CSE546 HW2

Haoxin Luo

December 8, 2022

Exercise A1-Conceptual

a).

True. Using PCA to project onto a k dimensional space will have zero construction error because our covariance matrix only have k non zero eigenvalues

b).

False, rows of V^T and columns of V equal to the eigenvectors of $X^T X$

c).

False, when k = n, each cluster will only contain 1 data point and the objective function will give us a value of 0 which is meaningless in terms of generalization

d).

False, SVD of a matrix A is not always unique, let USV^T be the SVD of A, only the singular values are unique. For a distinct positive singular values $S_{jj} > 0$, the jth corresponding column of U and V are not unique because we can also have: $A = (-U)S(-V)^T$.

e).

False. The rank of a square matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ has eigenvalue 1 and rank 2.

Exercise A2-Real World ML

a).

We might need to do some transformations on categorical variables such as gender, race and etc. using one hot encoding. And we can normalize our numerical variables to make sure they can be analyzed on the same scale. In this particular study, our outcome variable (labels) is if a person get the disease or not, so I will use a binary classification algorithm to train the data. Since our goal is to determine how susceptible someone is to this disease when they enter their personal information and the system is expected to take limited amount of personal data from a new user to infer more on details, I would use a simple neural network to handle missing data. It is capable of dealing with nonlinearlity, high dimensionality and outliers. Finally, we choose the label with highest probability from the results to be the predicted label.

b).

When we make predictions, there are 2 types of errors that could happen: (False Positives) predictions that some neighborhoods are dangerous but they are not. (False Negatives) predictions that some neighborhoods are not dangerous but they do.

The neighborhoods ultimately aren't dangerous at equal rates. We will get a biased result by training a model that assumes equal rate, but the data actually have different rates. Moreover, hiddding correlation between variables might undermine the statistical significance of an independent variable. Those issues are very concerning because it increase the chances of misunderstanding of real crime rates. results could affect the allocation of police resources. Obviously, we wish to assign more police power to areas that have higher crimes rates.

c).

Our previous training process assumes that the crimes are reported at the same rates in different neighborhoods, that police respond roughly the same to the same crime reported or observed in different neighborhoods, and that police spend the same time patrolling in some neighborhoods than others. But in fact, the neighborhoods ultimately aren't dangerous at equal rates.

To resolve this problem, we can build a tool that is both well calibrated and have similar false positive rates. We could score the risk of those neighborhood from 1-10 so that the crime rates for different neighborhood at any one particular risk score is very similar. We can then determine a cutoff somewhere between the range where scores below the cutoff are treated as low risk, and those above are high risk. Changing the threshold is similar to changing the degrees of freedom we have. In this way, we are turning from the regression problem to a binary classification problem.

Exercise A3-K means

a).

The Lloyd's algorithm runs as follows: 1. choose the number of cluster (e.g. k=10) 2. random assign k cluster center locations 3. each data points find out the center closest to them 4. each center finds the centroid of points they owns 5. update the center 6. repeat untile terminated

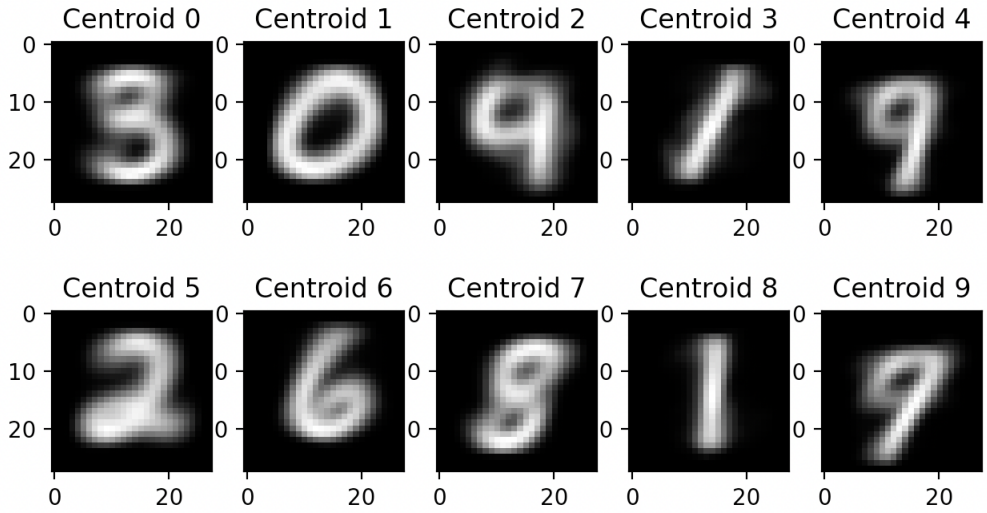
```
def lloyd_algorithm( data: np.ndarray, num_centers: int, epsilon: float = 10e-3,
    nIter=None) -> Tuple[np.ndarray, List[float]]:

    c = data[:num_centers]
    prev_c = c + np.inf
    err = []

    while np.linalg.norm(prev_c - c, ord=np.inf ) > epsilon:
        prev_c = np.copy(c)
        clusters = cluster_data(data, c)
        c = calculate_centers(data, clusters, num_centers)
        err.append(calculate_error(data, c))

    return c, err
```

b).



Exercise A4 - PCA

a).

$\lambda_1 = 5.1168, \lambda_2 = 3.7413, \lambda_{10} = 1.2427, \lambda_{30} = 0.3643, \lambda_{50} = 0.1697$

Sum of λ is about 52.725

b).

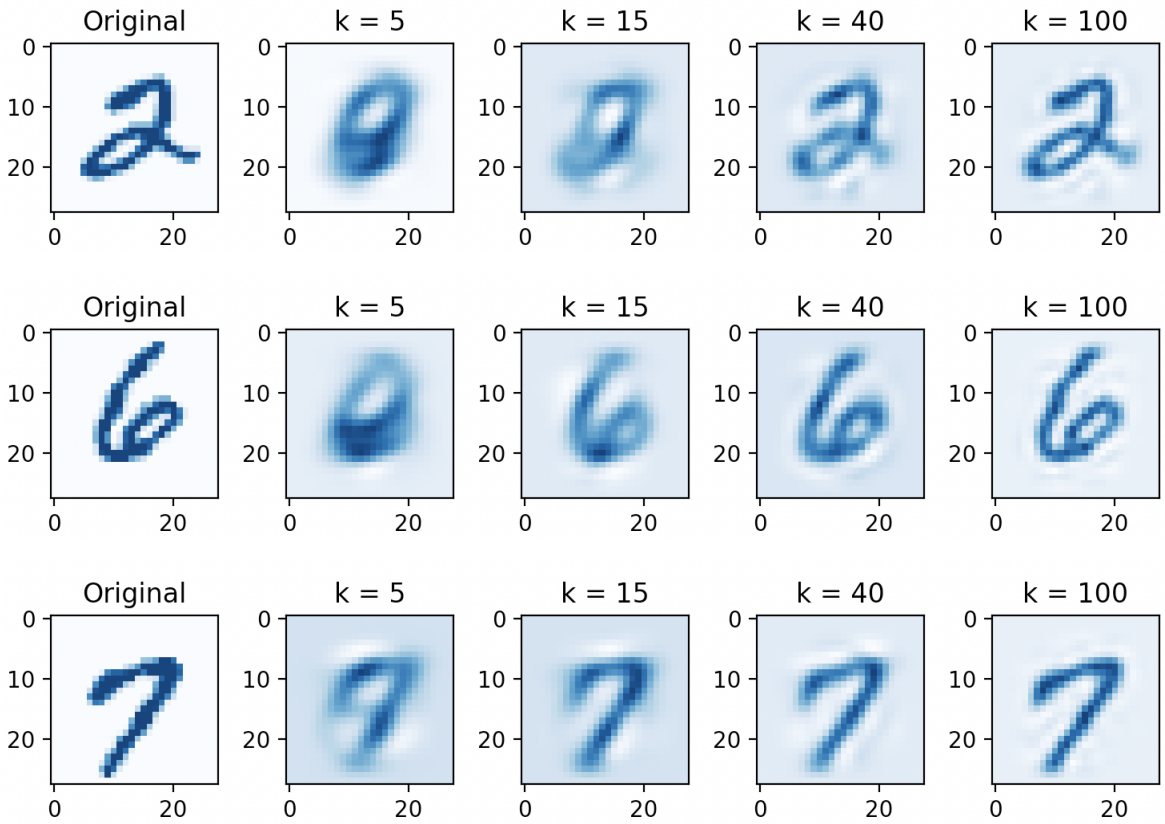
Given $x_1, ...x_n \in R^d$, for k « d, the compressed representation with $\lambda_1, ...\lambda_n \in R^k$ such that:

$$x_i \approx \mu + V_k \lambda_i$$

with $V_k^T V_k = I$, fix V_k and solve for μ_i, λ_i, we get $\mu = \bar{x}$ and $\lambda_i = V_k^T(x_i - \bar{x})$ Thus, the formula for the rank-k PCA approximation of x is:

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

c).

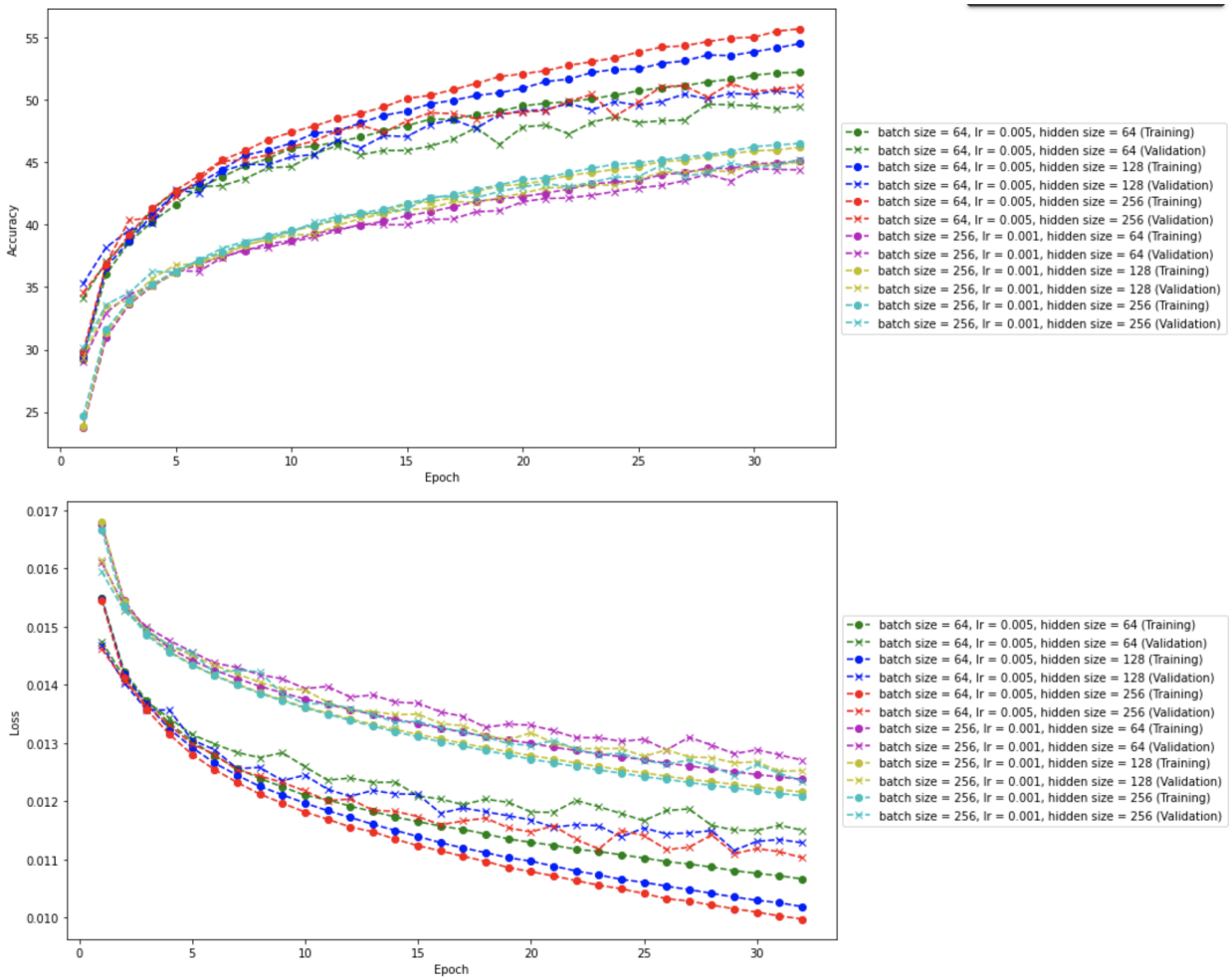


From the reconstructed images, we can see that the pictures get clearer when k gets bigger. To be specific, the first 5 eigenvectors are not enough to reconstruct the digits (2 and 6 both looks like 0). Starting from 15 eigenvectors, the image becomes more readable. With 100 eigenvectors, the digits becomes pretty clear to read.

Exercise A5 - Image Classification

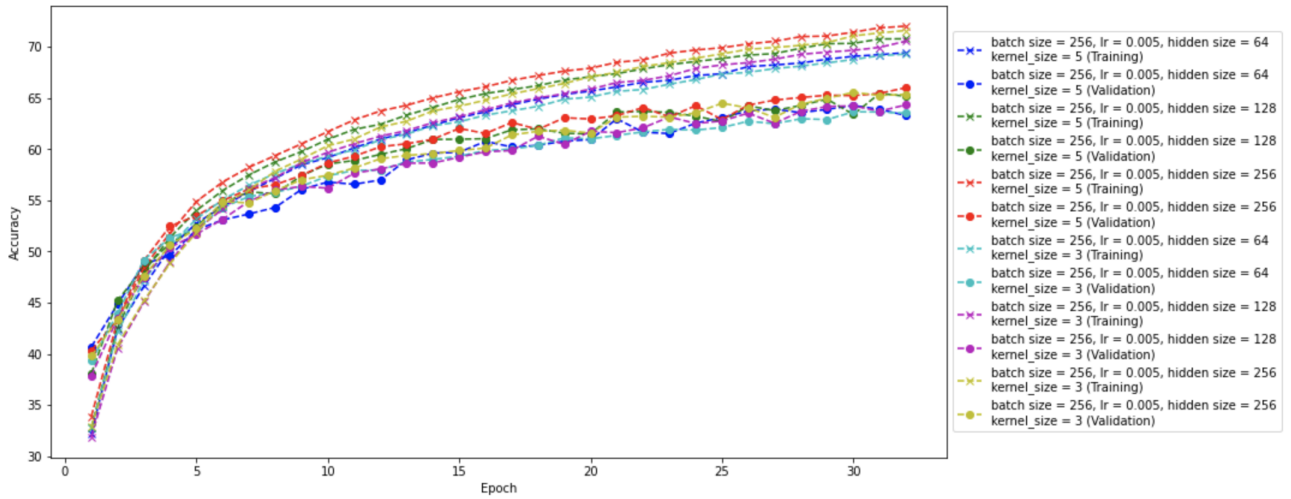
a). I used grid search with batch size 256, learning rate (0.001, 0.003, 0.005), hidden size 64, 128, 256 respectively and kernel size 3, 5

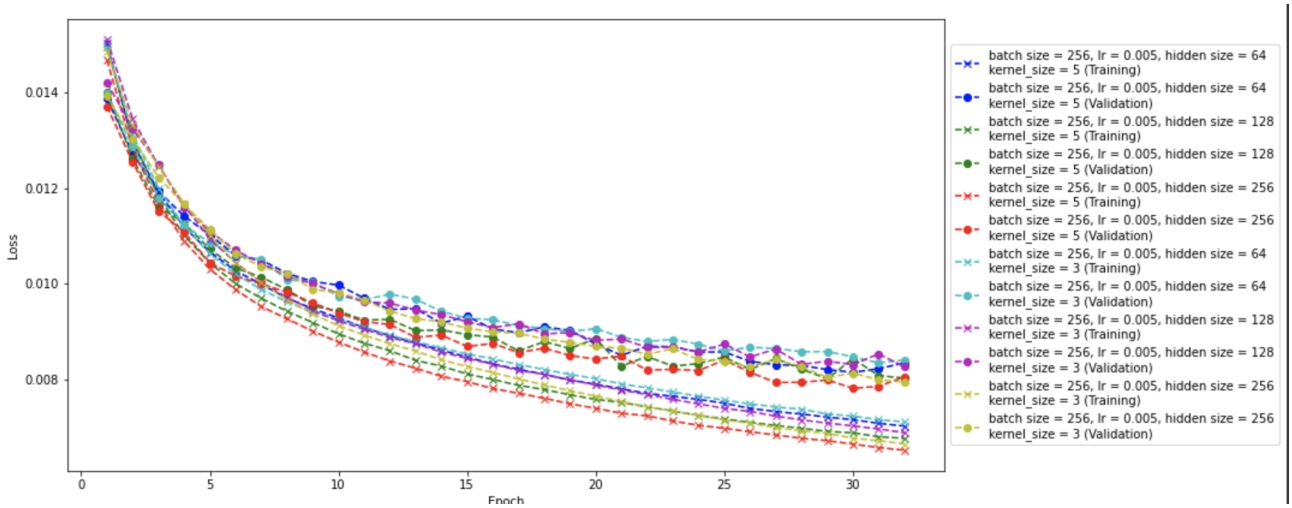
The best results we achieved is Train Accuracy: 55.684, Train Loss: 0.010 Val Accuracy: 51.060, Val Loss: 0.011 with batch size: 64, learning rate: 0.005, hidden size: 256, kernel size: 3. (If I have a stronger GPU and more time to train, I think I could get a better results)



b). I used grid search with batch size 256, learning rate (0.001, 0.003, 0.005), hidden size 64, 128, 256 respectively and kernel size 3, 5

The best results we achieved is train Accuracy: 70.176, Train Loss: 0.007, Val Accuracy: 65.820, Val Loss: 0.008 with batch size: 256, learning rate: 0.005, hidden size: 64, and kernel size: 3





c). 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

Let's load CIFAR-10 data. This is how we load datasets using PyTorch in the real world!
"""

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())

# 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]}")

"""In this problem, we will attempt to predict what class an image is labeled as.
```

```
"""

def linear_model() -> nn.Module:
  """Instantiate a linear model and send it to device."""
  model =  nn.Sequential(
            nn.Flatten(),
            nn.Linear(d, 10)
          )
  return model.to(DEVICE)

"""Let's define a method to train this model using SGD as our optimizer."""

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]]:

  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 ** (-6), 10 ** (-1), num_iter)

  for lr in lrs:
    print(f"trying learning rate {lr}")
    model = model_fn()
    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

"""Now that we have everything, we can train and evaluate our model."""

best_lr = parameter_search(train_loader, val_loader, linear_model)

model = linear_model()
optimizer = SGD(model.parameters(), best_lr)

# 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, 20)

"""Plot the training and validation accuracy for each epoch."""

epochs = range(1, 21)
plt.plot(epochs, train_accuracy, label="Train Accuracy")
plt.plot(epochs, val_accuracy, label="Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Logistic Regression Accuracy for CIFAR-10 vs Epoch")
plt.show()

"""The last thing we have to do is evaluate our model on the testing data."""

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}")

# ***Code Start Here*****************************************************************
"""

import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm

class A5a(nn.Module):
  # Fully-connected output, 1 fully-connected hidden layer:
  def __init__(self, hidden_size=64):
    super().__init__()
    self.h = hidden_size
    self.fc1 = nn.Linear(32 * 32 * 3, self.h)
    self.fc2 = nn.Linear(self.h, 10)

  def forward(self, x):
    x = torch.flatten(x, 1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)

    return F.log_softmax(x, dim=1)

class A5b(nn.Module):
  # Convolutional layer with max-pool and fully-connected output
  def __init__(self, hidden_size=64, kernel_size=5):
    super().__init__()
```

```
    self.h = hidden_size
    self.k = kernel_size
    self.conv1 = nn.Conv2d(3, self.h, self.k)
    self.pool = nn.MaxPool2d(2, 2)
    self.fc1 = nn.Linear(self.h * ((33 - self.k)//2) ** 2, 10)

  def forward(self, x):
    x = self.conv1(x)
    x = F.relu(x)
    x = self.pool(x)
    x = torch.flatten(x, 1)
    x = self.fc1(x)

    return F.log_softmax(x, dim=1)

# modified version of training (adding criterion and optimizer as parameters)
def train(epochs, model, train_loader, val_loader, criterion, optimizer, batch_size):
  train_losses = []
  train_accuracy = []

  val_losses = []
  val_accuracy =[]

  model.train()
  for epoch in range(epochs):
      train_loss = 0.0
      train_acc = 0
      total = 0
      for i, data in enumerate(train_loader):
          inputs, labels = data[0].to("cuda"), data[1].to("cuda")
          optimizer.zero_grad()

          outputs = model(inputs)
          loss = criterion(outputs, labels)
          loss.backward()
          torch.nn.utils.clip_grad_norm_(model.parameters(), 1)
          optimizer.step()
          train_loss += loss.item()

          _, predicted = torch.max(outputs.data, 1)
          train_acc += (predicted == labels).sum().item()
          total += labels.size(0)

      train_accuracy.append(100 * train_acc / total)
      train_losses.append(train_loss / total)
      print('[%d] Train Accuracy: %.3f %% Train Loss: %.3f' % (epoch
      + 1, 100 * train_acc / total, train_loss / total))
      train_loss = 0.0
```

```
      val_acc, val_loss = eval(epoch, model, val_loader, criterion)
      val_accuracy.append(val_acc)
      val_losses.append(val_loss)

  return train_accuracy, val_accuracy, train_losses, val_losses

# modified evaluate function
def eval(epoch, model, eval_loader, criterion):
  test_loss = 0.0
  test_acc = 0
  total = 0

  for i, data in enumerate(eval_loader):
    inputs, labels = data[0].to("cuda"), data[1].to("cuda")
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    test_loss += loss.item()

    _, predicted = torch.max(outputs.data, 1)
    test_acc += (predicted == labels).sum().item()
    total += labels.size(0)

  print('[%d] Val Accuracy: %.3f %% Val Loss: %.3f' % (epoch + 1,
  100 * test_acc / total, test_loss / total))
  return 100 * test_acc / total, test_loss / total

def plot_acc(train_acc, val_acc, labels, figname):
  epsx = [int(x) for x in np.arange(1, len(train_acc[0])+1)]
  plt.figure(figsize=(15, 6))
  COLORS = ['g', 'b', 'r', 'm', 'y', 'c']
  for tacc, vacc, label, color in zip(train_acc, val_acc, labels, COLORS):
    plt.plot(epsx, tacc, '--o', label=label + ' (Training)', color=color)
    plt.plot(epsx, vacc, '--x', label=label + ' (Validation)', color=color)
  plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.tight_layout()
  plt.savefig(figname)

def plot_loss(train_loss, val_loss, labels, figname):
  epsx = [int(x) for x in np.arange(1, len(train_loss[0])+1)]
  plt.figure(figsize=(15, 6))
  COLORS = ['g', 'b', 'r', 'm', 'y', 'c']
  for tloss, vloss, label, color in zip(train_loss, val_loss, labels, COLORS):
      plt.plot(epsx, tloss, '--o', label=label + ' (Training)', color=color)
      plt.plot(epsx, vloss, '--x', label=label + ' (Validation)', color=color)
  plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
  plt.xlabel('Epoch')
```

```
  plt.ylabel('Loss')
  plt.tight_layout()
  plt.savefig(figname)

def run_A5a(epochs=32):

  hyper_params = [
      {'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': 128, 'lr': 0.005, 'hidden_size': 64},
      {'batch_size': 128, 'lr': 0.005, 'hidden_size': 128},
      {'batch_size': 128, 'lr': 0.005, 'hidden_size': 256},
      {'batch_size': 128, 'lr': 0.003, 'hidden_size': 64},
      {'batch_size': 128, 'lr': 0.003, 'hidden_size': 128},
      {'batch_size': 128, 'lr': 0.003, 'hidden_size': 256},
      {'batch_size': 256, 'lr': 0.003, 'hidden_size': 64},
      {'batch_size': 256, 'lr': 0.003, 'hidden_size': 128},
      {'batch_size': 256, 'lr': 0.003, 'hidden_size': 256},
      {'batch_size': 128, 'lr': 0.001, 'hidden_size': 64},
      {'batch_size': 128, 'lr': 0.001, 'hidden_size': 128},
      {'batch_size': 128, 'lr': 0.001, 'hidden_size': 256},
      {'batch_size': 256, 'lr': 0.001, 'hidden_size': 64},
      {'batch_size': 256, 'lr': 0.001, 'hidden_size': 128},
      {'batch_size': 256, 'lr': 0.001, 'hidden_size': 256}
  ]

  overrall_train_acc = []
  overrall_train_losses = []
  overrall_val_acc = []
  overrall_val_losses = []
  labels = []
  for param in hyper_params:
      batch_size = param['batch_size']
      learning_rate = param['lr']
      hidden_size = param['hidden_size']

      model = A5a(hidden_size=hidden_size)
      criterion = nn.CrossEntropyLoss()
      optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)
      model.to("cuda")

      train_acc, val_acc, train_loss, val_loss = train(epochs, model,
      train_loader, val_loader, criterion, optimizer, batch_size)

      overrall_train_acc.append(train_acc)
      overrall_train_losses.append(train_loss)
      overrall_val_acc.append(val_acc)
```

```
        overall_val_losses.append(val_loss)
        labels.append('batch size = {}, lr = {},
        hidden size = {}'.format(batch_size, lr, hidden_size))
        eval(-1, model, test_loader, criterion)

    plot_acc(overall_train_acc, overall_val_acc, labels, 'A5a_acc.png')
    plot_loss(overall_train_losses, overall_val_losses, labels, 'A5a_loss.png')

run_A5a()

def run_A5b(epochs=32):

    hyper_params = [
        {'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': 128, 'lr': 0.005, 'hidden_size': 64},
        {'batch_size': 128, 'lr': 0.005, 'hidden_size': 128},
        {'batch_size': 128, 'lr': 0.005, 'hidden_size': 256},
        {'batch_size': 128, 'lr': 0.003, 'hidden_size': 64},
        {'batch_size': 128, 'lr': 0.003, 'hidden_size': 128},
        {'batch_size': 128, 'lr': 0.003, 'hidden_size': 256},
        {'batch_size': 256, 'lr': 0.003, 'hidden_size': 64},
        {'batch_size': 256, 'lr': 0.003, 'hidden_size': 128},
        {'batch_size': 256, 'lr': 0.003, 'hidden_size': 256},
        {'batch_size': 128, 'lr': 0.001, 'hidden_size': 64},
        {'batch_size': 128, 'lr': 0.001, 'hidden_size': 128},
        {'batch_size': 128, 'lr': 0.001, 'hidden_size': 256},
        {'batch_size': 256, 'lr': 0.001, 'hidden_size': 64},
        {'batch_size': 256, 'lr': 0.001, 'hidden_size': 128},
        {'batch_size': 256, 'lr': 0.001, 'hidden_size': 256}
    ]

    best_param = [
        {'batch_size': 256, 'lr': 5e-3, 'hidden_size': 256, 'kernel_size': 5}
    ]

    overall_train_acc = []
    overall_train_losses = []
    overall_val_acc = []
    overall_val_losses = []
    labels = []
    for param in hyper_params:
        print(param)
        batch_size = param['batch_size']
        learning_rate = param['lr']
        hidden_size = param['hidden_size']
        kernel_size = param['kernel_size']
```

```
        model = A5b(hidden_size=hidden_size, kernel_size=kernel_size)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,
        weight_decay=1e-5)
        model.to("cuda")

        train_accuracies, val_accuracies, train_losses, val_losses = train(epochs, model,
        train_loader, val_loader, criterion, optimizer, batch_size)

        overrall_train_acc.append(train_accuracies)
        overrall_val_acc.append(val_accuracies)
        overrall_train_losses.append(train_losses)
        overrall_val_losses.append(val_losses)
        labels.append('batch size = {}, lr = {},
        hidden size = {}\nkernel_size = {}'.format(batch_size, lr, hidden_size, kernel_size))
        eval(-1, model, test_loader, criterion)

    plot_acc(overrall_train_acc, overrall_val_acc, labels, 'A5b_acc.png')
    plot_loss(overrall_train_losses, overrall_val_losses, labels, 'A5b_loss.png')

run_A5b()
```

Exercise B1

a).

i). We know that the Euler's formula is: $e^{iy} = cos(y) + isin(y)$ and $k_p(x, x') = Re[E_w[e^{iw^T(x-x')}]]$. Even though $E[z(x)z(x')]$ is real, the sampled z(x)z(x') will in general be complex. Since both our distribution and the kernel are real-valued, we can write the integral as:

$$k_p(x, x') = E_w[cos(w^T(x - x')) + isin(w^T(x - x'))]$$

$$= \int p(w)(cos(w^T(x - x')) + isin(w^T(x - x')))dw$$

$$= \int p(w)cos(w^T(x - x'))dw$$

$$= E_w[cos(w^T(x - x'))]$$

The above has a real-valued integrand, so a Monte Carlo estimate of it will always be real valued.

ii). Using the fact that $E_b[cos(z + nb)] = 0$, for all $z \in R$ and $n \in N^+$, $b \sim Unif(0, 2\pi)$, we can re-write:

$$E_{w,b}[z_w(x)z_w(x')] = E_{w,b}[\sqrt{2}cos(w^T x + b)\sqrt{2}cos(w^T x' + b)]$$

$$= E_{w,b}[2cos(w^T x + b)cos(w^T x' + b)]$$

$$= E_{w,b}[cos(w^T x + b + w^T x' + b) + cos(w^T x + b - w^T x' - b)]$$

$$= E_{w,b}[cos(w^T x + w^T x' + 2b) + cos(w^T x - w^T x')]$$

$$= E_{w,b}[cos(w^T(x - x'))]$$

$$= k_p(x, x')$$

iii). By drawing D independent pairs of w,b and computing the estimate:

$$E_w[z(x)^T z(x')] = \frac{1}{D}\sum_{b=1}^{D} z_{w_1,b_1}(x)z_{w_1,b_1}(x') \approx E_{w,b}[z_w(x)z_w(x')]$$

$$E_w[E_{w,b}[z_w(x)z_w(x')]] = E_{w,b}[z_w(x)z_w(x')] = k_p(x, x')$$

$$E_w[z(x)^T z(x')] = k_p(x, x')$$

b).

$$E_w[z(x)^T z(x')] = k_p(x, x') = \int p(w)e^{iw^T(x-x')}dw$$

Let $(x - x') = \lambda$

$$= \int (\frac{2\pi}{\gamma^2})^{-D/2} e^{-\frac{\gamma^2\|w\|_2^2}{2}} e^{iw^T\lambda}dw$$

$$= \int (\frac{2\pi}{\gamma^2})^{-D/2} e^{-\frac{\gamma^2(w^T w - \frac{2iw^T\lambda}{\gamma^2})}{2}}dw$$

$$= \int (\frac{2\pi}{\gamma^2})^{-D/2} e^{-\frac{\gamma^2(w^T w - \frac{2iw^T\lambda}{\gamma^2} - \frac{1}{\gamma^4})}{2} - \frac{\lambda^T\lambda}{2\gamma^2}} dw$$

$$= (\frac{2\pi}{\gamma^2})^{-\frac{D}{2}} exp(-\frac{\lambda^T\lambda}{2\gamma^2}(\frac{2\pi}{\gamma^2})^{\frac{D}{2}})$$

$$= exp(-\frac{-||x - x'||_2^2}{2\gamma^2})$$

Since the integral of Gaussian distribution is 1, we have:

$$E_w[z(x)^T z(x')] = e^{-\frac{||x-x'||_2^2}{2\gamma^2}}$$

c).

Since we have independent random variables such that $-\sqrt{2} \leq Z_{w,b} \leq \sqrt{2}$, and $k(x, x') = E_{w,b}[z_w(x)z_w(x')]$ we can use Hoeffding's inequality to obtain the following high-probability bound on the absolute error on our estimate of k. The RFF estimator of k, using D pairs of w,b, obeys $p(|z(x)^T z(x') - k(x, x')| \geq \varepsilon) \leq 2exp(-D\varepsilon^2/8)$

Collaborator: Andy Chen, Yi-Ling Chen