is a square matrix so by the spectral theorem it can by unitarily diagnalized. Let $U = TIV^T$ such that $T, I, V \in \mathbb{R}^{n \times n}$ where T, V have singular values equal to 1 and I is the identity matrix. Because T and V are unitary,

$$TT^T = T^T T = VV^T = V^T V = I$$

Therefore

$$UU^T = TIV^T VIT^T = TIIT^T = TT^T = I$$

and

$$U^T U = VIT^T TIV^T = IT^T TI = T^T T = I$$

Thus

$$UU^T = U^T U = I$$

c. Show that $\|Ux\|_2 = \|x\|_2$ for any $x \in \mathbb{R}^n$.
Let $x \in \mathbb{R}^n$. Then

$$\|Ux\|_2^2 = (Ux)^T (Ux) = x^T U^T U x = x^T x = \|x\|_2^2$$

Taking the square root, it follows that

$$\|Ux\|_2 = \|x\|_2$$

K-Means Clustering

Problem 3: Answers

a. See Problem 3: Code

b. See Figures 1 and 2

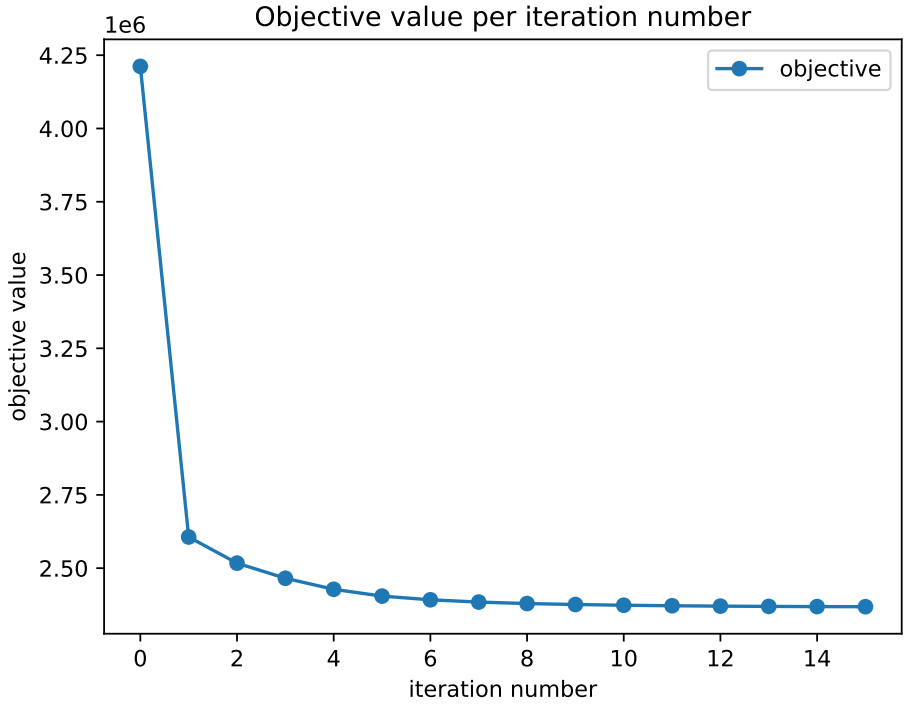


Figure 1

c. See Figure 3

Problem 3: Code

```
# HW4 Problem 3 - K-Means Clustering

from mnist import MNIST
import numpy as np
import matplotlib.pyplot as plt
```

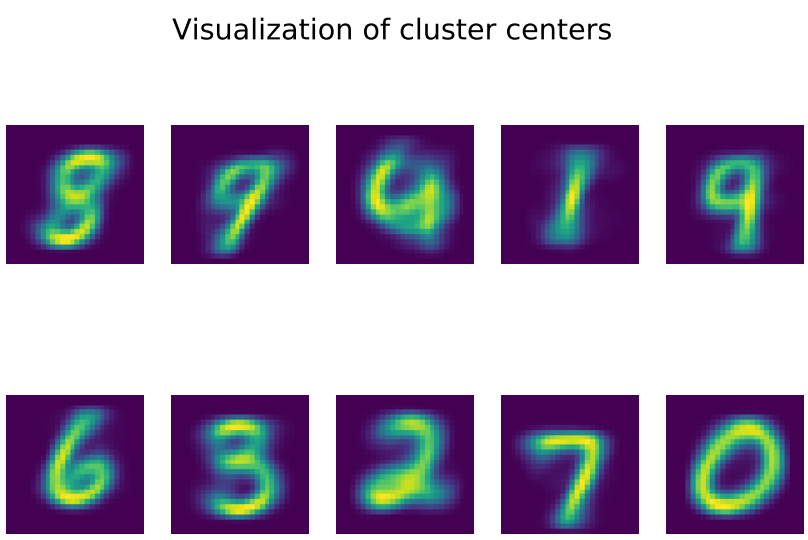


Figure 2

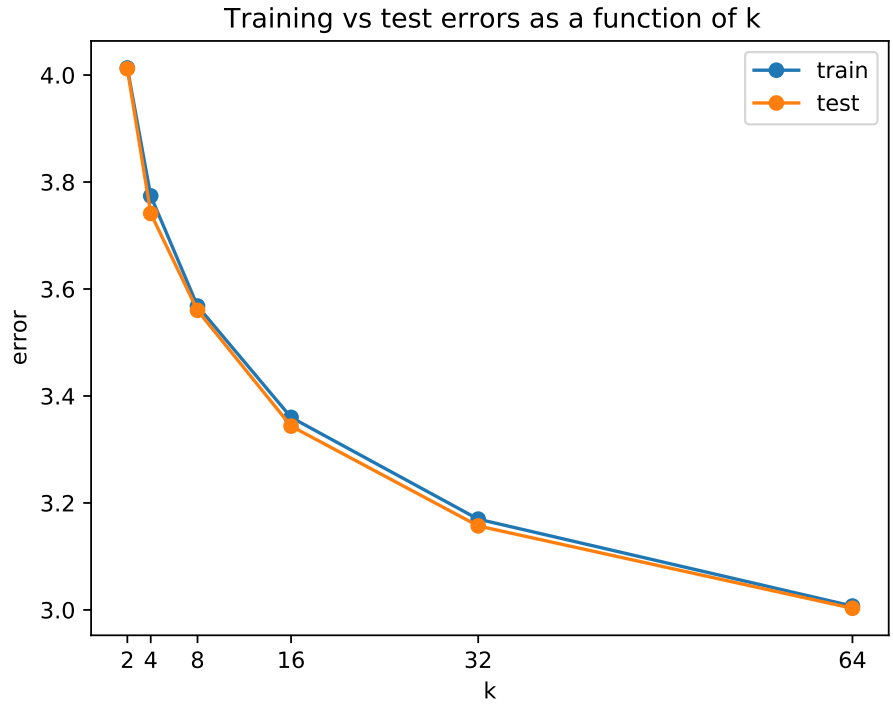


Figure 3

```
6  import pickle
7
8  def load_dataset(path):
9      ''' Load the mnist handwritten digits dataset. '''
10     mndata = MNIST(path)
```

```
    X_train, y_train = map(np.array, mndata.load_training())
    X_test, y_test = map(np.array, mndata.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    return X_train, y_train, X_test, y_test

def save(output, path):
    ''' Save the output to the file path. '''
    with open(path, 'wb') as f:
        pickle.dump(output, f)

def plot(*series, title, x_label, y_label, file_path, x_ticks=None):
    ''' Plot the data. '''
    plt.title(title)
    for x, y, label in series:
        if not x:
            plt.plot(y, label=label, marker='o')
        else:
            plt.plot(x, y, label=label, marker='o')
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    if x_ticks:
        plt.xticks(x_ticks)
    plt.legend()
    plt.savefig(file_path)
    plt.show()

def visualize(title, centers, file_path, fig_size=(8, 5), dim=(28, 28), font_size=16):
    ''' Visualize the cluster centers. '''
    fig, ax = plt.subplots(2, 5, figsize=fig_size)
    for j in range(len(centers)):
        ax.ravel()[j].imshow(centers[j, :].reshape(dim))
        ax.ravel()[j].axis('off')
    fig.suptitle(title, fontsize=font_size)
    plt.savefig(file_path)
    plt.show()

class K_Means:
    def train(self, X, k, conv_distance=0.01, verbose=False):
        '''
        Use Lloyds algorithm to compute K-Means clusters over the data X and
        return the resultant centers and each iteration's measured objective values.

        Arguments:
            X -- Numpy array of data
            k -- Integer of clusters to compute
            conv_distance -- Float minimum distance for determining convergence
            verbose -- Boolean display progress output if true
        Returns:
            centers -- k length array of computed center values
            objectives -- Array of objective values per iteration
        '''
        if verbose:
            print(f'Training with {k} clusters')

        self.centers = X[np.random.choice(np.arange(len(X)), size=k, replace=False)]

        # Only used for plotting results.
        objectives = []

        converged = False
        while not converged:
            # Compute the point distribution
            point_dist, objective = self._get_point_distribution(X)

            # Move each of the k centers closer to the center of mass.
            prev_centers = self.centers
            self.centers = np.array([np.average(X[i], axis=0) for i in point_dist])

```

```
            # Check whether the centers have converged.
            max_distance = np.max(np.sum((prev_centers - self.centers) ** 2, axis=1))
            converged = max_distance < conv_distance

            if verbose:
                print(max_distance)

            objectives.append(objective)

        return self.centers, objectives

    def _get_point_distribution(self, X):
        '''
        Return a distribution of points from X such that for each point i in X
        i is assigned to the closest of the k centers.

        Arguments:
            X -- Numpy array of data
        Returns:
            point_dist -- Nested list; distribution of points from X onto each of k points
            objective -- Float; the measured objective for this iteration
        '''
        objective = 0
        k = len(self.centers)

        point_dist = [[] for _ in range(k)]

        # Compute the sum of euclidean distances between two arrays.
        distance = lambda x, y: np.sum((x - y) ** 2)

        # Find which center j each point i is closest to.
        for i in range(len(X)):
            distances = [distance(X[i], self.centers[j]) for j in range(k)]
            j = np.argmin(distances)
            point_dist[j].append(i)
            objective += distances[j]

        return point_dist, objective

    def error(self, X):
        ''' Compute the error of the trained model on the dataset X. '''
        objective = 0

        for i in X:
            min_distance = float('inf')
            for j in self.centers:
                distance = np.linalg.norm((i - j) ** 2)
                min_distance = min(distance, min_distance)

            objective += min_distance

        return objective / len(X)

if __name__ == '__main__':
    print('Loading data')
    X_train, y_train, X_test, y_test = load_dataset('../data/python-mnist/data/')

    # Run the algorithm on the MNIST training dataset with k = 10.
    model = K_Means()
    centers, objectives = model.train(X_train, k=10, verbose=True)

    # Save the results.
    save(centers, path='../data/a3_part_b_centers.pickle')
    save(objectives, path='../data/a3_part_b_objectives.pickle')

    # Plot the objective as a function of the iteration number.
    plot((None, objectives, 'objective'),
        title='Objective value per iteration number',
        x_label='iteration number',
```

```
        y_label='objective value',
        file_path='../figures/a3_plot_b.pdf')

    # Visualize the cluster centers as a 28 x 28 image.
    visualize(title='Visualization of cluster centers',
        centers=centers,
        file_path='../figures/a3_clusters.pdf')

    # For k = {2, 4, 8, 16, 32, 64} run the algorithm on the training dataset to obtain centers.
    k_vals = [2, 4, 8, 16, 32, 64]

    train_errors = []
    test_errors = []

    for k in k_vals:
        model = K_Means()
        centers, objectives = model.train(X_train, k, verbose=True)
        train_errors.append(model.error(X_train))
        test_errors.append(model.error(X_test))

    # Save the results.
    save(train_errors, path='../data/a3_part_c_train_errors.pickle')
    save(test_errors, path='../data/a3_part_c_test_errors.pickle')

    # Plot the training and test error as a function of k.
    plot((k_vals, train_errors, 'train'),
        (k_vals, test_errors, 'test'),
        title='Training vs test errors as a function of k',
        x_label='k',
        y_label='error',
        x_ticks=k_vals,
        file_path='../figures/a3_plot_c.pdf')
```

PCA

Problem 4: Answers

a. λ_1: 5.148333441485338
λ_2: 3.7299894906022617
λ_{10}: 1.250010757212031
λ_{30}: 0.36492647617793883
λ_{50}: 0.16962131566853172
$\sum_{i=1}^{d} \lambda_i$: 52.83384400094484

b. Show a general formula for the rank-k PCA approximation of x.

$$
\begin{aligned}
\Sigma &= \frac{1}{n}\left(X_{\text{train}} - \mathbf{1}\mu^T\right)^T\left(X_{\text{train}} - \mathbf{1}\mu^T\right) \\
&= \frac{1}{n}\left(USV^T\right)^T\left(USV^T\right) && \text{Substitute SVD of } X_{\text{train}} - \mathbf{1}\mu^T \\
&= \frac{1}{n}VSU^TUS^TV^T \\
&= \frac{1}{n}VS^2V^T \\
&= \frac{1}{n}\left(X_{\text{train}} - \mathbf{1}\mu^T\right)\left(X_{\text{train}} - \mathbf{1}\mu^T\right)V \\
&= \frac{1}{n}VS^2 \\
&= \frac{1}{n}VD
\end{aligned}
$$

Now consider that since S is a diagonal matrix,

$$S^2 = \begin{bmatrix} s_1^2 & 0 & 0 & ... \\ 0 & s_2^2 & ... & 0 \\ 0 & 0 & ... & ... \\ 0 & 0 & ... & 0 \end{bmatrix}$$

Which implies that

$$\frac{1}{n}D = \begin{bmatrix} \frac{1}{n}d_1 & 0 & 0 & ... \\ 0 & \frac{1}{n}d_2 & ... & 0 \\ 0 & 0 & ... & ... \\ 0 & 0 & ... & 0 \end{bmatrix}$$

Therefore, let $\lambda_i = \frac{1}{n}d_i$ where λ_i is the i-th eigenvalue of V

$$\begin{aligned} \Sigma V &= \frac{1}{n}VD \\ &= V \begin{bmatrix} \lambda_1 & 0 & 0 & ... \\ 0 & \lambda_2 & ... & 0 \\ 0 & 0 & ... & ... \\ 0 & 0 & ... & 0 \end{bmatrix} \\ \Sigma v_i &= \lambda_i v_i \end{aligned}$$

So $\left(X_{\text{train}} - \mathbf{1}\mu^T\right)$ where $x_i \in \mathbb{R}$ can be projected onto k dimensions by using the first k eigenvectors of V to compute the minimum reconstruction error. Thus, a general formula for the rank-k PCA approximation for x can be written

$$\bar{x} = V_k V_k^T \left(x_i - \mu^T\right) + \mu$$

c. See Figures 4 and 5

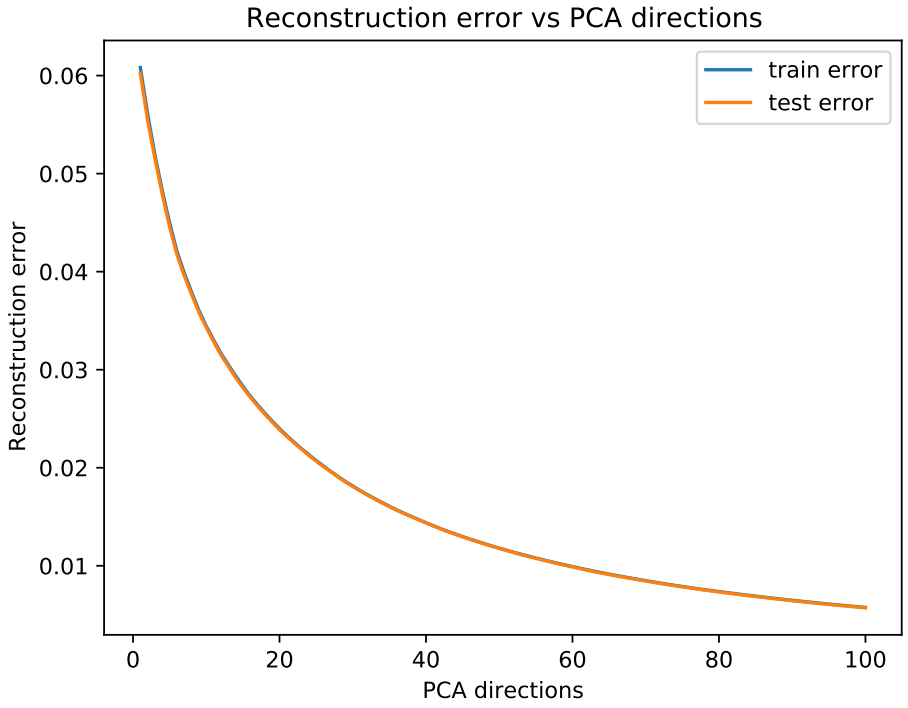


Figure 4

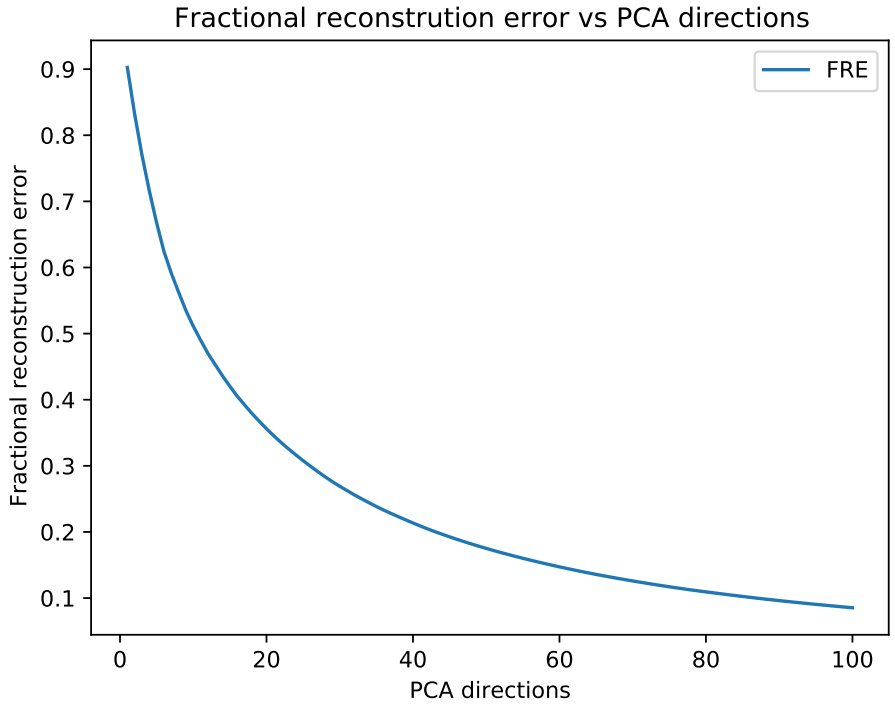


Figure 5

d. See Figure 6. These represent the first 10 eigenvectors in the single value decomposition of the sample covariance of training examples. They correspond to the most significant entries in the training dataset and so contain overlapping, repeated, and missing digits.

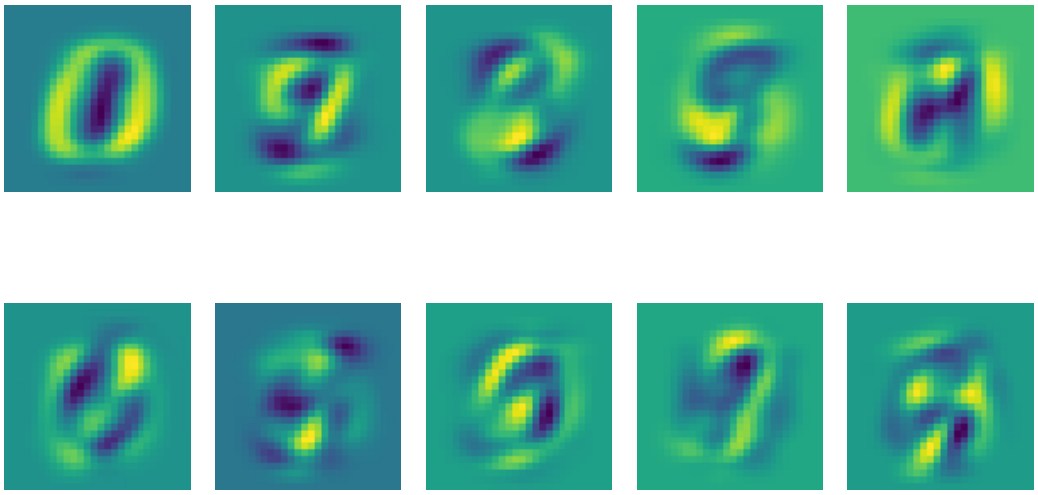
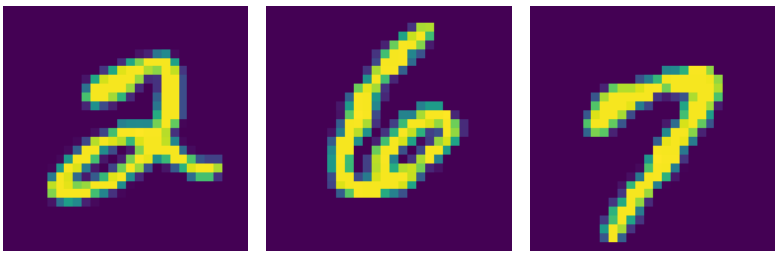


Figure 6

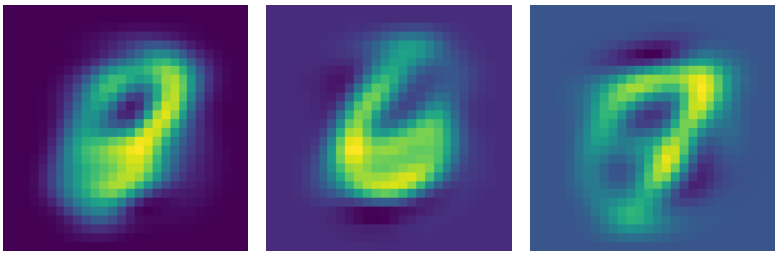
e. See Figure 7. Increasing k improves the fidelity of the reconstruction.

Problem 4: Code

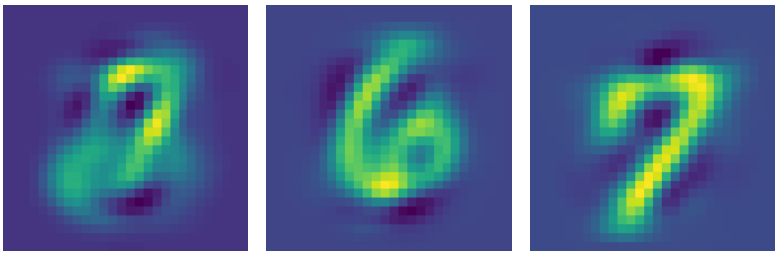
Figure 7



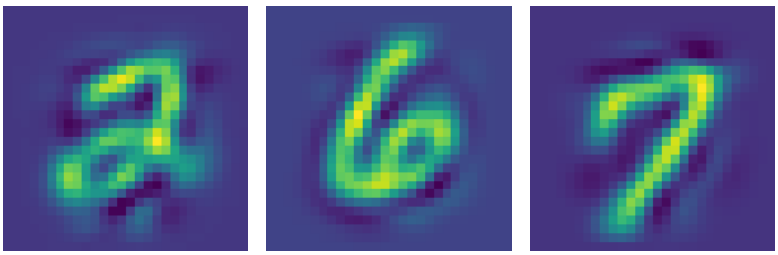
Actual



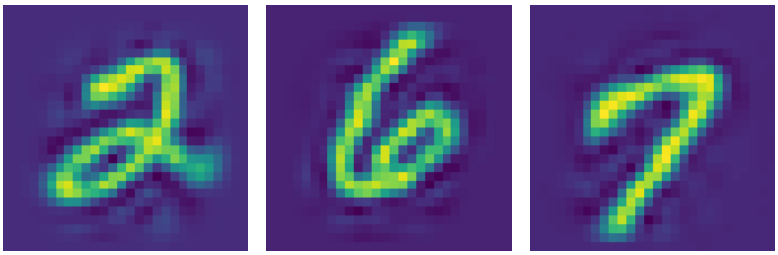
k=5



k=15



k=40



k=100

```
# HW4 Problem 4 - PCA

from mnist import MNIST
```

```
4  import matplotlib.pyplot as plt
5  import numpy as np
6  from tqdm import tqdm
7
8  TQDM_FORM = '{l_bar}{bar:10}{r_bar}{bar:-10b}'
9
10 def load_dataset(path):
11     ''' Load the mnist handwritten digits dataset. '''
12     mndata = MNIST(path)
13     X_train, y_train = map(np.array, mndata.load_training())
14     X_test, y_test = map(np.array, mndata.load_testing())
15     X_train = X_train / 255.0
16     X_test = X_test / 255.0
17     return X_train, y_train, X_test, y_test
18
19 def split(arr, index):
20     ''' Split numpy array into separate sets on the given index. '''
21     return arr[:index, :], arr[index:, :]
22
23 def reconstruct(X, V, mu, k):
24     '''
25     Return the PCA reconstruction from the principle components.
26
27     Arguments:
28         X -- Numpy array of data
29         V -- Numpy array of components
30         mu -- Float of the data mean
31         k -- Integer of top components to use
32     Returns:
33         Numpy array of reconstructed data
34     '''
35     # Include only top k components.
36     V = V[:k + 1].T
37     pc_scores = np.matmul(X - mu, V)
38     eigenvectors = np.matmul(pc_scores, V.T)
39     return eigenvectors + mu
40
41 def error(X, V, mu, k):
42     '''
43     Return the mean-squared reconstruction errors of the data over range k.
44
45     Arguments:
46         X -- Numpy array of data
47         V -- Numpy array of components
48         mu -- Float of the data mean
49         k -- Integer of top components to use
50     '''
51     error = []
52     for i in tqdm(range(k), bar_format=TQDM_FORM):
53         X_hat = reconstruct(X, V, mu, i)
54         error.append(np.square(X - X_hat).mean())
55     return error
56
57 def fractional_reconstruction_error(eigenvalues, k):
58     ''' Return a k length array of fractional reconstruction error of the eigenvalues. '''
59     fre = []
60     for i in tqdm(range(k), bar_format=TQDM_FORM):
61         fre.append(1. - np.sum(eigenvalues[:(i + 1)]) / np.sum(eigenvalues))
62     return fre
63
64 def matching_index(X, digit):
65     ''' Find the first index in the data where the image matches the digit. '''
66     for i in tqdm(range(len(X)), bar_format=TQDM_FORM):
67         if X[i] == digit:
68             break
69     return i
70
71 def plot(*series, title, x_label, y_label, save_path):
72     ''' Plot the data. '''
```

```
    plt.title(title)
    for x, y, label in series:
        plt.plot(x, y, label=label)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend()
    plt.savefig(save_path)
    plt.show()

def visualize(data, iterable, save_path=None, n_rows=2, n_cols=5, dim=(28, 28)):
    ''' Visualize the data. '''
    # Minimize the margins
    plt.gca().set_axis_off()
    plt.subplots_adjust(top=1, bottom=0, right=1, left=0, hspace=0, wspace=0)
    plt.margins(0, 0)
    plt.gca().xaxis.set_major_locator(plt.NullLocator())
    plt.gca().yaxis.set_major_locator(plt.NullLocator())

    for i, j in enumerate(iterable):
        plt.subplot(n_rows, n_cols, i + 1)
        imgplot = plt.imshow(data[j].reshape(dim))
        plt.axis('off')
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, bbox_inches='tight', pad_inches=0)
    plt.show()

if __name__ == '__main__':
    print('Load data')
    X_train, y_train, X_test, y_test = load_dataset('../data/python-mnist/data/')
    X_train, X_test = split(X_train, index=50000)

    mu = np.mean(X_train, axis=0)

    print('Get eigenvalues')
    U, S, V = np.linalg.svd(X_train - mu, False)
    n = X_train.shape[0]
    eigenvalues = S ** 2 / n

    print('\nPart a')
    print('What are the eigenvalues 1, 2, 10, 30, and 50?')
    for i in [0, 1, 9, 29, 49]:
        print(f'lambda{i + 1}: {eigenvalues[i]}')

    print('What is the sum of eigenvalues?')
    print(f'Sum of eigenvalues: {np.sum(eigenvalues)}')

    print('\nPart c')
    print('Compute reconstruction error')
    train_error = error(X_train, V, mu, k=100)
    test_error = error(X_test, V, mu, k=100)

    print('Plot reconstruction error')
    plot((range(1, 101), train_error, 'train error'),
        (range(1, 101), test_error, 'test error'),
        title='Reconstruction error vs PCA directions',
        x_label='PCA directions',
        y_label='Reconstruction error',
        save_path='../figures/a4_re.pdf')

    print('Compute fractional reconstruction error')
    fre = fractional_reconstruction_error(eigenvalues, k=100)

    print('Plot fractional reconstruction error')
    plot((range(1, 101), fre, 'FRE'),
        title='Fractional reconstrution error vs PCA directions',
        x_label='PCA directions',
        y_label='Fractional reconstruction error',
        save_path='../figures/a4_fre.pdf')
```

```
142
143     print('\nPart d')
144     print('Display the first 10 eigenvectors')
145     visualize(V, range(0, 10),
146         save_path='../figures/a4_eigenvectors.pdf')
147
148     print('\nPart e')
149     print('Show reconstructions for digits 2, 6, 7 with values k = 5, 15, 40, 100')
150     indices = []
151     for digit in [2, 6, 7]:
152         indices.append(matching_index(y_train, digit))
153
154     # Display the actual digits.
155     visualize(X_train, indices,
156         save_path=f'../figures/a4_actual.pdf',
157         n_rows=1,
158         n_cols=3)
159
160     # Display the reconstructions for different k.
161     for k in [5, 15, 40, 100]:
162         reconstruction = reconstruct(X_train, V, mu, k)
163         visualize(reconstruction, indices,
164             save_path=f'../figures/a4_recon_{k}.pdf',
165             n_rows=1,
166             n_cols=3)
167
168     # Problem 5 Part d.
169     # Re-run PCA with different k to compare results against AutoEncoder.
170
171     # Indices for digits 0-9.
172     digit_indices = [1, 14, 16, 12, 9, 11, 13, 15, 17, 4]
173
174     # Display the actual digits.
175     visualize(X_train, digit_indices,
176         save_path=f'../figures/a5_d_actual.pdf',
177         n_rows=1,
178         n_cols=10)
179
180     # Display the reconstructions for different k.
181     for k in [32, 64, 128]:
182         reconstruction = reconstruct(X_train, V, mu, k)
183         visualize(reconstruction, digit_indices,
184             save_path=f'../figures/a5_d_recon_{k}.pdf',
185             n_rows=1,
186             n_cols=10)
```

Unsupervised Learning with Autoencoders

Problem 5: Answers

a. See Figure 8

b. See Figure 9

c.

Test Errors on Linear and Non-linear networks

	h=32	h=64	h=128
Linear	1096.89	590.39	275.92
Non-linear	958.09	545.69	315.90

d. See Figure 10. Because of hidden layers and non-linearities, the Autoencoder is better able to capture digit features than the PCA model. Compare and contrast how increasing the number of hidden layers improves the fidelity of the reconstructed digits faster than increasing k in PCA. Further, the Autoencoder

Figure 8: Linear network digit reconstructions

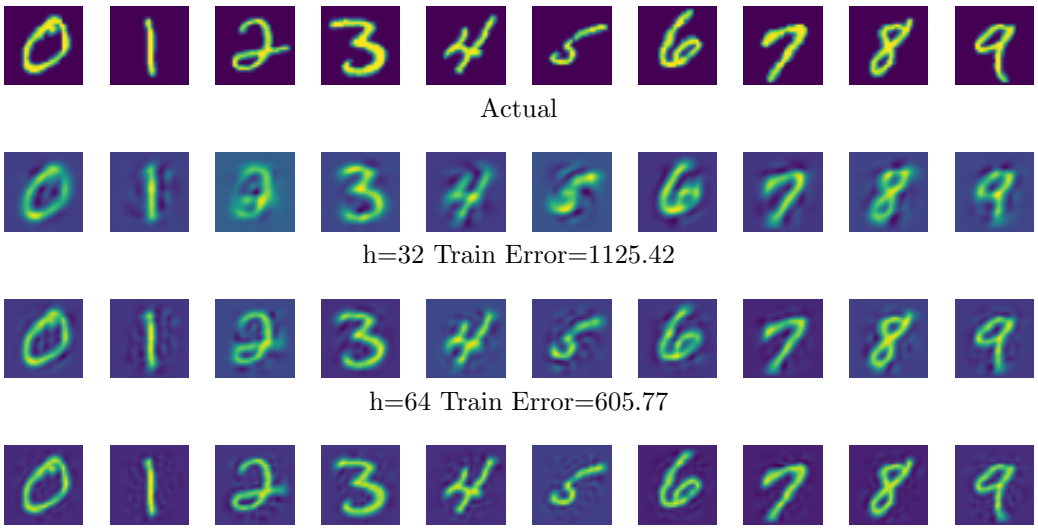
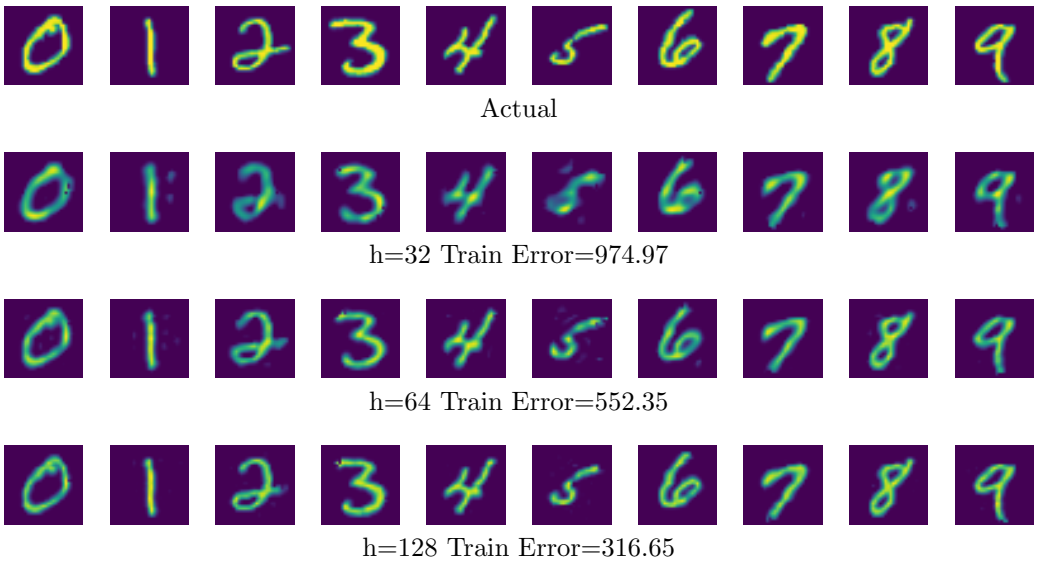


Figure 9: Non-linear network digit reconstructions



uses activation functions for better non-linear boundaries given the non-linear model superior performance over PCA for all values of k/h.

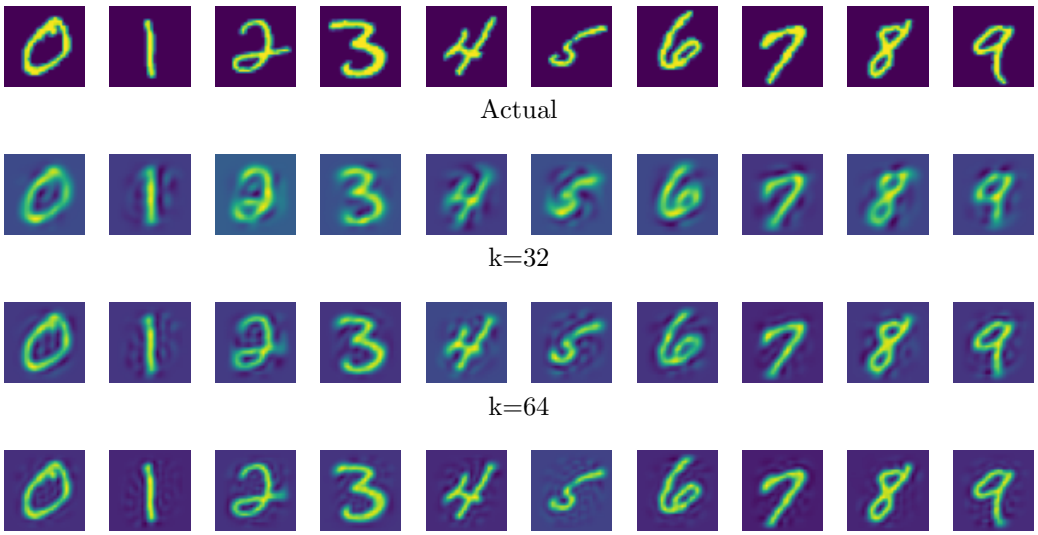
Problem 5: Code

```
# HW4 Problem 5 - AutoEncoders

import torch
import torch.nn as nn
import torchvision.datasets as datasets
from torchvision.datasets import MNIST
import matplotlib.pyplot as plt
from tqdm import tqdm

def load_dataset():
```

Figure 10: PCA digit reconstructions




```
    ''' Load the mnist handwritten digits dataset. '''
    train = datasets.MNIST(root='./', train=True, download=True, transform=None)
    test = datasets.MNIST(root='./', train=False, download=True, transform=None)
    X_train = train.data.view(-1, 784).float()
    y_train = train.targets
    X_test = test.data.view(-1, 784).float()
    y_test = test.targets
    return X_train, y_train, X_test, y_test

def visualize(data, iterable, save_path=None, n_rows=1, n_cols=10, dim=(28, 28)):
    ''' Visualize the data. '''
    # Minimize the margins
    plt.gca().set_axis_off()
    plt.subplots_adjust(top=1, bottom=0, right=1, left=0, hspace=0, wspace=0)
    plt.margins(0, 0)
    plt.gca().xaxis.set_major_locator(plt.NullLocator())
    plt.gca().yaxis.set_major_locator(plt.NullLocator())

    for i, j in enumerate(iterable):
        plt.subplot(n_rows, n_cols, i + 1)
        imgplot = plt.imshow(data[j].reshape(dim))
        plt.axis('off')
    plt.tight_layout()
    if save_path:
        plt.savefig(save_path, bbox_inches='tight', pad_inches=0)
    plt.show()

def save(output, path):
    ''' Save the output to the file path. '''
    with open(path, 'w') as f:
        f.write(output)

class AutoEncoder:
    def __init__(self, d, h, is_linear=True):
        self.h = h
        self.model = self._get_model(d, h, is_linear)
        self.loss_fn = nn.MSELoss()

    def _get_model(self, d, h, is_linear):
        ''' Return a linear or non-linear model with the dimensions d and h. '''
        if is_linear:
            return torch.nn.Sequential(nn.Linear(d, h), nn.Linear(h, d))
```

```
        else:
            return torch.nn.Sequential(nn.Linear(d, h), nn.ReLU(), nn.Linear(h, d), nn.ReLU())

    def train(self, X, n_epochs, learning_rate=1E-3, verbose=False):
        ''' Train the model and return a list of training losses. '''
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=learning_rate)
        losses = []
        for i in tqdm(range(n_epochs)):
            loss = self.get_loss(X)
            losses.append(loss.item())
            if verbose:
                print(f'iter: {i+1}\tloss: {loss.item()}')
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
        return losses

    def get_loss(self, X):
        ''' Return the model loss against the given data. '''
        X_hat = self.model(X)
        return self.loss_fn(X_hat, X)

if __name__ == '__main__':
    # Model parameters
    d = 784
    h_vals = [32, 64, 128]
    n_epochs = 2000

    # Indices for digits 0-9.
    digit_indices = [1, 14, 16, 12, 9, 11, 13, 15, 17, 4]

    # Determine training device; GPU or CPU.
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    # Load the datasets.
    X_train, y_train, X_test, y_test_target = load_dataset()

    # Save the losses.
    train_losses = ''
    test_losses = ''

    # Build linear and non-linear models.
    for is_linear in (True, False):
        # Build models with d x h dimensions.
        models = [AutoEncoder(d, h, is_linear) for h in h_vals]

        # Display the actual digits.
        visualize(X_train, digit_indices, save_path=f'../figures/a5_{is_linear}_actual.pdf')

        for i, m in enumerate(models):
            print(f'h = {h_vals[i]}')

            # Train the model.
            losses = m.train(X_train, n_epochs, verbose=False)

            # For parts a and b.
            # Display each reconstructed digit.
            with torch.no_grad():
                visualize(m.model(X_train), digit_indices, save_path=f'../figures/a5_{is_linear}_{
                    h_vals[i]}.pdf')

            # For part c.
            # Measure the model loss over the test data.
            test_loss = m.get_loss(X_test)

            # Display the losses.
            print(f'Train loss: {losses[-1]}')
            print(f'Test loss: {test_loss.item()}')
```

```
            # Update the loss measures.
            train_losses += f'{is_linear} {h_vals[i]} {losses[-1]}\n'
            test_losses += f'{is_linear} {h_vals[i]} {test_loss.item()}\n'

    save(train_losses, path=f'../data/a5_train_losses.txt')
    save(test_losses, path=f'../data/a5_test_losses.txt')
```

Text classification on SST-2

Problem 7: Answers

a. See Problem 7: Code

b. See Problem 7: Code

c. See Problem 7: Code

d.

Accuracy and Loss of Neural Networks (8 epochs)

	Training		Validation	
	Accuracy (%)	Loss	Accuracy (%)	Loss
RNN	0.8987	0.2719	0.7695	0.5934
GRU	0.9305	0.1866	0.7993	0.6079
LSTM	0.9382	0.1677	0.7259	0.8714

e. Output each hidden state. That is, each word produces an output. Then we build a model around predicting the tag of each token from the exposed hidden states.

Problem 7: Code

```
# HW4 Problem 7 - Text Classification

import torch
import torch.nn as nn

def collate_fn(batch):
    """
    Create a batch of data given a list of N sequences and labels.
    Sequences are stacked into a single tensor of shape (N, max_sequence_length),
    where max_sequence_length is the maximum length of any sequence in the batch.
    Sequences shorter than this length should be filled up with 0's.
    Also returns a tensor of shape (N, 1)containing the label of each sequence.

    :param batch: A list of size N, where each element is a tuple containing a sequence tensor
    and a single item tensor containing the true label of the sequence.

    :return: A tuple containing two tensors. The first tensor has shape
    (N, max_sequence_length) and contains all sequences.
    Sequences shorter than max_sequence_length are padded with 0s at the end.
    The second tensor has shape (N, 1) and contains all labels.
    """
    sentences, labels = zip(*batch)
    sentences, labels = list(sentences), torch.stack(list(labels))

    max_sequence_length = max(len(s) for s in sentences)

    for i, sentence in enumerate(sentences):
        length_diff = max_sequence_length - len(sentence)
        sentences[i] = torch.nn.functional.pad(sentence, [0, length_diff])

    sentences_tensor = torch.stack(sentences)
    return sentences_tensor, labels


class RNNBinaryClassificationModel(nn.Module):
    def __init__(self, embedding_matrix, hidden_size=64):
```

```
37          super().__init__()
38          embedding_dim = embedding_matrix.shape[1]
39          self.num_layers = 6
40
41          # Construct embedding layer and initialize with given embedding matrix. Do not modify this code.
42          self.embedding = nn.Embedding(num_embeddings=embedding_matrix.shape[0],
43                                        embedding_dim=embedding_dim,
44                                        padding_idx=0)
45          self.embedding.weight.data = embedding_matrix
46
47          # Construct 3 different types of RNN for comparison.
48          self.RNN = nn.RNN(input_size=embedding_dim,
49                            hidden_size=hidden_size,
50                            num_layers=self.num_layers,
51                            batch_first=True)
52
53          self.GRU = nn.GRU(input_size=embedding_dim,
54                            hidden_size=hidden_size,
55                            num_layers=self.num_layers,
56                            batch_first=True)
57
58          self.LSTM = nn.LSTM(input_size=embedding_dim,
59                            hidden_size=hidden_size,
60                            num_layers=self.num_layers,
61                            batch_first=True,
62                            bidirectional=True)
63
64          self.linear = nn.Linear(in_features=hidden_size, out_features=1)
65          self.sigmoid = nn.Sigmoid()
66
67      def forward(self, inputs):
68          """
69          Takes in a batch of data of shape (N, max_sequence_length).
70          Returns a tensor of shape (N, 1), where each
71          element corresponds to the prediction for the corresponding sequence.
72          :param inputs: Tensor of shape (N, max_sequence_length) containing N
73              sequences to make predictions for.
74          :return: Tensor of predictions for each sequence of shape (N, 1).
75          """
76          # Un-comment for training other models.
77          #return self.sigmoid(self.linear(self.RNN(self.embedding(inputs))[1][-1].squeeze(0)))
78          #return self.sigmoid(self.linear(self.GRU(self.embedding(inputs))[1][-1].squeeze(0)))
79          return self.sigmoid(self.linear(self.LSTM(self.embedding(inputs))[1][0][-1].squeeze(0)))
80
81      def loss(self, logits, targets):
82          """
83          Computes the binary cross-entropy loss.
84          :param logits: Raw predictions from the model of shape (N, 1)
85          :param targets: True labels of shape (N, 1)
86          :return: Binary cross entropy loss between logits and targets as a scalar tensor.
87          """
88          return nn.BCELoss()(logits, targets.float())
89
90      def accuracy(self, logits, targets):
91          """
92          Computes the accuracy, i.e number of correct predictions / N.
93          :param logits: Raw predictions from the model of shape (N, 1)
94          :param targets: True labels of shape (N, 1)
95          :return: Accuracy as a scalar tensor.
96          """
97          correct = 0
98
99          for i in range(len(logits)):
100             prediction = torch.round(logits[i])
101             correct += (prediction == targets[i]).sum().item()
102
103         accuracy = correct / len(logits)
104         return torch.tensor(accuracy)
105
```

```
106 # Training parameters
107 TRAINING_BATCH_SIZE = 32
108 NUM_EPOCHS = 8
109 LEARNING_RATE = 0.0001
110
111 # Batch size for validation, this only affects performance.
112 VAL_BATCH_SIZE = 128
```