Homework #4

CSE 546: Machine Learning
Cassia Cai
December 8, 2022

Collaborators: Apoorva Kalaskar, Madeleine Grunde-McLaughlin, Cadence Pearce, Javier Marin, and Sri Sakthinarayanan

Conceptual Questions

A1.

a. TRUE. rank(X) is the number of non-zero eigenvalues and at maximum is equal to d. A linear combination of these vectors can represent the other entries of X. Thus, if we project our data onto a k-dimensional subspace using PCA, our projection will have zero reconstruction error.

b. FALSE. If we suppose that an $n \times n$ matrix X has a singular value decomposition of $USV^\top$, where S is a diagonal $n \times n$ matrix, then the COLUMNS of V are equal to the eigenvectors of $X^\top X$.

c. FALSE. Choosing k to minimize the k-means objective will not necessarily find meaningful clusters. K-means optimizes some mathematical statistic and would need information about what we are modeling to decide whether the clusters are meaningful.

d. FALSE. The singular value decomposition of a matrix is NOT unique. The singular values are unique, but the columns of U and V are only unique up to a complex sign.

e. FALSE. The rank of a square matrix equals the number of its non-zero eigenvalues. We can be repeated non-zero eigenvalues.

Think before you train

A2.

a. **Scenario: Disease Susceptibility Predictor**

The dataset is tabular (demographic information, location information, risk factors, and whether a person has the disease or not). We can make x_{train} (which consists of the demographic information, location information, and risk factors), and y_{train}, which is whether a person has the disease or not. We can hot-one encode our demographic data (this is gender, age, ethnicity, income, etc.). This data can be used directly or normalize. Because this is personal demographic information, we may encounter instances of incomplete data so we may need to filter this out in an appropriate way. We can approach this problem using logistic regression for binary classification. We should also evaluate the binary classifier. We can use simple neural networks, and we would not need to remove redundant features should there be any and could handle missing inputs. Our output would be some metric (probably a scalar representing probability) about the susceptibility of this person to the disease.

b. Datasets describing crime have many shortcomings in describing the entire landscape of illegal behavior in a city, and that these shortcomings often fall disproportionately on minority communities. Some of these shortcomings include that crimes are reported at different rates in different neighborhoods, that police respond differently to the same crime reported or observed in different neighborhoods, and that police spend more time patrolling in some neighborhoods than others. What real-world implications might follow from ignoring these issues?

Some real-world implications that might follow from ignoring the fact that datasets describing crime have many shortcomings is that certain neighborhoods may be penalized and viewed as unsafe to a large extent when they are not when compared taking into consideration that in different neighborhoods, different types of crimes are reported at different rates and that police respond to these crimes differently. These shortcomings in the dataset can then affect high-level decisions about infrastructure (for example, where schools are built and who teaches at those schools, where resources are allocated and what those kinds of resources are, where people want to move, the housing market, the stigma associated with the neighborhood, etc.). In different neighborhoods, the definition of crime can vary. Does a fined infraction count as a crime? Is speeding? What about different drugs that are legal in some places but not others? Crime statistics and the shortcomings associated with crime statistics disproportionately affect minority communities and can instill fear of and in those communities, which then affects how that community grows (in terms of resource allocation, whether or not it is an attractive neighborhood to which people want to move and are willing to pay to move) and is viewed.

c. Briefly describe (1) some potential shortcomings of your training process from the aforementioned scenario that may result in your algorithm having different accuracy on different populations, and (2) how you may modify your procedure to address these shortcomings.

In my training process, I stated that we may encounter instances of incomplete data and would need a way to either filter this out or consider it. One line of inquiry is to look at which inputs participants are not providing (so which features we have the least amount of data for). This could be some demographic data that participants are either not comfortable sharing or likely think will have some sort of negative impact. We should also check for the correlation of some features, so for example more information about risk factors is helpful. Risk factors, I think, include lifestyle factors such as aging, nutrition levels, infection, and exposure to toxicants. Exposure to toxicants, nutrition levels, stress levels, and income (among some others) could be linked to perhaps location. So some factors that the algorithm associates with the disease may be redundant/overemphasized/underemphasized. We also need to keep in mind that correlation is not causation. I would modify my procedure by doing more feature engineering (what if we have too many features? Then, we probably want to reduce the dimensions with PCA). Another option could be to modify our goal and create different models for different populations and evaluating these models. This may give us different information about genetic disposition and may change our accuracies. In general, our approach to genetic profiling should be thoughtful because it could lead to different patient/provider interactions and access to health and life insurance.

k-means clustering

A3. Given a dataset $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$ and an integer $1 \leq k \leq n$, recall the following k-means objective function

$$\min_{\pi_1,...,\pi_k} \sum_{i=1}^{k} \sum_{j \in \pi_i} \|\mathbf{x}_j - \mu_i\|_2 \ , \quad \mu_i = \frac{1}{|\pi_i|} \sum_{j \in \pi_i} \mathbf{x}_j \ . \tag{1}$$

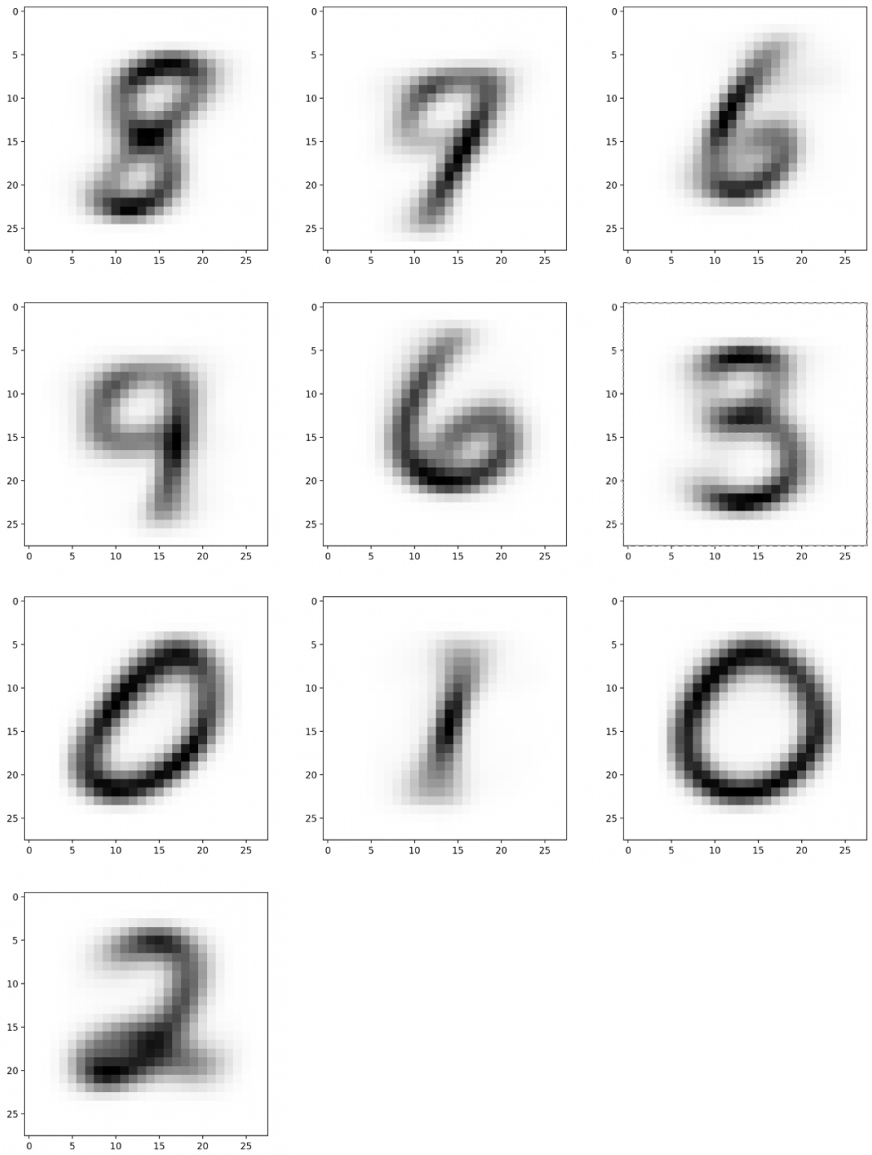
Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, ..., n\}$. The objective (1) is NP-hard[1] to find a global minimizer of. Nevertheless, Lloyd's algorithm (discussed in lecture) typically works well in practice.

a. Implement Lloyd's algorithm for solving the k-means objective (1).

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

def calculate_centers(data, classifications, num_centers):
    idx_sort = np.argsort(classifications)
    sorted_array = classifications[idx_sort]
    sorted_data = data[idx_sort]
    vals, idx_start = np.unique(sorted_array, return_counts=False, return_index=True)
    split_sorted_data = np.split(sorted_data, idx_start[1:])
    centers = np.array([np.mean(cluster,axis=0) for cluster in split_sorted_data])
    return centers

def cluster_data(data, centers):
    distances = [np.linalg.norm(data - center, axis=1) for center in centers]
    distance_array_t = np.asarray(distances).transpose()
    return np.argmin(distance_array_t, axis=1)

def calculate_error(data, centers):
    distances = np.zeros((data.shape[0], centers.shape[0]))
    for idx, center in enumerate(centers):
        distances[:, idx] = np.sqrt(np.sum((data - center) ** 2, axis=1))
    return np.mean(np.min(distances, axis=1))

def lloyd_algorithm(data, num_centers, epsilon):
    old_centers = data[np.random.permutation(np.arange(len(data)))[:num_centers]]
    old_clusters = cluster_data(data,old_centers)
    current_centers = calculate_centers (data, old_clusters, num_centers)
    current_clusters = cluster_data(data,current_centers)
    center_dist = np.max(np.abs(current_centers - old_centers))
    errors_array = []
    while center_dist > epsilon:
        print(center_dist)
        old_centers = current_centers
        new_classifications = cluster_data(data, old_centers)
        current_centers = calculate_centers(data, new_classifications, num_centers)
        center_dist = np.max(np.abs(current_centers - old_centers))
        errors = calculate_error(data, current_centers)
        errors_array.append(errors)
    return current_centers,errors_array
```

[1]To be more precise, it is both NP-hard in d when $k = 2$ and k when $d = 2$. See the references on the Wikipedia page for k-means for more details.

b. Run Lloyd's algorithm on the *training* dataset of MNIST with $k = 10$. Show the image representing the center of each cluster, as a set of k 28×28 images.



```
if __name__ == "__main__":
    from k_means import calculate_error, lloyd_algorithm
else:
    from .k_means import lloyd_algorithm, calculate_error
import matplotlib.pyplot as plt; import numpy as np
from utils import load_dataset, problem

def main():
    (x_train, _), (x_test, _) = load_dataset("mnist")
    k = 10; centers, errors = lloyd_algorithm(x_train, k)
    for i in range(k):
        image = centers[i].reshape((28,28))
        fig = plt.figure; plt.imshow(image, cmap='Greys'); plt.show()

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

PCA on MNIST

A4. Let's do PCA on MNIST dataset and reconstruct the digits in the dimensionality-reduced PCA basis. Compute your PCA basis using the training dataset, and evaluate the quality of the basis on the test set. We have $n_{train} = 50,000$ training examples of size 28×28. Begin by flattening each example to a vector to obtain $X_{train} \in \mathbb{R}^{50,000 \times d}$ and $X_{test} \in \mathbb{R}^{10,000 \times d}$ for $d = 784$.

Let $\mu \in \mathbb{R}^d$ denote the average of the training examples in X_{train}, i.e., $\mu = \frac{1}{n_{train}} X_{train}^\top \mathbf{1}$. Now let $\Sigma = (X_{train} - \mathbf{1}\mu^\top)^\top (X_{train} - \mathbf{1}\mu^\top)/50000$ denote the sample covariance matrix of the training examples, and let $\Sigma = UDU^T$ denote the eigenvalue decomposition of Σ.

a. If λ_i denotes the ith largest eigenvalue of Σ, what are the eigenvalues λ_1, λ_2, λ_{10}, λ_{30}, and λ_{50}? What is the sum of eigenvalues $\sum_{i=1}^{d} \lambda_i$?

$\lambda_1 = 5.11678$, $\lambda_2 = 3.74133$, $\lambda_{10} = 1.24273$, $\lambda_{30} = 0.364256$, $\lambda_{50} = 0.169708$

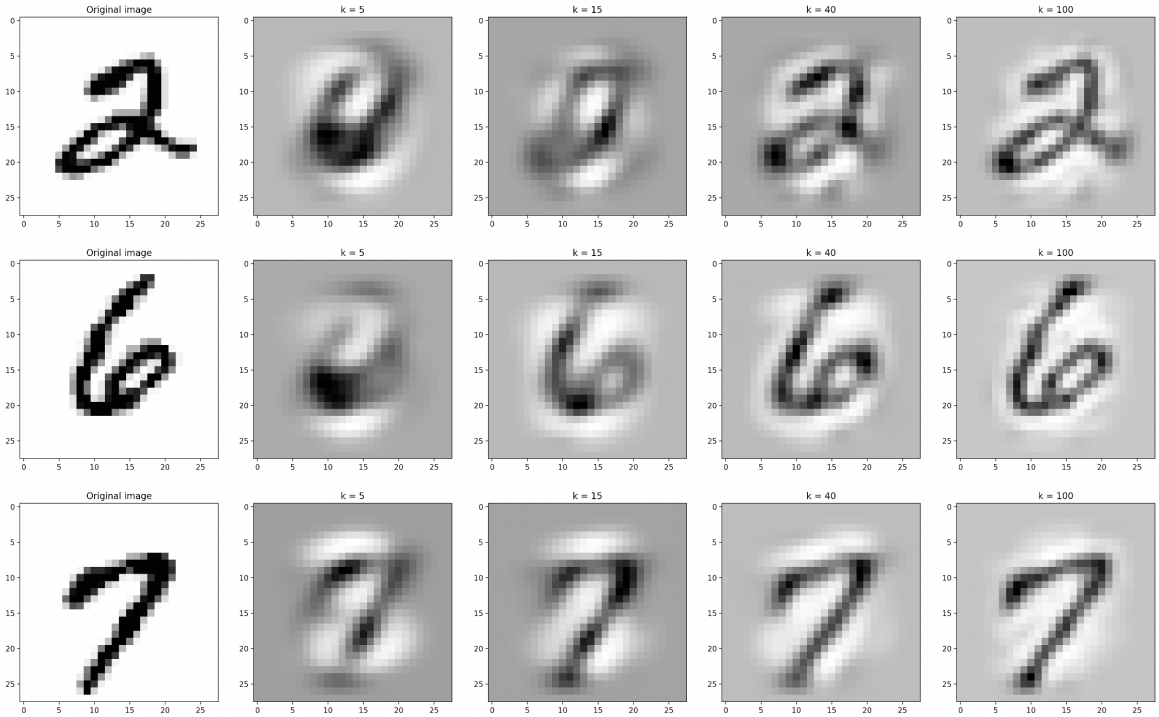
$$\sum_{i=1}^{d} \lambda_i = 52.7250$$

b. Let $x \in \mathbb{R}^d$ and $k \in 1, 2, \ldots, d$. The formula for the rank-k PCA approximation of x is:

$$\hat{x} \approx \mu + (x - \mu^T) V_k V_k^T$$

We can thus approximate an x as $\hat{x}$ with μ and the first k eigenvalue-eigenvectors where in the above equation V_k columns are formed by the first k eigenvectors.

c. We see that the for k = 5, this is not enough eigenvectors to reconstruct the digits. However, it does show where the different digits overlap in the image space (the shared characteristics). As we increase the number of eigenvectors, we begin to see the the portions of the digits that are different (so inner and outer). The less important eigenvectors capture the distinctions. At k = 100, the digit edges start looking clearer.



```
from typing import Tuple
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
from utils import load_dataset, problem

def reconstruct_demean(uk, demean_data):
    return np.dot(np.dot(demean_data, uk), uk.T)

def reconstruction_error(uk, demean_data):
    tmp = demean_data - reconstruct_demean(uk, demean_data)
    res = np.mean(np.linalg.norm(tmp, axis=1) ** 2)
    return res

def calculate_eigen(demean_data):
    x_tr = demean_data
    n, d = x_tr.shape
    I = np.ones((n,1))
    mu = np.dot(x_tr.T, I)/n
    sigma_val = x_tr - np.dot(I, mu.T)
    sigma = np.dot(sigma_val.T, sigma_val)/n
    eigenvalues, eigenvectors = np.linalg.eigh(sigma)
    indx = np.argsort(-1*eigenvalues)
    eigenvalues = eigenvalues[indx]
    eigenvectors = eigenvectors[:, np.argsort(eigenvalues)]
    return eigenvalues, eigenvectors

def main():
    (x_tr, y_tr), (x_test, _) = load_dataset("mnist")
    eigenvalues, eigenvectors = calculate_eigen(x_tr)
    eigenvaluesum = np.sum(eigenvalues)
    print(eigenvaluesum)
    for i in (1, 2, 10, 30, 50):
        print (f"{i}th:{eigenvalues[i-1]}")

    digits = [2,6,7]
    idxDigits = [np.where(y_tr== digit)[0][0] for digit in digits] # get the first inde
    k = [5,15,40,100] # number of components

    mu_array = np.stack((mu,) * n, axis=0)
    X_meaned = np.subtract(x_tr,np.squeeze(mu_array))

    for i in range(len(idxDigits)):
         original = x_tr[idxDigits][i].reshape((28,28))
         plt.imshow(original,cmap='Greys')
         plt.title('Original_image'); plt.show()

    reconstruct = reconstruct_demean(eigenvectors[:,:100], X_meaned)
    # change to 5, 15, 40,100
    for i in range(len(idxDigits)):
        original = np.real(reconstruct[idxDigits][i].reshape((28,28)))
        plt.imshow(original, cmap='Greys'); plt.show()

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

Image Classification on CIFAR-10

A5. We will explore different deep learning architectures for image classification on the CIFAR-10 dataset.

a. **Fully-connected output, 1 fully-connected hidden layer:** this network has one hidden layer denoted as $x^{\text{hidden}} \in \mathbb{R}^M$ where M will be a hyperparameter you choose. The nonlinearity applied to the hidden layer will be the `relu` ($\text{relu}(x) = \max\{0, x\}$. This network can be written as

$$x^{out} = W_2 \text{relu}(W_1(x^{in}) + b_1) + b_2$$

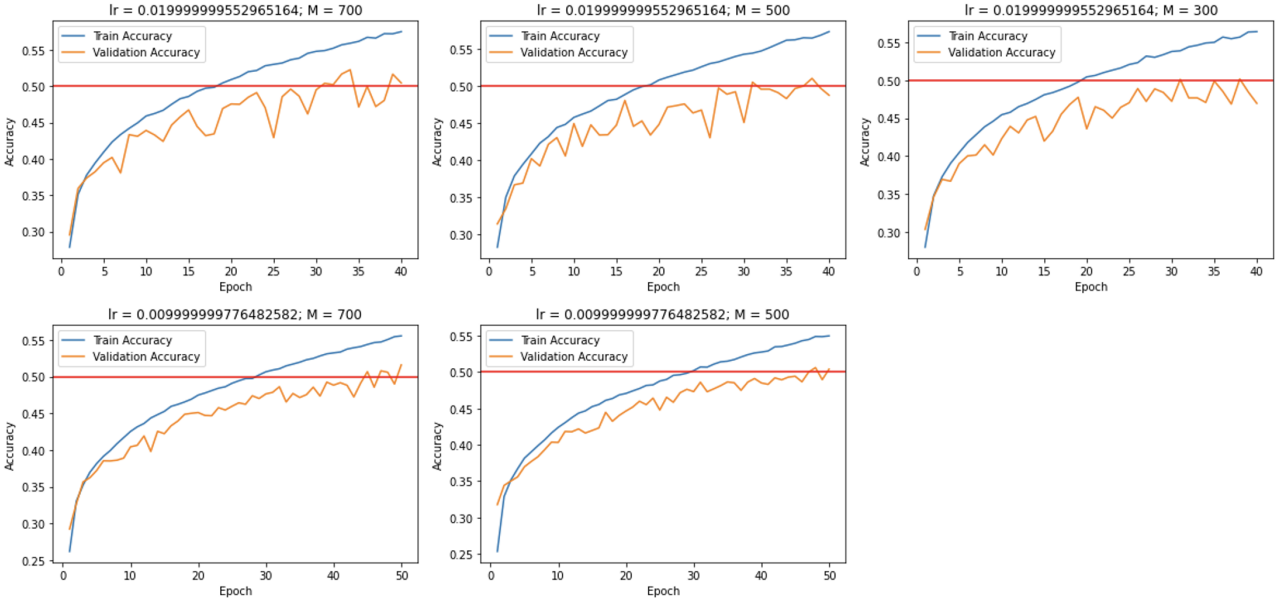
where $W_1 \in \mathbb{R}^{M \times 3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10 \times M}$, $b_2 \in \mathbb{R}^{10}$. Tune the different hyperparameters and train for a sufficient number of epochs to achieve a *validation accuracy* of at least 50%. Provide the hyperparameter configuration used to achieve this performance.

The hyperparameter configurations I searched over using stochastic gradient descent are shown below.

```
learning_rates = [1e-6, 1.1112e-2, 2.2223e-2, 3.3334e-2, 4.4445e-2,
5.5556e-2, 6.6667e-2, 7.7778e-2, 8.8889e-2, 1e-1]
hidden_sizes = [100, 300, 500, 700]
```

My top 5 hyperparameter configurations are:

learning rate	hidden size	test accuracy
0.019999999552965164	700	0.5034612341772152
0.019999999552965164	500	0.49535205696202533
0.019999999552965164	300	0.47339794303797467
0.009999999776482582	700	0.5183939873417721
0.009999999776482582	500	0.5099881329113924



b. **Convolutional layer with max-pool and fully-connected output:** for a convolutional layer W_1 with filters of size $k \times k \times 3$, and M filters (reasonable choices are $M = 100$, $k = 5$), we have that $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$.

- Each convolution will have its own offset applied to each of the output pixels of the convolution; we denote this as $\text{Conv2d}(x^{\text{in}}, W) + b_1$ where b_1 is parameterized in $\mathbb{R}^M$. Apply a `relu` activation to the result of the convolutional layer.

- Next, use a max-pool of size $N \times N$ (a reasonable choice is $N = 14$ to pool to 2×2 with $k = 5$) we have that $\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{in}}, W_1) + b_1)) \in \mathbb{R}^{\lfloor \frac{33-k}{N} \rfloor \times \lfloor \frac{33-k}{N} \rfloor \times M}$.
- We will then apply a fully-connected layer to the output to get a final network given as

$$x^{output} = W_2(\text{MaxPool}(\text{relu}(\text{Conv2d}(x^{\text{input}}, W_1) + b_1))) + b_2$$

  where $W_2 \in \mathbb{R}^{10 \times M(\lfloor \frac{33-k}{N} \rfloor)^2}$, $b_2 \in \mathbb{R}^{10}$.

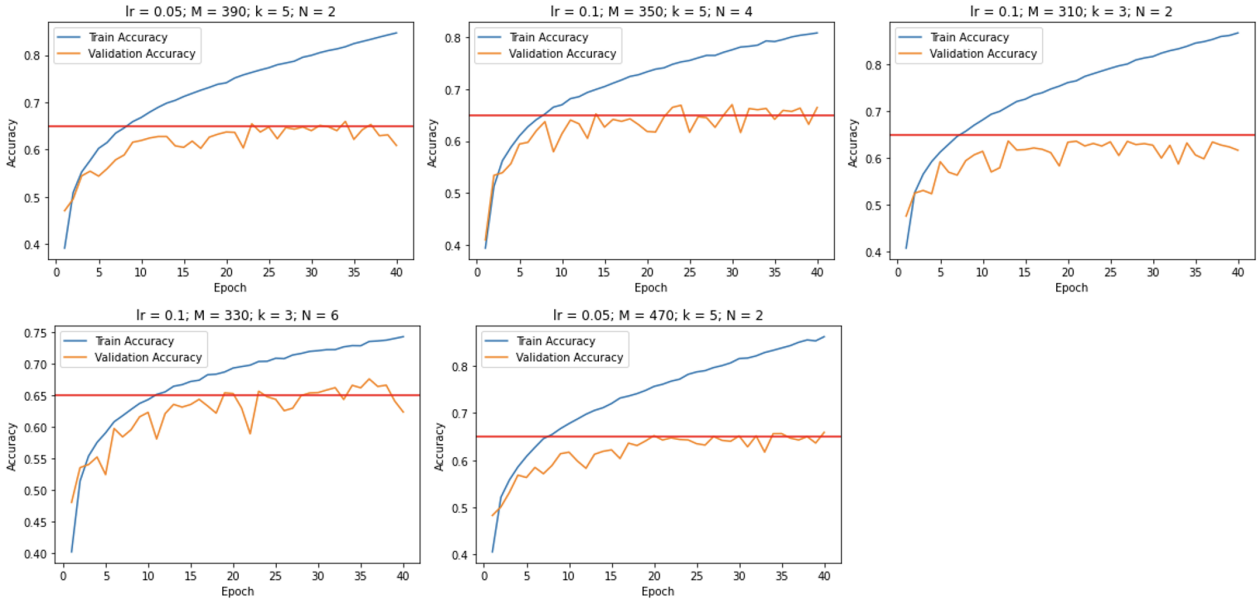
The parameters M, k, N (in addition to the step size and momentum) are all hyperparameters, but you can choose a reasonable value. Tune the different hyperparameters (number of convolutional filters, filter sizes, dimensionality of the fully-connected layers, step size, etc.) and train for a sufficient number of epochs to achieve a *validation accuracy* of at least 65%. Provide the hyperparameter configuration used to achieve this performance. Make sure to save the best model during the hyperparameter tuning so that you can evaluate test accuracy without retraining.

The hyperparameter configurations I searched over using stochastic gradient descent are shown below.

```
lrs = [1e-7, 5e-7, 1e-6, 5e-6, 1e-5, 5e-5, 1e-4,
    5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1]
M = [300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420,
       430, 440, 450, 460, 470, 480, 490]
k = [3, 4, 5, 6]
N = [2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

My top 5 hyperparameter configurations are:

learning rate	hidden size	k	N	test accuracy
0.05	390	5	2	0.62262658227848
0.10	350	5	4	0.6731606012658228
0.10	310	3	2	0.631131329113924
0.10	330	3	6	0.6203520569620253
0.05	470	5	2	0.6203520569620253




```
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

assert torch.cuda.is_available(), "GPU not available, check the directions above"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
print(DEVICE)

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

batch_size = 128
train_dataset, val_dataset = random_split(train_dataset, [int(0.9 * len(train_dataset))
    int( 0.1 * len(train_dataset))])
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)

def linear_model() -> nn.Module:
  model =  nn.Sequential(
            nn.Flatten(),
            nn.Linear(d, 10))
  return model.to(DEVICE)

def A5a_model(m) -> nn.Module:
  model = nn.Sequential(
      nn.Flatten(),
      nn.Linear(32*32*3, m),
      nn.ReLU(),
      nn.Linear(m, 10))
  return model.to(DEVICE)

def A5b_model(m,k,N) -> nn.Module:
  model = nn.Sequential(
      nn.Conv2d(3, m, k),
      nn.ReLU(),
      nn.MaxPool2d(N),
      nn.Flatten(),
      nn.Linear(m * (int((33 - k)/N)) ** 2, 10))
  return model.to(DEVICE)

def train(
    model: nn.Module, optimizer: SGD,
    train_loader: DataLoader, val_loader: DataLoader,
    epochs: int = 20
    ):
  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
    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]):
num_iter = 10
best_loss = torch.inf
best_lr = 0.0
best_hidden_size = 0.0
best_nM = 1.0
best_k = 1.0
best_N = 9.0

#lrs = torch.linspace(10 ** (-2), 10 ** (-1), num_iter) # for model A5a
lrs = [1e-7, 5e-7, 1e-6, 5e-6, 1e-5, 5e-5, 1e-4, 5e-4,
  1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1] # for model A5b
# hidden_sizes = np.arange(100,900,200)
nMs = np.arange(300,500,10)
```

```
  ks = np.arange(3,7,1)
  Ns = np.arange(2,30,2)

  for lr in lrs:
    # for hidden_size in hidden_sizes:
    for nM in nMs:
      for k in ks:
        for N in Ns:
          print(f"trying learning rate {lr}")
          # print(f"trying learning rate {hidden_size}")
          print(f"trying Nm {nM}")
          print(f"trying k {k}")
          print(f"trying N {N}")
          model = model_fn(nM, k, N)
          # model = model_fn(hidden_size)
          optim = SGD(model.parameters(), lr)

          train_loss, train_acc, val_loss, val_acc = train(
                  model,
                  optim,
                  train_loader,
                  val_loader,
                  epochs=40
                  )

          if min(val_loss) < best_loss:
            best_loss = min(val_loss)
            best_lr = lr
            # best_hidden_size = hidden_size
            best_nM = nM
            best_k = k
            best_N = N
            print(f"last val accuracy {val_acc[-1]}")
            print(val_acc)
            print('*****')

  return best_lr

  best_lr = parameter_search(train_loader, val_loader, A5b_model)
  # best_lr = parameter_search(train_loader, val_loader, A5a_model)

  # getting a plot
    model = A5b_model(330, 3, 6)
    optimizer = SGD(model.parameters(),0.1)
    train_loss, train_accuracy, val_loss, val_accuracy = train(
    model, optimizer, train_loader, val_loader, 40)

epochs = range(1, 41)
plt.plot(epochs, train_accuracy, label="Train Accuracy")
plt.plot(epochs, val_accuracy, label="Validation Accuracy")
plt.axhline(y=0.65, c='red')
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

```
def evaluate(model: nn.Module, loader: DataLoader):
  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}")
```