

Homework #4

CSE 446/546: Machine Learning
Profs. Kevin Jamieson and Ludwig Schmidt
Due: Wednesday 8 June, 2022 11:59pm
131 points(A)/90 points(B)

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

- 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 points for each problem improperly tagged.** For instructions, see https://www.gradescope.com/get_started#student-submission.
- If you collaborate on this homework with others, you must indicate who you worked with on your homework. Failure to do so may result in accusations of plagiarism.
- **The coding problems are available in a .zip file on the course website**, with some starter code. All coding questions in this class will have starter code. **Before attempting these problems make sure your coding environment is working.** See instructions in README file in the .zip file.
- For every problem involving code, please include the code as part of your PDF for the PDF submission *in addition to* submitting your code to the separate assignment on Gradescope created for code.

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 $X \in R^{n \times d}$ 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).
- 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 $X^\top X$.
- c. [2 points] True or False: choosing k to minimize the k -means objective (see Equation (1) below) is a good way to find meaningful clusters.
- d. [2 points] True or False: The singular value decomposition of a matrix is unique.
- e. [2 points] True or False: The rank of a square matrix equals the number of its unique nonzero eigenvalues.
- f. [2 points] True or False: Autoencoders, where the encoder and decoder functions are both neural networks with nonlinear activations, can capture more variance of the data in its encoded representation than PCA using the same number of dimensions.

What to Submit:

- a). True, because d is much smaller than n the covariance matrix has enough space for K , so PCA will minimize that reconstruction error.
- b). False, The columns of V are equal to the eigenvectors of $X^\top X$.
- c). False if $K = n$.
- d). False if negative the eigenvectors.
- e). False if the matrix has one eigenvalue.
- f). True, non-linear activation function auto encoders can capture more variance than linear model PCA.

Think before you train

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

- (1) any pre-processing steps you would take (i.e., data acquisition and 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. *[5 points]* **Scenario 1: 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.

b. *[5 points]* **Scenario 2: Social Media App Facial Recognition Technology**

- Setting: You are tasked with developing a machine learning pipeline that can quickly map someone's face for the application of filters (i.e., Snapchat, Instagram).
- Dataset: A set of face images compiled from the company's employees and their families.
- Result: The company wants an algorithm that can quickly identify the key features of a person's face to be used to apply a filter. (**Note:** Do not worry about describing the actual filter application).

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

- 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 various 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?
- d. *[5 points]* Pick one of either Scenario 1 or Scenario 2 (in parts a and b). Briefly describe (1) some potential shortcomings of your training process that may result in your algorithm having different accuracy on different populations, and (2) how you may modify your procedure to address these shortcomings.

What to Submit:

- a). First collect data carefully based on where the data came from. For example, data came from a hospital is more valuable than a YouTube survey advertise. Since we want to get which factors/features that contribute most to acquiring a specific disease, so I will use some regression algorithm maybe lasso regression to train the database.
- b). First convert each picture to pixel array data and apply multiple filters to extra useful information such as facial structure and skin color. Then use CNN to train.
- c). If we are training a large database and ignore this issue, this will make the predication on some edge cases became not accuracy.
- d).

k -means clustering

A3. Given a dataset $x_1, \dots, x_n \in \mathbb{R}^d$ and an integer $1 \leq k \leq n$, recall the following k -means objective function

$$\min_{\pi_1, \dots, \pi_k} \sum_{i=1}^k \sum_{j \in \pi_i} \|x_j - \mu_i\|_2, \quad \mu_i = \frac{1}{|\pi_i|} \sum_{j \in \pi_i} x_j. \quad (1)$$

Above, $\{\pi_i\}_{i=1}^k$ is a partition of $\{1, 2, \dots, n\}$. The objective (1) is NP-hard¹ to find a global minimizer of. Nevertheless the commonly-used algorithm we discussed in lecture (Lloyd's algorithm), typically works well in practice.

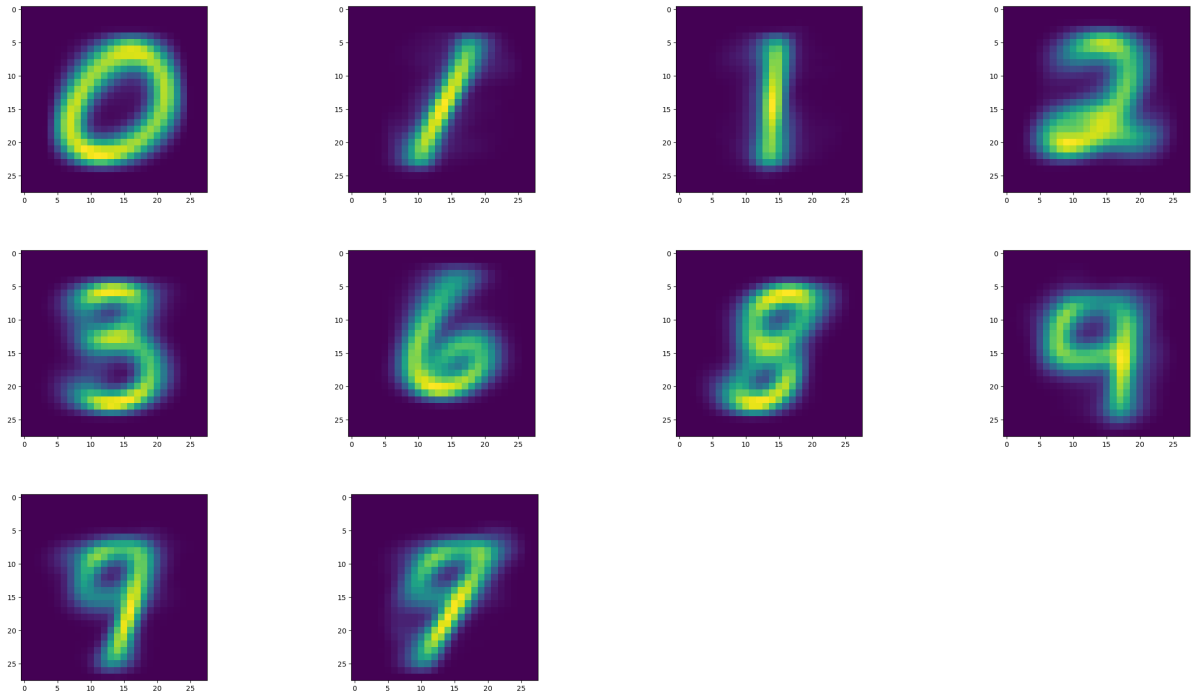
Note on Time to Run — The runtime of a good implementation for this problem should be fairly fast (a few minutes); if you find it taking upwards of one hour, please check your implementation! (Hint: **For loops are costly**. Can you vectorize it or use Numpy operations to make it faster in some ways?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

- [5 points]** Implement Lloyd's algorithm for solving the k -means objective (1). Do not use any off-the-shelf implementations, such as those found in `scikit-learn`. Include your code in your submission.
- [5 points]** Run the algorithm on the *training* dataset of MNIST with $k = 10$. Visualize (and include in your report) the cluster centers as a set of k 28×28 images.
- [5 points]** For $k = \{2, 4, 8, 16, 32, 64\}$ run the algorithm on the *training* dataset to obtain centers $\{\mu_i\}_{i=1}^k$. If $\{(x_i, y_i)\}_{i=1}^n$ and $\{(x'_i, y'_i)\}_{i=1}^m$ denote the training and test sets, respectively, plot the training error $\frac{1}{n} \sum_{i=1}^n \min_{j=1, \dots, k} \|x_i - \mu_j\|_2$ and test error $\frac{1}{m} \sum_{i=1}^m \min_{j=1, \dots, k} \|x'_i - \mu_j\|_2$ as a function of k on the same plot.

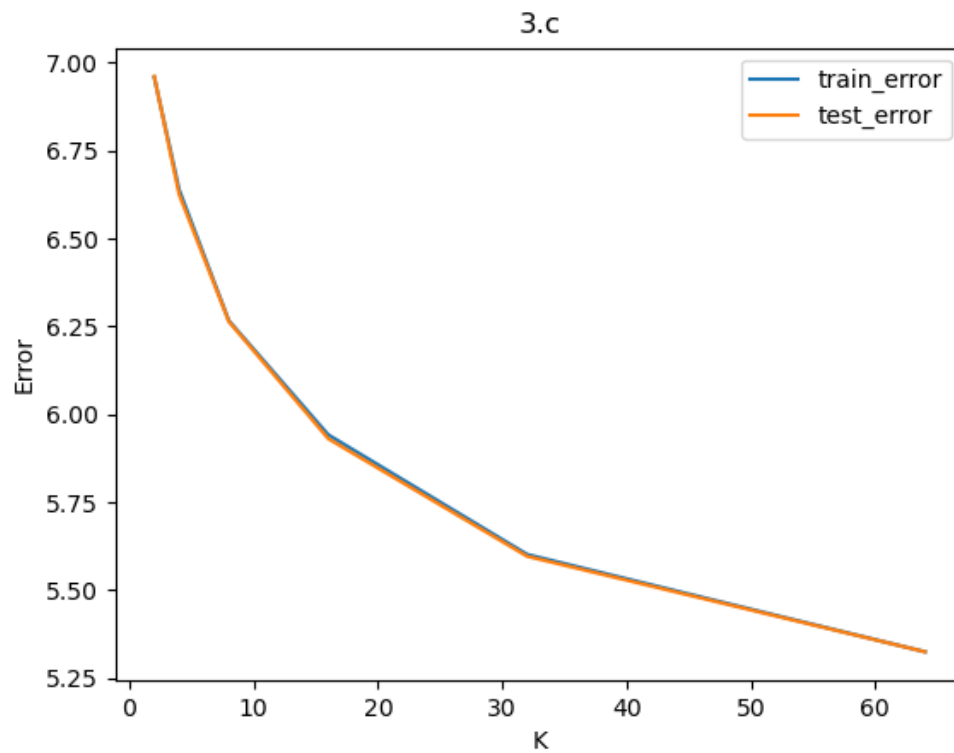
¹To be more precise, it is both NP-hard in d when $k = 2$ and k when $d = 2$. See the references on the wikipedia page for k -means for more details.

What to Submit:

- For part (b): 10 images of cluster centers.



- For part (c): Plot of training and test error as function of k .



Listing 1: A3 code

```

from calendar import c
from typing import List, Tuple

import numpy as np

from utils import problem

@problem.tag("hw4-A")
def calculate_centers(
    data: np.ndarray, classifications: np.ndarray, num_centers: int
) -> np.ndarray:
    """
    Sub-routine of Lloyd's algorithm that calculates the centers given datapoints and their
    num_centers is additionally provided for speed-up purposes.

    Args:
        data (np.ndarray): Array of shape (n, d). Training data set.
        classifications (np.ndarray): Array of shape (n,) full of integers in range {0, 1, ...,
num_centers - 1}.
        Data point at index i is assigned to classifications[i].
        num_centers (int): Number of centers for reference.
        Might be usefull for pre-allocating numpy array (Faster than appending to list).

    Returns:
        np.ndarray: Array of shape (num_centers, d) containing new centers.
    """
    #raise NotImplementedError("Your Code Goes Here")
    d = data.shape[1]
    new_centers = np.zeros((num_centers, d))
    for i in range(num_centers):
        indices = np.where(classifications == i)
        center = np.mean(data[indices], 0)
        new_centers[i] = center
    return new_centers

@problem.tag("hw4-A")
def cluster_data(data: np.ndarray, centers: np.ndarray) -> np.ndarray:
    """
    Sub-routine of Lloyd's algorithm that clusters datapoints to centers given datapoints and
    centers.

    Args:
        data (np.ndarray): Array of shape (n, d). Training data set.
        centers (np.ndarray): Array of shape (k, d). Each row is a center to which a datapoint
    is assigned.

    Returns:
        np.ndarray: Array of integers of shape (n,), with each entry being in range {0, 1, ..., k-1}.
        Entry j at index i should mean that j-th center is the closest to data[i] datapoint.
    """
    #raise NotImplementedError("Your Code Goes Here")
    distances = [np.linalg.norm(data - center, axis=1) for center in centers]
    closest_clusters = np.argmin(np.array(distances), axis=0)
    return closest_clusters

```

```

@problem.tag("hw4-A")
def calculate_error(data: np.ndarray, centers: np.ndarray) -> float:
    """Calculates error/objective function on a provided dataset, with trained centers.

    Args:
        data (np.ndarray): Array of shape (n, d). Dataset to evaluate centers on.
        centers (np.ndarray): Array of shape (k, d). Each row is a center to which a datapoint
            These should be trained on training dataset.

    Returns:
        float: Single value representing mean objective function of centers on a provided dataset.
    """
    #raise NotImplementedError("Your Code Goes Here")
    n = data.shape[0]
    classification = cluster_data(data, centers)
    error = 0

    for i, c in zip(data, classification):
        error += np.linalg.norm(i - centers[c])

    return error / n

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

    Args:
        data (np.ndarray): Array of shape (n, d). Training data set.
        num_centers (int): Number of centers to train/cluster around.
        epsilon (float, optional): Epsilon for stopping condition.
            Training should stop when max(abs(centers - previous_centers)) is smaller or equal to epsilon.
            Defaults to 10e-3.

    Returns:
        np.ndarray: Tuple of 2 numpy arrays:
            Element at index 0: Array of shape (num_centers, d) containing trained centers.
            Element at index 1: List of floats of length # of iterations
                containing errors at the end of each iteration of lloyd's algorithm.

    Note:
        - For initializing centers please use the first 'num_centers' data points.
    """
    #raise NotImplementedError("Your Code Goes Here")
    d = data.shape[1]
    prev_centers = np.ones((num_centers, d))
    centers = data[:num_centers]

    while np.max(np.abs(centers - prev_centers)) > epsilon:
        prev_centers = centers
        c = cluster_data(data, centers)
        centers = calculate_centers(data, c, num_centers)

```



```

    return centers

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

import matplotlib.pyplot as plt
import numpy as np

from utils import load_dataset, problem

@problem.tag("hw4-A", start_line=1)
def main():
    """Main function of k-means problem

    You should:
        a. Run Lloyd's Algorithm for k=10, and report 10 centers returned.
        b. For ks: 2, 4, 8, 16, 32, 64 run Lloyd's Algorithm,
           and report objective function value on both training set and test set.
           (All one plot, 2 lines)

    NOTE: This code takes a while to run. For debugging purposes you might want to change:
        x_train to x_train[:10000]. CHANGE IT BACK before submission.
    """
    (x_train, _), (x_test, _) = load_dataset("mnist")
    #raise NotImplementedError("Your Code Goes Here")

    """

    # Part b
    centers = lloyd_algorithm(x_train, 10)
    for i in range(10):
        plt.imshow(centers[i].reshape(28,28))
        plt.show()
    """

    # Part c
    ks = [2, 4, 8, 16, 32, 64]
    train_error = np.zeros(6)
    test_error = np.zeros(6)
    i = 0
    for k in ks:
        print("k={}, k".format(k), k)
        centers = lloyd_algorithm(x_train, k)
        train_error[i] += calculate_error(x_train, centers)
        test_error[i] += calculate_error(x_test, centers)
        i += 1

    plt.figure(1)
    plt.plot(ks, train_error)
    plt.plot(ks, test_error)

```

```
plt.xlabel('K')
plt.ylabel('Error')
plt.title('3.c')
plt.legend(["train_error", "test_error"])
plt.show()

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

PCA

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

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

- [2 points]** If λ_i denotes the i th largest eigenvalue of Σ , what are the eigenvalues $\lambda_1, \lambda_2, \lambda_{10}, \lambda_{30}$, and λ_{50} ? What is the sum of eigenvalues $\sum_{i=1}^d \lambda_i$?
- [5 points]** Let $x \in \mathbb{R}^d$ and $k \in 1, 2, \dots, d$. Write a formula for the rank- k PCA approximation of x .
- [5 points]** Using this approximation, plot the reconstruction error from $k = 1$ to 100 (the X -axis is k and the Y -axis is the mean-squared error reconstruction error) on the training set and the test set (using the μ and the basis learned from the training set). On a separate plot, plot $1 - \frac{\sum_{i=1}^k \lambda_i}{\sum_{i=1}^d \lambda_i}$ from $k = 1$ to 100.
- [3 points]** Now let us get a sense of what the top PCA directions are capturing. Display the first 10 eigenvectors as images, and provide a brief interpretation of what you think they capture.
- [3 points]** Finally, visualize a set of reconstructed digits from the training set for different values of k . In particular provide the reconstructions for digits 2, 6, 7 with values $k = 5, 15, 40, 100$ (just choose an image from each digit arbitrarily). Show the original image side-by-side with its reconstruction. Provide a brief interpretation, in terms of your perceptions of the quality of these reconstructions and the dimensionality you used.

What to Submit:

- For part (a):** The 0 th largest eigenvalue is: 5.116787728342063
The 1 th largest eigenvalue is: 3.7413284788648
The 9 th largest eigenvalue is: 1.2427293764173146
The 29 th largest eigenvalue is: 0.36425572027888997
The 49 th largest eigenvalue is: 0.16970842700671648
The sum of eigenvalues is: 52.725035495127734
- For part (b):** $x = (x - \mu^T)V_k V_k^T + \mu$
- For part (c):** Plot containing reconstruction error on train and test sets. Plot of $1 - \frac{\sum_{i=1}^k \lambda_i}{\sum_{i=1}^d \lambda_i}$
- For part (d):** 10 eigenvectors as images.
- For part (e):** 15 total images, including 3 original and 12 reconstructed ones. Each reconstructed image corresponds to a certain digit (2, 6 or 7) and k value (5, 15, 40 or 100).
- Code for parts c-e

Unsupervised Learning with Autoencoders

A5. In this exercise, we will train two simple autoencoders to perform dimensionality reduction on MNIST. As discussed in lecture, autoencoders are a long-studied neural network architecture comprised of an encoder component to summarize the latent features of input data and a decoder component to try and reconstruct the original data from the latent features.

Weight Initialization and PyTorch

Last assignment, we had you refrain from using `torch.nn` modules. For this assignment, we will use these modules, and recommend using `nn.Linear` for your linear layers. You will not need to initialize the weights yourself; the default initialization in PyTorch will be sufficient for this problem. *Hint: we also recommend using the `nn.Sequential` module to organize your network class and simplify the process of writing the forward pass. However, you may choose to organize your code however you'd like.*

Training

Use `optim.Adam` for this question. Feel free to experiment with different learning rates, though you can use $5 \cdot 10^{-5}$ as mentioned in the code. Use mean squared error (`nn.MSELoss()` or `F.mse_loss()`) for the loss function.

- a. *[10 points]* Use a network with a single linear layer. Let $W_e \in \mathbb{R}^{h \times d}$ and $W_d \in \mathbb{R}^{d \times h}$. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathcal{F}_1(x) = W_d W_e x.$$

Run experiments for $h \in \{32, 64, 128\}$. For each of the different h values, report your final training error and visualize a set of 10 reconstructed digits, side-by-side with the original image. *Note:* we omit the bias term in the formulation for notational convenience since `nn.Linear` learns bias parameters alongside weight parameters by default.

- b. *[10 points]* Use a single-layer network with non-linearity. Let $W_e \in \mathbb{R}^{h \times d}$, $W_d \in \mathbb{R}^{d \times h}$, and activation $\sigma : \mathbb{R} \mapsto \mathbb{R}$, where σ is the ReLU function. Given some $x \in \mathbb{R}^d$, the forward pass is formulated as

$$\mathcal{F}_2(x) = \sigma(W_d \sigma(W_e x)).$$

Report the same findings as asked for in part a (for $h \in \{32, 64, 128\}$).

- c. *[5 points]* Now, evaluate $\mathcal{F}_1(x)$ and $\mathcal{F}_2(x)$ (use $h = 128$ here) on the test set. Provide the test reconstruction errors in a table.
- d. *[5 points]* In a few sentences, compare the quality of the reconstructions from these two autoencoders with those of PCA from problem A5. You may need to re-run your code for PCA using the ranks $k \in \{32, 64, 128\}$ to match the h values used above.

What to Submit:

- **For parts (a, b):** Final training error and set of 10 reconstructed images of digits, side-by-side with the original image (10 images for each h).
- **For part (c):** Errors of networks from part a and b on testing set.
- **For part (d):** 2-3 sentences on differences in quality of solutions between PCA and Autoencoders, with example images
- Code for parts a-c

Image Classification on CIFAR-10

A6. 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 tensors, two-dimensional convolutions (`nn.Conv2d`) and fully-connected layers (`nn.Linear`), ReLU non-linearities (`F.relu`), pooling (`nn.MaxPool2d`), and tensor reshaping (`view`).

A few preliminaries:

- Each network f maps an image $x^{\text{in}} \in \mathbb{R}^{32 \times 32 \times 3}$ (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 would usually 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 5 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, included with the code files for this assignment. 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 (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

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

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

- 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 $\text{Conv2d}(x^{\text{in}}, W_1) \in \mathbb{R}^{(33-k) \times (33-k) \times M}$.

- 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

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

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

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

The 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.

What to Submit:

- Parts a-b: Plot of training and validation accuracy for each TOP 5 hyperparameter configurations evaluated. (10 lines total). If it took less than 5 hyperparameter configurations to pass performance threshold plot all hyperparameter configurations evaluated. List of the hyperparameter values you searched over, and your search method (random, grid, etc.).
- Parts a-b: Values of best performing hyperparameters, and accuracy of best models on test data.
- Code

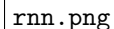
Text classification on SST-2

B1. The Stanford Sentiment Treebank (SST-2) is a dataset of movie reviews. Each review is annotated with a label indicating whether the sentiment of the review is positive or negative. Below are some examples from the dataset. Note that often times the reviews are only partial sentences or even single words.

Sequence	Sentiment
is one big , dumb action movie .	Negative
perfectly executed and wonderfully sympathetic characters ,	Positive
until then there 's always these rehashes to feed to the younger generations	Negative
is like nothing we westerners have seen before .	Positive

In this problem you will use a Recurrent Neural Network (RNN) for classifying reviews as either Positive (1) or Negative (0).

Using an RNN for binary classification



rnn.png

Above is a simplified visualization of the RNN you will be building. Each token of the input sequence (*CSE*, *446*, ...) is fed into the network sequentially. Note that in reality, we convert each token to some integer index. But training with discrete values does not work well, so we also "embed" the words in a high-dimensional continuous space. We already provided an `nn.Embedding` layer to you to do this. Each RNN cell (squares above) generates a hidden state h_i . We then feed the last hidden state into a simple fully-connected layer, which then produces a single prediction between 0 and 1.

YOu may wish to refer to the famous Understanding LSTMs blog post to dive deeper into the logic and math of LSTMs.

Setup

- a. Update environment by running:

```
conda env update -f environment.yaml
conda activate cse446
pip install -e .
```

You only need to modify `problems.py`, however you are free to also modify the other two files. You only need to submit `problems.py`, but if you make changes to the other files you should also include them in your submission.

Problems

- a. *[10 points]* In Natural Language Processing (NLP), we usually have to deal with variable length data. In SST-2, each data point corresponds to a review, and reviews often have different lengths. The issue is that to efficiently train a model with textual data, we need to create batches of data that are fixed size matrices. In order to store a batch of sequences in a single matrix, we add padding to shorter sequences so that each sequence has the same length. Given a list of N sequences, we:

- Find the length of the longest sequence in the batch, call it `max_sequence_length`
- Append padding tokens to the end of each sequence so that each sequence has length `max_sequence_length`

(c) Stack the sequences into a matrix of size $(N, \text{max_sequence_length})$

In this process, words are mapped to integer ids, so in the above process we actually store integers rather than strings. For the padding token, we simply use id 0. In the file `problems.py`, fill out `collate_fn` to perform the above batching process. Details are provided in the comment of the function.

- b. *[15 points]* Implement the constructor and forward method for `RNNBinaryClassificationModel`. You will use three different types of recurrent neural networks: the vanilla RNN (`nn.RNN`), Long Short-Term Memory (`nn.LSTM`) and Gated Recurrent Units (`nn.GRU`). For the hidden size, use 64 (Usually this is a hyperparameter you need to tune). We have already provided you with the embedding layer to turn token indices into continuous vectors of size 50, but you will need a linear layer to transform the last hidden state of the recurrent layer to a shape that can be interpreted as a label prediction.
- c. *[5 points]* Implement the `loss` method of `RNNBinaryClassificationModel`, which should compute the binary cross-entropy loss between the predictions and the target labels. Also implemented the `accuracy` method, which given the predictions and the target labels should return the accuracy.
- d. *[15 points]* We have already provided all of the data loading, training and validation code for you. Choose appropriate parameters in `get_parameters` function, by setting the constants: `TRAINING_BATCH_SIZE`, `NUM_EPOCHS`, `LEARNING_RATE`. With a good learning rate, you shouldn't have to train for more than 16 epochs. Report your best validation loss and corresponding validation accuracy, corresponding training loss and training accuracy for each of the three types of recurrent neural networks.
- e. *[5 points]* Currently we are only using the final hidden state of the RNN to classify the entire sequence as being either positive or negative sentiment. But can we make use of the other hidden states? Suppose you wanted to use the same architecture for a related task called tagging. For tagging, the goal is to predict a tag for each token of the sequence. For instance, we might want predict the part of speech tag (verb, noun, adjective etc.) of each token. In a few sentences, describe how you would modify the current architecture to predict a tag for each token.

What to Submit:

- Parts a-c: Code in **PDF** (in addition to code submission).
- Part d: Loss and accuracy on both validation and training sets for each of 3 three different types of models. Also what parameters were used to achieve these values.
- Part e: Few sentences on modification of architecture.
- Code for parts a-d

Random Fourier Features

B2. Kernel methods such as Support Vector Machines 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 $\mathbf{w} = [\mathbf{w}_1, \dots, \mathbf{w}_N]$, 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} :

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

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 $\langle \mathbf{z}(\mathbf{x}), \mathbf{z}(\mathbf{x}') \rangle$.

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^N \omega_i k(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^N \omega_i \langle \psi(x), \psi(x') \rangle \approx \sum_{i=1}^N \omega_i \langle \mathbf{z}(x_i), \mathbf{z}(x') \rangle = \left(\sum_{i=1}^N \omega_i \mathbf{z}(x_i)^T \right) \mathbf{z}(x') = \beta^T \mathbf{z}(x)$$

Assuming $\mathbf{z}(\mathbf{x}) = \sigma(M\mathbf{x} + b)$ for some nonlinear function σ , this “approximate” SVM *can potentially be evaluated much quicker* than the kernel SVM. To see why, note that the left-hand-side requires evaluating $k(\mathbf{x}_i, \mathbf{x})$ for all $i \in \{1, \dots, N\}$, in general, if ω_i is not sparse. On the other hand, the right-hand-side just requires computing $\mathbf{z}(x) = \sigma(M\mathbf{x} + b)$ 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 SVM 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

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

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

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

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

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

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

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 $z_{\mathbf{w}} : \mathbb{R}^d \rightarrow \mathbb{R}$ as

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

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.

$$\mathbb{E}_{\mathbf{w}, b} [z_{\mathbf{w}}(\mathbf{x}) z_{\mathbf{w}}(\mathbf{x}')] = k_p(\mathbf{x}, \mathbf{x}')$$

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 $\mathbf{z} : \mathbb{R}^d \rightarrow \mathbb{R}^D$. Let \mathbf{z} be the D -dimensional concatenation of $z_{\mathbf{w}}(\mathbf{x})$ s:

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

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.

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

- 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.

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

- c. **[5 points] Concentration Bounds** In part (b) 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 why we can apply Hoeffding's inequality to obtain

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

- d. **[15 points] Empirical evidence:** In this sub-problem, we'll be comparing a support vector machine classifier (**SVM**) to a classifier with random Fourier features (**RFF**). Let's get our hands dirty with some code!

- i Implement both a **SVM** and a **RFF** classifiers on the a subset of 10000 samples from the MNIST dataset. Use the RBF kernel with γ for both models.

Although you may use any library of your choice for this problem, we recommend looking at **sklearn's** **SVC** and **RBFSampler** classes. Pay attention to the **kernel** and **gamma** parameters of both classes.

- ii Plot the test accuracy of the **RFF** for $D \in \{100, 500, 1000, 5000, 10000, 15000, 20000\}$. Draw a horizontal line for the test accuracy of the standard **SVM**. What do you observe? What value of d would you pick for **RFF**? For your choice of d , what's the gap between the testing accuracy of both models? (Note: read part iii before completing this; you will need to record the runtime of these training runs.)
- iii Repeat the same experiment as in part ii, but instead of recording the test accuracy of both models, record the runtime (in milliseconds) of training them. What do you observe now as D increases? Would you still pick the same value of D ? What is the trade-off between the runtime and correctness of the **RFF**?

To record the time elapsed between two points in your code, you can use python's time library. Only record the runtime required to learn the corresponding model on the training data. Learning an RFF with Sklearn involves a 2-stage approach: fitting the RBFSampler and fitting a linear model the the resulting RFF features in \mathbb{R}^D . The runtime should include both steps.

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 [...]".
- Part d:
 - code snippet for all experiments (in Latex, there is no autograder submission for this problem).
 - Plot for subproblem (ii) of test accuracy of SVM and RFF as a function of the number of fourier features D . 2-3 sentences describing observed behavior, the gap between models, and justifying choice of d
 - Plot for subproblem (iii) of model training runtime of SVM and RFF as a function of the number of fourier features D . 2-3 sentences describing observed behavior, justifying new choice of d , and describing the runtime-correctness tradeoff.