Homework #4

Hyungseok Choi
Student No. : 2250166

June 30, 2022

Collaborated with Kevin Tong, Grace Chen.

Conceptual Questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. True. Each data belongs to a span of k base vectors, so the reconstruction error of the vector projected in k dimension can be said to be zero.

b. False. The columns of V are equal to the eigenvectors of $X^\top X$.

c. False. If we choose k equal to the number of data (=n), the k-means objective is 0 but it doesn't represent a meaningful clusters.

d. False. The singular value matrix is unique. However, the singular decomposition of a matrix could differ.

e. True. By definition, the rank of a square matrix equals the number of its unique nonzero eigenvalues.

f. True. While PCA only projecting data to a lower demension, Autoencoders could capture more variance using neural networks with nonlinear activation functions.

A2.

a. First, check the data to see which data are important, how each data are distributed, whether there is missing or incorrect information, whether the units of measurements are unified, no columns to be normalized, and whether sufficient amount of data is collected. After sufficient preprocessing is completed, split the data into a training dataset and test dataset. The most suitable model for this task is the tree model through supervised learning. Then, in order to tune the Hyper-parameter, validated the model by using the CVs of LOO and K-fold methods, and the most appropriate parameters are found using grid or random search. After creating such a suitable model, the performance of the model can be evaluated through test dataset, and then, when new data is collected, the expected season susceptibility of the new user can be predicted based on the learned model.

b. First, it is necessary to detect the face from the picture. Assuming that there are face data from various angles and sizes, the face is detected from the picture through CNN or HOG method. Use affine transformation according to the difference in face rotation, angle, etc, and calculate the difference with the front face. Then, we train the face model through extracting the general features of the face, which allows us to detect the face in a short time. Then, apply a filter to the extracted face features.

c. In addition to the shortcoming that problem mentioned, it was found that all data, such as factors that change over time, were not always measured from the same standard. Therefore, although general trends can be identified from the data, it can be seen that the results or prediction cannot be applied strictly.

d. If only information about employees in a particular country is used, overfitting might occur, so not only face information for the company's employees and their families, but also more general human face data should be collected. In addition, data according to various situations, such as wearing accessories like wearing a mask or glasses, and changes in facial expressions should also collected.

k-means clustering

A3.

a.
```
@problem.tag("hw4-A")
def lloyd_algorithm(
    data: np.ndarray, num_centers: int, epsilon: float = 10e-3
) -> Tuple[np.ndarray, List[float]]:

    centers = data[:num_centers]

    isConverged = False
    errors = []

    while not isConverged:
        classification = cluster_data(data, centers)
        previous_centers = centers
        centers = calculate_centers(data, classification, num_centers)

        errors.append(calculate_error(data, centers))

        if np.max(abs(centers - previous_centers)) < epsilon:
            isConverged = True

    return (centers, errors)
```

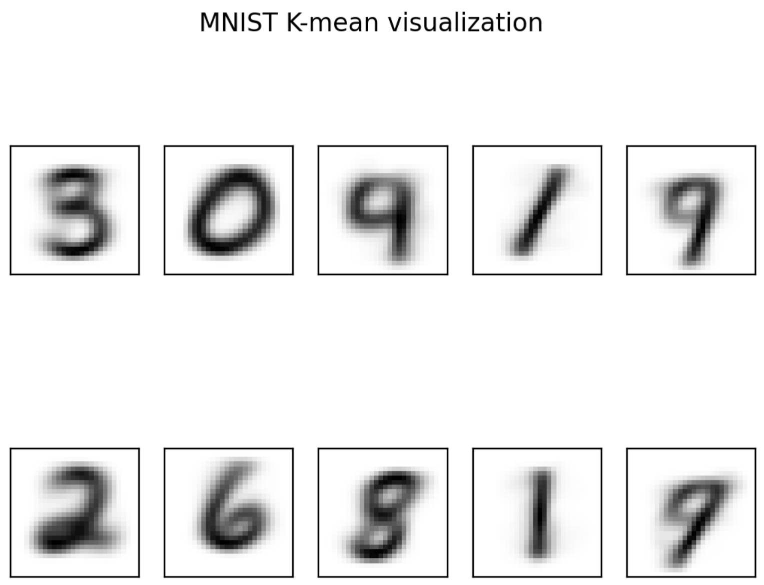

Figure 1: 10 images of cluster centers

b.

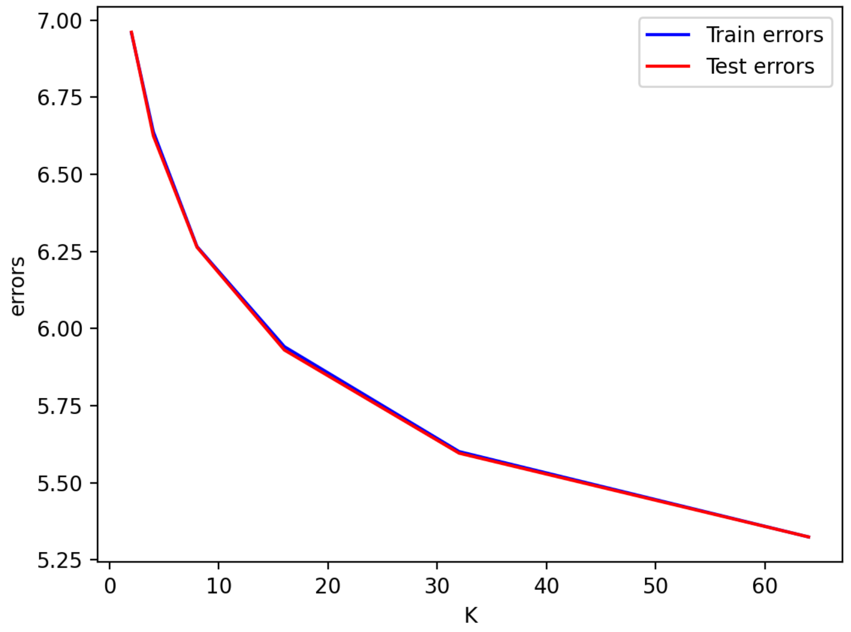

Figure 2: Plot of training and test error as function of k

c.

```
# /homeworks/k_means/k_means.py

from typing import List, Tuple
import numpy as np
from utils import problem

@problem.tag("hw4-A")
def calculate_centers(
    data: np.ndarray, classifications: np.ndarray, num_centers: int
) -> np.ndarray:

    return np.array([data[classifications==k].mean(axis=0) for k in range(num_centers)])


@problem.tag("hw4-A")
def cluster_data(data: np.ndarray, centers: np.ndarray) -> np.ndarray:

    return np.argmin(np.sqrt(((data - centers[:, np.newaxis])**2).sum(axis=2)), axis=0)


@problem.tag("hw4-A")
def calculate_error(data: np.ndarray, centers: np.ndarray) -> float:
    errors = 0.0

    for d in data:
        min_distance = float('inf')
        for c in centers:
            distance = np.linalg.norm((d-c))
            min_distance = min(distance, min_distance)

        errors += min_distance

    return errors / len(data)

@problem.tag("hw4-A")
    def lloyd_algorithm(
        data: np.ndarray, num_centers: int, epsilon: float = 10e-3
    ) -> Tuple[np.ndarray, List[float]]:

        centers = data[:num_centers]

        isConverged = False
        errors = []

        while not isConverged:
            classification = cluster_data(data, centers)
            previous_centers = centers
            centers = calculate_centers(data, classification, num_centers)

            errors.append(calculate_error(data, centers))

            if np.max(abs(centers - previous_centers)) < epsilon:
                isConverged = True

        return (centers, errors)
```

```
# /homeworks/k_means/main.py

if __name__ == "__main__":
    from k_means import calculate_error, lloyd_algorithm  # type: ignore
else:
    from .k_means import lloyd_algorithm, calculate_error

import matplotlib.pyplot as plt
import numpy as np
import pickle

from utils import load_dataset, problem


@problem.tag("hw4-A", start_line=1)
def main():

    (x_train, _), (x_test, _) = load_dataset("mnist")

    run_for = "ct"

    if run_for == "b":
        centers, errors = lloyd_algorithm(x_train, 10)
        with open('A2b-centers.pickle','wb') as mysavedata:
            pickle.dump(centers, mysavedata)

        with open('A2b-errors.pickle','wb') as mysavedata:
            pickle.dump(errors, mysavedata)


    if run_for == "bt":
        with open('A2b-centers.pickle','rb') as myloaddata:
            centers = pickle.load(myloaddata)

        with open('A2b-errors.pickle','rb') as myloaddata:
            errors = pickle.load(myloaddata)

        fig, ax = plt.subplots(2, 5)
        for i in range(len(centers)):
            ax.ravel()[i].imshow(centers[i, :].reshape((28,28)), cmap='gray_r')
            ax.ravel()[i].axes.xaxis.set_visible(False)
            ax.ravel()[i].axes.yaxis.set_visible(False)
        fig.suptitle('MNIST K-mean visualization')
        plt.show()

    if run_for == "c":

        kList = [2, 4, 8, 16, 32, 64]

        train_error = []
        test_error = []

        for k in kList:
            centers, errors = lloyd_algorithm(x_train, k)
            with open('A2c-centers-' + str(k) + '.pickle','wb') as file:
                pickle.dump(centers, file)

            with open('A2c-trainerrors-' + str(k) + '.pickle','wb') as file:
                pickle.dump(errors, file)

            t_error = calculate_error(x_test, centers)

            with open('A2c-testerrors-' + str(k) + '.pickle', 'wb') as file:
                pickle.dump(t_error, file)

            train_error.append(errors[-1])
            test_error.append(t_error)

        plt.plot(kList, train_error, "bo", label="Train␣errors")
        plt.plot(kList, test_error, "ro", label="Test␣errors")
        plt.xlabel("K")
        plt.ylabel("errors")
        plt.legend()
        plt.show()

    if run_for == "ct":

        kList = [2, 4, 8, 16, 32, 64]

        train_error = []
        test_error = []

        for k in kList:
            # centers, errors = lloyd_algorithm(x_train, k)
            # t_error = calculate_error(x_test, centers)

            with open('A2c-centers-' + str(k) + '.pickle','rb') as file:
                centers = pickle.load(file)

            with open('A2c-trainerrors-' + str(k) + '.pickle','rb') as file:
                errors = pickle.load(file)

            with open('A2c-testerrors-' + str(k) + '.pickle', 'rb') as file:
                t_error = pickle.load(file)

            train_error.append(errors[-1])
            test_error.append(t_error)

            print(errors[-1], t_error)

        plt.plot(kList, train_error, "b--", label="Train␣errors")
        plt.plot(kList, test_error, "r--", label="Test␣errors")
        plt.xlabel("K")
        plt.ylabel("errors")
        plt.legend()
        plt.show()


if __name__ == "__main__":
    main()
```

A4.

a. $\lambda_1 = 5.11678773, \lambda_2 = 3.74132848, \lambda_10 = 1.24272938, \lambda_30 = 0.36425572, \lambda_50 = 0.16970843$
$\sum_{i=1}^{d} \lambda_i = 52.725035$

b. Let V_k is the first k vectors of eigenvector matrix. $(X - \mathbf{1}\mu) \cdot V_k \cdot {V_k}^T$

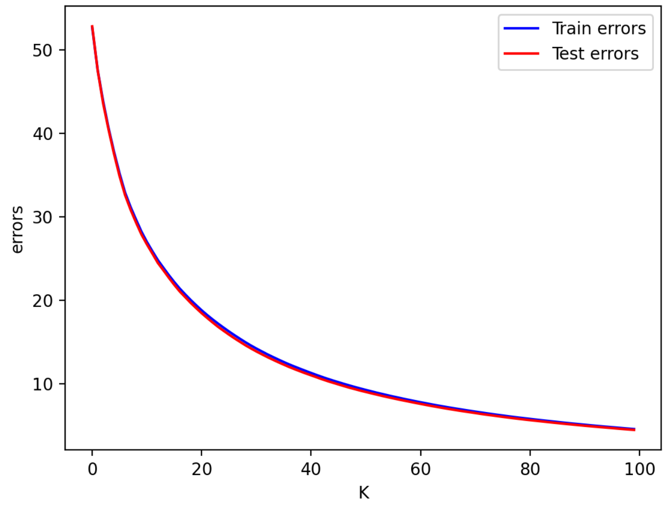

Figure 3: Reconstruction error on train and test

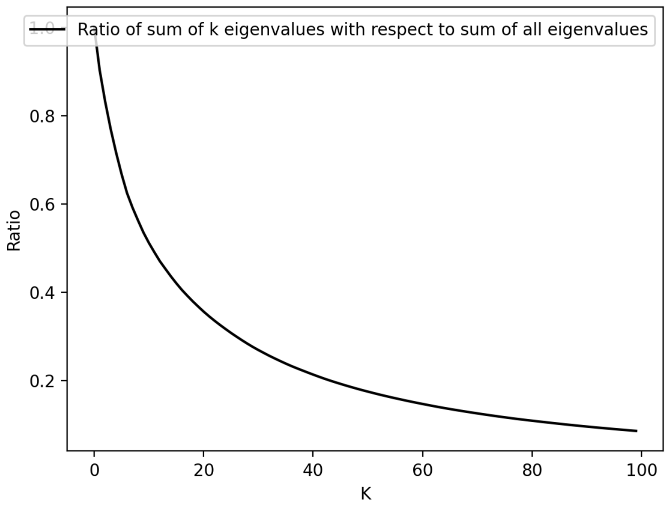

Figure 4: Plot of $1 - \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$

c.

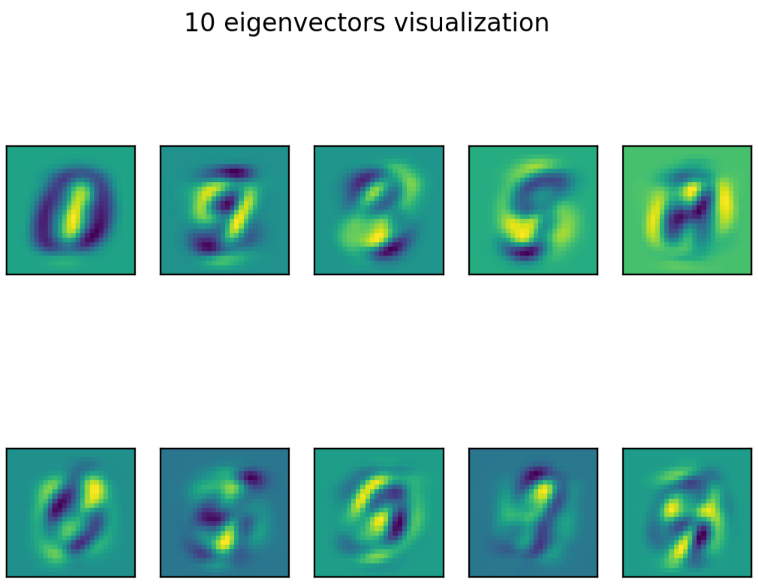

Figure 5: 10 eigenvectors as images

d.

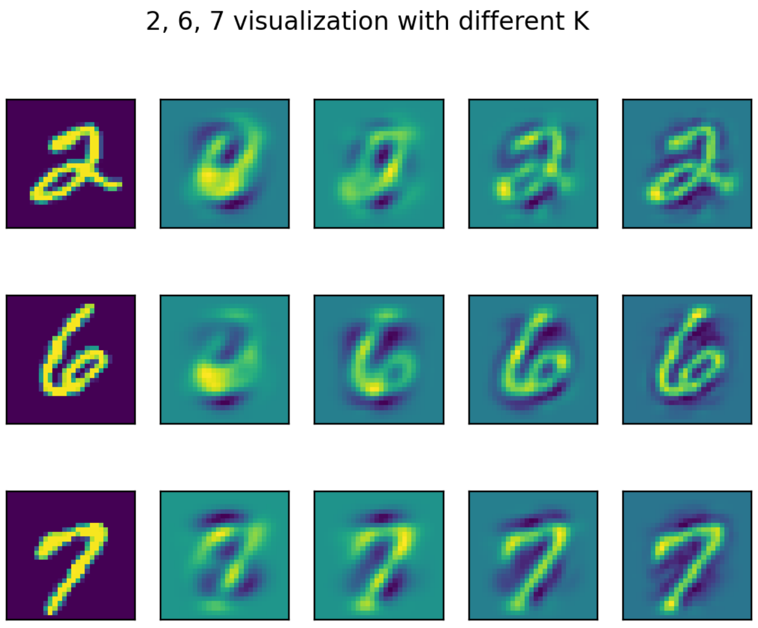

Figure 6: Reconstructed images

e.

```
# /homeworks/pca/main.py

from typing import Tuple

import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
import pickle

from utils import load_dataset, problem


@problem.tag("hw4-A")
def reconstruct_demean(uk: np.ndarray, demean_data: np.ndarray) -> np.ndarray:

    return demean_data@uk@uk.T


@problem.tag("hw4-A")
def reconstruction_error(uk: np.ndarray, demean_data: np.ndarray) -> float:

    return np.mean(np.linalg.norm(reconstruct_demean(uk, demean_data)-demean_data, axis=1)*


@problem.tag("hw4-A")
def calculate_eigen(demean_data: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:

    return np.linalg.eig(np.cov(demean_data.T, bias=True))


@problem.tag("hw4-A", start_line=2)
def main():

    (x_tr, y_tr), (x_test, _) = load_dataset("mnist")

    run_for = "e"

    if run_for == "a":
        idices = [0, 1, 9, 29, 49]
        demean_xtr = x_tr - np.mean(x_tr, axis=0)
        eigens = calculate_eigen(demean_xtr)

        with open('A4a-eigens.pickle','wb') as file:
            pickle.dump(eigens, file)

        print(eigens[0][idices])
        print(np.sum(eigens[0]))

    if run_for == "at":
        idices = [0, 1, 9, 29, 49]
        with open('A4a-eigens.pickle','rb') as file:
            eigens = pickle.load(file)

        print(eigens[0][idices])
        print(np.sum(eigens[0]))
```

```
if run_for == "c":
    with open('A4a-eigens.pickle','rb') as file:
        eigens = pickle.load(file)

    k_list = list(range(100))
    demean_xtr = x_tr - np.mean(x_tr, axis=0).reshape(1, -1)
    demean_xte = x_test - np.mean(x_test, axis=0).reshape(1, -1)

    train_errors = []
    test_errors = []
    ratios = []

    tot_lambda = np.sum(eigens[0])

    for k in k_list:
        print(k)
        train_err = reconstruction_error(eigens[1][:, :k], demean_xtr)
        test_err = reconstruction_error(eigens[1][:, :k], demean_xte)
        ratio = 1 - np.sum(eigens[0][:k])/tot_lambda

        train_errors.append(train_err)
        test_errors.append(test_err)
        ratios.append(ratio)


    with open('A4c-train_errors.pickle','wb') as file:
        pickle.dump(train_errors, file)

    with open('A4c-test_errors.pickle','wb') as file:
        pickle.dump(test_errors, file)

    with open('A4c-ratios.pickle','wb') as file:
        pickle.dump(ratios, file)

    plt.plot(k_list, train_errors, "b-", label="Train errors")
    plt.plot(k_list, test_errors, "r-", label="Test errors")
    plt.xlabel("K")
    plt.ylabel("errors")
    plt.legend()
    plt.show()

    plt.plot(k_list, ratios, "k-", label="Ratio of sum of k eigenvalues with respect to
    plt.xlabel("K")
    plt.ylabel("ratios")
    plt.legend()
    plt.show()

if run_for == "ct":
    k_list = list(range(100))

    with open('A4c-train_errors.pickle','rb') as file:
        train_errors = pickle.load(file)

    with open('A4c-test_errors.pickle','rb') as file:
            test_errors = pickle.load(file)

        with open('A4c-ratios.pickle','rb') as file:
            ratios = pickle.load(file)

        plt.plot(k_list, train_errors, "b-", label="Train errors")
        plt.plot(k_list, test_errors, "r-", label="Test errors")
        plt.xlabel("K")
        plt.ylabel("errors")
        plt.legend()
        plt.show()

        plt.plot(k_list, ratios, "k-", label="Ratio of sum of k eigenvalues with respect to 
        plt.xlabel("K")
        plt.ylabel("Ratio")
        plt.legend()
        plt.show()

    if run_for == "d":
        with open('A4a-eigens.pickle','rb') as file:
            eigens = pickle.load(file)

        fig, ax = plt.subplots(2, 5)
        for i in range(10):
            ax.ravel()[i].imshow(eigens[1][:, i].reshape((28,28)))
            ax.ravel()[i].axes.xaxis.set_visible(False)
            ax.ravel()[i].axes.yaxis.set_visible(False)
        fig.suptitle('10 eigenvectors visualization')
        plt.show()

    if run_for == "e":
        with open('A4a-eigens.pickle','rb') as file:
            eigens = pickle.load(file)

        data_idx = [5, 13, 15]
        k_val = [5, 15, 40, 100]
        demean_xtr = x_tr - np.mean(x_tr, axis=0)

        fig, ax = plt.subplots(3, 5)
        for i in range(3):
            ax.ravel()[i*5].imshow(x_tr[data_idx[i], :].reshape((28,28)))
            ax.ravel()[i*5].axes.xaxis.set_visible(False)
            ax.ravel()[i*5].axes.yaxis.set_visible(False)
            for j in range(4):
                ax.ravel()[i*5+j+1].imshow(reconstruct_demean(eigens[1][:, :k_val[j]], deme
                ax.ravel()[i*5+j+1].axes.xaxis.set_visible(False)
                ax.ravel()[i*5+j+1].axes.yaxis.set_visible(False)
        fig.suptitle('2, 6, 7 visualization with different K')
        plt.show()



if __name__ == "__main__":
    main()
```

A5.

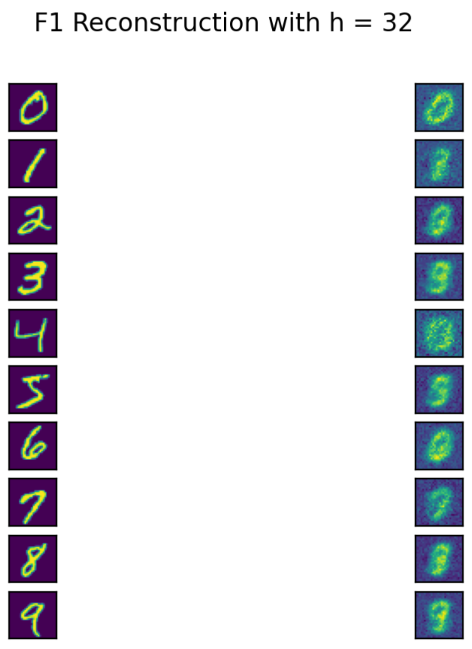

Figure 7: k=32

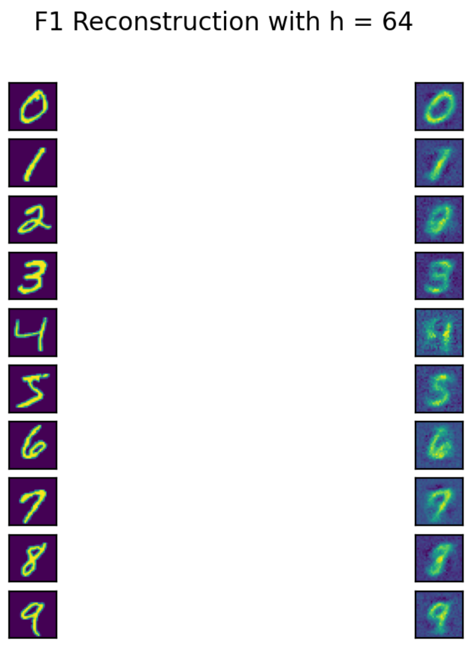

Figure 8: k=64

a.

b.

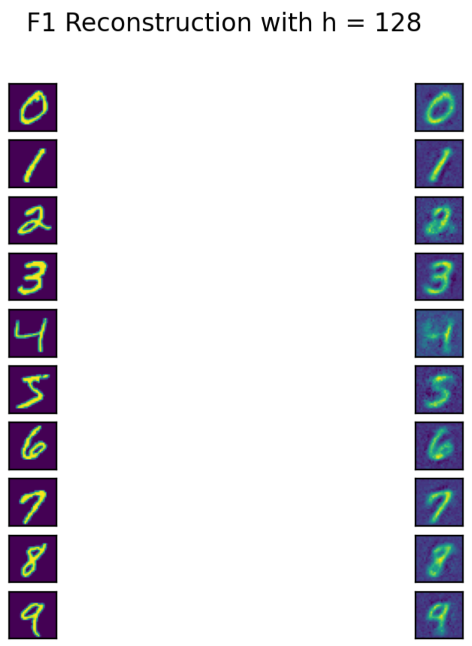

Figure 9: k=128

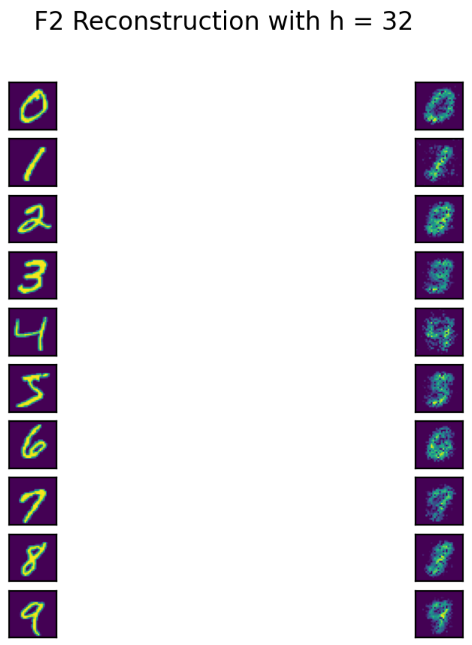

Figure 10: k=32

b.

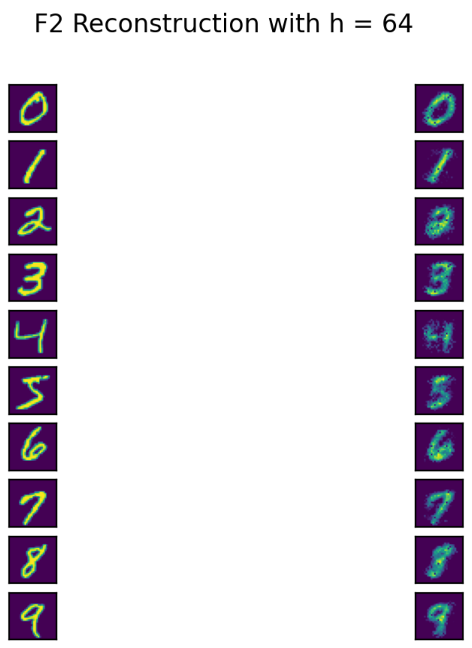

Figure 11: k=64

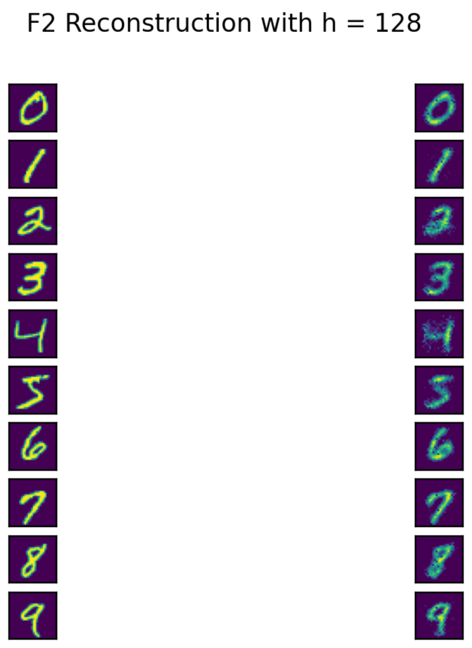

Figure 12: k=128

c. F1: 0.0236318, F2: 0.0225958

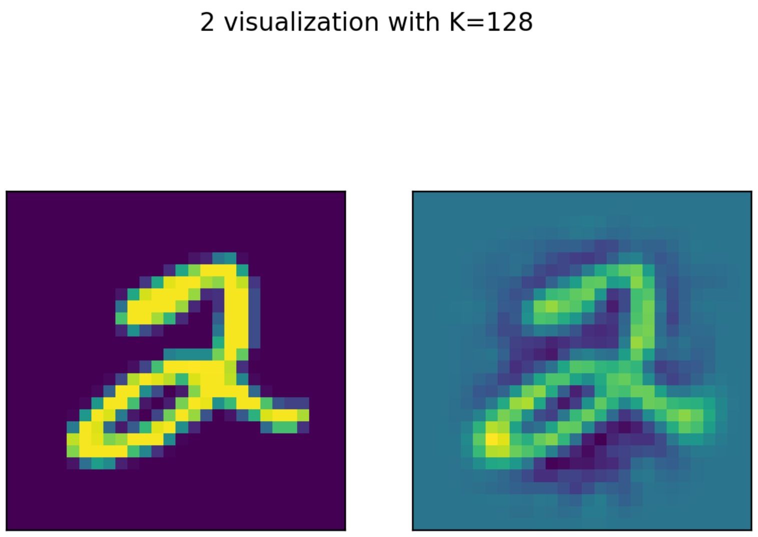

Figure 13: PCA with k=128

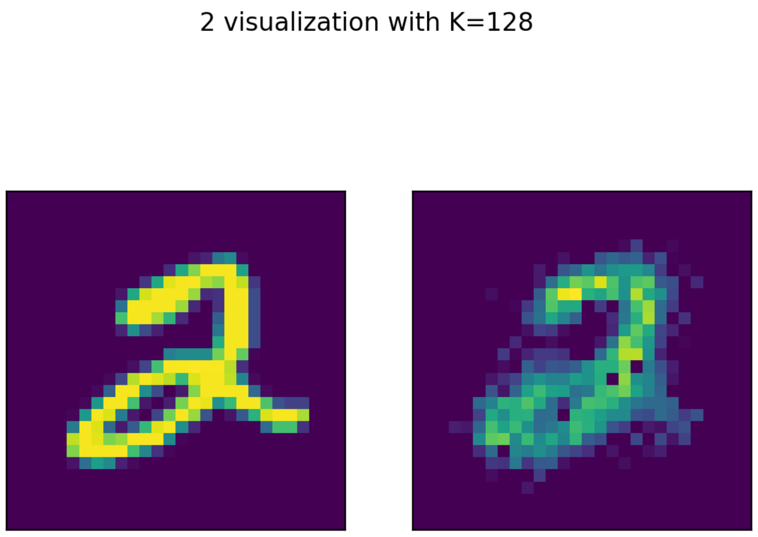

Figure 14: Autoencoder with k=128

d. In this case, it was found that the result of reconstructing using PCA was more clear and similar to the original image than that of using autoencoder.

```
# /homeworks/autoencoders/main.py

import matplotlib.pyplot as plt
import numpy as np
import torch
from torch import nn
from torch.optim import Adam
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F

from utils import load_dataset, problem


@problem.tag("hw4-A")
def F1(h: int) -> nn.Module:

    return nn.Sequential(
        nn.Linear(784, h),
        nn.Linear(h, 784)
    )


@problem.tag("hw4-A")
def F2(h: int) -> nn.Module:

    return nn.Sequential(
        nn.Linear(784, h),
        nn.ReLU(),
        nn.Linear(h, 784),
        nn.ReLU()
    )


@problem.tag("hw4-A")
def train(
    model: nn.Module, optimizer: Adam, train_loader: DataLoader, epochs: int = 40
) -> float:

    e = 0.0
    for i in range(epochs):
        for (x_batch,) in train_loader:
            optimizer.zero_grad()
            predictions = model(x_batch)
            if i == epochs -1:
                loss = nn.MSELoss()
                loss = loss(predictions, x_batch)
                loss.backward()
                e = loss.item()
            optimizer.step()
    return e


@problem.tag("hw4-A")
def evaluate(model: nn.Module, loader: DataLoader) -> float:
    loss = nn.MSELoss()
    model.eval()
    test_loss = 0.0
    with torch.no_grad():
        for (x_batch,) in loader:
            x_batch_pred = model(x_batch)
            batch_loss = loss(x_batch_pred, x_batch)
            test_loss = test_loss + batch_loss.item()
        test_loss = test_loss / len(loader)
        return test_loss

@problem.tag("hw4-A", start_line=9)
def main():

    (x_train, y_train), (x_test, _) = load_dataset("mnist")
    x = torch.from_numpy(x_train).float()
    x_test = torch.from_numpy(x_test).float()

    # Neat little line that gives you one image per digit for visualization in parts a and b
    images_to_visualize = x[[np.argwhere(y_train == i)[0][0] for i in range(10)]]


    train_loader = DataLoader(TensorDataset(x), batch_size=32, shuffle=True)
    test_loader = DataLoader(TensorDataset(x_test), batch_size=32, shuffle=True)

    run_for = "c"

    hs = [32, 64, 128]

    if run_for =="a":
        for h in hs:
            model = F1(h)
            optimizer = torch.optim.Adam(params=model.parameters(), lr=5*10**(-5))
            e = train(model, optimizer, train_loader, 40)

            fig, ax = plt.subplots(10, 2)
            for i in range(10):
                for j in range(2):
                    if j == 0:
                        ax.ravel()[i*2+j].imshow(images_to_visualize[i].reshape((28,28)))
                    else:
                        ax.ravel()[i*2+j].imshow(model(images_to_visualize[i]).detach().num
                    ax.ravel()[i*2+j].axes.xaxis.set_visible(False)
                    ax.ravel()[i*2+j].axes.yaxis.set_visible(False)
            fig.suptitle('F1␣Reconstruction␣with␣h␣=␣' + str(h))
            plt.show()

    if run_for =="b":
        for h in hs:
            model = F2(h)
            optimizer = torch.optim.Adam(params=model.parameters(), lr=5*10**(-5))
            e = train(model, optimizer, train_loader, 40)

            fig, ax = plt.subplots(10, 2)
            for i in range(10):
                for j in range(2):
                    if j == 0:
                        ax.ravel()[i*2+j].imshow(images_to_visualize[i].reshape((28,28)))
                    else:
                        ax.ravel()[i*2+j].imshow(model(images_to_visualize[i]).detach().nump
                    ax.ravel()[i*2+j].axes.xaxis.set_visible(False)
                    ax.ravel()[i*2+j].axes.yaxis.set_visible(False)
            fig.suptitle('F2 Reconstruction with h = ' + str(h))
            plt.show()

    if run_for =="c":
        for m in [F1, F2]:
            model = m(128)
            optimizer = torch.optim.Adam(params=model.parameters(), lr=5*10**(-5))
            train(model, optimizer, train_loader, 40)
            val_test = evaluate(model, test_loader)
            print(val_test)



if __name__ == "__main__":
    main()
```

A6.

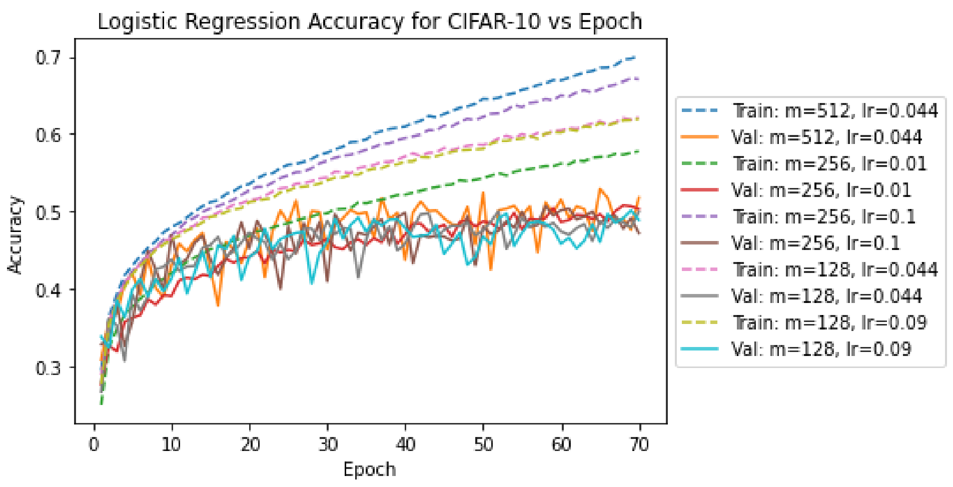

Figure 15: Plot of training and validation accuracy

Learning rate	Hidden layer dimensions
0.044	512
0.01	256
0.1	256
0.044	128
0.09	128

a. I used grid search over learning rate with fixing hidden layer dimension.
Best performing hyperparameters are learning rate 0.044 with hidden layer dimension 512 with max val$_{a}ccuracy$0.507628

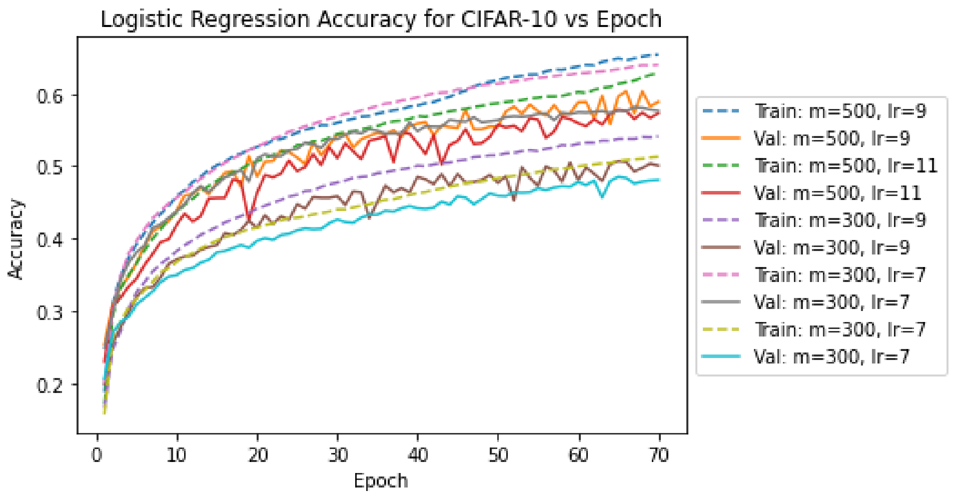

Figure 16: Plot of training and validation accuracy

b. I used grid search over learning rate with fixing hidden layer dimension.
Best performing hyperparameters are learning rate 0.012 with M, N and K value each 500, 9 and 12 with max val$_{a}ccuracy$0.602493

Learning rate	M value	N value	K value
0.12	500	9	12
0.08	500	11	11
0.1	300	9	12
0.05	300	7	5
0.02	300	7	4

```
import torch
from torch import nn

from typing import Tuple, Union, List, Callable
from torch.optim import SGD
import torchvision
from torch.utils.data import DataLoader, TensorDataset, random_split
import matplotlib.pyplot as plt
from tqdm import tqdm, trange
import math
import torch.nn.functional as F

assert torch.cuda.is_available(), "GPU is not available, check the directions above (or disa

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print(DEVICE)  # this should print out CUDA

train_dataset = torchvision.datasets.CIFAR10("./data", train=True, download=True, transform=
test_dataset = torchvision.datasets.CIFAR10("./data", train=False, download=True, transform=

batch_size = 128

train_dataset, val_dataset = random_split(train_dataset, [int(0.9 * len(train_dataset)), int

# Create separate dataloaders for the train, test, and validation set
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True
)

val_loader = DataLoader(
    val_dataset,
    batch_size=batch_size,
    shuffle=True
)

test_loader = DataLoader(
    test_dataset,
    batch_size=batch_size,
    shuffle=True
)

imgs, labels = next(iter(train_loader))
print(f"A single batch of images has shape: {imgs.size()}")
example_image, example_label = imgs[0], labels[0]
c, w, h = example_image.size()
```

```
print(f"A single RGB image has {c} channels, width {w}, and height {h}.")

# This is one way to flatten our images
batch_flat_view = imgs.view(-1, c * w * h)
print(f"Size of a batch of images flattened with view: {batch_flat_view.size()}")

# This is another equivalent way
batch_flat_flatten = imgs.flatten(1)
print(f"Size of a batch of images flattened with flatten: {batch_flat_flatten.size()}")

# The new dimension is just the product of the ones we flattened
d = example_image.flatten().size()[0]
print(c * w * h == d)

# View the image
t =  torchvision.transforms.ToPILImage()
plt.imshow(t(example_image))

# These are what the class labels in CIFAR-10 represent. For more information,
# visit https://www.cs.toronto.edu/~kriz/cifar.html
classes = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog",
           "horse", "ship", "truck"]
print(f"This image is labeled as class {classes[example_label]}")

def hidden_relu_model(M) -> nn.Module:
  """Instantiate a hidden_relu model and send it to device."""
  model =  nn.Sequential(
             nn.Flatten(),
             nn.Linear(d, M),
             nn.ReLU(),
             nn.Linear(M, 10),
             nn.Softmax()
           )
  return model.to(DEVICE)

def conv_model(M, k, N) -> nn.Module:
model =  nn.Sequential(
        nn.Conv2d(3, M, k),
        nn.ReLU(),
        nn.MaxPool2d(N),
        nn.Flatten(),
        nn.Linear(M*(math.floor((33-k)/N))**2, 10),
        nn.Softmax()
    )
return model.to(DEVICE)

def train(
    model: nn.Module, optimizer: SGD,
    train_loader: DataLoader, val_loader: DataLoader,
    epochs: int = 20
    )-> Tuple[List[float], List[float], List[float], List[float]]:
  """
  Trains a model for the specified number of epochs using the loaders.

  Returns:
    Lists of training loss, training accuracy, validation loss, validation accuracy for each
"""

  loss = nn.CrossEntropyLoss()
  train_losses = []
  train_accuracies = []
  val_losses = []
  val_accuracies = []
  for e in range(epochs):
    model.train()
    train_loss = 0.0
    train_acc = 0.0

    # Main training loop; iterate over train_loader. The loop
    # terminates when the train loader finishes iterating, which is one epoch.
    for (x_batch, labels) in train_loader:
    x_batch, labels = x_batch.to(DEVICE), labels.to(DEVICE)
    optimizer.zero_grad()
    labels_pred = model(x_batch)
    batch_loss = loss(labels_pred, labels)
    train_loss = train_loss + batch_loss.item()

    labels_pred_max = torch.argmax(labels_pred, 1)
    batch_acc = torch.sum(labels_pred_max == labels)
    train_acc = train_acc + batch_acc.item()

    batch_loss.backward()
    optimizer.step()
    train_losses.append(train_loss / len(train_loader))
    train_accuracies.append(train_acc / (batch_size * len(train_loader)))

    # Validation loop; use .no_grad() context manager to save memory.
    model.eval()
    val_loss = 0.0
    val_acc = 0.0

    with torch.no_grad():
    for (v_batch, labels) in val_loader:
        v_batch, labels = v_batch.to(DEVICE), labels.to(DEVICE)
        labels_pred = model(v_batch)
        v_batch_loss = loss(labels_pred, labels)
        val_loss = val_loss + v_batch_loss.item()

        v_pred_max = torch.argmax(labels_pred, 1)
        batch_acc = torch.sum(v_pred_max == labels)
        val_acc = val_acc + batch_acc.item()
    val_losses.append(val_loss / len(val_loader))
    val_accuracies.append(val_acc / (batch_size * len(val_loader)))

  return train_losses, train_accuracies, val_losses, val_accuracies

def parameter_search(train_loader: DataLoader,
                     val_loader: DataLoader,
                     model_fn:Callable[[], nn.Module]) -> float:
  """
  Parameter search for our linear model using SGD.

  Args:
    train_loader: the train dataloader.
    val_loader: the validation dataloader.
    model_fn: a function that, when called, returns a torch.nn.Module.

  Returns:
    The learning rate with the least validation loss.
    NOTE: you may need to modify this function to search over and return
      other parameters beyond learning rate.
  """
  num_iter = 10  # This will likely not be enough for the rest of the problem.
  best_loss = torch.inf
  best_lr = 0.0

  lrs = torch.linspace(10 ** (-1), 10 ** (0), num_iter)

  for lr in lrs:
    print(f"trying learning rate {lr}")
    model = model_fn(512)
    optim = SGD(model.parameters(), lr)

    train_loss, train_acc, val_loss, val_acc = train(
        model,
        optim,
        train_loader,
        val_loader,
        epochs=20
        )

    if min(val_loss) < best_loss:
    best_loss = min(val_loss)
    best_lr = lr

  return best_lr

best_lr = parameter_search(train_loader, val_loader, hidden_relu_model)

model = hidden_relu_model(256)
optimizer = SGD(model.parameters(), 0.01)

# We are only using 20 epochs for this example. You may have to use more.
train_loss, train_accuracy, val_loss, val_accuracy = train(
    model, optimizer, train_loader, val_loader, 70)

def evaluate(
    model: nn.Module, loader: DataLoader
) -> Tuple[float, float]:
  """Computes test loss and accuracy of model on loader."""
  loss = nn.CrossEntropyLoss()
  model.eval()
  test_loss = 0.0
  test_acc = 0.0
  with torch.no_grad():
    for (batch, labels) in loader:
      batch, labels = batch.to(DEVICE), labels.to(DEVICE)
      y_batch_pred = model(batch)
      batch_loss = loss(y_batch_pred, labels)
      test_loss = test_loss + batch_loss.item()

      pred_max = torch.argmax(y_batch_pred, 1)
      batch_acc = torch.sum(pred_max == labels)
      test_acc = test_acc + batch_acc.item()
    test_loss = test_loss / len(loader)
    test_acc = test_acc / (batch_size * len(loader))
    return test_loss, test_acc

test_loss, test_acc = evaluate(model, test_loader)
print(f"Test␣Accuracy:␣{test_acc}")

params = [(500, 9, 12, 0.12),
          (500, 11, 11, 0.08),
          (300, 9, 12, 0.1),
          (300, 7, 5, 0.05),
          (300, 7, 5, 0.02)]

epochs = range(1,71)

train_accs = []
val_accs = []

plt.figure()

for p in params:
  model = conv_model(p[0], p[1], p[2])
  optimizer = SGD(model.parameters(), p[3])

  train_loss, train_accuracy, val_loss, val_accuracy = train(
    model, optimizer, train_loader, val_loader, 70)

  train_accs.append(train_accuracy)
  val_accs.append(val_accuracy)

plt.figure()

for i in range(len(params)):
  plt.plot(epochs, train_accs[i], '--', label=f"Train:␣m={params[i][0]},␣lr={params[i][1]}")
  plt.plot(epochs, val_accs[i], label=f"Val:␣m={params[i][0]},␣lr={params[i][1]}")

plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(loc='center␣left',bbox_to_anchor=(1,0.5))
plt.title("Logistic␣Regression␣Accuracy␣for␣CIFAR-10␣vs␣Epoch")
plt.show()
```