Homework #4

CSE 446/546: Machine Learning

Ray Chen
work with:Kevin Shao

A1:

- **Part a:**
  True. Because we use k non-zero eigen-values to do the construction, PCA will result in zero reconstruction error.
- **Part b:**
  False. The columns of V should equal to the eigen-vectors of the given matrix.
- **Part c:**
  False. Choose k to minimize the k-means could cause the outfitting problem, which will generate meaningless clusters.
- **Part d:**
  False. We can choose the decomposition as we want.
- **Part e:**
  False. We take the 2$\times$2 identity matrix as an example, which rank is 1 and eigenvalue only have 1.

A2:

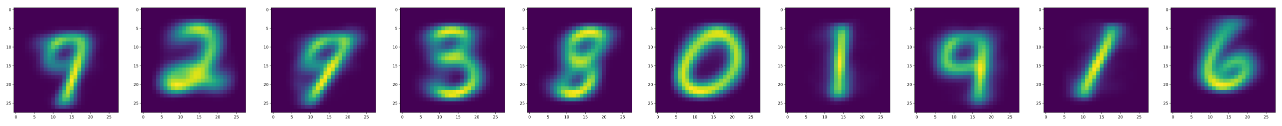
- **Part a:**
  • The features need to be collected are race, gender, age, income, educational background, height and weight. By using those features, we can come up with a disease susceptibility predictor.
  • The Neural Networks should works for those features. As some features may not be completed, we need the Neural Networks to finish the missing part for us. In that way, the result can be more accurate.
  • The model should not be hard to learn, as the input features is not too much. We can check the last five years actual disease rate to determine whether our predict model or the data source is good or not. After get the result, we can adjust our features to best match the last five years' data to achieve the goal.

- **Part b:**
  • Firstly, ignoring those issues may cause the crime model is not accurate when compare to the real-world scenario.
  • Second, the police force may not be able to respond on time as the police force are not distributed by the correct model. This will cause more crime and safety issues. Like in a model considered safe area, the police are limited; however, in the real word, that area is not safe because data is not collected completely.

- **Part c:**
  • Some people's family may have the genetic disease, but they didn't report that, which could cause the model is not accurate. Besides, some people intentionally blur some information, which can cause the model is not accurate.
  • For the first issue, we can add the new feature: family genetic disease. If we take this in to consideration, the issue can be addressed. For the second one, when we collect thee data we should let the user know the data is confidential and is vital to provide vaild information in order to get the accurate result.

A3:

- **Part a:**
  See Code section.
- **Part b:**



- **Code:**

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

def calculate_centers(
    data: np.ndarray, classifications: np.ndarray, num_centers: int
) -> np.ndarray:
    n_f = data.shape[1]
    new_cent = np.zeros((num_centers, n_f))
    for j in range(0, num_centers):
      X_C = data[np.where(classifications == j)]
      new_cent[j] = X_C.mean(axis = 0)
    return new_cent

def cluster_data(data: np.ndarray, centers: np.ndarray) -> np.ndarray:
    n_samples = data.shape[0]
    center = len(centers)
    classifications = np.zeros(n_samples)
    for i in range(0, n_samples):
        distances = np.zeros(center)
        for j in range(0, center):
            distances[j] = np.sqrt(np.sum(np.power(data[i, :] - centers[j], 2)))
        classifications[i] = np.argmin(distances)
    return(classifications)

def calculate_error(data: np.ndarray, centers: np.ndarray) -> float:
    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: np.ndarray, num_centers: int, epsilon: float = 10e-3
) -> Tuple[np.ndarray, List[float]]:
    n_samples = data.shape[0]
    n_features = data.shape[1]
    classifications = np.zeros(n_samples, dtype = np.int64)
    # Choose initial cluster centroids randomly
    I = np.random.choice(n_samples, k)
    centroids = data[I, :]
    classifications = np.zeros(n_samples)
    for i in range(0, n_samples):
        distances = np.zeros(num_centers)
        for j in range(0, num_centers):
            distances[j] = np.sqrt(np.sum(np.power(data[i, :] - centers[j], 2)))
        classifications[i] = np.argmin(distances)
        new_centroids = np.zeros((num_centers, n_features))
        for j in range(0, num_centers):
            J = np.where(classifications == j)
            X_C = data[J]
            new_centroids[j] = X_C.mean(axis = 0)
        centroids = new_centroids
    return centroids, classifications
```

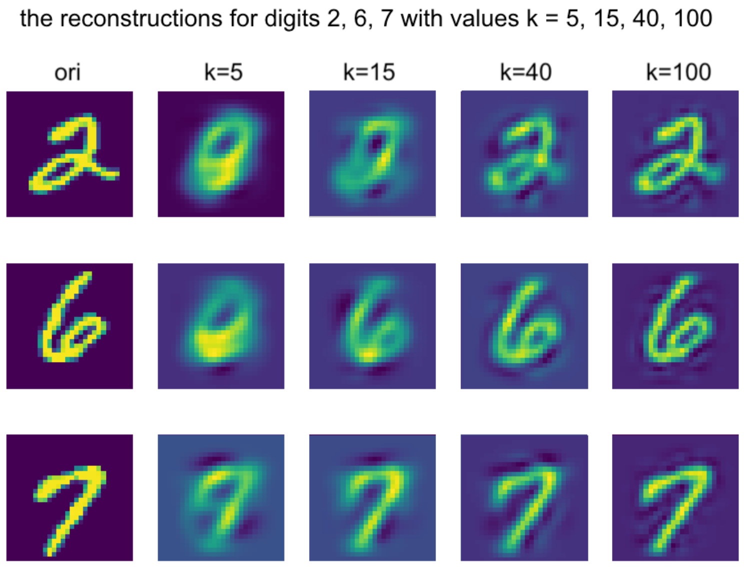
A4:

- **Part a:**
$\lambda_1 = 5.148333$, $\lambda_2 = 3.729989$, $\lambda_10 = 1.25272$, $\lambda_{30} = 0.364266$, $\lambda_{50} = 0.169612$, $\sum_{i=1}^{d} \lambda_i = 52.733847$.

- **Part b:**
The following formula should be the the rank-k PCA approximation of x:

$$x_{i,PCA} \approx U_k U_k^T (x_i - \mu) + \mu$$

- **Part c:**



the reconstructions for digits 2, 6, 7 with values k = 5, 15, 40, 100
ori
k=5
k=15
k=40
k=100

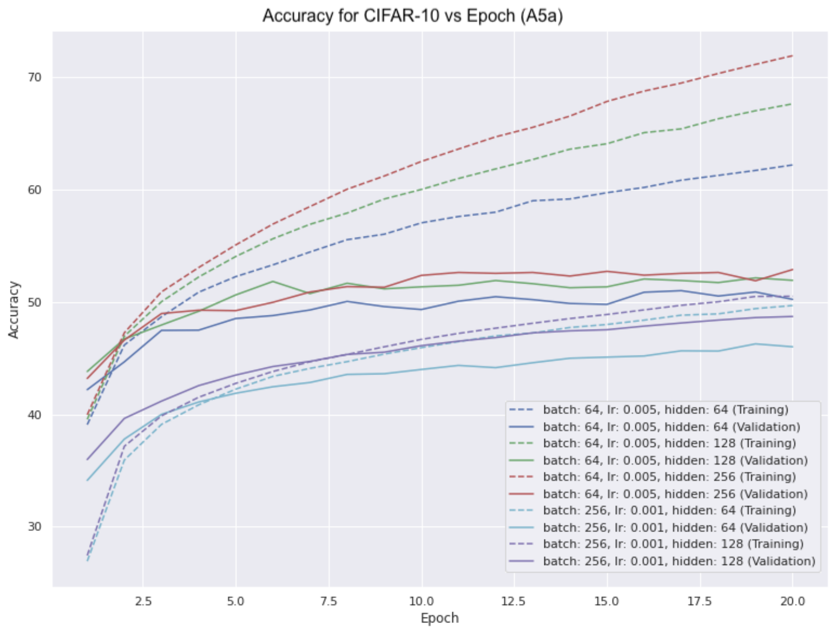
From the image above, we conclude that with bigger value of k, the construction is better, which means the image gets more clear. When k=5 or k=15, the digits are very blur. When k reaches 40, the digits is recognizable. As k=100, We can certainly say that the constructions for digits is good and enough for human to recognize digits.

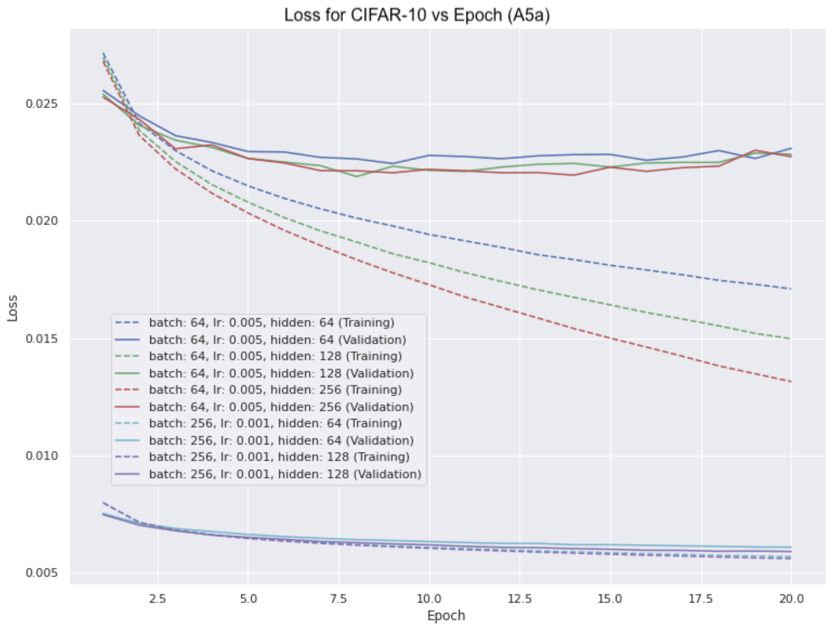
- **Code::**

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


@problem.tag("hw4-A")
def reconstruct_demean(uk: np.ndarray, demean_data: np.ndarray) -> np.ndarray:
    reconstruction_data = np.dot(demean_data, uk)
    reconstruction_data = np.dot(reconstruction_data , uk.T)
    return reconstruction_data


# @problem.tag("hw4-A")
def reconstruction_error(uk: np.ndarray, demean_data: np.ndarray) -> float:
    tmp = demean_data - reconstruct_demean(uk, demean_data)
    res = np.mean(np.linalg.norm(tmp, axis=1) ** 2)
    return res


@problem.tag("hw4-A")
def calculate_eigen(X_train: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
    num = X_train.shape[0]
    Imatrix = np.ones((num, 1))
    mu = np.dot(X_train.T, Imatrix) / num
    sigma = np.dot((X_train - np.dot(Imatrix, mu.T)).T, (X_train - np.dot(Imatrix, mu.T))) / num
    eigen_values, eigen_vectors = np.linalg.eigh(sigma)
    index = np.argsort(eigen_values)[::-1]
    eigen_values, eigen_vectors = eigen_values[index], eigen_vectors[:, index]
    return eigen_values, eigen_vectors



@problem.tag("hw4-A", start_line=2)
def main():
    """
    Main function of PCA problem. It should load data, calculate eigenvalues/-vectors,
    and then answer all questions from problem statement.

    If the handout instructs you to implement the following sub-problems, you should:

    Part A:
        - Report 1st, 2nd, 10th, 30th and 50th largest eigenvalues
        - Report sum of eigenvalues

    Part C:
        - For each of digits 2, 6, 7 plot original image, and images reconstruced from PCA with
            k values of 5, 15, 40, 100.
    """
    (X_train, y_tr), (X_test, _) = load_dataset("mnist")
    X_train = X_train/255.0
    X_test = X_test/255.0
    (X_train_raw, labels_train_raw), (X_test_raw, labels_test_raw) = (X_train, y_tr), (X_test, _)
    X_train = X_train_raw[:50000]
    X_test = X_train_raw[50000:]
    n_train, d = X_train.shape
    mu = 1 / n_train * np.sum(X_train, axis=0).reshape(d, 1)
    demean_X_train = X_train - np.repeat(mu.T, n_train, axis=0)
    sigma = demean_X_train.T @ demean_X_train / n_train
    n_test, _ = X_test.shape
    demean_X_test = X_test - np.repeat(mu.T, n_test, axis=0)
    eig, vec = np.linalg.eig(sigma)
    # calculate eigen values
    eig = calculate_eigen(X_train)
    print(eig)
    tmp = [0, 1, 9, 29, 49]
    for x in tmp:
        print(eig[x])
    print (np.sum(eig))
    # index of a sample of 2, 6, 7
    idx2, idx6, idx7  = 5, 13 ,15
    k_sample = [5, 15, 40, 100]
    sample_res = []
    # compute array for 2 6 7
    for k in range(1, 101):
        uk = vec.T[:k].T # first k columns
        if k in k_sample:
            tmp = {}
            tmp[2] = reconstruct_demean(uk, demean_X_train[idx2].reshape((d, 1))) + mu
            tmp[6] = reconstruct_demean(uk, demean_X_train[idx6].reshape((d, 1))) + mu
            tmp[7] = reconstruct_demean(uk, demean_X_train[idx7].reshape((d, 1))) + mu
            sample_res.append(tmp)
    # plot the image of 2, 6, 7
    idxs = [5, 13, 15]
    k_sample = [2, 6, 7]
    fig, ax = plt.subplots(3, 5)
    for i in range(5):
        ax[i][0].imshow(X_train[idxs[i]].reshape((28, 28)).astype('float64'))
        ax[i][1].imshow(sample_res[0][k_sample[i]].reshape((28, 28)).astype('float64'))
        ax[i][2].imshow(sample_res[1][k_sample[i]].reshape((28, 28)).astype('float64'))
        ax[i][3].imshow(sample_res[2][k_sample[i]].reshape((28, 28)).astype('float64'))
        ax[i][4].imshow(sample_res[3][k_sample[i]].reshape((28, 28)).astype('float64'))
        for j in range(3):
            ax[i][j].axis('off')
    plt.suptitle("the reconstructions for digits 2, 6, 7 with values k = 5, 15, 40, 100")
    plt.show()

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

A5:

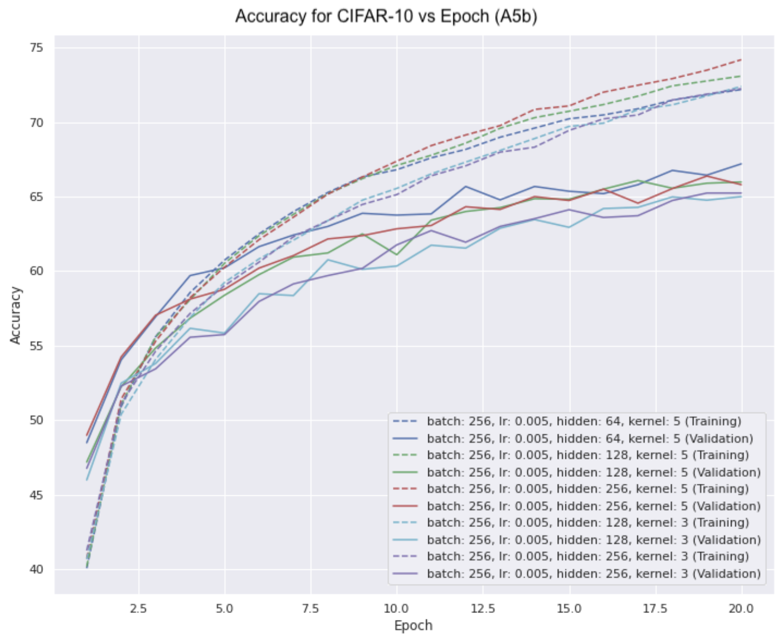
- **Part a:**
  Fully-connected output, 1 fully-connected hidden layer

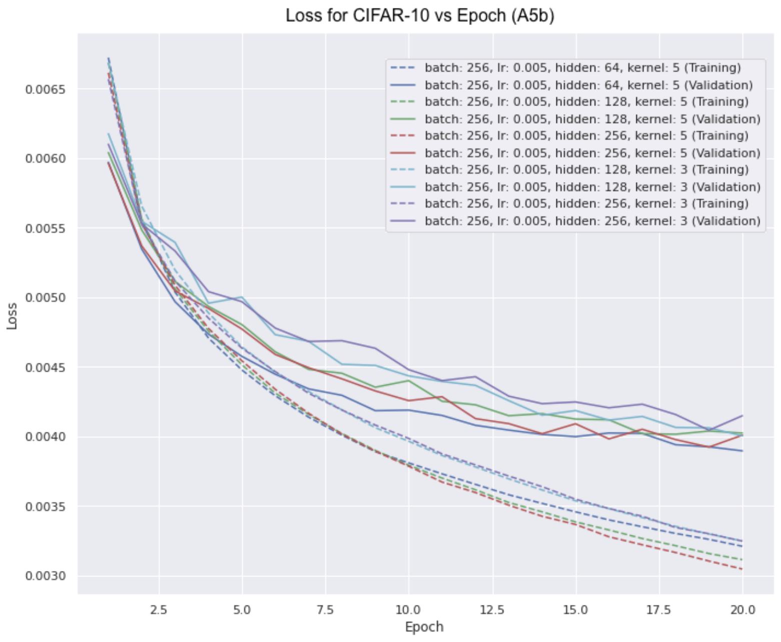




A5a hyperparameters:
Batch size: 64, lr: 0.005, Hidden size: 64;
Batch size: 64, lr: 0.005, Hidden size: 128;
Batch size: 64, lr: 0.005, Hidden size: 256;
Batch size: 256, lr: 0.001, Hidden size: 64;
Batch size: 256, lr: 0.001, Hidden size: 128.
A5a best performing hyperparameters:
Batch size: 64, lr: 0.005, Hidden size: 256;
The Accuracy of that model is 52.89%.
Search Method is grid.

- **Part b:**
  Convolutional layer with max-pool and fully-connected output





A5b hyperparameters:
Batch size: 256, lr: 0.005, Hidden size: 64, Kernel size: 5;
Batch size: 256, lr: 0.005, Hidden size: 128, Kernel size: 5;
Batch size: 256, lr: 0.005, Hidden size: 256, Kernel size: 5;
Batch size: 256, lr: 0.005, Hidden size: 128, Kernel size: 3;
Batch size: 256, lr: 0.005, Hidden size: 256, Kernel size: 3.
A5b best performing hyperparameters
Batch size: 256, lr: 0.005, Hidden size: 256, Kernel size: 5;
The Accuracy of that model is 65.96%.
Search Method is grid.

- **Code:**

```
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 is not available, check the directions above (or disable this

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

# moudle
def A5a() -> nn.Module:
  """Instantiate model and send it to device."""
  model =  nn.Sequential(
            torch.flatten(x, 1),
            hidden_size=64,
            nn.Linear(32 * 32 * 3, self.hidden_size),
            nn.Linear(self.hidden_size, 10))
  return model.to(DEVICE)

def A5b() -> nn.Module:
  """Instantiate model and send it to device."""
  model =  nn.Sequential(
            hidden_size=64,
            kernel_size=5,
            nn.Conv2d(3, self.hidden_size, self.kernel_size),
            nn.MaxPool2d(2, 2),
            nn.Linear(self.hidden_size * ((33 - self.kernel_size)//2) ** 2, 10))
  return model.to(DEVICE)

def prepare_dataset(batch_size=64, train_val_split_ratio=0.9):
    transform = transforms.Compose([,transforms.ToTensor(),transforms.Normalize(mean=[0.485, 0.456,
    cifar10_set = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
    train_size = int(len(cifar10_set) * train_val_split_ratio)
    val_size = len(cifar10_set) - train_size
    cifar10_trainset, cifar10_valset = torch.utils.data.random_split(cifar10_set, [train_size, val_s
    cifar10_testset = datasets.CIFAR10(root='./data', train=False, download=False, transform=transfo
    train_loader = torch.utils.data.DataLoader(cifar10_trainset, batch_size=batch_size, shuffle=True
    val_loader = torch.utils.data.DataLoader(cifar10_valset, batch_size=batch_size, shuffle=True)
    test_loader = torch.utils.data.DataLoader(cifar10_testset, batch_size=batch_size, shuffle=True)
    return train_loader, val_loader, test_loader


def train(epochs, model, train_loader, val_loader, criterion, optimizer, batch_size):
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    model.train()
    for epoch in range(epochs):
        run_loss, correct, total = 0.0, 0, 0
        for i, data in enumerate(train_loader):
            inputs, labels = data[0].to(DEVICE), data[1].to(DEVICE)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1)
            optimizer.step()
            run_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)
        train_accs.append(100 * correct / total)
        train_losses.append(run_loss / total)
        print('[%d] Tri Accuracy: %.3f %% Tri Loss: %.3f' % (epoch + 1, 100 * correct / total, run_l
        run_loss = 0.0 #reset the loss
        val_acc, val_loss = eval(epoch, model, val_loader, criterion)
        val_accs.append(val_acc)
        val_losses.append(val_loss)
    return train_accs, val_accs, train_losses, val_losses

def eval(epoch, model, eval_loader, criterion):
    run_loss, correct, total = 0.0, 0, 0
    for i, data in enumerate(eval_loader):
        inputs, labels = data[0].to(DEVICE), data[1].to(DEVICE)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        run_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)
    print('[%d] Val Accuracy: %.3f %% Val Loss: %.3f' % (epoch + 1, 100 * correct / total, run_loss
    return 100 * correct / total, run_loss / total

def plot_acc(train_acc, val_acc, labels, figname):
    epochsx = [int(x) for x in np.arange(1, len(train_acc[0])+1)]
    plt.figure(figsize=(15, 8))
    for tacc, vacc, label in zip(train_acc, val_acc, labels):
        plt.plot(epochsx, tacc, '--', label=label + ' (Training)')
        plt.plot(epochsx, vacc, '-', label=label + ' (Validation)')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.tight_layout()
    plt.savefig(figname)

def plot_loss(train_loss, val_loss, labels, figname):
    epochsx = [int(x) for x in np.arange(1, len(train_loss[0])+1)]
    plt.figure(figsize=(15, 8))
    for tloss, vloss, label in zip(train_loss, val_loss, labels):
        plt.plot(epochsx, tloss, '--', label=label + ' (Training)')
        plt.plot(epochsx, vloss, '-', label=label + ' (Validation)')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.tight_layout()
    plt.savefig(figname)
```

```
def part_a (epochs=20):
    T_acc ,T_los, V_acc, V_los, labels = [], [], [], [], []
    for param in hyper_params:
        print(param)
        batch_size = param['batch_size']
        lr = param[lr]
        hidden_size = param['hidden_size']
        train_loader, val_loader, test_loader = prepare_dataset(batch_size=batch_size)
        model = A5a(hidden_size=hidden_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
        model.to(DEVICE)
        Ta, Va, Tl, Vl = train(epochs, model, train_loader, val_loader, criterion, optimizer, batch_
        T_acc.append(Ta)
        V_acc.append(Va)
        T_los.append(Tl)
        V_los.append(Vl)
        labels.append('batch: {}, lr: {}, hidden: {}'.format(batch_size, lr, hidden_size))
        eval(-1, model, test_loader, criterion)
    plot_acc(T_acc, V_acc, labels, 'A5a_acc.png')
    plot_loss(T_los, V_los, labels, 'A5a_loss.png')

def part_b(epochs=20):
    T_acc ,T_los, V_acc, V_los, labels = [], [], [], [], []
    for param in hyper_params:
        print(param)
        batch_size = param['batch_size']
        lr = param[lr]
        hidden_size = param['hidden_size']
        kernel_size = param['kernel_size']
        train_loader, val_loader, test_loader = prepare_dataset(batch_size=batch_size)
        model = A5b(hidden_size=hidden_size, kernel_size=kernel_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=1e-5)
        model.to(DEVICE)
        Ta, Va, Tl, Vl = train(epochs, model, train_loader, val_loader, criterion, optimizer, batch_
        T_acc.append(Ta)
        V_acc.append(Va)
        T_los.append(Tl)
        V_los.append(Vl)
        labels.append('batch: {}, lr: {}, hidden: {}, kernel: {}'.format(batch_size, lr, hidden_size
        eval(-1, model, test_loader, criterion)
    plot_acc(T_acc, V_acc, labels, 'A5b_acc.png')
    plot_loss(T_los, V_los, labels, 'A5b_loss.png')

def main():
    part_a()
    part_b()

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