Homework #4

CSE 446/546: Machine Learning
Prof. Matt Golub & Pang Wei Koh
Due: **Wednesday** Dec 4, 2024 11:59pm
Points A: 96; B: 25

Please review all homework guidance posted on the website before submitting to Gradescope. Reminders:

- All code must be written in Python and all written work must be typeset (e.g. LaTeX).
- Make sure to read the "What to Submit" section following each question and include all items.
- Please provide succinct answers and supporting reasoning for each question. Similarly, when discussing experimental results, concisely create tables and/or figures when appropriate to organize the experimental results. All explanations, tables, and figures for any particular part of a question must be grouped together.
- For every problem involving generating plots, please include the plots as part of your PDF submission.
- When submitting to Gradescope, please link each question from the homework in Gradescope to the location of its answer in your homework PDF. Failure to do so may result in deductions of up to 10% of the value of each question not properly linked. For instructions, see https://www.gradescope.com/get_started#student-submission.

Important: By turning in this assignment (and all that follow), you acknowledge that you have read and understood the collaboration policy with humans and AI assistants alike: https://courses.cs.washington.edu/courses/cse446/24au/assignments/. Any questions about the policy should be raised at least 24 hours before the assignment is due. There are no warnings or second chances. If we suspect you have violated the collaboration policy, we will report it to the college of engineering who will complete an investigation. Not adhering to these reminders may result in point deductions.

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. *[2 points]* True or False: Given a data matrix $X \in \mathbb{R}^{n\times d}$ where d is much smaller than n and $k = \text{rank}(X)$, if we project our data onto a k-dimensional subspace using PCA, our projection will have zero reconstruction error (in other words, we find a perfect representation of our data, with no information loss).

> Solution:
> True, if X has rank k, then X lies entirely in the k-dimensional subspace of the d dimensional space as those are all the linearly independent dimensions of the data. Thus, by projecting onto the k-dimensional subspace, we're essentially representing the data in its full dimensionality, meaning the reconstruction error is zero.

b. *[2 points]* True or False: 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 rows of V are equal to the eigenvectors of $X^\top X$.

> Solution:
> False, the columns of V are equal to the eigenvectors of $X^\top X$, not the rows.

c. *[2 points]* True or False: choosing k to minimize the k-means objective (see Equation (4) below) is a good way to find meaningful clusters.

> Solution:
> False, choosing k to minimize the k-means objective is not a good way to find meaningful clusters due to the fact that as k increases, the objective decreases until it reaches zero, when k = number of data points. Also k is not being penalized in the objective function.

d. *[2 points]* True or False: The singular value decomposition of a matrix is unique.

> False, while the singular values themselves are unique for a given matrix, the decomposition as a whole is not unique. This is because The left (U) and right (V) singular vectors corresponding to distinct singular values are unique up to a sign change. (i.e. if u_i and v_i are singular vectors, replacing them with $-u_i$ and $-v_i$ still satisfies the SVD equation.

e. *[2 points]* True or False: The rank of a square matrix equals the number of its unique nonzero eigenvalues.

> False, the rank of a square matrix equals the number of its nonzero eigenvalues including multiplicity. For example, if we take the 2x2 identity matrix, we find it has two eigenvalues, both of which are 1.The number of eigenvalues of this matrix equals its rank, but not the number of unique eigenvalues.

What to Submit:

- **Parts a-e:** 1-2 sentence explanation containing your answer.

Think before you train

A2. **The first part of this problem (part a)** explores how you would apply machine learning theory and techniques to a real-world problem. There is one scenario detailing a setting, a dataset, and a specific result we hope to achieve. Your job is to describe how you would handle the scenario with the tools we've learned in this class. Your response should include:

(1) any pre-processing steps you would take (e.g. any data processing),

(2) the specific machine learning pipeline you would use (i.e., algorithms and techniques learned in this class),

(3) how your setup acknowledges the constraints and achieves the desired result.

You should also aim to leverage some of the theory we have covered in this class. Some things to consider may be: the nature of the data (i.e., *How hard is it to learn? Do we need more data? Are the data sources good?*), the effectiveness of the pipeline (i.e., *How strong is the model when properly trained and tuned?*), and the time needed to effectively perform the pipeline.

a. *[10 points]* **Scenario: Disease Susceptibility Predictor**

- Setting: You are tasked by a research institute to create an algorithm that learns the factors that contribute most to acquiring a specific disease.
- Dataset: A rich dataset of personal demographic information, location information, risk factors, and whether a person has the disease or not.
- Result: The company wants a system that can determine how susceptible someone is to this disease when they enter in their own personal information. The pipeline should take limited amount of personal data from a new user and infer more detailed metrics about the person.

Solution:

(1) Pre-processing:
To begin, the data will be cleaned by handling missing values (through imputation or removal of entry), removing outliers, and feature encoding categorical labels. Dimensionality reduction techniques like SVD would also be applied to reduce feature redundancy and extract key data patterns. Finally, the dataset will be split into training, validation, and test sets to ensure good generalization.

(2) ML Pipeline: Gradient Boosted Decision Trees
After pre-processing, a Gradient Boosted Trees model would be built and trained. These can be effective for this task because they can model complex, non-linear relationships in the data and handle mixed feature types (categorical and numerical) without extensive preprocessing. Hyperparameter tuning (e.g., tree depth, learning rate) would be done using cross-validation.

(3) Results: Gradient Boosted Trees efficiently handle sparse and incomplete data, making them suitable for limited user inputs. SVD aids dimensionality reduction for better inference, while cross-validation ensures robust generalization. Although less interpretable than simpler models, Gradient Boosted Trees balance accuracy and efficiency, delivering fast, detailed predictions and insights from minimal data.

The second part of this problem (parts b, c) focuses on exploring possible shortcomings of machine learning models, and what real-world implications might follow from ignoring these issues.

b. *[5 points]* Briefly describe (1) some potential shortcomings of your training process from the disease susceptibility predictor scenario above that may result in your algorithm having different accuracy on different populations, and (2) how you may modify your procedure to address these shortcomings.

Solution:
Shortcomings: The training process may suffer from bias due to imbalanced or unrepresentative data. For instance, if certain populations (e.g., specific age groups, locations, or socioeconomic backgrounds) are underrepresented, the model may generalize poorly for these groups. Additionally, data quality may vary across populations, with missing or noisy data disproportionately affecting certain subgroups. These issues can lead to disparities in accuracy and fairness, potentially disadvantaging vulnerable populations.

Modifications: To address these issues, the training dataset would be checked to make sure it is diverse and representative. Re-sampling techniques could also be used. Additionally, based on model evaluations model, the model could go through post-processing adjustments to ensure the model is equitable across different demographic groups and to address any gaps.

c. *[5 points]* Recall in Homework 2 we trained models to predict crime rates using various features. It is important to note that **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?

Solution:
There are a variety of real world implications that might follow from ignoring these issues. To begin, if crime reporting rates differ across neighborhoods, the model may not accurately reflect the true level of crime in those areas, leading to underreporting or overreporting in certain communities. Additionally, if police response varies by neighborhood, the model may not account for differential treatment, which could skew predictions and reinforce biases in the criminal justice system. Altogether, biased patrolling patterns might result in over-policing of certain communities while neglecting others.

What to Submit:

- **For part (a):** One clearly-written short paragraph (approximately 4-7 sentences).
- **For part (b):** Clearly-written and well-thought-out answers addressing (1) and (2) (as described in the problem). Two short paragraphs or one medium paragraph suffice.
- **For part (c):** One clearly-written short paragraph on real-world implications that may follow from ignoring dataset issues.

Image Classification on CIFAR-10

A3. In this problem we will explore different deep learning architectures for image classification on the CIFAR-10 dataset. Make sure that you are familiar with `torch.Tensor`s, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`).

Hint: For loops are costly. Can you vectorize it or use Numpy operations to make it faster in some ways?

A few preliminaries:

- Make sure to read the "Tips for HW4" EdStem post for additional tips about training your models.
- Each network f maps an image $x^{\text{in}} \in \mathbb{R}^{32\times32\times3}$ (3 channels for RGB) to an output $f(x^{\text{in}}) = x^{\text{out}} \in \mathbb{R}^{10}$. The class label is predicted as $\arg\max_{i=0,1,\dots,9} x_i^{\text{out}}$. An error occurs if the predicted label differs from the true label for a given image.
- The network is trained via multiclass cross-entropy loss.
- Create a validation dataset by appropriately partitioning the train dataset. *Hint*: look at the documentation for `torch.utils.data.random_split`. Make sure to tune hyperparameters like network architecture and step size on the validation dataset. Do **NOT** validate your hyperparameters on the test dataset.
- At the end of each epoch (one pass over the training data), compute and print the training and validation classification accuracy.
- While one could train a network for hundreds of epochs to reach convergence and maximize accuracy, this can be prohibitively time-consuming, so feel free to train for just a dozen or so epochs.

For parts (a) and (b), apply a hyperparameter tuning method (e.g. random search, grid search, etc.) using the validation set, report the hyperparameter configurations you evaluated and the best set of hyperparameters from this set, and plot the training and validation classification accuracy as a function of epochs. Produce a separate line or plot for each hyperparameter configuration evaluated (top 3 configurations is sufficient to keep the plots clean). Finally, evaluate your best set of hyperparameters on the test data and report the test accuracy.

Note 1: Please refer to the provided notebook with starter code for this problem, on the course website. That notebook provides a complete end-to-end example of loading data, training a model using a simple network with a fully-connected output and no hidden layers (this is equivalent to logistic regression), and performing evaluation using canonical Pytorch. We recommend using this as a template for your implementations of the models below.
Note 2: If you are attempting this problem and do not have access to GPU we highly recommend using Google Colab. The provided notebook includes instructions on how to use GPU in Google Colab.

Here are the network architectures you will construct and compare.

a. *[18 points]* **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 (M could be in the hundreds). 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 $W_1 \in \mathbb{R}^{M\times3072}$, $b_1 \in \mathbb{R}^M$, $W_2 \in \mathbb{R}^{10\times M}$, $b_2 \in \mathbb{R}^{10}$. 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.

Solution:
Note: Top 3 Configurations are ranked, bolded and have test accuracy below.

Hyperparameter Search List: Method - Random Search (Epochs = 25)
Trying learning rate: 1.7904477307362864e-05 hidden size: 128 momentum: 0.8729248031707446 - Validation Accuracy: 0.25742187

Trying learning rate: 0.0006528940689729769 hidden size: 256 momentum: 0.772012992498581 - Validation Accuracy: 0.3919921875

Trying learning rate: 5.882872432422778e-05 hidden size: 1024 momentum: 0.8932933909911871 - Validation Accuracy: 0.32578125

1. Trying learning rate: 0.00827048887116308 hidden size: 1024 momentum: 0.7153118637375578 - Validation Accuracy: 0.507421875
Test Accuracy: 0.504746835443038

Trying learning rate: 0.0002961433356233488 hidden size: 256 momentum: 0.9402098994601886 - Validation Accuracy: 0.42578125

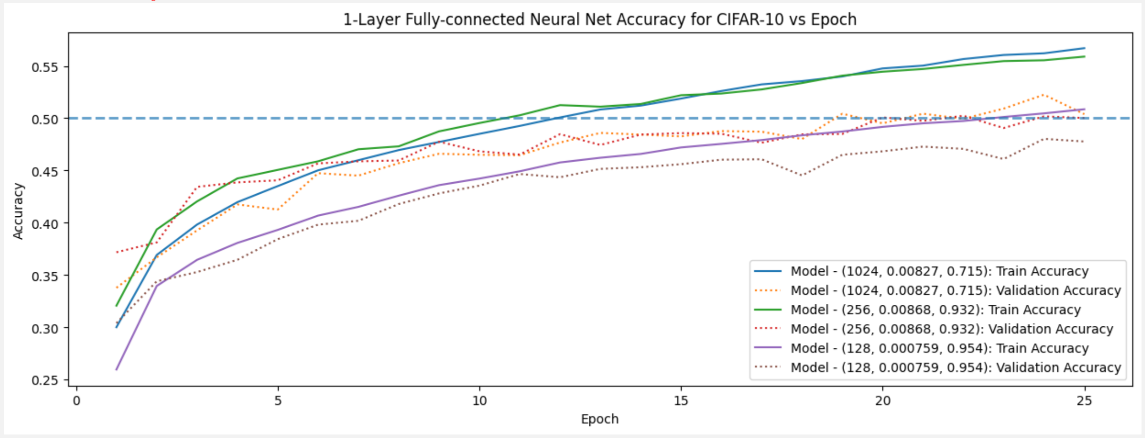
Trying learning rate: 3.1222413607757744e-05 hidden size: 128 momentum: 0.9860480438287756 - Validation Accuracy: 0.3740234375

Trying learning rate: 9.453965099370138e-06 hidden size: 1024 momentum: 0.7904681328703714 - Validation Accuracy: 0.198828125

Trying learning rate: 0.0008736756104278588 hidden size: 512 momentum: 0.8037009966115737 - Validation Accuracy: 0.42421875

2. Trying learning rate: 0.008681289340257909 hidden size: 256 momentum: 0.9316038103164421 - Validation Accuracy: 0.5046875
Test Accuracy: 0.49831882911392406

3. Trying learning rate: 0.0007586019807728006 hidden size: 128 momentum: 0.9544475457600203 - Validation Accuracy: 0.4939453125
Test Accuracy: 0.4810126582278481



b. *[18 points]* **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 $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k)\times(33-k)\times M}$.

Solution:
Note: Top 3 Configurations are ranked, bolded and have test accuracy below.

Hyperparameter Search List: Method - Random Search (Epochs = 30)
Trying learning rate: 0.00016087837155232316 hidden size: 80 momentum: 0.8544568592034737 kernel size: 7 pool size: 6 - Validation Accuracy: 0.3880859375

1. Trying learning rate: 0.00661301161427712 hidden size: 80 momentum: 0.972770973458486 kernel size: 3 pool size: 6 - Validation Accuracy: 0.666796875
Test Accuracy: 0.653989932

Trying learning rate: 0.0015932556810924032 hidden size: 100 momentum: 0.8335843196323389 kernel size: 5 pool size: 6 - Validation Accuracy: 0.55625

2. Trying learning rate: 0.008556426171577872 hidden size: 60 momentum: 0.7254331137969444 kernel size: 7 pool size: 6 - Validation Accuracy: 0.6234375
Test Accuracy: 0.61582619

3. Trying learning rate: 0.0029858531179861774 hidden size: 100 momentum: 0.8495980404349044 kernel size: 3 pool size: 6 - Validation Accuracy: 0.60703125
Test Accuracy: 0.59216785

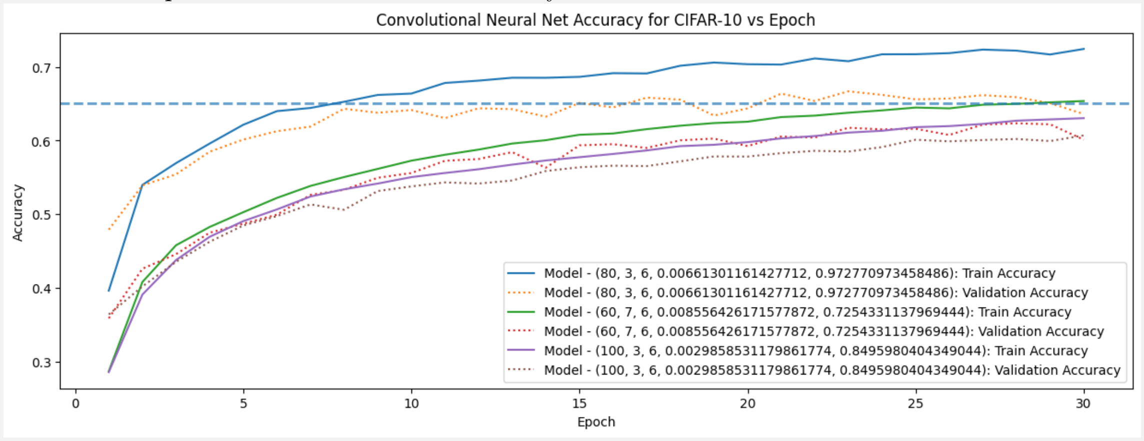
Trying learning rate: 0.0007262300949488417 hidden size: 60 momentum: 0.8510879232478871 kernel size: 7 pool size: 6 - Validation Accuracy: 0.504296875

Trying learning rate: 1.6420956998283162e-05 hidden size: 80 momentum: 0.8745893759124831 kernel size: 3 pool size: 8 - Validation Accuracy: 0.18046875

Trying learning rate: 0.0008866948388685951 hidden size: 100 momentum: 0.8998194727395175 kernel size: 5 pool size: 8 - Validation Accuracy: 0.519140625

Trying learning rate: 0.0005174979564089593 hidden size: 60 momentum: 0.9583223999896975 kernel size: 5 pool size: 8 - Validation Accuracy: 0.5396484375

Trying learning rate: 0.0025156316758898067 hidden size: 80 momentum: 0.7922161961946913 kernel size: 7 pool size: 6 - Validation Accuracy: 0.5916015625



- 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.

The number of hyperparameters to tune, combined with the slow training times, will hopefully give you a taste of how difficult it is to construct networks with good generalization performance. State-of-the-art networks can have dozens of layers, each with their own hyperparameters to tune. Additional hyperparameters you are welcome to play with, if you are so inclined, include: changing the activation function, replace max-pool with average-pool, adding more convolutional or fully connected layers, and experimenting with batch normalization or dropout.

Code: hw4-a3.py

```
# -*- coding: utf-8 -*-
"""hw4_a3_cifar_image_classification.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/17HQFOC2lNI1tU2Fw1_o3YSr5Q7P8OSk9

# Homework 4: Image Classification on CIFAR-10

## Information before starting

In this problem, we will explore different deep learning architectures for image
classification on the CIFAR-10 dataset. Make sure that you are familiar with torch
`Tensor`s, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers
(`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor
reshaping (`view`). **Make sure to read through all instructions in both this notebook and
in the PDF while completing this problem!**

### Copying this Colab Notebook to your Google Drive

Since the course staff is the author of this notebook, you cannot make any lasting changes
to it. You should make a copy of it to your Google Drive by clicking **File -> Save a Copy
in Drive**.

### Problem Introduction

You've already had some practice using the PyTorch library in HW3, but this problem dives
into training more complex deep learning models.

The specific task we are trying to solve in this problem is image classification. We're
using a common dataset called CIFAR-10 which has 60,000 images separated into 10 classes:
* airplane
* automobile
* bird
* cat
* deer
* dog
* frog
* horse
* ship
* truck

We've provided an end-to-end example of loading data, training a model, and performing
evaluation. We recommend using this code as a template for your implementations of the more
complex models. Feel free to modify or reuse any of the functions we provide.

**Unlike other coding problems in the past, this one does not include an autograded
component.**

### Enabling GPU

We are using Google Colab because it has free GPU runtimes available. GPUs can accelerate
training times for this problem by 10-100x when compared to using CPU. To use the GPU
runtime on Colab, make sure to **enable** the runtime by going to **Runtime -> Change
runtime type -> Select T4 GPU under "Hardware accelerator"**.

Note that GPU runtimes are *limited* on Colab. We recommend limiting your training to
short-running jobs (under 15 minutes each) and spread your work over time, if possible.
Colab *will* limit your usage of GPU time, so plan ahead and be prepared to take breaks
during training. If you have used up your quota for GPU, check back in a day or so to be
able to enable GPU again.

Your code will still run on CPU, so if you are just starting to implement your code or have
been GPU limited by Colab, you can still make changes and run your code - it will just be
quite a bit slower. You can also choose to download your notebook and run locally if you
have a personal GPU or have a faster CPU than the one Colab provides. If you choose to do
this, you may need to install the packages that this notebook depends on to your `cse446`
conda environment or to another Python environment of your choice.

To check if you have enabled GPU, run the following cell. If `device` is `cuda`, it means
that GPU has been enabled successfully.
"""

import torch

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

"""### Submitting your assignment

Once you are done with the problem, make sure to put all of your necessary figures into your
PDF submission. Then, download this notebook as a Python file (`.py`) by going to **File ->
Download -> Download `.py`**. Rename this file as `hw4-a3.py` and upload to the Gradescope
submission for HW4 code.

## End-to-end Example
```

~~~
### Background and Setup

1. We first import all of the dependencies required for this problem:
"""

# Commented out IPython magic to ensure Python compatibility.
import torch
from torch import nn
import numpy as np

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.notebook import tqdm

# %matplotlib inline

"""2. And check if we are using GPU, if it is available. (Make sure to set your runtime to
enable GPU!)"""

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

"""*To use the GPU, you will need to send both the model and the data to a device; this
transfers the model from its default location on CPU to the GPU.*

*Note that torch operations on Tensors will fail if they are not located on the same device.
Here's a small example of how to send the model and data to your device:*

  ```python
  model = model.to(DEVICE)  # Sending a model to GPU

  for x, y in tqdm(data_loader):
    x, y = x.to(DEVICE), y.to(DEVICE)
  ```
*When reading tensors you may need to send them back to cpu, you can do so with `x =
x.cpu()`*

3. Now, let's load the CIFAR-10 data. We can take advantage of public datasets available
through PyTorch torchvision!
"""

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())

"""4. Finally, like we did in HW3, we'll use the PyTorch `DataLoader` to wrap our datasets.
You've already seen that `DataLoader`s handle batching, shuffling, and iterating over data,
and this is really useful for this problem as well!
~~~

```
*Since training on all of the 50,000 training samples can be prohibitively expensive, we
define a flag called* `SAMPLE_DATA` *that controls if we should make the dataset smaller for
faster training time. When* `SAMPLE_DATA=true`, *we'll only use 10% of our training data
when training and performing our hyperparameter searches.*  **Make sure that you've set
`SAMPLE_DATA=false` when you want to perform your final training loops for submission!**
"""

SAMPLE_DATA = False # set this to True if you want to speed up training when searching for
hyperparameters!

batch_size = 128

if SAMPLE_DATA:
  train_dataset, _ = random_split(train_dataset, [int(0.1 * len(train_dataset)), int(0.9 *
  len(train_dataset))]) # get 10% of train dataset and "throw away" the other 90%

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

# 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
)

"""Now, we're ready to train!

### Logistic Regression Example

Let's first take a look at our data to get an understanding of what we are doing. As a
reminder, CIFAR-10 is a dataset containing images split into 10 classes.
"""

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.

1. First, let's create our model. Note: for a linear model we could flatten the data before
passing it into the model, but that is not the case for convolutional neural networks.
"""

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)

"""2. 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]]:
    """
    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
    epoch.
    """

    loss = nn.CrossEntropyLoss()
    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []
    for e in tqdm(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

"""3. Now, let's define our hyperparameter search. For this problem, we will be using SGD.
The two hyperparameters for our linear model trained with SGD are the learning rate and
momentum. Only learning rate will be searched for in this example, but you will be tuning
multiple hyperparameters. **Feel free to experiment with hyperparameters and how you search.
We recommend implementing random search!**

*Note: We ask you to plot the accuracies for the 3 best models for each structure, so you
will need to return multiple sets of hyperparameters for the homework.*
"""

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
    best_loss = torch.tensor(np.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

"""4. 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 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)

"""5. We can also 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}")

"""The rest is yours to code. You can structure the code any way you would like.

We do advise making using code cells and functions (train, search, predict etc.) for each
subproblem, since they will make your code easier to debug.

Also note that several of the functions above can be reused for the various different models
you will implement for this problem; i.e., you won't need to write a new `evaluate()`.

## Your Turn!

The rest is yours to code. You are welcome to structure the code any way you would like.

We do advise making using code cells and functions (train, search, predict etc.) for each
subproblem, since they will make your code easier to debug.

Also note that several of the functions above can be reused for the various different models
you will implement for this problem; i.e., you won't need to write a new `evaluate()`.
Before you reuse functions though, make sure they are compatible with what the assignment is
asking for.

### Submitting Code
```

```
And as a last reminder, once you are done with the problem, make sure to put all of your
necessary figures into your PDF submission. Then, download this notebook as a Python file
(`.py`) by going to **File -> Download -> Download `.py`**. Rename this file as `hw4-a3.py`
and upload to the Gradescope submission for HW4 code.

**Part A. Fully-connected output, 1 fully-connected hidden layer.**

Define model.
"""

def FCHiddenLayer(M: int) -> nn.Module:
    model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(3072, M),
        nn.ReLU(),
        nn.Linear(M, 10)
    )
    return model.to(DEVICE)

"""Define method to train model."""

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
    epoch.
    """

    loss = nn.CrossEntropyLoss()
    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []
    for e in tqdm(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

"""Hyperparameter Search: Random Search"""

def nn_parameter_search(train_loader: DataLoader,
                     val_loader: DataLoader,
                     model_fn:Callable[[int], 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.
    """
    best_3_models = {}
    num_iter = 10

    for _ in range(num_iter):
        M = np.random.choice([128, 256, 512, 1024])
        lr = 10 ** np.random.uniform(-5, -2)
        momentum = np.random.uniform(0.7, 0.99)
        print(f"Trying hidden size: {M} learning rate: {lr} momentum: {momentum}")
        model = model_fn(M)
        optim = SGD(model.parameters(), lr, momentum=momentum)
        train_loss, train_acc, val_loss, val_acc = train(
            model,
            optim,
            train_loader,
            val_loader,
            epochs=25
            )

        print(f"Validation Accuracy: {max(val_acc)}")
        best_3_models[max(val_acc)] = [(M, lr, momentum), train_acc, val_acc]
        if len(best_3_models) > 3:
                best_3_models.pop(min(best_3_models))

    return best_3_models

"""Train and Evaluate."""

best_3_models = nn_parameter_search(train_loader, val_loader, FCHiddenLayer)

epochs = range(1, 26)
for i, (key, value) in enumerate(best_3_models.items()):
  plt.plot(epochs, value[1], label=f"Model - ({value[0][0]}, {value[0][1]}, {value[0][2]}):
  Train Accuracy")
  plt.plot(epochs, value[2], label=f"Model - ({value[0][0]}, {value[0][1]}, {value[0][2]}):
  Validation Accuracy", linestyle="dotted")
plt.axhline(y=0.50, linestyle='--', linewidth=2, alpha=0.7)
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.title("1-Layer Fully-connected Neural Net Accuracy for CIFAR-10 vs Epoch")
plt.show()

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
```

```
for (M, lr, momentum) in best_3_models:
  model = FCHiddenLayer(M)
  optim = SGD(model.parameters(), lr, momentum=momentum)
  _, _, _, _ = train(
      model, optim, train_loader, val_loader, 25)
  test_loss, test_acc = evaluate(model, test_loader)
  print(f"Test Accuracy: {test_acc}")

"""**Part B. Convolutional Layer with max-pool and fully-connected output.**

Define model.
"""

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

"""Define method to train model."""

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
    epoch.
    """

    loss = nn.CrossEntropyLoss()
    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []
    for e in tqdm(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

"""Hyperparameter Search: Random Search"""

def cnn_parameter_search(train_loader: DataLoader,
                     val_loader: DataLoader,
                     model_fn:Callable[[int, int, int], 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.
    """
    best_3_models = {}
    num_iter = 10

    for _ in range(num_iter):
        M = np.random.choice([60, 80, 100])
        k = np.random.choice([3, 5, 7])
        N = np.random.choice([4, 6, 8, 10])
        lr = 10 ** np.random.uniform(-5, -2)
        momentum = np.random.uniform(0.7, 0.99)
        print(f"Trying hidden size: {M} kernel size: {k} pool size: {N} learning rate: {lr}
        momentum: {momentum}")
        model = model_fn(M, k, N)
        optim = SGD(model.parameters(), lr=lr, momentum=momentum)
        train_loss, train_acc, val_loss, val_acc = train(
            model,
            optim,
            train_loader,
            val_loader,
            epochs=30
            )

        print(f"Validation Accuracy: {max(val_acc)}")
        best_3_models[max(val_acc)] = [(M, k, N, lr, momentum), train_acc, val_acc]
        if len(best_3_models) > 3:
                best_3_models.pop(min(best_3_models))

    return best_3_models

"""Train and Evaluate."""

best_3_models = cnn_parameter_search(train_loader, val_loader, CNNMaxpoolFC)

epochs = range(1, 31)
plt.figure(figsize=(15, 5))
for i, (key, value) in enumerate(best_3_models.items()):
  plt.plot(epochs, value[1], label=f"Model - ({value[0][0]}, {value[0][1]}, {value[0][2]},
  {value[0][3]}, {value[0][4]}): Train Accuracy")
  plt.plot(epochs, value[2], label=f"Model - ({value[0][0]}, {value[0][1]}, {value[0][2]},
  {value[0][3]}, {value[0][4]}): Validation Accuracy", linestyle="dotted")
plt.axhline(y=0.65, linestyle='--', linewidth=2, alpha=0.7)
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.title("Convolutional Neural Net Accuracy for CIFAR-10 vs Epoch")
plt.show()

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

for (M, k, N, lr, momentum) in best_3_models:
  model = CNNMaxpoolFC(M, k, N)
  optim = SGD(model.parameters(), lr=lr, momentum=momentum)
  _, _, _, _ = train(
      model, optim, train_loader, val_loader, 30)
  test_loss, test_acc = evaluate(model, test_loader)
  print(f"Test Accuracy: {test_acc}")
```

What to Submit:

- **For parts (a)-(b):** A single plot of the training and validation accuracy for the top 3 hyperparameter configurations you evaluated (x-axis is training epoch; y-axis is accuracy; this plot should contain 6 lines total). If it took fewer than 3 hyperparameter configurations to pass the performance threshold, plot all hyperparameter configurations you evaluated. A horizontal line should be plotted at the targeted threshold (50% or 65%). Validation lines should be dotted, and training lines should be solid.
- **For parts (a)-(b):** List the hyperparameter values you searched over and your search method (random, grid, etc.). Provide the values of best performing hyperparameters, and accuracy of best models on test data.
- **For parts (a)-(b):** Code. You should convert your code (the .ipynb notebook) into a Python (.py) file, rename it to `hw4-a3.py`, and submit it to the corresponding Gradescope submission. To download the file from Google Colab, you can go to File ¿ Download ¿ Download as .py.

Matrix Completion and Recommendation System

A4.

Note: Please refer to the provided notebook with starter code for this problem, on the course website. The notebook provides a template for each part of the problem, and includes code to help load the data properly. We recommend creating a copy of the starter notebook and completing the assignment by filling out the template.

Hint: For loops are costly. Can you vectorize it or use Numpy operations to make it faster in some ways?

You will build a personalized movie recommendation system. We will use the 100K MovieLens dataset available at https://grouplens.org/datasets/movielens/100k/. There are $m = 1682$ movies and $n = 943$ users. Each user rated at least 20 movies, but some watched many more. The total dataset contains $100,000$ total ratings from all users. The goal is to recommend movies the users haven't seen.

Consider a matrix $R \in \mathbb{R}^{m\times n}$ where the entry $R_{i,j} \in \{1, \dots, 5\}$ represents the jth user's rating on movie i. A higher value represents that the user is more satisfied with the movie.

We may think of our historical data as some observed entries of this matrix while many remain unknown, and we wish to estimate the unknown ratings that each user would assign to each movie. We could use these ratings to recommend the "best" movies for each user.

The dataset contains user and movie metadata which we will ignore. We strictly use the ratings contained in the `u.data` file. Use this data file and the following python code to construct a training and test set:

```
import csv
import numpy as np
data = []
with open('u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)

num_observations = len(data)   # num_observations = 100,000
num_users = max(data[:,0])+1   # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1   # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]
```

The arrays `train` and `test` contain R_{train} and R_{test}, respectively. Each line takes the form "j, i, s", where j is the user index, i is the movie index, and s is the user's score.

Using `train`, you will train a model that can predict $\widehat{R} \in \mathbb{R}^{m\times n}$, how every user would rate every movie. You will evaluate your model based on the average squared error on `test`:

$$\mathcal{E}_{\text{test}}(\widehat{R}) = \frac{1}{|\text{test}|} \sum_{(i,j,R_{i,j})\in\text{test}} (\widehat{R}_{i,j} - R_{i,j})^2.$$

Low-rank matrix factorization is a baseline method for personalized recommendation. It learns a vector representation $u_i \in \mathbb{R}^d$ for each movie and a vector representation $v_j \in \mathbb{R}^d$ for each user, such that the inner product

$\langle u_i, v_j \rangle$ approximates the rating $R_{i,j}$. You will build a simple latent factor model.

You will implement multiple estimators and use the inner product $\langle u_i, v_j \rangle$ to predict if user j likes movie i in the test data. For simplicity, we will put aside best practices and choose hyperparameters by using those that minimize the test error. You may use fundamental operators from `numpy` or `pytorch` in this problem (`numpy.linalg.lstsq, SVD, autograd,` etc.) but not any precooked algorithm from a package like `scikit-learn`. If there is a question whether some package is not allowed for use in this problem, it probably is not appropriate.

a. *[5 points]* Our first estimator pools all users together and, for each movie, outputs as its prediction the average user rating of that movie in `train`. That is, if $\mu \in \mathbb{R}^m$ is a vector where μ_i is the average rating of the users that rated the ith movie, write this estimator $\widehat{R}$ as a rank-one matrix.
Compute the estimate $\widehat{R}$. What is $\mathcal{E}_{\text{test}}(\widehat{R})$ for this estimate?

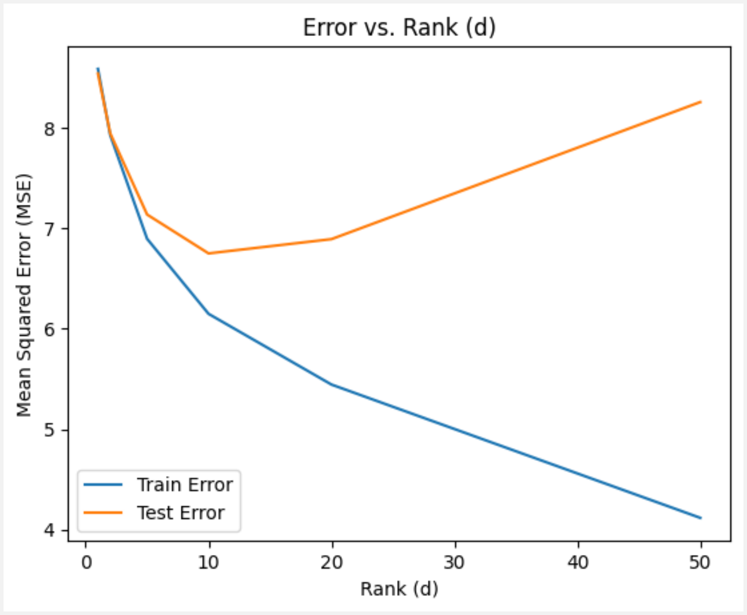
Solution:
$\widehat{R} = \mu \mathbf{1}^\top$ where $\mathbf{1} \in \mathbb{R}^n$ is a vector of ones with length equal to the number of users, n.
$\mathcal{E}_{\text{test}}(\widehat{R}) = 1.0636$

b. *[5 points]* Allocate a matrix $\widetilde{R}_{i,j} \in \mathbb{R}^{m \times n}$ and set its entries equal to the known values in the training set, and 0 otherwise. Let $\widehat{R}^{(d)}$ be the best rank-d approximation (in terms of squared error) approximation to $\widetilde{R}$. This is equivalent to computing the singular value decomposition (SVD) and using the top d singular values. This learns a lower-dimensional vector representation for users and movies, assuming that each user would give a rating of 0 to any movie they have not reviewed.

- For each $d = 1, 2, 5, 10, 20, 50$, compute the estimator $\widehat{R}^{(d)}$. We recommend using an efficient solver such as `scipy.sparse.linalg.svds`.
- Plot the average squared error of predictions on the training set and test set on a single plot, as a function of d.

Note that, in most applications, we would not actually allocate a full $m \times n$ matrix. We do so here only because our data is relatively small and it is instructive.

Solution:



c. *[10 points]* Replacing all missing values by a constant may impose strong and potentially incorrect assumptions on the unobserved entries of R. A more reasonable choice is to minimize the MSE (mean squared error) only on rated movies. Define a loss function:

$$\mathcal{L}\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) := \sum_{(i,j,R_{i,j})\in\text{train}} (\langle u_i, v_j\rangle - R_{i,j})^2 + \lambda \sum_{i=1}^m \|u_i\|_2^2 + \lambda \sum_{j=1}^n \|v_j\|_2^2 \tag{1}$$

where $\lambda > 0$ is the regularization coefficient. We will implement algorithms to learn vector representations by minimizing (1). Note: we define the loss function here as the sum of squared errors; be careful to calculate and plot the mean squared error for your results.

Since this is a non-convex optimization problem, the initial starting point and hyperparameters may affect the quality of $\widehat{R}$. You may need to tune λ and σ to optimize the loss you see.

- *Alternating minimization*: First, randomly initialize $\{u_i\}$ and $\{v_j\}$. Then, alternatate between (1) minimizing the loss function with respect to $\{u_i\}$ by treating $\{v_j\}$ as fixed; and (2) minimizing the loss function with respect to $\{v_j\}$ by treating $\{u_i\}$ as fixed. Repeat (1) and (2) until both $\{u_i\}$ and $\{v_j\}$ converge. Note that when one of $\{u_i\}$ or $\{v_j\}$ is given, minimizing the loss function with respect to the other part has a closed-form solution. Indeed, it can be shown that when minimizing with respect to a *single* u_i (with $\{v_j\}$ fixed), the gradient is given by:

$$\nabla_{u_i} L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) = 2\left(\sum_{j\in r(i)} v_j v_j^T + \lambda I\right) u_i - 2\sum_{j\in r(i)} R_{i,j} v_j \tag{2}$$

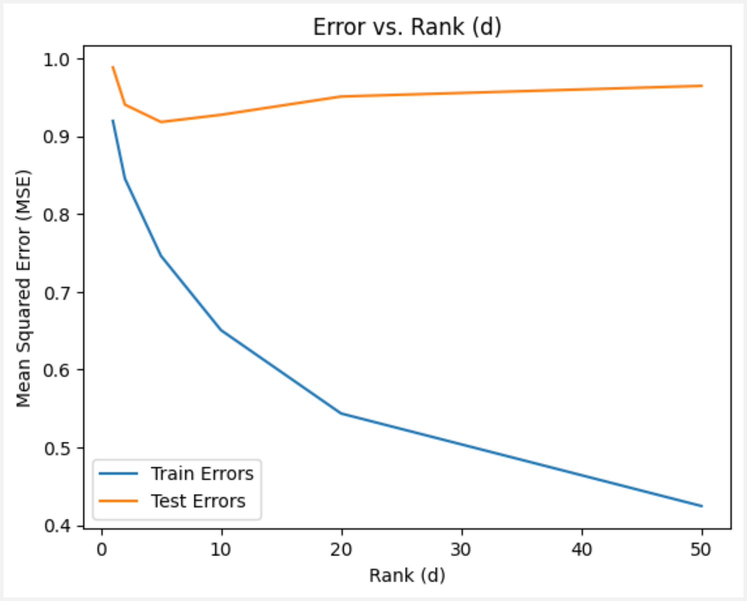
  where here $r(i)$ is a shorthand for the set of users who have reviewed movie i in the training set, or more formally, $r(i) = \{j : (j, i, R_{i,j}) \in \text{train}\}$. Setting the overall gradient to be equal to 0 gives us that

$$\arg\min_{u_i} L\Big(\{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n\Big) = \left(\sum_{j\in r(i)} v_j v_j^T + \lambda I\right)^{-1} \left(\sum_{j\in r(i)} R_{i,j} v_j\right) \tag{3}$$

  Note that this update rule is for a single vector u_i, whereas you should update all of the $\{u_i\}_{i=1}^m$ in one round. When it comes to the alternate step which involves fixing $\{u_i\}$ and minimizing $\{v_j\}$, an analogous calculation will give you a very similar update rule.

- Try $d \in \{1, 2, 5, 10, 20, 50\}$ and plot the mean squared error of train and test as a function of d.

Solution:



Some hints:

- Common choices for initializing the vectors $\{u_i\}_{i=1}^m$, $\{v_j\}_{j=1}^n$ include: entries drawn from `np.random.rand()` scaled by some scale factor $\sigma > 0$ (σ is an additional hyperparameter), or using one of the solutions from part b or c.
- The only $m \times n$ matrix you need to allocate is probably for $\widetilde{R}$.
- It is **crucial** that the squared error part of the loss is only defined w.r.t. $R_{i,j}$ that actually exist in the training set. Consider implementing some type of data structures that allow you to keep track of $r(i)$ as well as the reverse mapping $r^{-1}(j)$ from movies to relevant users.

Code: hw4-a4.py

```
# -*- coding: utf-8 -*-
"""hw4-a4-matrix-completion-and-recommendation-systems.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/11Bgh1tuvrNXCBEMamhJ37d4KC1F3xf-B

# Homework 4: Matrix Completion and Recommendation System


## Information before starting

In this problem, we will be building a personalized movie recommendation system! To make
these recommendations, we'll build on what we've learned in lecture about SVD and what we've
practiced so far with Python and Python packages such as NumPy and PyTorch.

### Copying this Colab Notebook to your Google Drive
```

```
Since the course staff is the author of this notebook, you cannot make any lasting changes
to it. You should make a copy of it to your Google Drive by clicking **File -> Save a Copy
in Drive**.

### Problem Introduction

We will use the 100K MovieLens dataset available at
https://grouplens.org/datasets/movielens/100k/ to estimate unknown user ratings given their
previous ratings. Run the code block below to download the dataset.
"""

# @title loading dataset
!rm -rf ml-100k*
!wget https://files.grouplens.org/datasets/movielens/ml-100k.zip
!unzip ml-100k.zip
!mv ml-100k/u.data .

"""### Compute

This problem should not require using GPU. Since Google Colab will limit your GPU usage, we
recommend saving your GPU quota for HW4 A3 and making sure that your runtime is set to CPU
by going to **Runtime -> Change runtime type -> Select CPU under "Hardware accelerator"**.

### Submitting your assignment

Once you are done with the problem, make sure to put all of your necessary figures into your
PDF submission. Then, download this notebook as a Python file (`.py`) by going to **File ->
Download -> Download `.py`**. Rename this file as `hw4-a4.py` and upload to the Gradescope
submission for HW4 code.

## Code: Setup

Let's start by importing the packages that we'll need to complete this problem.
"""

import csv
import numpy as np
from scipy.sparse.linalg import svds
import matplotlib.pyplot as plt
import torch

"""Now, let's load the 100K MovieLens data. If you have downloaded the `u.data` file and
uploaded to the "Files" tab, the following code block will construct training and test sets
for you. There are $m = 1682$ movies and $n = 943$ users in the dataset, and each user has
rated at least 20 movies. The total dataset has 100,000 total ratings from all users, and
our goal will be to estimate the unknown ratings that each user would assign to each movie.
These ratings can then be used to recommend the "best" movies for each user!"""

data = []
with open('u.data') as csvfile:
    spamreader = csv.reader(csvfile, delimiter='\t')
    for row in spamreader:
        data.append([int(row[0])-1, int(row[1])-1, int(row[2])])
data = np.array(data)
```

```
num_observations = len(data)  # num_observations = 100,000
num_users = max(data[:,0])+1  # num_users = 943, indexed 0,...,942
num_items = max(data[:,1])+1  # num_items = 1682 indexed 0,...,1681

np.random.seed(1)
num_train = int(0.8*num_observations)
perm = np.random.permutation(data.shape[0])
train = data[perm[0:num_train],:]
test = data[perm[num_train::],:]

print(f"Successfully loaded 100K MovieLens dataset with",
      f"{len(train)} training samples and {len(test)} test samples")

"""For this problem, we will consider a matrix $R \in \mathbb{R}^{m \times n}$ where the
entry $R_{i,j} \in \{1,...,5\}$ represents the $j$th user's rating on movie $i$. A higher
value represents that the user is more staisfied with the movie.

## Code: Assignment

The rest is yours to code! We provide some scaffolding for your implementation, but feel
free to modify it and implement however you would like to. You may use fundamental operators
from `NumPy` and `PyTorch` in this problem, such as `numpy.linalg.lstsq, SVD, autograd`,
etc., but you many not use any precooked algorithm from a package like `scikit-learn`.

### Part (a)

Our first estimator pools all users together and, for each movie, outputs as its prediction
the average user rating of that movie in ``train``. That is, if $\mu \in \mathbb{R}^m$ is a
vector where $\mu_i$ is the average rating of the users that rated the $i$-th movie. Write
this estimator $\widehat{R}$ as a rank-one matrix.

Compute the estimate $\widehat{R}$. What is $\mathcal{E}_{\rm test} (\widehat{R})$ for this
estimate?
"""

# Your code goes here. You should:

# 1. Compute estimate and

# 2. Evaluate test error
mu = np.zeros(num_items)
for i in range(num_items):
  ratings = train[train[:,1] == i,2]
  if len(ratings) > 0:
    mu[i] = np.mean(ratings)
  else:
    mu[i] = 0

r_hat = np.outer(mu, np.ones(num_users))

test_error = 0
for (user, movie, actual_rating) in test:
    predicted_rating = r_hat[movie, user]
    test_error += (predicted_rating - actual_rating) ** 2
test_error /= len(test)

print(f"Test Error (E_test): {test_error:.4f}")

"""### Part (b)
Allocate a matrix $\widetilde{R}_{i, j} \in \mathbb{R}^{m \times n}$ and set its entries
equal to the known values in the training set, and $0$ otherwise.

Let $\widehat{R}^{(d)}$ be the best rank-$d$ approximation (in terms of squared error)
approximation to $\widetilde{R}$. This is equivalent to computing the singular value
decomposition (SVD) and using the top $d$ singular values. This learns a lower-dimensional
vector representation for users and movies, assuming that each user would give a rating of
$0$ to any movie they have not reviewed.

- For each $d = 1, 2, 5, 10, 20, 50$, compute the estimator $\widehat{R}^{(d)}$. We
recommend using an efficient solver, such as ``scipy.sparse.linalg.svds``.
- Plot the average squared error of predictions on the training set and test set on a single
plot, as a function of $d$.
"""

# Your code goes here
# Create the matrix R twiddle (\widetilde{R}).
r_twiddle = np.zeros((num_items, num_users))
for (user, movie, rating) in train:
  r_twiddle[movie, user] = rating

from sre_constants import error
from re import S
# Your code goes here
def construct_estimator(d, r_twiddle):
  U, S, Vt = svds(r_twiddle, d)
  return (U @ np.diag(S)) @ Vt


def get_error(d, r_twiddle, dataset):
  error = 0
  r_hat = construct_estimator(d, r_twiddle)
  for (user, movie, actual_rating) in dataset:
    predicted_rating = r_hat[movie, user]
    error += (predicted_rating - actual_rating) ** 2
  return error / len(dataset)

# Your code goes here
# Evaluate train and test error for: d = 1, 2, 5, 10, 20, 50.
d_vals = [1, 2, 5, 10, 20, 50]
train_errors = []
test_errors = []

for d in d_vals:
  train_error = get_error(d, r_twiddle, train)
  test_error = get_error(d, r_twiddle, test)
  train_errors.append(train_error )
  test_errors.append(test_error)

# Your code goes here
# Plot both train and test error as a function of d on the same plot.
plt.plot(d_vals, train_errors, label='Train Error')
plt.plot(d_vals, test_errors, label='Test Error')
plt.xlabel('Rank (d)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Error vs. Rank (d)')
plt.legend()
plt.show()

"""### Part (c)
Replacing all missing values by a constant may impose strong and potentially incorrect
assumptions on the unobserved entries of $R$. A more reasonable choice is to minimize the
mean squared error (MSE) only on rated movies. Define a loss function:
$$
\mathcal{L} \left( \{u_i\}_{i=1}^m, \{v_j\}_{j=1}^n \right) :=
\sum_{(i, j, R_{i, j}) \in {\rm train}} (\langle u_i,v_j\rangle - R_{i,j})^2 +
\lambda \sum_{i=1}^m \|u_i\|_2^2 +
\lambda \sum_{j=1}^n \|v_j\|_2^2
$$
where $\lambda > 0$ is the regularization coefficient. We will implement algorithms to learn
vector representations by minimizing the above loss. You may need to tune $\lambda$ and
$\sigma$ to optimize the loss.

Implement alternating minimization (as defined in the homework spec) and plot the MSE of
``train`` and ``test`` for $d \in \{1, 2, 5, 10, 20, 50\}$.

*Note: we define the loss function here as the sum of squared errors; be careful to
calculate and plot the mean squared error for your results*
"""

# Your code goes here. You are welcome to change the parameter lists and/or write new
functions to complete this part of the assignment.
# In particular, you will likely also want to use R twiddle, and you may want to create
global data structures to store observed entries.
# These global data structures might look like mappings of users to the movies they've
reviewed, and of movies to the users who have reviewed that movie.

def closed_form_u(r_twiddle, V, U, l):
  for movie in range(num_items):
    users = MTU.get(movie, [])

    if users:
      user_prefs = V[users, :]
      ratings = r_twiddle[movie, users]
      U[movie, :] = np.linalg.solve((user_prefs.T @ user_prefs) + l *
      np.identity(user_prefs.shape[1]), (ratings @ user_prefs))
    else:
      U[movie, :] = np.zeros(U.shape[1])

  return U


def closed_form_v(r_twiddle, U, V, l):
  for user in range(num_users):
    movies = UTM.get(user, [])
    if movies:
      movie_info = U[movies, :]
      ratings = r_twiddle[movies, user]
      V[user, :] = np.linalg.solve((movie_info.T @ movie_info) + l *
      np.identity(movie_info.shape[1]), (ratings @ movie_info))
    else:
      V[user, :] = np.zeros(V.shape[1])

  return V


def construct_alternating_estimator(
    d, r_twiddle, l=10.0, delta=1e-1, sigma=0.1, U=None, V=None
):
  prevU = None
  prevV = None

  if U is None:
    U = sigma * np.random.randn(num_items, d)
  if V is None:
    V = sigma * np.random.randn(num_users, d)

  while (prevU is None and prevV is None) or np.max(np.abs(U - prevU)) >= delta or
  np.max(np.abs(V - prevV)) >= delta:
    prevU = np.copy(U)
    prevV = np.copy(V)
    U = closed_form_u(r_twiddle, V, U, l)
    V = closed_form_v(r_twiddle, U, V, l)

  return U @ V.T


# Your code goes here
# Any additional functions that you may write to help implement alternating minimization.
UTM = {}
MTU = {}

for user, movie, rating in train:
  UTM.setdefault(user, []).append(movie)
  MTU.setdefault(movie, []).append(user)

train_users = train[:, 0]
train_movies = train[:, 1]
train_ratings = train[:, 2]

r_twiddle = np.zeros((num_items, num_users))
r_twiddle[train_movies, train_users] = train_ratings

# Your code goes here
# Evaluate train and test error for: d = 1, 2, 5, 10, 20, 50.
d_values = [1, 2, 5, 10, 20, 50]
train_errors = []
test_errors = []
```

```
test_users = test[:, 0]
test_movies = test[:, 1]
test_ratings = test[:, 2]

for d in d_values:
  R_hat = construct_alternating_estimator(d, r_twiddle, sigma=5, U=None, V=None)
  train_errors.append(np.mean((R_hat[train_movies, train_users] - train_ratings)**2))
  test_errors.append(np.mean((R_hat[test_movies, test_users] - test_ratings)**2))

# Your code goes here
# Plot both train and test error as a function of d on the same plot.
plt.plot(d_values, train_errors, label='Train Errors')
plt.plot(d_values, test_errors, label='Test Errors')
plt.xlabel('Rank (d)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Error vs. Rank (d)')
plt.legend()
plt.show()
```

What to Submit:

- **For part a**: A mathematical expression for $\hat{R}$. Value for $\mathcal{E}_{test}(\hat{R})$.
- **For part b**: Plot of MSE on training and test set vs. d.
- **For part c**: Plot of MSE on training and test set vs. d.
- **For parts a-c**: Code. You should convert your code (the .ipynb notebook) into a Python (.py) file, rename it to `hw4-a4.py`, and submit it to the corresponding Gradescope submission. To download the file from Google Colab, you can go to File ¿ Download ¿ Download as .py.

k-means clustering

A5. Given a dataset $\mathbf{x}_1, ..., \mathbf{x}_n \in \mathbb{R}^d$ and an integer $1 \le k \le n$, recall the following k-means objective function

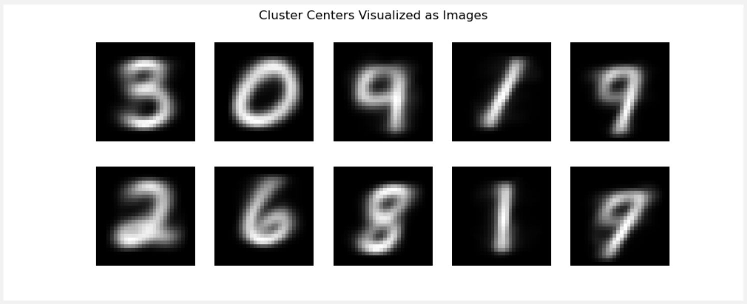
$$\min_{\pi_1,\dots,\pi_k} \sum_{i=1}^{k} \sum_{j\in\pi_i} \|\mathbf{x}_j - \mu_i\|_2^2 \,, \quad \mu_i = \frac{1}{|\pi_i|} \sum_{j\in\pi_i} \mathbf{x}_j \,. \tag{4}$$

Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, ..., n\}$. The objective (4) is NP-hard[1] to find a global minimizer of. Nevertheless, Lloyd's algorithm (discussed in lecture) typically works well in practice.[2]

a. *[5 points]* Implement Lloyd's algorithm for solving the k-means objective (4). Do not use any off-the-shelf implementations, such as those found in `scikit-learn`.

b. *[5 points]* 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.

Solution:



Note on Time to Run — The runtime of a good implementation for this problem should be fairly fast (a few minutes); if you find it taking upwards of one hour, please check your implementation! (Hint: **For loops are costly.** Can you vectorize it or use Numpy operations to make it faster in some ways? If not, is looping through data-points or through centers faster?)

What to Submit:

- **For part (a):** Nothing required in PDF submission.
- **For part (b):** 10 images of cluster centers.
- **For parts (a)-(b):** Code through corresponding Gradescope coding submission.

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

Random Fourier Features

B1. Kernel methods such as Logistic Regression are considered memory-based learners. Rather than learning a mapping from a set of input features $\mathcal{X} \subset \mathbb{R}^d$ to outputs in $\mathcal{Y}$, they *remember* all training examples $(\mathbf{x_i}, y_i)$ and learn a corresponding weight for them.

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{N} \omega_i k(\mathbf{x_i}, \mathbf{x})$$

After learning the weight vector $\mathbf{w} = [\mathbf{w}_1, ..., \mathbf{w}_N]$, we can make prediction on unseen samples using the *kernel function* k between all training samples and $\mathbf{x}$. Kernel methods are attractive because they rely on the *kernel trick*. Any positive definite function $k(\mathbf{x}, \mathbf{x}')$ with $\mathbf{x}, \mathbf{x}' \in \mathbb{R}^d$ defines a function ψ mapping $\mathbb{R}^d$ to a higher-dimensional space such that the inner product between datapoints can be quickly computed as $\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle = k(\mathbf{x}, \mathbf{x}')$. In essence, the kernel trick is an efficient way to learn a linear decision boundary in a higher dimension space than that of $\mathcal{X}$.

The kernel trick can be prohibitively expensive for large datasets. This is because the memory-based algorithm accesses the data through evaluations of the kernel matrix $k(x, x')$ which grows in proportion to the dataset size N.
Instead of relying on the implicit feature mapping ψ provided by the kernel trick, suppose we can approximate the kernel function k as the inner product of two vectors in $\mathbb{R}^D$. Mathematically, we would like to find a mapping $\mathbf{z}$:

$$\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D \qquad \text{such that} \qquad k_p(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle \approx \langle \mathbf{z}(\mathbf{x}), \mathbf{z}(\mathbf{x}') \rangle$$

With this approximation, we no longer require the *kernel trick* to express $\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$ as $k(\mathbf{x}, \mathbf{x}')$. Rather, we can approximate it by directly computing the tractable inner product $\langle \mathbf{z}(\mathbf{x}), \mathbf{z}(\mathbf{x}') \rangle$.

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{N} \omega_i k(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^{N} \omega_i \langle \psi(\mathbf{x}_i), \psi(\mathbf{x}) \rangle \approx \sum_{i=1}^{N} \omega_i \langle \mathbf{z}(\mathbf{x}_i), \mathbf{z}(\mathbf{x}) \rangle = \left(\sum_{i=1}^{N} \omega_i \mathbf{z}(\mathbf{x}_i)^T \right) \mathbf{z}(\mathbf{x}) = \beta^T \mathbf{z}(\mathbf{x})$$

Assuming $\mathbf{z}(\mathbf{x}) = \sigma(M\mathbf{x} + b)$ for some nonlinear function σ, this "approximate" Logistic Regression *can potentially be evaluated much quicker* than the kernel Logistic Regression. To see why, note that the left-hand-side requires evaluating $k(\mathbf{x_i}, \mathbf{x})$ for all $i \in \{1, \ldots, N\}$, in general, if ω_i is not sparse. On the other hand, the right-hand-side just requires computing $\mathbf{z}(\mathbf{x}) = \sigma(M\mathbf{x} + b)$ which is dominated by the time to compute a $D \times d$ matrix-vector product, and then inner product with β which is $\mathbb{R}^D$. Thus, the total computation time for the left-hand-side scales linearly with N, and the right-hand-side scales with just d and D, independent of N! When training the approximate Logistic Regression we also get similar computational savings if $N \gg \max\{d, D\}$.

a. *[15 points]* **Deriving Random Fourier Features**: Bochner's theorem states that a continuous kernel $k(\mathbf{x}, \mathbf{x}') = k(\mathbf{x} - \mathbf{x}')$ on $\mathbb{R}^d$ is positive definite if and only if k is the Fourier transform of a non-negative measure. While we won't delve into the logic of Fourier transforms here, this theorem lets us express the kernel as follows: for any probability distribution $p(\mathbf{w})$ define

$$k_p(\mathbf{x}, \mathbf{x}') := \int_{\mathbb{R}^d} p(\mathbf{w}) e^{i\mathbf{w}^T(\mathbf{x}-\mathbf{x}')} dw = \mathbb{E}_{\mathbf{w}} \left[e^{i\mathbf{w}^T(\mathbf{x}-\mathbf{x}')} \right]$$

where $i = \sqrt{-1}$, the imaginary unit. While any choice of $p(\mathbf{w})$ induces a valid kernel, in this problem we'll be focusing on the Gaussian distribution, namely

$$p(\mathbf{w}) = (2\pi\sigma^2)^{-\frac{D}{2}} e^{-\frac{1}{2\sigma^2}||\mathbf{w}||_2^2} = (2\pi/\gamma^2)^{-\frac{D}{2}} e^{-\gamma^2||\mathbf{w}||_2^2/2} \quad \text{where } \gamma = \frac{1}{\sigma}$$

In this sub-problem, we'll use this Fourier-transform interpretation of k to derive a randomized mapping $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D$ which is an unbiased estimate of the kernel function i.e.

$$\mathbb{E}_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = k_p(\mathbf{x}, \mathbf{x}')$$

If $\mathbf{z}(x)^T \mathbf{z}(x')$ serves as a good approximation to the kernel matrix, we can apply the aforementioned approximation algorithm.

i Use Euler's formula $e^{iy} = \cos(y) + i\sin(y)$ to show that $k_p(\mathbf{x}, \mathbf{x}') = E_{\mathbf{w}}\left[\cos(\mathbf{w}^T(\mathbf{x} - \mathbf{x}'))\right]$.

Hint: If both x and A are real, then $A = \int f(x) + ig(x)dx = \int f(x)dx$.

ii We begin by defining $z_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}$ as

$$z_{\mathbf{w}}(\mathbf{x}) = \sqrt{2}\cos(\mathbf{w}^T\mathbf{x} + b) \qquad \text{where } \mathbf{w} \sim p(\mathbf{w}),\ b \sim \text{Uniform}(0, 2\pi)$$

Note that this is not yet the mapping vector $\mathbf{z}$, but rather a mapping to $\mathbb{R}$. Use part (i) to show that the expected product of $z_{\mathbf{w}}(\mathbf{x})$s is an unbiased estimate of the kernel function i.e.

$$E_{\mathbf{w},b}\left[z_{\mathbf{w}}(\mathbf{x})z_{\mathbf{w}}(\mathbf{x}')\right] = k_p(\mathbf{x}, \mathbf{x}')$$

Hint: For this problem you may use the following identity: $2\cos(\alpha)\cos(\beta) = \cos(\alpha+\beta) + \cos(\alpha-\beta)$.

iii Now we're ready to define our random Fourier features $\mathbf{z} : \mathbb{R}^d \to \mathbb{R}^D$. Let $\mathbf{z}$ be the d-dimensional concatenation of $z_{\mathbf{w}}(\mathbf{x})$s:

$$\mathbf{z}(\mathbf{x}) = \left[\tfrac{1}{\sqrt{D}} z_{\mathbf{w}_1}(\mathbf{x}),\ \tfrac{1}{\sqrt{D}} z_{\mathbf{w}_2}(\mathbf{x}),\ \ldots,\ \tfrac{1}{\sqrt{D}} z_{\mathbf{w}_D}(\mathbf{x})\right]^T$$

Use parts (i) and (ii) to show that the expected inner product of the mapping $\mathbf{z}$ is an unbiased estimate of the kernel function i.e.

$$E_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = k_p(\mathbf{x}, \mathbf{x}')$$

b. *[5 points]* **Random Fourier Features and the RBF Kernel**. As mentioned in part (b), using different distributions $p(\mathbf{w})$ induces different valid kernels. Using the $p(\mathbf{w})$ given in part (a), show that expected value of the inner product between random Fourier features is the RBF kernel i.e.

$$E_{\mathbf{w}}[\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}')] = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||_2^2}{2\gamma^2}\right)$$

Hint: The PDF for a variable $X \in \mathbb{R}^d$ following normal distribution with mean μ and covariance matrix Σ is as follows:

$$P(X = x) = ((2\pi)^d |\Sigma|)^{-1/2} \exp(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu))$$

where $|\Sigma| = \det(\Sigma)$ denote the determinant of matrix Σ. In addition, if $\Sigma = \text{diag}(\sigma^2, \ldots, \sigma^2)$, then $|\Sigma| = \sigma^{2d}$, and $\Sigma^{-1} = \text{diag}(\sigma^{-2}, \ldots, \sigma^{-2})$.

c. *[5 points]* **Concentration Bounds** In part (a) we derived our function $\mathbf{z}$ which serve as a good approximation to the kernel function. Our results let us get an upper bound our approximation error for the kernel function. Explain <u>why</u> we can apply Hoeffding's inequality to obtain

$$p(|\mathbf{z}(\mathbf{x})^T \mathbf{z}(\mathbf{x}') - k(\mathbf{x}, \mathbf{x}')| \geq \epsilon) \leq 2\exp\left(-D\epsilon^2/8\right)$$

What to Submit:

- Part a: Separate proofs for subproblems (i), (ii) and (iii)
- Part b: proof
- Part c: proof, 1-2 sentence explanation about which conditions are met that allow us to apply Hoeffding's inequality e.g. "B is bounded by [...]".

Administrative

A6.

a. *[2 points]* About how many hours did you spend on this homework? There is no right or wrong answer :)

Solution:
12-16 hours