CSE446 HW3

Brian Kang[*†]

Conceptual Questions

A.1

a. False. By projecting our data on a k-dim subspace, k = full rank(X) maximum. Its is possible to not have full rank, then we definitely will lose information.

b. True. The max margin using support vectors are constructed such that the distance from the decision boundary is minimized, that is, lowest generalization error among all linear classifiers.

c. True. Bootstrap used sampling with replacement.

d. False, should be the rows of V^T.

e. False. Applying LASSO after feature reduction using PCA does not return coefficients that have meaning, so it is not interpretable for the linear model.

f. True. With varying k, the complexity of the clustering model varies. So, with the right k, we can find good clusters.

g. Decrease sigma to increase variance and decrease bias and increase the fit so we are not underfitting the training data. This allows us to reduce our error as long as we do not decrease so much that we end up overfitting.

[*]Collaborated with Cindy Wu

[†]Referred to Stackoverflow, Kaggle, and various other websites that helped structuring my codes.

Kernels and Bootstrap

A.2

Proof.

$$\begin{aligned}
\phi(x) \cdot \phi(x') &= \sum_{i=0}^{\infty} (\frac{1}{\sqrt{i!}} e^{\frac{-x^2}{2}} x^i) * (\frac{1}{\sqrt{i!}} e^{\frac{-x'^2}{2}} x'^i) \\
&= e^{\frac{-x^2-x'^2}{2}} \sum_{i=0}^{\infty} \frac{1}{i!} (xx')^i \\
&= e^{\frac{-x^2-x'^2}{2}} * e^{xx'} \text{ by taylor expansion of } e^z \\
&= e^{\frac{-x^2+2xx'-x'^2}{2}} = e^{-\frac{(x-x')}{2}^2}
\end{aligned}$$

□

A.3

Here is the code: Plots and answers are below the code!

```
# Brian Kang
import numpy as np
import matplotlib.pyplot as plt
from itertools import compress, combinations


# generate random data
def generateData(n):
    np.random.seed(446)
    X = np.random.uniform(size=n)
    e = np.random.randn(n)
    Y = f(X) + e
    return X, Y


# for easier plotting later
def f(X):
    return 4 * np.sin(np.pi * X) * np.cos(6 * np.pi * X ** 2)


# first kernel function
def kernelPoly(X, z, d):
    return np.power(1 + X * z, d)


# second kernel function
def kernelRBF(X, z, gamma):
    return np.exp(-gamma * np.square(X - z))


def kernelRidge(X, Y, kernelFunc, hyperparam, lmbda):
    Xi, Xj = np.meshgrid(X, X)  # get 2D array of all points made from [X by X]
    K = kernelFunc(Xi, Xj, hyperparam)  # do kernel evaluation
    # np.linalg.lstsq(a, b, rcond = 'warning'):
    # Solves the equation a*x = b by computing a vector x that
    # minimizes the squared Euclidean 2-norm of b-ax
    # https://stackoverflow.com/questions/54753132/understanding-numpys-lstsq
    alphahat, _, _, _ = np.linalg.lstsq(K.T @ K + lmbda * K, K.T @ Y, rcond=None
    )  # get alpha hat
    # return a lambda function of getting array of fhat = sum of alphahat *
    kernel poly or rbf
    return lambda xprime: np.array([np.sum(alphahat * kernelFunc(X, z,
    hyperparam)) for z in xprime])


# do both LOOCV and 10-fold by setting k=1 or 10
def kCV(k, X, Y, kernelFunc, hyperparam, lmbda):
    total = len(X)
    index = np.arange(total).astype(int)
    # mix up index and make n/k by k matrix, each row is then a fold
    fold = np.random.permutation(index).reshape(int(total / k), k)
    error = np.zeros(int(total / k))
```

```
    # do ridge
    for i, testindx in enumerate(fold):
        trainindx = np.ones(total).astype(int)
        trainindx[testindx] = 0
        # get training set from X using compress() in np.array format
        Xtrain = np.array([xtrain for xtrain in compress(X, trainindx)])
        Ytrain = np.array([ytrain for ytrain in compress(Y, trainindx)])
        # define predictor function
        fhat = kernelRidge(Xtrain, Ytrain, kernelFunc, hyperparam, lmbda)
        # SSR/n
        error[i] = np.sum(np.power(Y[testindx] - fhat(X[testindx]), 2)) / len(
    testindx)
    return np.mean(error)  # mean error


def problemsABC(n, nfold, B, figureNumber):
    X, Y = generateData(n)

    # Part 3.A.poly
    # make some lambda values
    LMBDA = np.power(10, np.linspace(-2, 2, 10))
    # make some hyperparameter d's in the Natural numbers
    D = np.arange(0, 15)
    savePolyResult = np.zeros([len(LMBDA) * len(D), 3])
    index = 0
    for d in D:
        for lmbda in LMBDA:
            savePolyResult[index] = np.array([d, lmbda, kCV(nfold, X, Y,
    kernelPoly, d, lmbda)])
            index = index + 1
    # get vars corresponding with smallest error
    bestd, bestlmbda, _ = savePolyResult[np.argmin(savePolyResult[:, 2])]
    print("Best d for poly kernel is " + str(bestd))
    print("Best lambda for poly kernel is " + str(bestlmbda))

    # Part 3.B & C.poly
    xvals = np.linspace(0, 1, 100)
    # bootstrap
    index = np.arange(n).astype(int)
    saveFhat = []
    for i in range(B):
        bootstrap = np.random.choice(index, n, replace=True)
        saveFhat.append(kernelRidge(X[bootstrap], Y[bootstrap], kernelPoly,
    bestd, bestlmbda))
    # confidence intervals using percentiles
    sortedFhat = np.sort(np.array([fhat(xvals) for fhat in saveFhat]), axis=0)
    CIdown = sortedFhat[int(0.025 * B), :]
    CIup = sortedFhat[int(0.975 * B), :]
    # define predictor function
    fhatPoly = kernelRidge(X, Y, kernelPoly, bestd, bestlmbda)
    plt.figure(figureNumber)
    ax = plt.gca()
    plt.scatter(X, Y)
    plt.plot(xvals, f(xvals), label="True f")  # true f(x)
    plt.plot(xvals, fhatPoly(xvals), label="Predicted f")  # fhat(x)
```

```
    ax.fill_between(xvals, CIdown, CIup, color='b', alpha=0.1)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.savefig("hw3_" + str(figureNumber) + ".png")

    # Part 3.A.rbf
    # make some lambda values
    LMBDA = np.power(10, np.linspace(-2, 2, 10))
    # make some hyperparameter gamma's
    GAMMA = np.linspace(0, 50, 20)
    saveRBFResult = np.zeros([len(LMBDA) * len(GAMMA), 3])
    index = 0
    for gamma in GAMMA:
        for lmbda in LMBDA:
            saveRBFResult[index] = np.array([gamma, lmbda, kCV(1, X, Y,
kernelRBF, gamma, lmbda)])
            index = index + 1
    # get vars corresponding with smallest error
    bestgamma, bestlmbda, _ = saveRBFResult[np.argmin(saveRBFResult[:, 2])]
    # if get gamma using heuristic method
    # bestgamma = 1/np.median([np.power(xi[0] - xi[1], 2) for xi in combinations
(X, 2)])
    print("Best gamma for rbf kernel is " + str(bestgamma))
    print("Best lambda for rbf kernel is " + str(bestlmbda))

    # Part 3.B & C.rbf
    xvals = np.linspace(0, 1, 100)
    # bootstrap
    index = np.arange(n).astype(int)
    saveFhat = []
    for i in range(B):
        bootstrap = np.random.choice(index, n, replace=True)
        saveFhat.append(kernelRidge(X[bootstrap], Y[bootstrap], kernelRBF,
bestgamma, bestlmbda))
    # confidence intervals using percentiles
    sortedFhat = np.sort(np.array([fhat(xvals) for fhat in saveFhat]), axis=0)
    CIdown = sortedFhat[int(0.025 * B), :]
    CIup = sortedFhat[int(0.975 * B), :]
    # define predictor function
    fhatRBF = kernelRidge(X, Y, kernelRBF, bestgamma, bestlmbda)
    plt.figure(figureNumber + 1)
    ax = plt.gca()
    plt.scatter(X, Y)
    plt.plot(xvals, f(xvals), label="True f")  # true f(x)
    plt.plot(xvals, fhatRBF(xvals), label="Predicted f")  # fhat(x)
    ax.fill_between(xvals, CIdown, CIup, color='b', alpha=0.1)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.savefig("hw3_" + str(figureNumber + 1) + ".png")

    return fhatPoly, fhatRBF  # for part E


```

```
154 # problem 3.abc.1
155 problemsABC(n=30, nfold=1, B=300, figureNumber=1)
156 print()
157 # problem 3.abc.2 = 3.d
158 fhatPoly, fhatRBF = problemsABC(n=300, nfold=10, B=300, figureNumber=3)
159
160 # problem 3.e
161 m = 1000
162 B = 300
163 X, Y = generateData(m)
164 # bootstrap
165 index = np.arange(m).astype(int)
166 saveMu = []
167 for i in range(B):
168     bootstrap = np.random.choice(index, m, replace=True)  # size m
169     error = np.mean(
170         np.power(Y[bootstrap] - fhatPoly(X[bootstrap]), 2) - np.power(Y[
    bootstrap] - fhatRBF(X[bootstrap]), 2))
171     saveMu.append(error)
172 # confidence intervals using percentiles
173 sortedMu = np.sort(np.array(saveMu), axis=0)
174 CIdown = sortedMu[int(0.025 * B)]
175 CIup = sortedMu[int(0.975 * B)]
176 print()
177 print("The CI is: (" + str(CIdown) + ", " + str(CIup) + ")")
178 if (CIdown <= 0) and (CIup >= 0):
179     print(
180         "The confidence interval includes 0, so there exists statistically
    significant\nevidence that one of f_rbf or f_poly is better at predicting Y
    from X.")
181 else:
182     print(
183         "The confidence interval does NOt include 0, so there is NO
    statistically significant\nevidence that one of f_rbf or f_poly is better at
    predicting Y from X.")
```

a. Refer to figure 1.

```
Best d for poly kernel is 2.0
Best lambda for poly kernel is 0.027825594022071243
Best gamma for rbf kernel is 50.0
Best lambda for rbf kernel is 0.0774263682681127
```

Figure 1: Parameter values for Poly and RBF Kernels

b. Refer to figure 2.

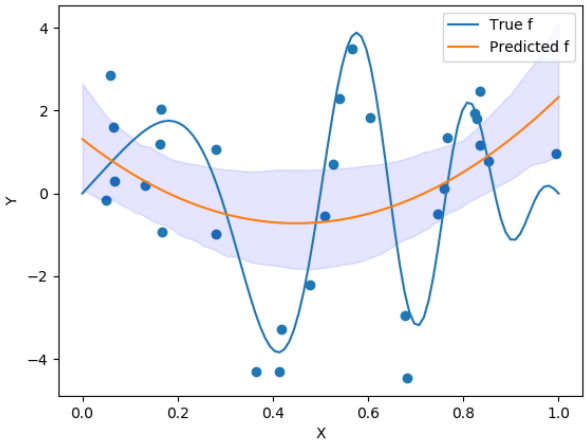


Figure 2: Poly Kernel v.s. True f

c. Refer to figure 3.

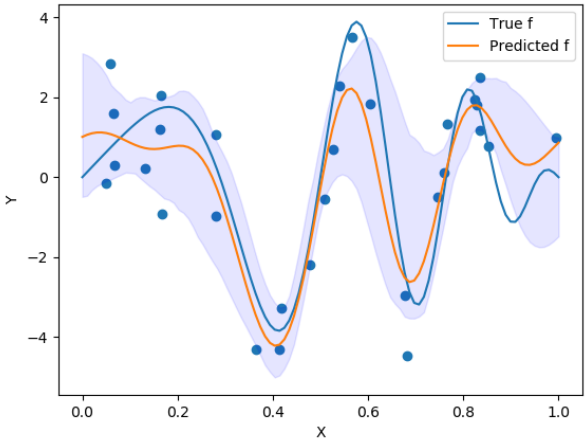


Figure 3: RBF Kernel v.s. True f

```
Best d for poly kernel is 13.0
Best lambda for poly kernel is 0.027825594022071243
Best gamma for rbf kernel is 50.0
Best lambda for rbf kernel is 0.01
```

Figure 4: Parameter values for Poly Kernel

d. Refer to figure 4 for parameter values.

Refer to figure 5 for poly kernel plot.

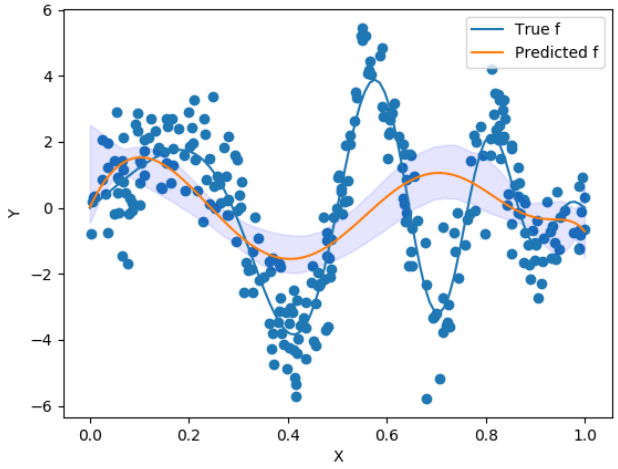


Figure 5: Poly Kernel v.s. True f

Refer to figure 6 for rbf kernel plot.

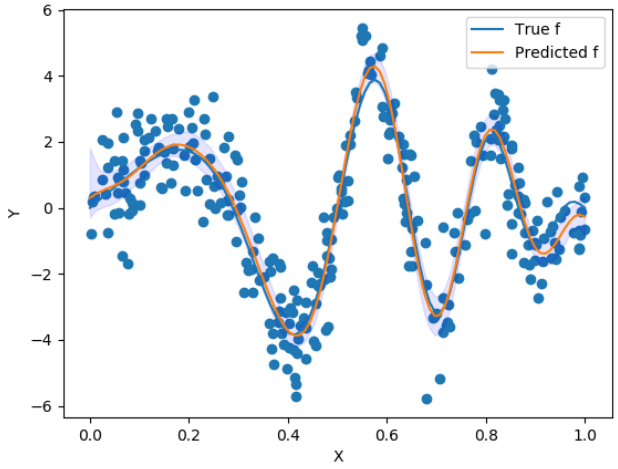


Figure 6: RBF Kernel v.s. True f

e. Refer to figure 7 for the confidence interval and its interpretation.
TYPO Correction: "There IS evidence supporting the fact that one is better than the other because 0 is not in the confidence interval."

```
The CI is: (2.782856608988435, 3.4650986308114233)
The confidence interval does NOt include 0, so there is NO statistically significant
evidence that one of f_rbf or f_poly is better at predicting Y from X.
```

Figure 7: Confidence Interval

k-means clustering

A.4

Here is the code: Plots and answers are below the code!

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


def load_dataset():
    mndata = MNIST('.\data')
    mndata.gz = True
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    return X_train, labels_train, X_test, labels_test


# Part A
class kmeans:
    def __init__(self, k):
        self.k = k
        self.objective = []
        self.cluster = None
        self.center = None

    # lloyds algorithm
    def train(self, X, maxIter):
        # get k starting center points randomly chosen
        n, _ = X.shape
        np.random.seed(446)
        startCenter = X[np.random.randint(0, n, size=self.k)]
        # center_i = mu_i
        center = np.copy(startCenter)  # use the copy
        dist = np.zeros((n, self.k))  # must have extra parentheses
        for iter in range(maxIter):
            for i in range(self.k):
                dist[:, i] = np.linalg.norm(X - center[i], axis=1) ** 2
            # find centeroids of each clusters
            centroids = []
            objective = 0
            # referred to some stackoverflow page & lecture notes
            for i in range(self.k):
                getCenter = X[np.argmin(dist, axis=1) == i]  # point with min
    dist
                objective += np.sum(np.linalg.norm(getCenter - center[i], axis
    =1) ** 2)
                centroid = np.mean(getCenter, axis=0)
                centroids.append(centroid)
            center = np.copy(np.array(centroids))  # updated center
            self.objective.append(objective)
        self.center = center
```

```
        self.cluster = np.argmin(dist, axis=1)

    def predict(self, X):
        n, _ = X.shape
        dist = np.zeros((n, self.k))  # must have extra parentheses
        for i in range(self.k):
            dist[:, i] = np.linalg.norm(X - self.center[i], axis=1) ** 2
        pred = self.center[np.argmin(dist, axis=1)]
        return pred


# Part B.1
x_train, y_train, x_test, y_test = load_dataset()  # load in data

kmean1 = kmeans(10)
kmean1.train(x_train, 75)
plt.figure(1)
plt.plot(kmean1.objective)
plt.xlabel("Iteration")
plt.ylabel("Objective Value")
plt.savefig("hw3_9.png")

# Part B.2
# visualize the cluster centers
# k=10=2*5
plt.figure(2)
fig, axes = plt.subplots(2, 5)
for i, ax in enumerate(axes.flatten()):
    ax.imshow(kmean1.center[i].reshape(28, 28), cmap='gray')
    ax.set_title('Cluster Center {}'.format(i))
plt.tight_layout()
plt.savefig("hw3_10.png")

# Part C
ks = np.array([2, 4, 8, 16, 32, 64])
trainError = []
testError = []
for k in ks:
    kmeani = kmeans(k)
    kmeani.train(x_train, 25)
    predTrain = kmeani.predict(x_train)
    predTest = kmeani.predict(x_test)
    trainError.append(np.mean(np.linalg.norm(x_train - predTrain, axis=1) ** 2))
    testError.append(np.mean(np.linalg.norm(x_test - predTest, axis=1) ** 2))

plt.figure(3)
plt.plot(ks, trainError, label='Train')
plt.plot(ks, testError, label='Test')
plt.xlabel("K")
plt.ylabel("Error")
plt.legend()
plt.savefig("hw3_11.png")
```

a. Code for Lloyd's algorithm is above.

b. Refer to figure below for objective function values.

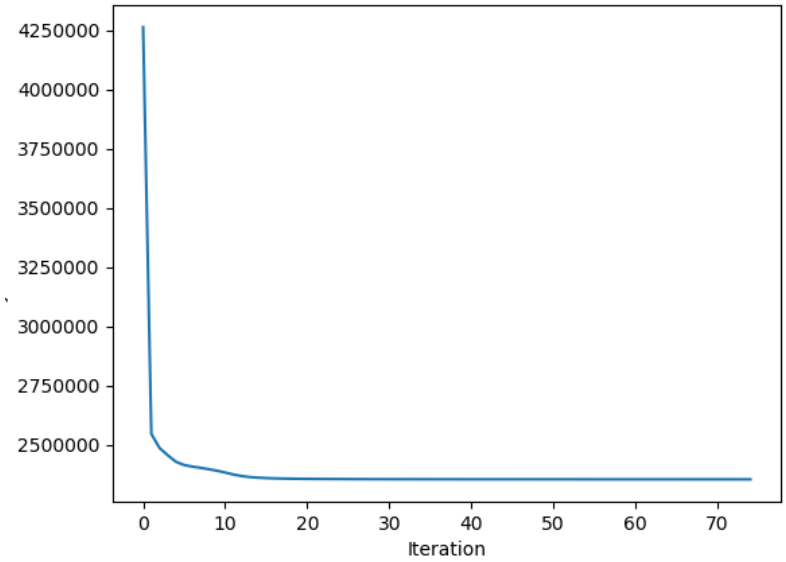


Figure 8: K-means Objective

Refer to figure below for Visualized Cluster Centers.

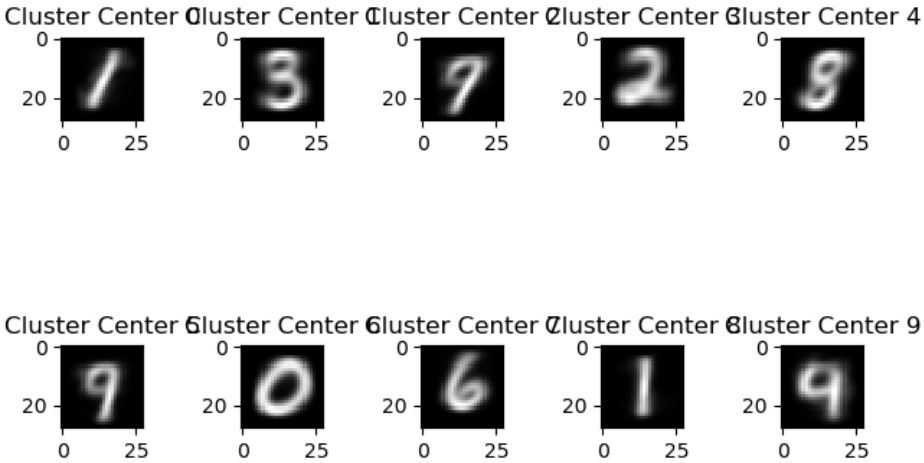


Figure 9: Visualized Cluster Centers

c. Refer to figure below for error values.

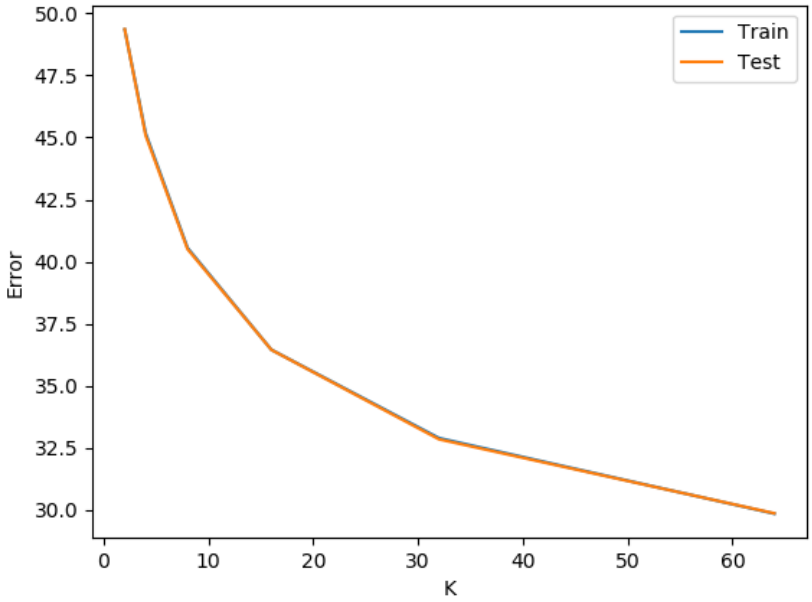


Figure 10: Model Errors per Varying K

Neural Networks for MNIST

A.5

Here is the code: Plots and answers are below the code!

```
# Brian Kang
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST


def load_dataset():
    mndata = MNIST('.\data')
    mndata.gz = True
    X_train, labels_train = map(np.array, mndata.load_training())
    X_test, labels_test = map(np.array, mndata.load_testing())
    X_train = X_train / 255.0
    X_test = X_test / 255.0
    return X_train, labels_train, X_test, labels_test


class nnet(nn.Module):
    def __init__(self, X, Y):
        super(nnet, self).__init__()
        self.X = X.T  # 784 x n
        self.Y = Y.T  # 10 x n
        d, n1 = self.X.shape  # 784
        k, n2 = self.Y.shape
        h = 64  # number of nodes in the hidden layer
        alpha = 1 / np.sqrt(d)
        torch.manual_seed(446)
        self.w0 = nn.Parameter(torch.FloatTensor(h, d).uniform_(-alpha, alpha), requires_grad=True)
        self.w1 = nn.Parameter(torch.FloatTensor(k, h).uniform_(-alpha, alpha), requires_grad=True)
        self.b0 = nn.Parameter(torch.FloatTensor(h, n1).uniform_(-alpha, alpha), requires_grad=True)
        self.b1 = nn.Parameter(torch.FloatTensor(k, n2).uniform_(-alpha, alpha), requires_grad=True)
        self.layer1 = torch.zeros(h, n1)
        self.output = torch.zeros(k, n2)
        self.reLU = nn.ReLU()  # we instantiate an instance of the ReLU module

    def forward(self, X):
        self.layer1 = self.reLU(torch.add(torch.mm(self.w0, X.T), self.b0))  # reLU(W*X+b)
        self.output = torch.add(torch.mm(self.w1, self.layer1), self.b1)
        return self.output


# Part A
learningRate = 0.001
h = 64
```

```
d = 10

x_train, y_train, x_test, y_test = load_dataset()  # load in data
num_class = 10  # this is k
onehot_label_train = np.eye(num_class)[y_train]  # one-hot code the labels
onehot_label_test = np.eye(num_class)[y_test]

minibatch_size = 600
torch.manual_seed(446)
batch = np.random.randint(0, len(x_train), minibatch_size)  # random index selection
model = nnet(np.array(x_train[batch, :]), np.array(onehot_label_train[batch, :]))

model.train()
optim = torch.optim.Adam(model.parameters(), lr=learningRate)
lossFunc = nn.CrossEntropyLoss()

i = 0
indx = []
lost = []
cont = True  # start training
while cont:
    optim.zero_grad()  # zero all the gradients to get new gradients soon
    outputs = model(torch.from_numpy(np.array(x_train[batch, :])).float())
    label = []  # get label
    for idx, vec in enumerate(np.array(onehot_label_train[batch, :])):
        label = np.append(label, np.where(vec == 1)[0])
    input = outputs.T
    loss = lossFunc(input,
                    torch.LongTensor(torch.from_numpy(np.array(label)).long()))
    loss.backward()  # computes derivatives of the loss with respect to W
    optim.step()  # make a step

    i = i + 1
    indx.append(i)
    lost.append(loss.item())
    if i % 1000 == 1:
        learningRate = learningRate + 0.00025  # slowly increase learning rate
    if i == 60000 and loss.item() >= 0.01:
        i = 0
    if loss.item() < 0.01:
        cont = False  # loss is small enough, stop training
    # next random batch to train on
    torch.manual_seed(446)
    batch = np.random.randint(0, len(x_train), minibatch_size)  # random index selection

# test loss
model.eval()
torch.manual_seed(446)
batch = np.random.randint(0, len(x_test), minibatch_size)
ii = 0
lostVal = []
for iter in range(0, i):
```

```
    outputs = model(torch.from_numpy(np.array(x_test[batch, :])).float())
    label = []
    for idx, vec in enumerate(np.array(onehot_label_test[batch, :])):
        label = np.append(label, np.where(vec == 1)[0])
    input = outputs.T
    loss = lossFunc(input,
                    torch.LongTensor(torch.from_numpy(np.array(label)).long()))
    ii = ii + 1
    lostVal.append(loss.item())
    torch.manual_seed(446)
    batch = np.random.randint(0, len(x_test), minibatch_size)

# loss plot
plt.figure(1)
plt.plot(indx, lost, label="Train", linewidth=0.5)
plt.plot(indx, lostVal, label="Test", linewidth=0.5)
plt.xlabel("Iteration")
plt.ylabel("Cross Entropy Training Loss")
plt.suptitle("Minibatch of Size 600")
plt.legend()
plt.savefig("hw3_5.png")

# accuracy plot
acc = [1 - x for x in lost]
accVal = [1 - x for x in lostVal]
plt.figure(2)
plt.plot(indx, acc, label="Train", linewidth=0.5)
plt.plot(indx, accVal, label="Test", linewidth=0.5)
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.suptitle("Minibatch of Size 600")
plt.legend()
plt.ylim(bottom=0)
plt.savefig("hw3_6.png")


# Part B
class nnet2(nn.Module):
    def __init__(self, X, Y):
        super(nnet2, self).__init__()
        self.X = X.T  # 784 x n
        self.Y = Y.T  # 10 x n
        d, n = self.X.shape  # 784 x minibatch
        k, _ = self.Y.shape
        h1 = 32  # number of nodes in the hidden layer
        h2 = 32  # second hidden layer
        alpha = 1 / np.sqrt(d)
        torch.manual_seed(446)
        self.w0 = nn.Parameter(torch.FloatTensor(h1, d).uniform_(-alpha, alpha),
    requires_grad=True)
        self.w1 = nn.Parameter(torch.FloatTensor(h2, h1).uniform_(-alpha, alpha)
    , requires_grad=True)
        self.w2 = nn.Parameter(torch.FloatTensor(k, h2).uniform_(-alpha, alpha),
    requires_grad=True)
        self.b0 = nn.Parameter(torch.FloatTensor(h1, n).uniform_(-alpha, alpha),
```

```
        requires_grad=True)
150         self.b1 = nn.Parameter(torch.FloatTensor(h2, n).uniform_(-alpha, alpha),
        requires_grad=True)
151         self.b2 = nn.Parameter(torch.FloatTensor(k, n).uniform_(-alpha, alpha),
    requires_grad=True)
152         self.layer1 = torch.zeros(h1, n)
153         self.layer2 = torch.zeros(h2, n)
154         self.output = torch.zeros(k, n)
155         self.reLU = nn.ReLU()  # we instantiate an instance of the ReLU module
156
157     def forward(self, X):
158         self.layer1 = self.reLU(torch.add(torch.mm(self.w0, X.T), self.b0))  #
    reLU(W*X+b)
159         self.layer2 = self.reLU(torch.add(torch.mm(self.w1, self.layer1), self.
    b1))
160         self.output = torch.add(torch.mm(self.w2, self.layer2), self.b2)
161         return self.output
162
163
164 learningRate = 0.001
165 h1 = 32
166 h2 = 32
167 d = 10
168
169 minibatch_size = 600
170 torch.manual_seed(446)
171 batch = np.random.randint(0, len(x_train), minibatch_size)  # random index
    selection
172 model = nnet2(np.array(x_train[batch, :]), np.array(onehot_label_train[batch,
    :]))
173
174 model.train()
175 optim = torch.optim.Adam(model.parameters(), lr=learningRate)
176 lossFunc = nn.CrossEntropyLoss()
177
178 i = 0
179 indx = []
180 lost = []
181 cont = True  # start training
182 while cont:
183     optim.zero_grad()  # zero all the gradients to get new gradients soon
184     outputs = model(torch.from_numpy(np.array(x_train[batch, :])).float())
185     label = []  # get label
186     for idx, vec in enumerate(np.array(onehot_label_train[batch, :])):
187         label = np.append(label, np.where(vec == 1)[0])
188     input = outputs.T
189     loss = lossFunc(input,
190                     torch.LongTensor(torch.from_numpy(np.array(label)).long()))
191     loss.backward()  # computes derivatives of the loss with respect to W
192     optim.step()  # make a step
193
194     i = i + 1
195     indx.append(i)
196     lost.append(loss.item())
197     if i % 1000 == 1:
```

```
        learningRate = learningRate + 0.00025  # slowly increase learning rate
    if i == 60000 and loss.item() >= 0.01:
        i = 0
    if loss.item() < 0.01:
        cont = False  # loss is small enough, stop training
    # next random batch to train on
    torch.manual_seed(446)
    batch = np.random.randint(0, len(x_train), minibatch_size)  # random index
    selection

# test loss
model.eval()
torch.manual_seed(446)
batch = np.random.randint(0, len(x_test), minibatch_size)
ii = 0
lostVal = []
for iter in range(0, i):
    outputs = model(torch.from_numpy(np.array(x_test[batch, :])).float())
    label = []
    for idx, vec in enumerate(np.array(onehot_label_test[batch, :])):
        label = np.append(label, np.where(vec == 1)[0])
    input = outputs.T
    loss = lossFunc(input,
                    torch.LongTensor(torch.from_numpy(np.array(label)).long()))
    ii = ii + 1
    lostVal.append(loss.item())
    torch.manual_seed(446)
    batch = np.random.randint(0, len(x_test), minibatch_size)

# loss plot
plt.figure(3)
plt.plot(indx, lost, label="Train", linewidth=0.5)
plt.plot(indx, lostVal, label="Test", linewidth=0.5)
plt.xlabel("Iteration")
plt.ylabel("Cross Entropy Training Loss")
plt.suptitle("Minibatch of Size 600")
plt.legend()
plt.savefig("hw3_7.png")

# accuracy plot
acc = [1 - x for x in lost]
accVal = [1 - x for x in lostVal]
plt.figure(4)
plt.plot(indx, acc, label="Train", linewidth=0.5)
plt.plot(indx, accVal, label="Test", linewidth=0.5)
plt.xlabel("Iteration")
plt.ylabel("Accuracy")
plt.suptitle("Minibatch of Size 600")
plt.legend()
plt.ylim(bottom=0)
plt.savefig("hw3_8.png")
```

a. Refer to figure below for loss values.

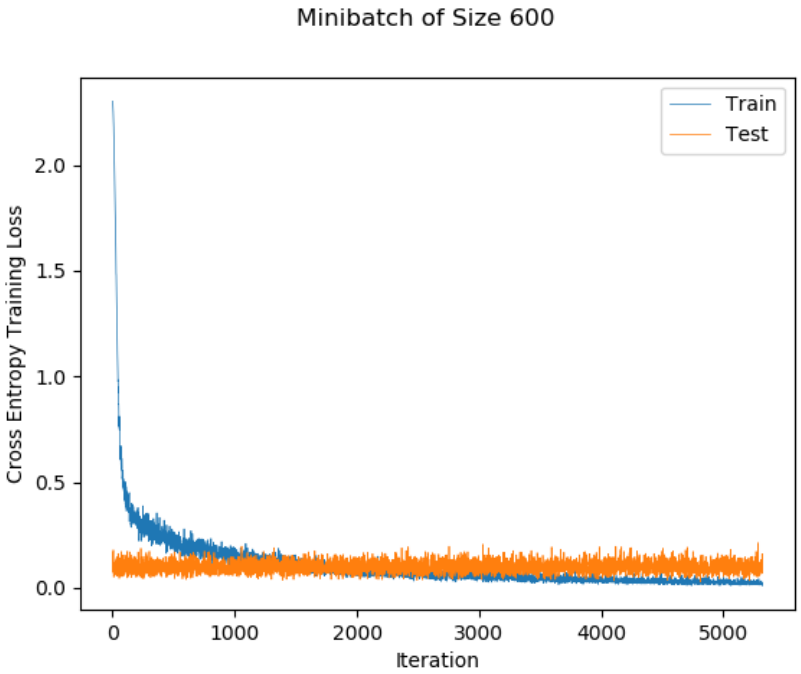


Figure 11: Loss: Wide and Shallow

Refer to figure below for accuracy values.

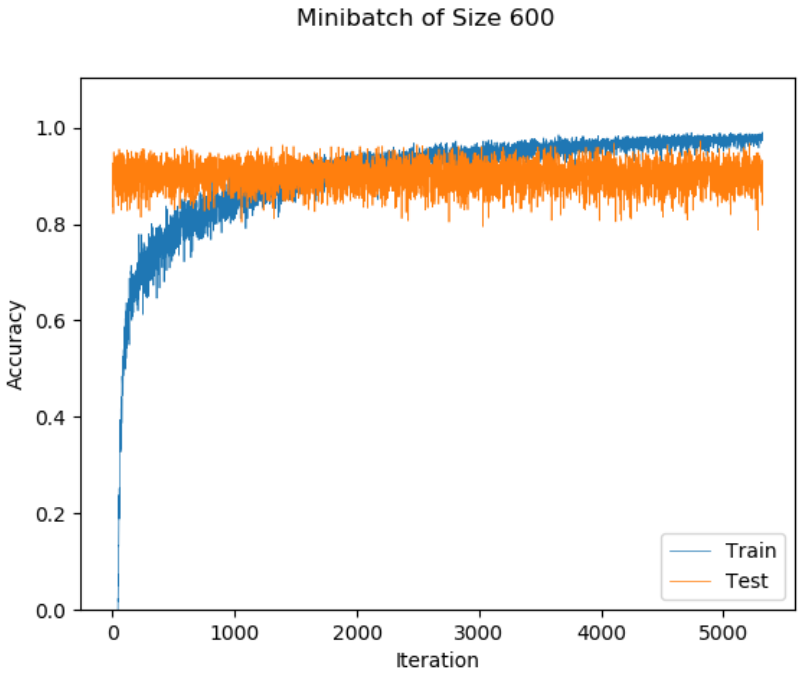


Figure 12: Accuracy: Wide and Shallow

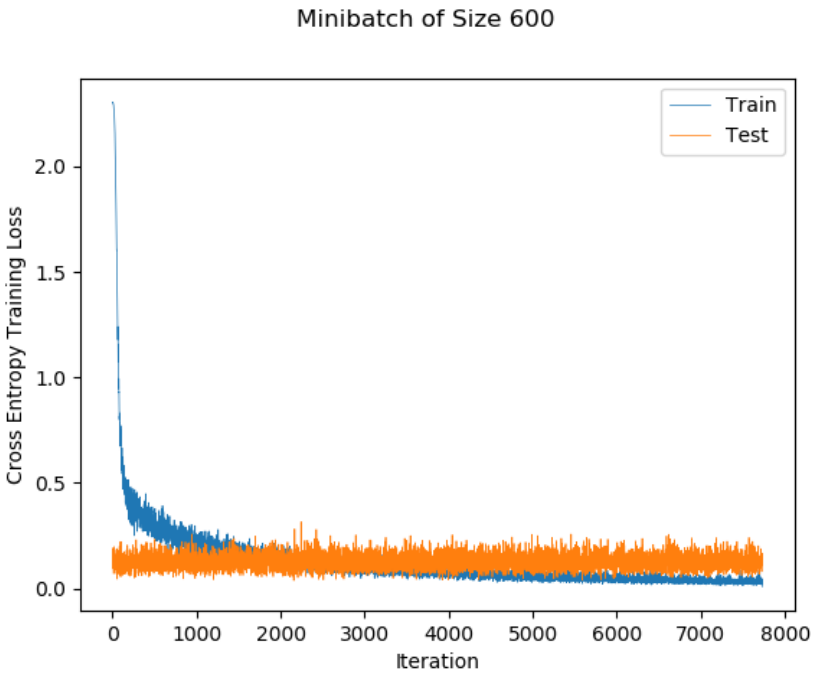


Figure 13: Loss: Narrow and Deep

b. Refer to figure above for loss values.

Refer to figure below for accuracy values.

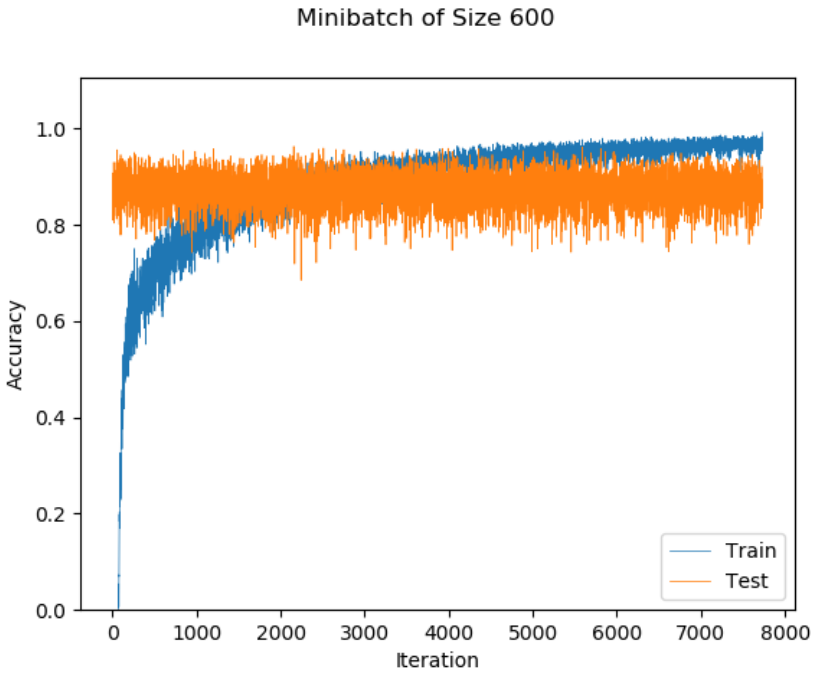


Figure 14: Accuracy: Narrow and Deep

c. NOTE: Before answering I would like to first acknowledge the behavior of the tests results. I expected the loss to start high and then decrease (or accuracy start low then increase) like the training set at first. But we see that it hovers around a single value like noise behaves. Not sure it this is implementation error, but I reason this behavior that it happened because I use the trained weight and bias on the test set. Such results in predicting the label "at its best" everytime, so the variation can be explained by the randomness of each mini-batch iteration.

Now my answer to the question, it looks like general behavior of Wide&Shallow and Narrow&Deep are very similar. But after running the neural networks multiple times, it seemed like the wide&shallow one computed faster than the deeper one even if it was narrower. This can be correlated to the smaller iterations done for wide&shallow (5500-ish) versus narrow&deep (7800-ish). Also, it's observable that test narrow&deep has more variation around a value than the other. This might be related to the idea that wider and shallower neural nets are better at "memorization" while narrow and deep neural nets are better at "generalization." In a sense, wide neural nets use more of the data to "memorize" the behavior, while deep neural nets use each layer to "generally understand" a component.

PCA

A.6

Here is the code: Plots and answers are below the code!

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


def load_dataset():
    mndata = MNIST('.\data')
    mndata.gz = True
    x_train, labels_train = map(np.array, mndata.load_training())
    x_test, labels_test = map(np.array, mndata.load_testing())
    x_train = x_train / 255.0
    x_test = x_test / 255.0
    return x_train, labels_train, x_test, labels_test


# Part A
x_train, y_train, x_test, y_test = load_dataset()
mu = np.mean(x_train, axis=0)
simga = np.dot((x_train - mu).T, x_train - mu) / len(x_train)  # covariance
    matrix
lmbda, V = np.linalg.eig(simga)
# sort in decreasing order
sortIndx = np.argsort(lmbda)[::-1]
lmbda = lmbda[sortIndx].astype('float')  # change type for later
V = V[:, sortIndx].astype('float')
indx = [0, 1, 9, 29, 49]
print("Eigenvalues:", lmbda[indx])  # get highest selected lmbda
sumlmbda = np.sum(lmbda)
print("Sum of Eigenvalues:", sumlmbda)

# Part C.1
trainError = []
testError = []
ks = np.arange(1, 101)
for k in ks:
    predTrain = mu + np.dot(np.dot(x_train - mu, V[:, :k]), V[:, :k].T)
    predTest = mu + np.dot(np.dot(x_test - mu, V[:, :k]), V[:, :k].T)
    trainError.append(np.mean(np.linalg.norm(x_train - predTrain, axis=1) ** 2))
     # mse
    testError.append(np.mean(np.linalg.norm(x_test - predTest, axis=1) ** 2))

plt.figure(1)
plt.plot(ks, trainError, label="Train")
plt.plot(ks, testError, label="Test")
plt.xlabel("K")
plt.ylabel("Error")
plt.legend()
plt.savefig("hw3_13.png")

```

```
# Part C.2
value = []
for k in ks:
    value.append(1 - np.sum(lmbda[:k]) / sumlmbda)

plt.figure(2)
plt.plot(ks, value)
plt.xlabel("K")
plt.ylabel("1-Sum(lambda 1...k) / Sum(lambda 1...d)")
plt.savefig("hw3_14.png")

# Part D
# visualize the PCA eigenvectors
# k=10=2*5
plt.figure(3)
fig, axes = plt.subplots(2, 5)
for i, ax in enumerate(axes.flatten()):
    ax.imshow(V[:, i].reshape(28, 28), cmap='gray')
    ax.set_title('PCA Eigen {}'.format(i))
plt.tight_layout()
plt.savefig("hw3_15.png")

# Part E
fignum = 16
for nums in [2, 6, 7]:
    digit = x_train[y_train == nums][0]  # arbitrary
    ks = [5, 15, 40, 100]
    # keep the brackets
    reconstruct = [digit] + [mu + np.dot(np.dot(digit - mu, V[:, :k]), V[:, :k].
    T) for k in ks]
    fig, axes = plt.subplots(1, 5)
    for i, ax in enumerate(axes.flatten()):
        ax.imshow(reconstruct[i].reshape(28, 28), cmap='gray')
        if i == 0:
            ax.set_title("Real Num.")
        else:
            ax.set_title("k=" + str(ks[i - 1]))
    plt.tight_layout()
    plt.savefig("hw3_" + str(fignum) + ".png")
    fignum = fignum + 1
```

```
Eigenvalues: [5.11678773 3.74132848 1.24272938 0.36425572 0.16970843]
Sum of Eigenvalues: 52.72503549512705
```

a.

b. For any k,

$$x = \mu + (x - \mu)V_k V_k^T$$

V_k is the matrix of eigenvectors of the first k eigenvalues for any $k = 1, \ldots, d$.

c. Refer to the plots below.

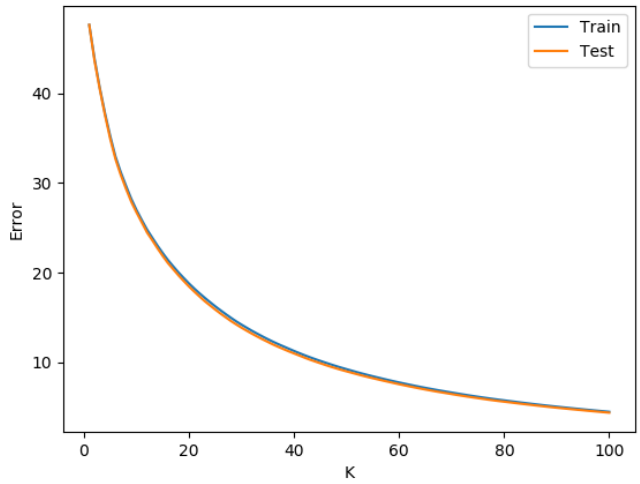


Figure 15: Reconstruction Error

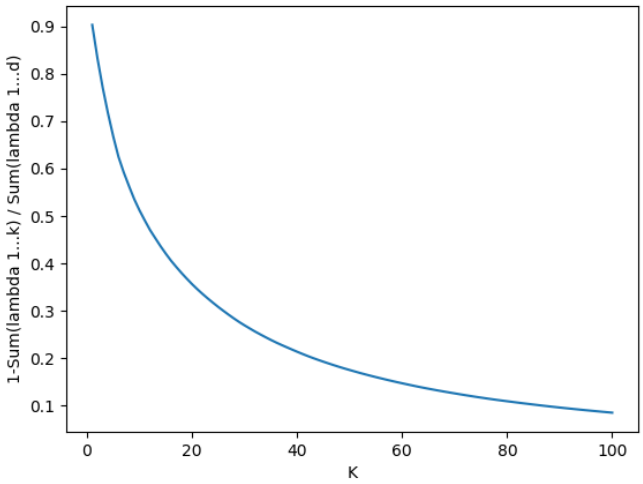
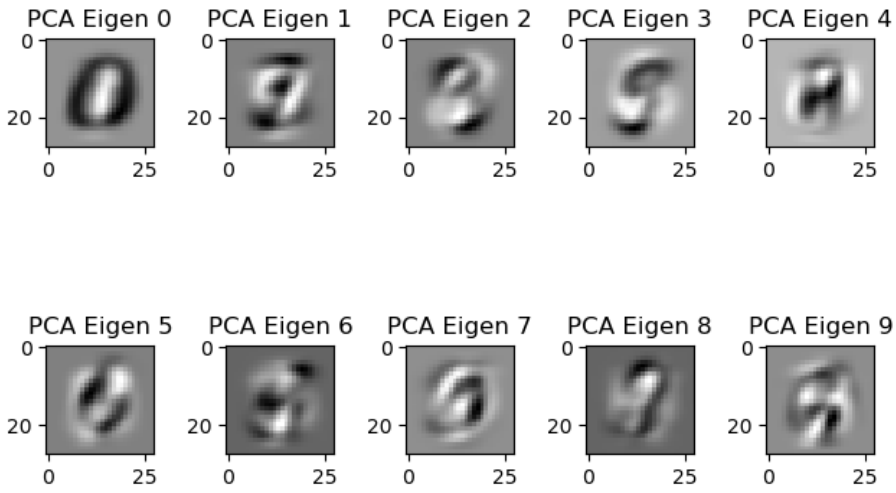
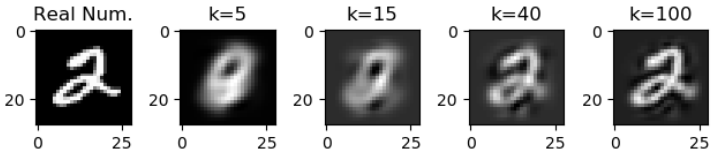


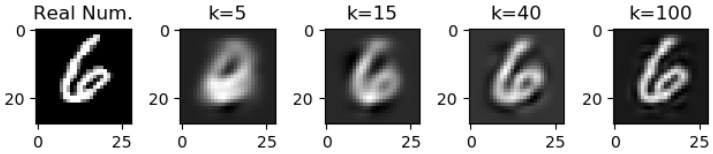
Figure 16: Plot of $1 - \frac{\sum_{i=1}^{k} \lambda}{\sum_{i=1}^{d} \lambda}$

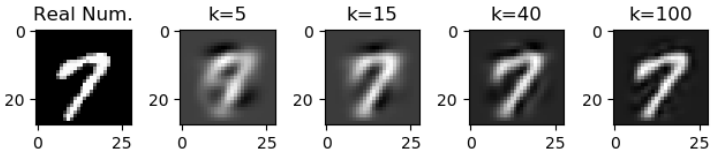
d. Above is what the PCA directions captured. PCA is alternaively known in the math world as SVD, or Singular Value Decomposition. Vectors are decomposed into certain number of orthogonal vectors, which allows to get a more accurate picture as we multiply more of these orthogonal vectors (leading to higher dimensions). We are seeing this. PCA captured parts of the characteristics of the digits. Each plot captures a different characteristic, and when these are all multiplied to each other, it is likely that we will get a accurate representation of the digits.



e. Below is presented the numbers 2, 6, 7 and what PCA (or SVD) captured with various k's, i.e., various dimensions of orthogonal vectors. As said in (d), with the increase of k, we can see that each digit is appears more clear.




Real Num.
k=5
k=15
k=40
k=100
0
20
0
25


Real Num.
k=5
k=15
k=40
k=100
0
20
0
25