CS M146 Fall 2023 Homework 4 Solutions

UCLA ID: 605900342
Name: Caleb Traxler
Collaborators: N/A

By turning in this assignment, I agree by the UCLA honor code and declare that all of this is my own work.

Problem 1 (PCA [10 PTS])

You have the following data:

data #	1	2	3	4	5	6	7	8
x_1	5.51	20.82	-0.77	19.30	14.24	9.74	11.59	-6.08
x_2	5.35	24.03	-0.57	19.38	12.77	9.68	12.06	-5.22

You want to reduce the data into a single-dimension representation. You are given the first principal component $\mathbf{v}_1 = (0.694, 0.720)$.

(a) 4 What is the representation (projected coordinate) along the first principal direction for data $\mathbf{x}^{(1)}$?

From the lecture notes, I see that for $U = [v_1]$ as the first principal component, such that $v_1 = (0.694, 0.720)$, I know that for any point x, its reduced representation is $y = Wx = U^T x$.

$U^T = \begin{bmatrix}0.694 & 0.720\end{bmatrix}$, $x^{(1)} = \begin{bmatrix}5.51\\5.35\end{bmatrix}$, thus $y = \begin{bmatrix}0.694 & 0.720\end{bmatrix}\begin{bmatrix}5.51\\5.35\end{bmatrix} = 7.676$.

Thus, the representation (projected coordinate) along the first principal direction for data $x^{(1)}$ is approximately 7.676.

(b) 3 What are the feature values in the original space reconstructed using this first principal representation $y_1^{(1)}$?

From part (a) I found that $y = 7.676$ and the original data point at index 1 is $x^{(1)} = \begin{bmatrix}5.51\\5.35\end{bmatrix}$

Thus, I can compute the feature values in the original space reconstructed using the first representation as: $y_1^{(1)} = yv_1 = 7.676\begin{bmatrix}0.694\\0.720\end{bmatrix} = \begin{bmatrix}5.327\\5.527\end{bmatrix}$

(c) 3 What is the principal component score for $\mathbf{x}^{(1)}$ along the second principal direction? You can answer up to a change of sign.

To calculate the second principal direction, I need to recall Lemma 2 from the lecture notes:

Lemma 2: The solution to the PCA objective is given by the first m eigenvectors of the empirical covariance matrix. Let mean vector $\mu = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}$. The empirical covariance matrix $A = \frac{1}{(n-1)}\sum_{i=1}^{n}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$.

Let $v_i \in R^d$ denote the eigenvector for the ith largest eigenvalue of A. Define $V = [v_1, v_2, ..., v_d]$ as matrix of eigenvectors.

$\mu_1 = \frac{1}{8}(5.51 + 20.82 + (-0.77) + 19.30 + 14.25 + 9.74 + 11.59 + (-6.08)) = 9.295$

$\mu_2 = \frac{1}{8}(5.35 + 24.03 + (-0.57) + 19.38 + 12.77 + 9.68 + 12.06 + (-5.22)) = 9.685$

Thus, $\mu = \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix}$

Furthermore, I will compute the covariance matrix:

$A = \frac{1}{7}\sum_{i=1}^{8}(x^{(i)} - \mu)(x^{(i)} - \mu)^T$.

$\sum_{i=1}^{8}(x^{(i)} - \mu)(x^{(i)} - \mu)^T :=$

$i = 1 \implies \begin{bmatrix} 5.51 \\ 5.35 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 5.51 \\ 5.35 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} -3.785 \\ -4.335 \end{bmatrix})(\begin{bmatrix} -3.785 & -4.335 \end{bmatrix}) = \begin{bmatrix} 14.326225 & 16.407975 \\ 16.407975 & 18.792225 \end{bmatrix}$

$i = 2 \implies (\begin{bmatrix} 20.82 \\ 24.03 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 20.82 \\ 24.03 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} 11.525 \\ 14.345 \end{bmatrix})(\begin{bmatrix} 11.525 & 14.345 \end{bmatrix}) = \begin{bmatrix} 132.825625 & 165.326125 \\ 165.326125 & 205.779025 \end{bmatrix}$

$i = 3 \implies (\begin{bmatrix} -0.77 \\ -0.57 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} -0.77 \\ -0.57 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} -10.065 \\ -10.255 \end{bmatrix})(\begin{bmatrix} -10.065 & -10.255 \end{bmatrix}) = \begin{bmatrix} 101.304225 & 103.216575 \\ 103.216575 & 105.165025 \end{bmatrix}$

$i = 4 \implies (\begin{bmatrix} 19.30 \\ 19.38 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 19.30 \\ 19.38 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} 10.005 \\ 9.695 \end{bmatrix})(\begin{bmatrix} 10.005 & 9.695 \end{bmatrix}) = \begin{bmatrix} 100.100025 & 96.998475 \\ 96.998475 & 93.993025 \end{bmatrix}$

$i = 5 \implies (\begin{bmatrix} 14.24 \\ 12.77 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 14.24 \\ 12.77 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} 4.945 \\ 3.085 \end{bmatrix})(\begin{bmatrix} 4.945 & 3.085 \end{bmatrix}) = \begin{bmatrix} 24.453025 & 15.255325 \\ 15.255325 & 9.517225 \end{bmatrix}$

$i = 6 \implies (\begin{bmatrix} 9.74 \\ 9.68 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 9.74 \\ 9.68 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} 0.445 \\ -0.005 \end{bmatrix})(\begin{bmatrix} 0.445 & -0.005 \end{bmatrix}) = \begin{bmatrix} 0.198025 & -0.002225 \\ -0.002225 & 0.000025 \end{bmatrix}$

$i = 7 \implies (\begin{bmatrix} 11.59 \\ 12.06 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} 11.59 \\ 12.06 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} 2.295 \\ 2.375 \end{bmatrix})(\begin{bmatrix} 2.295 & 2.375 \end{bmatrix}) = \begin{bmatrix} 5.267025 & 5.450625 \\ 5.450625 & 5.640625 \end{bmatrix}$

$i = 8 \implies (\begin{bmatrix} -6.08 \\ -5.22 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})(\begin{bmatrix} -6.08 \\ -5.22 \end{bmatrix} - \begin{bmatrix} 9.295 \\ 9.685 \end{bmatrix})^T = (\begin{bmatrix} -15.375 \\ -14.905 \end{bmatrix})(\begin{bmatrix} -15.375 & -14.905 \end{bmatrix}) = \begin{bmatrix} 236.390625 & 229.164375 \\ 229.164375 & 222.159025 \end{bmatrix}$

$$\begin{bmatrix} 14.326225 & 16.407975 \\ 16.407975 & 18.792225 \end{bmatrix} + \begin{bmatrix} 132.825625 & 165.326125 \\ 165.326125 & 205.779025 \end{bmatrix} + \begin{bmatrix} 101.304225 & 103.216575 \\ 103.216575 & 105.165025 \end{bmatrix} + \begin{bmatrix} 100.100025 & 96.998475 \\ 96.998475 & 93.993025 \end{bmatrix}$$
$$+ \begin{bmatrix} 24.453025 & 15.255325 \\ 15.255325 & 9.517225 \end{bmatrix} + \begin{bmatrix} 0.198025 & -0.002225 \\ -0.002225 & 0.000025 \end{bmatrix} + \begin{bmatrix} 5.267025 & 5.450625 \\ 5.450625 & 5.640625 \end{bmatrix} + \begin{bmatrix} 236.390625 & 229.164375 \\ 229.164375 & 222.159025 \end{bmatrix}$$
$$= \begin{bmatrix} 614.864710 & 631.81725 \\ 631.81725 & 661.0462 \end{bmatrix}$$

Thus, $\frac{1}{7}\begin{bmatrix} 614.864710 & 631.81725 \\ 631.81725 & 661.0462 \end{bmatrix} = \begin{bmatrix} 87.83781571 & 90.25960714 \\ 90.25960714 & 94.43517143 \end{bmatrix}$

Therefore, I can conclude that the covariance matrix:

$A = \frac{1}{7}\sum_{i=1}^{8}(x^{(i)} - \mu)(x^{(i)} - \mu)^T = \begin{bmatrix} 87.83781571 & 90.25960714 \\ 90.25960714 & 94.43517143 \end{bmatrix}$.

Next, I can compute the eigenvalues of matrix A:

$\det(\begin{bmatrix} 87.83781571 - \lambda & 90.25960714 \\ 90.25960714 & 94.43517143 - \lambda \end{bmatrix}) = 0$

$\implies (87.83781571 - \lambda)(94.43517143 - \lambda) - (90.25960714)(90.25960714) = 0$

$\implies 8294.979185 - 87.83781571\lambda - 94.43517143\lambda + \lambda^2 - 8146.796681 = 0$

$\implies 148.182504 - 182.2729871\lambda + \lambda^2 = 0$

$\implies \lambda = 181.456$ and $\lambda = 0.816629$.

The larger eigenvalue, $\lambda = 181.456$, with corresponding eigenvector, $v = \begin{bmatrix} -0.694 \\ -0.720 \end{bmatrix}$.

Furthermore, note that $\lambda = 0.816629$ has a corresponding eigenvector of $v = \begin{bmatrix} -0.71990351 \\ 0.69407416 \end{bmatrix}$, which represents the second principal direction.

Thus, I can further compute the principal component score for $x^{(1)}$ along the second principal direction, which is $v = \begin{bmatrix} -0.71990351 \\ 0.69407416 \end{bmatrix}$

Similar to part(a), I see that $U = [v]$ as the second principal direction $x^{(1)}$. $y = Wx = U^T x$, where $U^T = [-0.71990351, 0.69407416]$, $x^{(1)} = \begin{bmatrix} 5.51 \\ 5.35 \end{bmatrix}$, thus, $y = [-0.71990351, 0.69407416] \begin{bmatrix} 5.51 \\ 5.35 \end{bmatrix}$ = -0.253372.

However, recall that the question mentioned that I can answer up to a change of sign change, thus = 0.253372.

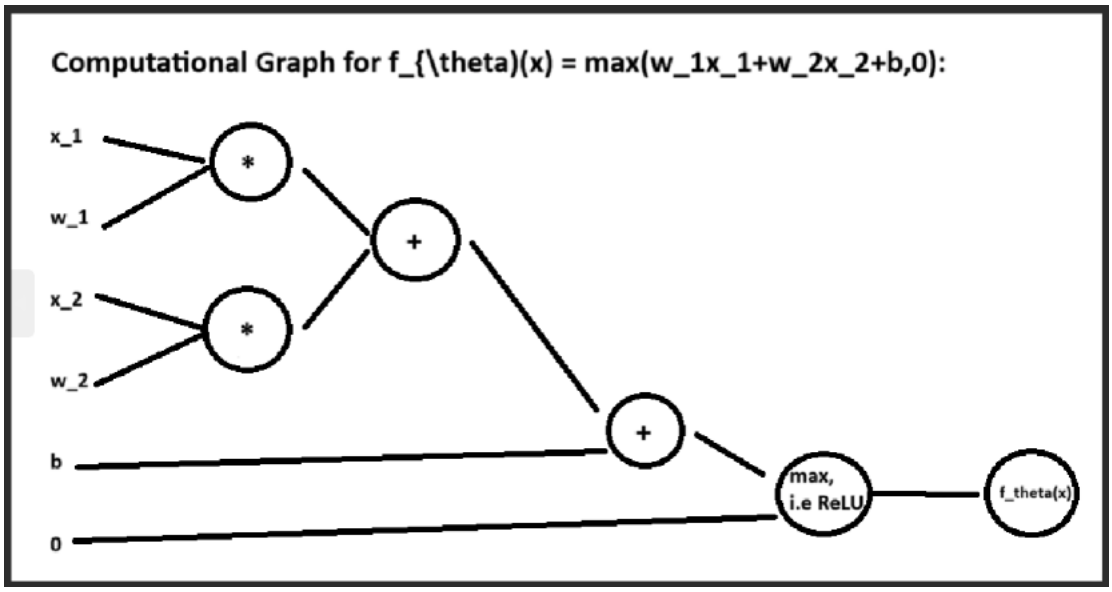
Conclusion:

I see that the score along the first principal direction is a strong indicator of the data points position in the reduced dimension space, which reflects the major trend or pattern in the data. The score along the second direction provides additional detail, capturing variance not accounted for by the first principal component.

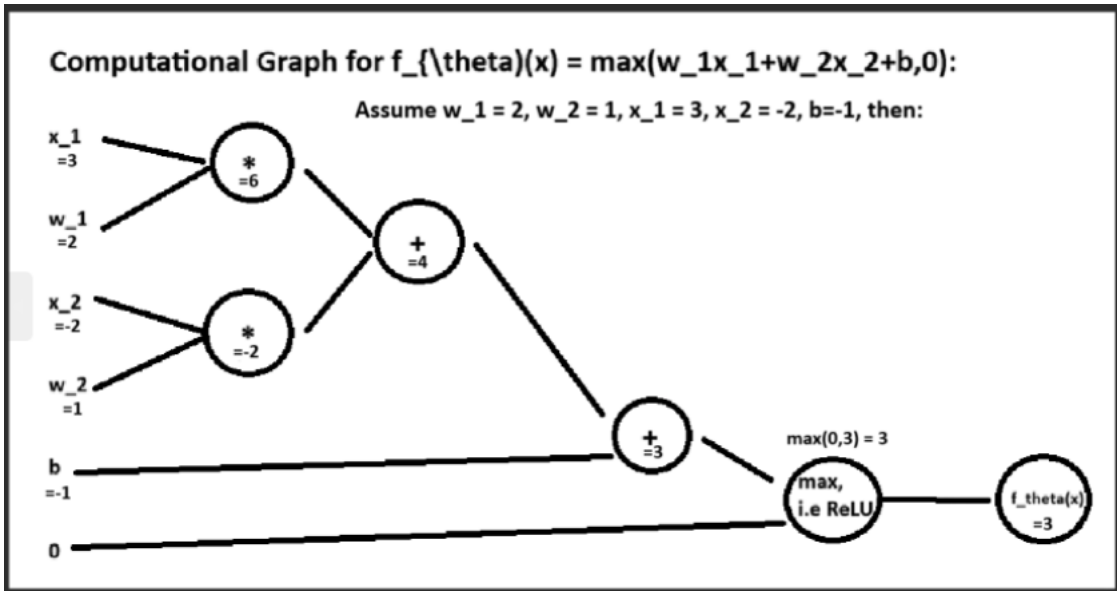
Problem 2 (Neural Networks [10 PTS])

Suppose you have the following neural network described by the function $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \max(w_1 x_1 + w_2 x_2 + b, 0)$, where $\boldsymbol{x} = (x_1, x_2)^T \in \mathbb{R}^2$ is the input to the network and $\boldsymbol{\theta} = (w_1, w_2, b)^T \in \mathbb{R}^3$ are the network's parameters.

(a) 3 Draw the computation graph for $f_{\boldsymbol{\theta}}(\boldsymbol{x})$.



(b) 3 Let $w_1 = 2, w_2 = 1, x_1 = 3, x_2 = -2, b = -1$. Compute the forward pass for $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ (i.e. compute $f_{\boldsymbol{\theta}}(\boldsymbol{x})$).



$(x_1 = 3)(w_1 = 2) = 6$

$(x_2 = -2)(w_2 = 1) = -2$

$((x_1 = 3)(w_1 = 2)) + ((x_2 = -2)(w_2 = 1)) = (6) + (-2) = 4$

$((x_1 = 3)(w_1 = 2)) + ((x_2 = -2)(w_2 = 1)) + b = (4) + (-1) = 3$
Thus, $f_\theta = 3$.

(c) 6 Let $w_1 = 2, w_2 = 1, x_1 = 3, x_2 = -2, b = -1$. Compute the backward pass for $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ with respect to w_1, w_2, b (i.e. compute $\frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x})}{\partial w_1}, \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x})}{\partial w_2}, \frac{\partial f_{\boldsymbol{\theta}}(\boldsymbol{x})}{\partial b}$). Recall $f_\theta(x) = \max(w_1 x_1 + w_2 x_2 + b, 0)$.
General derivative for the ReLU function:
Assume $ReLU(x) = max(0, x)$, then the derivative is $f(x) = \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x > 0 \end{cases}$

From the chain rule I see that $\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial L}{\partial z}x_1 \implies$
$\frac{\partial f_\theta(x)}{\partial w_1} = \frac{\partial}{\partial w_1}(max(w_1 x_1 + w_2 x_2 + b, 0))|_{w_1=2, x_1=3} = x_1 = 3$

From the chain rule I see that $\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial w_2} = \frac{\partial L}{\partial z}x_2 \implies$
$\frac{\partial f_\theta(x)}{\partial w_2} = \frac{\partial}{\partial w_2}(max(w_1 x_1 + w_2 x_2 + b, 0))|_{w_2=1, x_2=-2} = x_2 = -2$

From the chain rule I see that $\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial L}{\partial z}1 \implies$
$\frac{\partial f_\theta(x)}{\partial b} = \frac{\partial}{\partial b}(max(w_1 x_1 + w_2 x_2 + b, 0))|_{b=-1} = 1$

Problem 3 (A Two-Layer Neural Network for Binary Classification [12 PTS])

In this exercise, we will walk through the forward and backward propagation process for a two-layer fully connected neural network. We will use the same data as in Homework 1, the MNIST dataset. The data you will be using is under the `MNIST` folder.

(a) 0 Load and visualize the data by running the cells for Part (a) in the notebook.

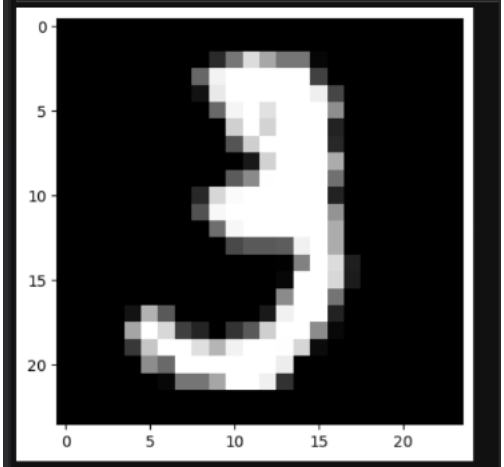
Code:

```
import pandas as pd
import numpy as np
import os
import gzip
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from tqdm import tqdm

# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
## Load fashionMNIST. This is the same code with homework 1.
##
def crop_center(img,cropped):
    img = img.reshape(-1, 28, 28)
    start = 28//2-(cropped//2)
    img = img[:, start:start+cropped, start:start+cropped]
    return img.reshape(-1, cropped*cropped)

def load_mnist(path, kind='train'):

    """Load MNIST data from `path`"""
    labels_path = os.path.join(path, '%s-labels-idx1-ubyte.gz' % kind)
    images_path = os.path.join(path, '%s-images-idx3-ubyte.gz' % kind)

    with gzip.open(labels_path, 'rb') as lbpath:
        labels = np.frombuffer(lbpath.read(), 'B', offset=8)

    with gzip.open(images_path, 'rb') as imgpath:
        images = np.frombuffer(imgpath.read(),'B', offset=16).reshape(-1, 784)
        images = crop_center(images, 24)
    return images, labels
X_train_and_val, y_train_and_val = load_mnist('./data/mnist', kind='train')
X_test, y_test = load_mnist('./data/mnist', kind='t10k')
X_train, X_val = X_train_and_val[:50000], X_train_and_val[50000:]
y_train, y_val = y_train_and_val[:50000], y_train_and_val[50000:]
print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Val data shape: ', X_val.shape)
print('Val target shape: ', y_val.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)
```

```
Train data shape:  (50000, 576)
Train target shape:  (50000,)
Val data shape:  (10000, 576)
Val target shape:  (10000,)
Test data shape:  (10000, 576)
Test target shape:  (10000,)
```

```
# PART (a):
# To Visualize a point in the dataset
index = 10
X = np.array(X_train[index], dtype='uint8').reshape([24, 24])
fig = plt.figure()
plt.imshow(X, cmap='gray')
plt.show()
if y_train[index] in set([1, 3, 5, 7, 9]):
    label = 'Odd'
else:
    label = 'Even'
print('Label is', label)
```




```
Label is Odd
```

(b) 2 Implement the forward pass of the two-layer feed forward neural network with a ReLU non-linear layer. Specifically, implement code of "part (b)" to compute `scores` in the `loss()` function, take a screenshot of your code and paste it here. Further information and guidance are provided in the comments of the code.
(*Hint*: Our network has four set of parameters to be updated: First layer weights, first layer biases, second layer weights, and second layer biases. The weights and bias parameters are initialized in the `__init__()` function of the `TwoLayerNet` class.)

I will display both the __init__ and loss functions:

```
class TwoLayerNet(object):
    """
    A two-layer fully-connected neural network for binary classification.
    We train the network with a softmax output and cross entropy loss function
    with L2 regularization on the weight matrices. The network uses a ReLU
    nonlinearity after the first fully connected layer.
    Input: X
    Hidden states for layer 1: h1 = XW1 + b1
    Activations: a1 = ReLU(h1)
    Hidden states for layer 2: h2 = a1W2 + b2
    Probabilities: s = softmax(h2)

    ReLU function:
    (i) x = x if x >= 0  (ii) x = 0 if x < 0

    The outputs of the second fully-connected layer are the scores for each class.
    """

    def __init__(self, input_size, hidden_size, output_size, std=1e-4):
        """
        Initialize the model. Weights are initialized to small random values and
        biases are initialized to zero. Weights and biases are stored in the
        variable self.params, which is a dictionary with the following keys:

        W1: First layer weights; has shape (D, H)
        b1: First layer biases; has shape (H,)
        W2: Second layer weights; has shape (H, C)
        b2: Second layer biases; has shape (C,)

        Inputs:
        - input_size: The dimension D of the input data.
        - hidden_size: The number of neurons H in the hidden layer.
        - output_size: The number of classes C.
        """
        self.params = {}
        self.params['W1'] = std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)
```

```python
def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optional; if it
      is not passed then we only return scores, and if it is passed then we
      instead return the loss and gradients.
    - reg: Regularization strength.

    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function; has the same keys as self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass

    ### ========== TODO : START ========== ###
    #   Calculate the output of the neural network using forward pass.
    #   The expected result should be a matrix of shape (N, C), where:
    #      - N is the number of examples in the input dataset 'X'.
    #      - C is the number of classes.
    #   Use 'h1' as the first hidden layer output
    h1=np.dot(X, self.params['W1']) + self.params['b1']
    #   Apply the ReLU activation function to 'h1' to get 'a1'. Use np.maximum for ReLU implementation.
    a1 = np.maximum(0, h1)
    #   The output 'scores' is the result of the second layer (before applying softmax).
    scores = np.dot(a1, self.params['W2']) + self.params['b2']
    #   Refer to the model architecture comments at the beginning of this class for more details.
    #   Note: Do not use a for loop in your implementation.
    ##  Part (b): Implement the forward pass and compute scores.

    ### ========== TODO : END ========== ###
```

(c) 1 What is the formula for calculating ℓ_2 regularization? Write it down here only the regularization term with factor $\frac{\lambda}{2}$. Then, implement the ℓ_2 regularization term in "part (c)", take a screenshot of your code and paste it here.

Recall from the generalization lecture the definition of l_2 regularization:

$regularization = \frac{\lambda}{2} \sum_i W_i^2 = \frac{\lambda}{2}(\sum(W_1 * W_1) + (\sum(W_2 * W_2))$

```
### ========== TODO : START ========== ###
#   Calculate the regularization loss. Multiply the regularization
#   loss by 0.5 (in addition to the regularization factor 'reg').
##  Part (c): Implement the regularization loss
reg_loss = 0.5 * reg * (np.sum(W1 * W1) + np.sum(W2 * W2))

### ========== TODO : END ========== ###
```

(d) 2 Recall that we can express the final hypothesis for a binary classifier using sigmoid (only 1 output unit) or softmax (2 output units). Implement the softmax cross entropy loss and the gradient w.r.t the inputs in the function `softmax_loss(x,y)`, take a screenshot of your code and paste it here.
Suppose your softmax cross entropy loss function takes $\boldsymbol{X}$ and $\boldsymbol{Y}$ as inputs. Specifically, input $\boldsymbol{X}$ is the output score after the second layer, where $x_j^{(i)}$ is the output score of sample i being in class j; input $\boldsymbol{Y}$ is the target label represented as one-hot vectors, where $y^{(i)} \in \mathbb{R}^C$ is the one-hot encoded label of sample i, e.g, for binary classification, $y^{(i)} = [0, 1]$ or $y^{(i)} = [1, 0]$. What is the formula for calculating the softmax cross-entropy loss and gradient? Write it down here. You can write your formula using $x_j^{(i)}$ and $y_j^{(i)}$.

Recall from lecture the following information:

Inputs: $x \in R^d, y \in R^C$

Representation: $W \in R^{Cxd}$, $b \in R^C$

score vector: $s_\theta = Wx + b$

$h = max(0, W_1x + b_1), s = W_2h + b_2$

hypothesis: $f_\theta(x) = softmax(s_\theta) = \frac{exp(s_\theta)}{\sum_{j\in\{1,\dots C\}} exp(s_{\theta,j})}$

$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_c^{(i)} log f_\theta(x^{(i)})$

Thus, upon taking a partial derivative w.r.t $s_{\theta,j}$, I can obtain the gradient:

$\frac{\partial J(\theta)}{\partial s_{\theta,j}} = \frac{\partial}{\partial s_{\theta,j}}(-\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_c^{(i)} log f_\theta(x^{(i)})) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C}(\frac{\partial}{\partial s_{\theta,j}} y_c^{(i)} log f_\theta(x^{(i)})))$

$= -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C}((y_c^{(i)})(\frac{\partial}{\partial s_{\theta,j}} log f_\theta(x^{(i)})) + (\frac{\partial}{\partial s_{\theta,j}} y_c^{(i)})(log f_\theta(x^{(i)}))$

$= -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C}(y_c^{(i)})(\frac{\partial}{\partial s_{\theta,j}} log f_\theta(x^{(i)})).$

However I know that generally, the derivative of the logarithm of the softmax function is:

$\frac{\partial}{\partial s_{\theta,j}} log(f_{\theta,j}(x^{(i)}) = \frac{\partial}{\partial s_{\theta,j}}(log(exp(s_{\theta,j})) - log(\sum_{k=1}^{C} exp(s_{\theta,k}))) = 1 - \frac{exp(s_{\theta,j})}{\sum_{k=1}^{C} exp(s_{\theta,k})} = 1 - f_{\theta,j}(x^{(i)})$

$\implies \frac{\partial J(\theta)}{\partial x_j^{(i)}} = -\frac{1}{n}\sum_{i=1}^{n}(y_j^{(i)} - f_{\theta,j}(x^{(i)}))$, such that $f_{\theta,j}(x^{(i)}) = \frac{exp(x_j^{(i)})}{\sum_{k=1}^{C} exp(x_k^{(i)})}$

```
# If the targets are not given then jump out, we're done
if y is None:
    return scores

if y.ndim == 1:
    y_one_hot = np.zeros((N, W2.shape[1]))
    y_one_hot[np.arange(N), y] = 1
else:
    y_one_hot = y

# Compute the loss
loss = None

# scores is num_examples by num_classes (N, C)
def softmax_loss(x, y):
    ### ========== TODO : START ========== ###
    #   Calculate the cross entropy loss after softmax output layer.
    #   This function should return loss and dx
    #probs = np.exp(x - np.max(x, axis=1, keepdims=True)) # Other Notes:
    #this operation is called stable softmax: numerically more stable as it reduces
    #overflow issues by not letting the numerator and denominator grow too big.
    #probs /= np.sum(probs, axis=1, keepdims=True)


    shifted_logits = x - np.max(x, axis=1, keepdims=True)
    Z = np.sum(np.exp(shifted_logits), axis=1, keepdims=True)
    log_probs = shifted_logits - np.log(Z)
    probs = np.exp(log_probs)

    N = x.shape[0]
    ##  Part (d): Implement the CrossEntropyLoss
    loss = -np.sum(y * log_probs) / N
    ##  Part (d): Implement the gradient of y wrt x
    dx = (probs - y) / N

    ### ========== TODO : END ========== ###
    return loss, dx


data_loss, dscore = softmax_loss(scores, y_one_hot)
```

(e) 3 Implement the back-propagation process by completing the gradient computation for W2 and b2 in "part (e)", take a screenshot of your code and paste it here. What is the formula for calculating the gradient of W2 and b2? Write it down here.

Recall that in general, from the lecture notes I know that:

Inputs: $x \in R^d, y \in R^C$

Representation: $W \in R^{Cxd}$, $b \in R^C$

score vector: $s_\theta = Wx + b, \;\; h = max(0, W_1 x + b_1)$

$s = W_2 h + b_2$, or more specifically, I can write $s_\theta^{(i)} = W_2 h^{(i)} + b_2$

D-Score: The d-score can be denoted as $\frac{\partial J}{\partial s}$ and refers to the gradient of the loss J w.r.t the score vector s.

$\frac{\partial J(\theta)}{\partial W_2} = \frac{1}{n}\sum_{i=1}^{n}(h^{(i)T})\frac{\partial J(\theta)}{\partial s_\theta^{(i)}}$, such that the gradient $\frac{\partial J(\theta)}{\partial s_\theta^{(i)}}$ is a the d-score vector.

$\frac{\partial J(\theta)}{\partial b_2} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial J(\theta)}{\partial s_\theta^{(i)}}$, such that the gradient $\frac{\partial J(\theta)}{\partial s_\theta^{(i)}}$ is a the d-score vector.

WITH REGULARIZATION:

$\frac{\partial J(\theta)}{\partial W_2} = \frac{1}{n}\sum_{i=1}^{n}(h^{(i)T})\frac{\partial J(\theta)}{\partial s_\theta^{(i)}} + \lambda W_2$

$\frac{\partial J(\theta)}{\partial b_2} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial J(\theta)}{\partial s_\theta^{(i)}}$

```
loss = data_loss + reg_loss


grads = {}

### ========== TODO : START ========== ###
#  Compute backpropagation
#  Remember the loss contains two parts: cross-entropy and regularization.
# The computation for gradients of W1 and b1 shown here can be regarded as a reference.
## Part (e): Implement the computations of gradients for W2 and b2.
grads['W2'] = np.dot(a1.T, dscore) + reg * W2
grads['b2'] = np.sum(dscore, axis=0)

dh = np.dot(dscore, W2.T)
dh[a1 <= 0] = 0

grads['W1'] = np.dot(X.T, dh) + reg * W1
#grads['b1'] = np.ones(N).dot(dh)
grads['b1'] = np.sum(dh, axis=0)
### ========== TODO : END ========== ###

return loss, grads
```

(f) 4 Implement the prediction function in "part (f)" and check the correctness of your implementation above by running the predictions on validation and test sets. Adjust the learning rate in $10^{-5}, 10^{-4}, 10^{-3}, 5 \times 10^{-3}, 10^{-1}$. Report the best accuracy you get and the corresponding learning rate. Briefly discuss your observations of using different learning rates.

```
def predict(self, X):
    """
    Use the trained weights of this two-layer network to predict labels for
    data points. For each data point we predict scores for each of the C
    classes, and assign each data point to the class with the highest score.

    Inputs:


    - X: A numpy array of shape (N, D) giving N D-dimensional data points to
      classify.

    Returns:
    - y_pred: A numpy array of shape (N,) giving predicted labels for each of
      the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
      to have class c, where 0 <= c < C.
    """
    y_pred = None

    ### ========== TODO : START ========== ###
    #   Predict the class given the input data.
    ##  Part (f): Implement the prediction function
    h = np.maximum(0, np.dot(X, self.params['W1']) + self.params['b1'])
    scores = np.dot(h, self.params['W2']) + self.params['b2']
    y_pred = np.argmax(scores, axis=1)

    ### ========== TODO : END ========== ###

    return y_pred
```

```
input_size = 576
hidden_size = 50
num_classes = 2
net = TwoLayerNet(input_size, hidden_size, num_classes)

# Train the network
for learning_rate in [1e-5, 1e-4, 1e-3, 5e-3, 1e-1]:
    print('learning_rate: ', learning_rate)
    stats = net.train(X_train, y_train, X_val, y_val,
              num_iters=1000, batch_size=200,
              learning_rate=learning_rate, learning_rate_decay=0.95,
              reg=0.1, verbose=True)

    # Predict on the validation set
    val_acc = (net.predict(X_val) == y_val).mean()
    print('Validation accuracy: ', val_acc)

    # Save this net as the variable subopt_net for later comparison.
    subopt_net = net
    test_acc = (subopt_net.predict(X_test) == y_test).mean()
    print('Test accuracy (subopt_net): ', test_acc)
    print('\n')
```

Question: Report the best accuracy you get and the corresponding learning rate.

Answer: The best accuracy is achieved with a learning rate of 0.001, resulting in a validation accuracy of 97.24% and a test accuracy of 96.95%.

Question: Briefly discuss your observations of using different learning rates.

Answer:

1. Learning Rate: 10^{-5}: The leaning rate is low, leading to slow convergence. The loss decreases gradually, indicating that the model is learning but at a slower pace. The accuracy is decent but not optimal.

2. Learning Rate: 10^{-4}: This learning rate shows a significantly improvement in both loss reduction and accuracy compared to the previous one. It has a good balance between learning speed and stability, leading to a much better model performance.

3. Learning Rate: 10^{-3}: This leaning rate provides the best results among all the rates tested. The loss decreases rapidly, and the model achieves high accuracy of both validation and test sets. I indicates an optimal learning rate for this specific model and dataset.

4. Learning Rate: $5x10^{-3}$: This leaning rate seems too high, as indicated by the initial decrease in loss followed by an increase. This model might be skipping the optimal points during training., leading to lower accuracy.

5. Learning Rate: 0.1: This learning rate is excessively high, causing the model to perform poorly. The initially decreasing but then fluctuates, and the accuracy is barely better than random guessing. This is a case of overshooting this minimum during optimization.

```
learning_rate:  1e-05
iteration 0 / 1000: loss 0.6931571803587853
iteration 100 / 1000: loss 0.6931449366352727
iteration 200 / 1000: loss 0.6931272330289089
iteration 300 / 1000: loss 0.6930858008653276
iteration 400 / 1000: loss 0.6930517525585971
iteration 500 / 1000: loss 0.6930273879532359
iteration 600 / 1000: loss 0.6929132827323534
iteration 700 / 1000: loss 0.6927762913144732
iteration 800 / 1000: loss 0.6926119023276132
iteration 900 / 1000: loss 0.6923681827118727
Validation accuracy:  0.7433
Test accuracy (subopt_net):  0.7457


learning_rate:  0.0001
iteration 0 / 1000: loss 0.6921234023765391
iteration 100 / 1000: loss 0.6515572939426283
iteration 200 / 1000: loss 0.4886364551759082
iteration 300 / 1000: loss 0.35563760267235994
iteration 400 / 1000: loss 0.35062075153918604
iteration 500 / 1000: loss 0.2771702613170189
iteration 600 / 1000: loss 0.27621207924038427
iteration 700 / 1000: loss 0.27729886920878116
iteration 800 / 1000: loss 0.3036217516011145
iteration 900 / 1000: loss 0.33391356950337014
Validation accuracy:  0.8849
Test accuracy (subopt_net):  0.8812


learning_rate:  0.001
iteration 0 / 1000: loss 0.30654927525755254
iteration 100 / 1000: loss 0.3169978149747196
iteration 200 / 1000: loss 0.19507573837642797
iteration 300 / 1000: loss 0.14846130272704358
iteration 400 / 1000: loss 0.1007710858387622
iteration 500 / 1000: loss 0.16938307890777807
iteration 600 / 1000: loss 0.11427258492315304
iteration 700 / 1000: loss 0.09075696823210015
iteration 800 / 1000: loss 0.08510954976295412
iteration 900 / 1000: loss 0.1111473419444934
Validation accuracy:  0.9724
Test accuracy (subopt_net):  0.9695


learning_rate:  0.005
iteration 0 / 1000: loss 0.10379700398175276
iteration 100 / 1000: loss 0.538793983951545
iteration 200 / 1000: loss 0.47537758626866344
iteration 300 / 1000: loss 0.3932618040082146
iteration 400 / 1000: loss 0.40029318385360807
iteration 500 / 1000: loss 0.3813438261789614
iteration 600 / 1000: loss 0.3444563062640672
iteration 700 / 1000: loss 0.32586570097968903
iteration 800 / 1000: loss 0.2791761047790897
iteration 900 / 1000: loss 0.297466193315009
Validation accuracy:  0.9496
Test accuracy (subopt_net):  0.9492


learning_rate:  0.1
iteration 0 / 1000: loss 0.27516480083140776
iteration 100 / 1000: loss 32.55998922322143
iteration 200 / 1000: loss 5.414354427158727
iteration 300 / 1000: loss 1.4249506715258873
iteration 400 / 1000: loss 0.8120683533732994
iteration 500 / 1000: loss 0.7133101805349776
iteration 600 / 1000: loss 0.6980098964943378
iteration 700 / 1000: loss 0.6935968700949952
iteration 800 / 1000: loss 0.6902145084052099
iteration 900 / 1000: loss 0.6931821530433099
Validation accuracy:  0.506
Test accuracy (subopt_net):  0.5074
```

Problem 4 (k-Means Clustering [15 PTS])

In this problem, we shall apply k-means Algorithm to a popular visual classification dataset CIFAR-10 [1]. CIFAR-10 dataset consists of 60K labeled 32 x 32 colored images i.e., each image would have 3 channels corresponding to RGB colors. Out of the total 60K images, 50K images belong to the training set and 10K belong to the testing set. Within the scope of this problem, we shall work with the testing set only for computational efficiency purposes. Additionally, as k-means is an unsupervised learning algorithm, we shall discard the labels associated with the images.

Please find the code skeleton for this problem in `Fall23-CS146-HW4.ipynb` notebook shared with you as a part of this HW.

A Implementing *k*-means

Rather than training k-means from scratch, we are going to make use of a pre-existing implementation available in scikit-learn.[2]

(a) 5 Make use of the `dataloader()` function to load the data in our notebook. Within this function, the data is loaded using `datasets` package in Tensorflow. [3] **Tip**: Run this notebook in Google Colab as it saves you a lot of time spent in installing Tensorflow.

This function will return 10K testing images of CIFAR-10 dataset along with their ground truth label. We provide `visualize(X, ind)` function to visualize index `ind` in the data `X`. Before using k-means algorithm, we need to make sure that the data has the shape (10000, N), where N = 32 x 32 x 3, instead of (10000, 32, 32, 3). Your first task would be to implement `reshape` function that will convert the tensor form of the input to a 2D matrix of the form 10000 x N. Take a screenshot of your code and paste it here.

[1] https://www.cs.toronto.edu/ kriz/cifar.html
[2] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
[3] https://www.tensorflow.org/datasets/catalog/cifar10: :text=The%20CIFAR%2D10%20dataset%20consists,images%20and%2010000%20test%2

```
[1]  ## Function to load the CIFAR10 data
     ## Documentation of CIFAR10: https://www.cs.toronto.edu/~kriz/cifar.html
     def dataloader():
       import tensorflow as tf
       cifar10 = tf.keras.datasets.cifar10
       (_, _), (X, y) = cifar10.load_data()
       return X, y
```

```
[2]  ## simple utility function to visualize the data
     def visualize(X, ind):
       from PIL import Image
       plt.imshow(Image.fromarray(X[ind], 'RGB'))
```

```
[3]  X, y = dataloader()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 10s 0us/step
```

```
[4]  # 10K images of size 32 x 32 x 3
     # where 32 x 32 is the height and width of the image
     # 3 is the number of channels 'RGB'
     X.shape, y.shape
```

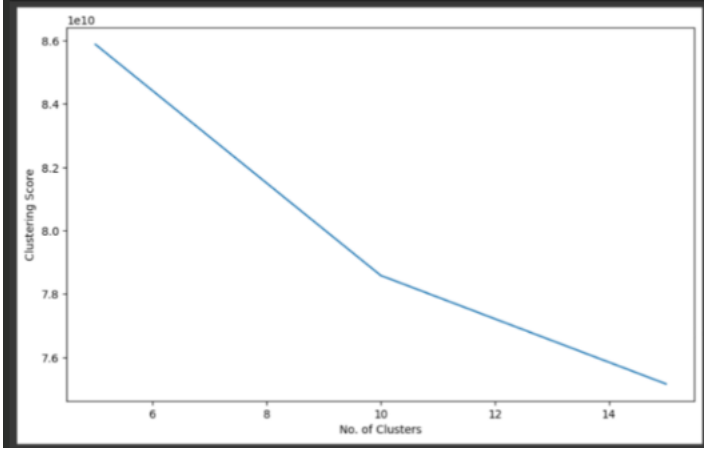
```
((10000, 32, 32, 3), (10000, 1))
```

```
[5]  '''
       Implement this function to form a 10000 x N matrix
       from 10000 x 32 x 32 x 3 shape input.
     '''
     def reshape(X):
       '''
         Write one line of code here
       '''
       ### ========== TODO : START ========== ###
       # part (a)

       X = X.reshape(X.shape[0], -1)

       ### ========== TODO : END ========== ###
       return X
```

```
[6]  X = reshape(X)
```

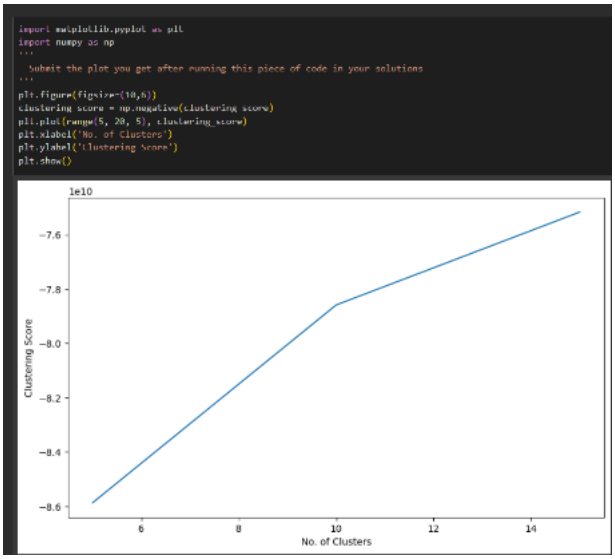
(b) 5 Once you have reshaped your input, you are ready to apply k-means algorithm to your data. Note that the number of cluster centers k is a hyperparameter. As we are in the unsupervised regime, we do not have access to any labeled validation set that could be used to decide the best for k. Thus, we use some pre-defined metrics to decide what choice of k should be the best. In this HW, we are going to use the *Sum of squared distances of samples to their closest cluster center* as our score metric. However, we are also aware that k-means algorithm can lead to different convergence based on the initialization of the clusters. To account for the differences between the score due to various initializations, we apply the algorithm across 3 random seeds (`random_state` in the code). Here, your task is to write a few lines of code to fit the k-means algorithm and calculate the score metric for all three runs, take a screenshot of your code and paste it here. Finally, submit the results graph you get after running the plotting block in the notebook that takes in your calculated scores as the input.

```
import matplotlib.pyplot as plt
'''
  Submit the plot you get after running this piece of code in your solutions
'''
plt.figure(figsize=(10,6))
plt.plot(range(5, 20, 5), clustering_score)
plt.xlabel('No. of Clusters')
plt.ylabel('Clustering Score')
plt.show()
```



Here is the negative of the graph above:

```
import matplotlib.pyplot as plt
import numpy as np
'''
  Submit the plot you get after running this piece of code in your solutions
'''
plt.figure(figsize=(10,6))
clustering_score = np.negative(clustering_score)
plt.plot(range(5, 20, 5), clustering_score)
plt.xlabel('No. of Clusters')
plt.ylabel('Clustering Score')
plt.show()
```



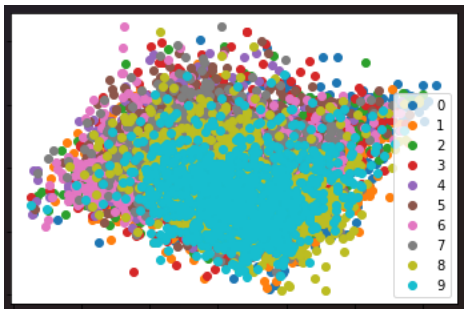


Figure 1: PCA of CIFAR-10 test data in 2-D with ground truth labels.

B Visualization

5 Another aspect of applying k-means algorithm involves visualizing the cluster assignments. As described above, the CIFAR-10 data is 3092 dimensional (32 x 32 x 3) which is quite impossible to visualize for us. However, we can project our high dimensional input data into lower dimensional space using PCA algorithm discussed in the class. Once you have the PCA output of the data, we can visualize the transformed low dimensional data easily.

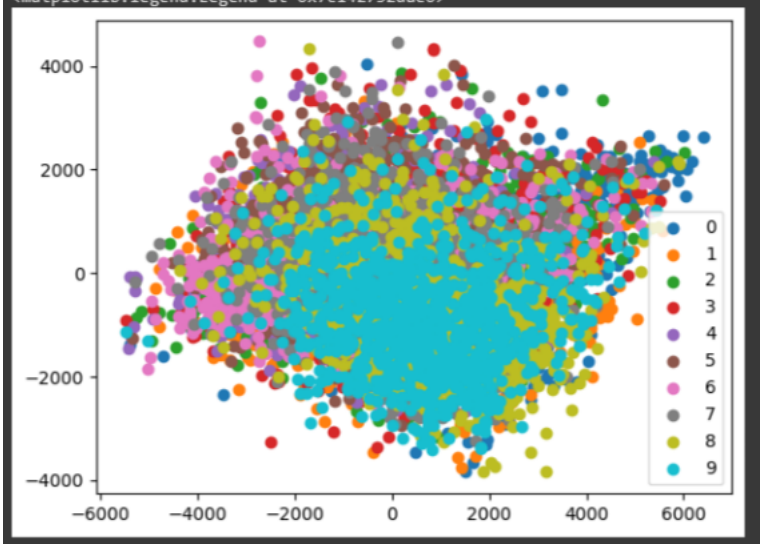
We provide a simple implementation of PCA using its support in `sklearn.decomposition`. To get some sense of how PCA can be beneficial in plotting the high dimensional data, we provide a code block that provides you the PCA of the input data along with its ground truth labels. Running that block should output a plot which looks like Figure 1.

Your task is to first apply k-means algorithm with $k = 10$ and `random_state = 42` to the PCA-transformed data, and get the predicted label assignments to each data sample. Careful inspection of the k-means documentation should help you to get the predicted labels in one-line of code. Using these predicted labels, implement a code to get a scatter plot of the low-dimensional PCA data with these new assignments. Take a screenshot of your code and paste it here. Submit the final graph you get for this part.

```
from sklearn.decomposition import PCA
pca = PCA(2)
#Transform the data
df = pca.fit_transform(X)
```

```
import numpy as np
### Analyzing the input data in 2D based on its true labels

u_labels = np.unique(y[:, 0])

for i in u_labels:
    plt.scatter(df[y[:, 0] == i , 0] , df[y[:, 0] == i , 1] , label = i)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7c142732dae0>
```



```
'''
  Submit the output plot as a part of the solutions
'''

kmeans = KMeans(n_clusters = 10, init = 'random', random_state = 42)
'''
  Write 1 - 2 line of code to get the predicted labels of the 10-clusters
'''
### ========== TODO : START ========== ###

kmeans.fit(X)
label = kmeans.labels_

### ========== TODO : END ========== ###

u_labels = np.unique(label)

#plotting the results:

for i in u_labels:
    '''
      Write one line of code to get a scatter plot for i-th cluster.
      Have its label = i
    '''
    ### ========== TODO : START ========== ###

    plt.scatter(df[label == i, 0], df[label == i, 1], label = i)

    ### ========== TODO : END ========== ###

plt.legend()
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: Fut
  warnings.warn(
```