Problem 1:

(a) $\vec{x}^{(1)} \cdot \vec{v}_1 = [5.51 \quad 5.35]\begin{bmatrix}0.694\\0.720\end{bmatrix} = 7.67594$

(b) $\tilde{x}^{(i)} = Uy^{(i)} = [0.694 \quad 0.720] \cdot 7.67594 = [5.327102 \quad 5.526677]$

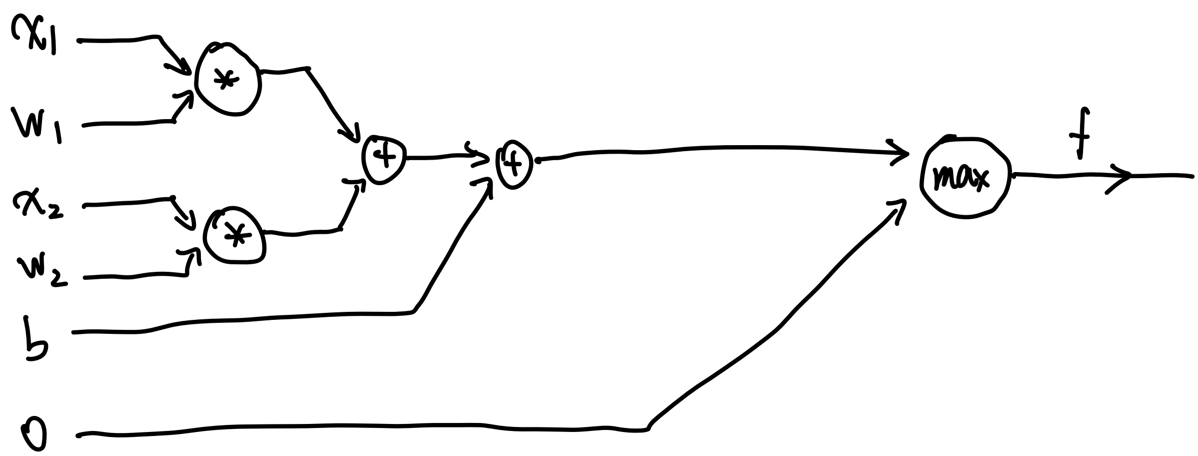
(c) The second principle direction should be orthogonal to the first 1.

So $\vec{v}_2 = [-0.720 \quad 0.694]$

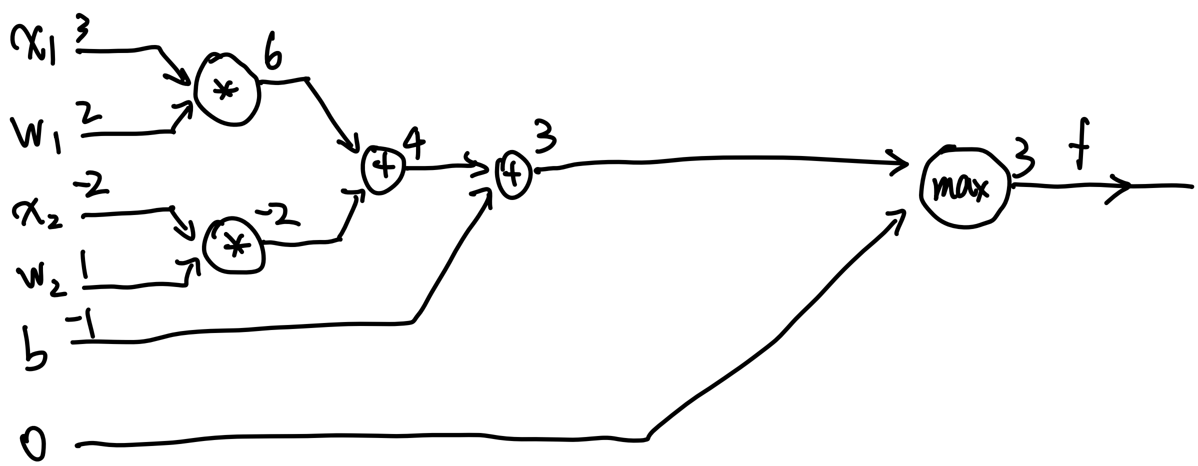
Then principal component score $= \vec{v}_2^{\,T} x = \begin{bmatrix}-0.720\\0.694\end{bmatrix}[5.51 \quad 5.35] = -0.2543$

Problem 2:

(a)

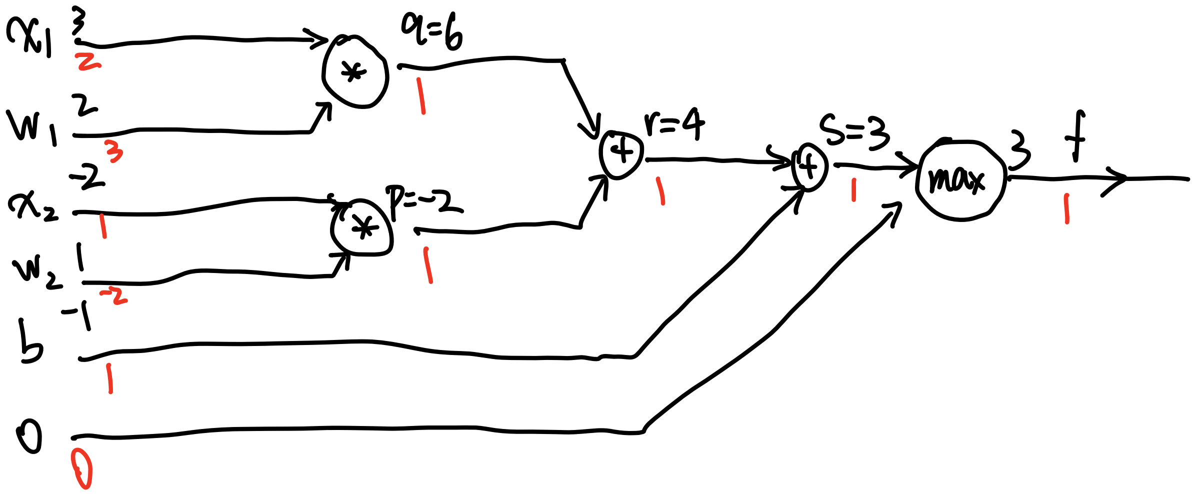



(b)




So we have $f_\theta(x) = 3$

(c)




let's define $q = x_1 \cdot w_1 = 6$, $p = x_2 \cdot w_2 = -2$, $r = q + p = 4$, $s = r + b = 3$

Then we have $\frac{\partial f_\theta(x)}{\partial f_\theta(x)} = 1$

$$\frac{\partial f_\theta(x)}{\partial s} = 1$$

$$\frac{\partial f_\theta(x)}{\partial r} = \frac{\partial f_\theta(x)}{\partial b} = \frac{\partial f_\theta(x)}{\partial s} = 1$$

$$\frac{\partial f_\theta(x)}{\partial q} = \frac{\partial f_\theta(x)}{\partial p} = \frac{\partial f_\theta(x)}{\partial r} = 1$$

$$\frac{\partial f_\theta(x)}{\partial x_1} = 1 \times 2 = 2$$

$$\frac{\partial f_\theta(x)}{\partial w_1} = 1 \times 3 = 3$$

$$\frac{\partial f_\theta(x)}{\partial x_2} = 1 \times 1 = 1$$

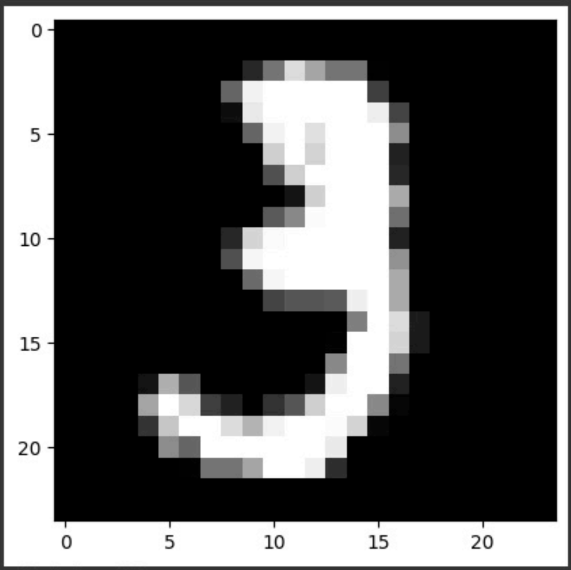
$$\frac{\partial f_\theta(x)}{\partial w_2} = 1 \times (-2) = -2$$

Thus, we have $\frac{\partial f_\theta(x)}{\partial w_1} = 3$, $\frac{\partial f_\theta(x)}{\partial w_2} = -2$, $\frac{\partial f_\theta(x)}{\partial b} = 1$

Problem 3.

(a)

```python
# 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)

```python
### ========== 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
#   Apply the ReLU activation function to 'h1' to get 'a1'. Use np.maximum for ReLU implementation.
#   The output 'scores' is the result of the second layer (before applying softmax).
#   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.

h1 = np.dot(X, W1) + b1
a1 = np.maximum(0, h1)
scores = np.dot(a1, W2) + b2


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

(c)

$$R = \frac{\lambda}{2} \cdot \left( \Sigma_{ij} W_{1ij}^2 + \Sigma_{ij} W_{2ij}^2 + \cdots + \Sigma_{ij} W_{nij}^2 \right)$$

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

### ========== 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)

$$loss = -\frac{1}{N} \Sigma_{i=1}^{N} \Sigma_{j=1}^{C} y_j^{(i)} \log\left( \frac{e^{x_j^{(i)}}}{\Sigma_{k=1}^{C} e^{x_k^{(i)}}} \right)$$

$$\text{Gradient} = \text{predicted\_y} - \text{actual\_y} = \frac{e^{x_j^{(i)}}}{\Sigma_{k=1}^{C} e^{x_k^{(i)}}} - y_j^{(i)}$$

```
# 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:
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    ##  Part (d): Implement the CrossEntropyLoss
    logprobs = -np.log(probs[np.arange(N), y])
    loss = np.sum(logprobs) / N
    ##  Part (d): Implement the gradient of y wrt x
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N

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

(e)

$$\frac{dL}{dW_2} = \frac{1}{N} a_1^T \cdot dscore + reg \cdot W_2$$

$$\frac{dL}{db_2} = \frac{1}{N} \Sigma_{i=1}^{T} dscore^{(i)}$$

```
### ========== TODO : START ========== ###
#  Compute backpropagation
#  Remember the loss contains two parts: cross-entropy and regularization.
## 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)
### ========== TODO : END ========== ###

return loss, grads
```

(f) The performance are approaching the best when learning rate is medium. And when learning rate is too high or too low, the performance is getting poorer.

```python
### ========== TODO : START ========== ###
#   Predict the class given the input data.
##  Part (f): Implement the prediction function

W1, b1 = self.params['W1'], self.params['b1']
W2, b2 = self.params['W2'], self.params['b2']

# Forward pass
# First layer
h1 = np.dot(X, W1) + b1
a1 = np.maximum(0, h1)  # ReLU activation

# Second layer
scores = np.dot(a1, W2) + b2

# Predicted class
y_pred = np.argmax(scores, axis=1)

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

Best learning rate: $\alpha = 0.001$

validation accuracy: 0.9733

Test accuracy: 0.9678

Very Low learning rate results in slow convergence.

Very high learning rate may result in divergent.

```
learning_rate:  1e-05
iteration 0 / 1000: loss 0.6931649940072726
iteration 100 / 1000: loss 0.693154359458996
iteration 200 / 1000: loss 0.6931526464820942
iteration 300 / 1000: loss 0.6931347649910297
iteration 400 / 1000: loss 0.693103343905047
iteration 500 / 1000: loss 0.6931010596462825
iteration 600 / 1000: loss 0.6930638379042646
iteration 700 / 1000: loss 0.6930082301083845
iteration 800 / 1000: loss 0.6929382968535166
iteration 900 / 1000: loss 0.6929072867648536
Validation accuracy:  0.7899
Test accuracy (subopt_net):  0.7858


learning_rate:  0.0001
iteration 0 / 1000: loss 0.692801731114866
iteration 100 / 1000: loss 0.6719827463398109
iteration 200 / 1000: loss 0.5013737246731946
iteration 300 / 1000: loss 0.3241831299039786
iteration 400 / 1000: loss 0.36179172183527714
iteration 500 / 1000: loss 0.3308763959715828
iteration 600 / 1000: loss 0.3308903579575143
iteration 700 / 1000: loss 0.3145584885036203
iteration 800 / 1000: loss 0.3389736095308798
iteration 900 / 1000: loss 0.24493988512407197
Validation accuracy:  0.8845
Test accuracy (subopt_net):  0.8804


learning_rate:  0.001
iteration 0 / 1000: loss 0.3402363861952303
iteration 100 / 1000: loss 0.25044070293827736
iteration 200 / 1000: loss 0.14953351193130388
iteration 300 / 1000: loss 0.13545031903867275
iteration 400 / 1000: loss 0.09973515145836007
iteration 500 / 1000: loss 0.14104951597705032
iteration 600 / 1000: loss 0.11138129430291509
iteration 700 / 1000: loss 0.1143827758435182
iteration 800 / 1000: loss 0.15094791984968528
iteration 900 / 1000: loss 0.07903575570694399
Validation accuracy:  0.9733
Test accuracy (subopt_net):  0.9678
```

```
learning_rate:  0.005
iteration 0 / 1000: loss 0.10997640442696606
iteration 100 / 1000: loss 0.4957102099970299
iteration 200 / 1000: loss 0.46130670394323536
iteration 300 / 1000: loss 0.4059288527255275
iteration 400 / 1000: loss 0.3700568077290139
iteration 500 / 1000: loss 0.40863506773802044
iteration 600 / 1000: loss 0.3059037670573398
iteration 700 / 1000: loss 0.3225089167819696
iteration 800 / 1000: loss 0.2962714211639297
iteration 900 / 1000: loss 0.23732551453505904
Validation accuracy:  0.944
Test accuracy (subopt_net):  0.9458


learning_rate:  0.1
iteration 0 / 1000: loss 0.2549922226451809
iteration 100 / 1000: loss 20.888060524622333
iteration 200 / 1000: loss 3.6870699442665633
iteration 300 / 1000: loss 1.1601519169336312
iteration 400 / 1000: loss 0.7698893353634444
iteration 500 / 1000: loss 0.704939406146651
iteration 600 / 1000: loss 0.694959048833996
iteration 700 / 1000: loss 0.6949299189723591
iteration 800 / 1000: loss 0.6922929120402739
iteration 900 / 1000: loss 0.6938488314335312
Validation accuracy:  0.506
Test accuracy (subopt_net):  0.5074
```

Problem 4:

(a)

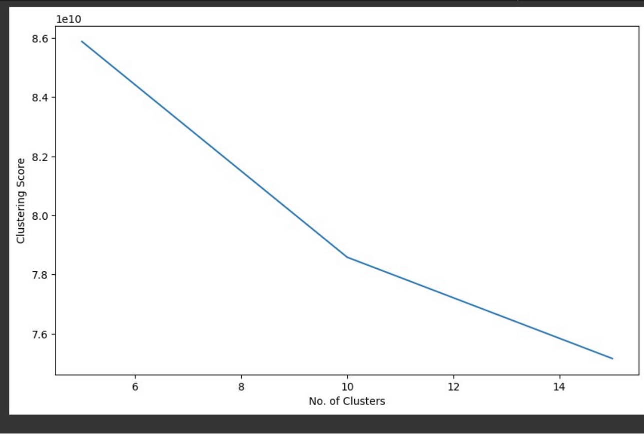
```
### ========== TODO : START ========== ###
# part (a)

X = X.reshape(10000, -1)

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

(b)

```
'''
  ### ========== TODO : START ========== ###
  # part (b)

  kmeans.fit(X)
  score += kmeans.inertia_

  ### ========== TODO : END ========== ###
clustering_score.append(score/3) ## divide by 3 because 3 random states
```

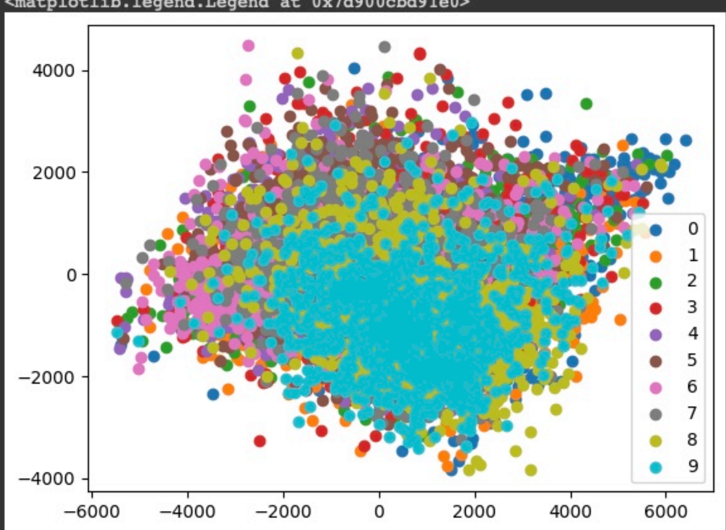
```
  0%|          | 0/3 [00:00<?, ?it/s]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: F
  warnings.warn(

 33%|███       | 1/3 [00:58<01:57, 58.60s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

 67%|██████    | 2/3 [01:50<00:54, 54.84s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

100%|██████████| 3/3 [03:01<00:00, 60.60s/it]
 33%|███       | 1/3 [03:01<06:03, 181.82s/it]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: F
  warnings.warn(

 33%|███       | 1/3 [01:11<02:22, 71.12s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

 67%|██████    | 2/3 [02:15<01:06, 66.96s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

100%|██████████| 3/3 [03:21<00:00, 67.00s/it]
 67%|██████    | 2/3 [06:22<03:13, 193.11s/it]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: F
  warnings.warn(

 33%|███       | 1/3 [01:32<03:05, 92.81s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

 67%|██████    | 2/3 [02:58<01:28, 88.57s/it]/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.p
  warnings.warn(

100%|██████████| 3/3 [05:00<00:00, 100.11s/it]
100%|██████████| 3/3 [11:23<00:00, 227.73s/it]
```

```
'''
  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()
```




```
### 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()
```

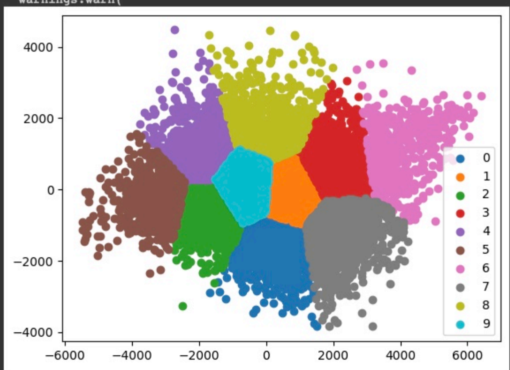
```
<matplotlib.legend.Legend at 0x7d900cbd91e0>
```



B visulization

```
'''
  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(df)
labels = kmeans.labels_

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

u_labels = np.unique(labels)

#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[labels == i , 0], df[labels == i , 1], label = i)

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

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

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` wi
  warnings.warn(
```



# Problem 1: A Two-Layer Neural Network for Binary Classification

```
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-ipyth
%load_ext autoreload
%autoreload 2
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

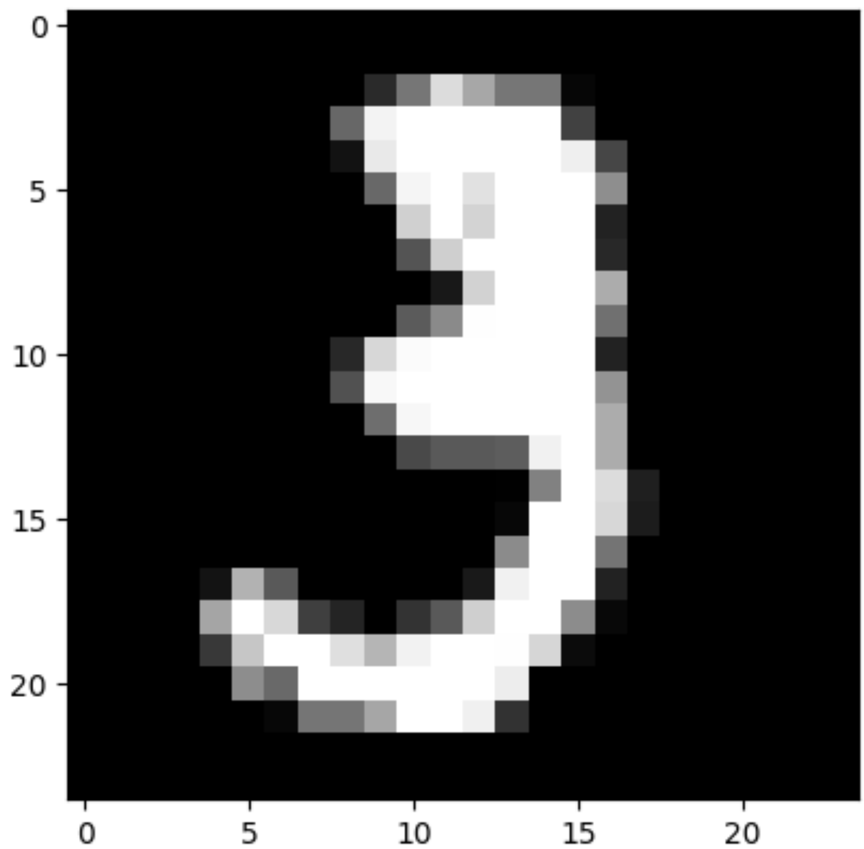
```
os.getcwd()
```

```
'/content'
```

```
## 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('/content/drive/MyDrive/Colab Not
X_test, y_test = load_mnist('/content/drive/MyDrive/Colab Notebooks/data/mnist'
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
```

In the following cells, you will build a two-layer neural network.

```
# convert to binary label
y_train = y_train.astype(int) % 2
y_val = y_val.astype(int) % 2
y_test = y_test.astype(int) % 2
```

```
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 cla
"""

def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values ar
    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)

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|
      an integer in the range 0 <= y[i] < C. This parameter is optional; if
```

```
      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]
    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 trai
      samples.
    - grads: Dictionary mapping parameter names to gradients of those param
      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
    scores = None

    ### ========== 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
    #   Apply the ReLU activation function to 'h1' to get 'a1'. Use np.maxi
    #   The output 'scores' is the result of the second layer (before apply
    #   Refer to the model architecture comments at the beginning of this c
    #   Note: Do not use a for loop in your implementation.
    ##  Part (b): Implement the forward pass and compute scores.

    h1 = np.dot(X, W1) + b1
    a1 = np.maximum(0, h1)
    scores = np.dot(a1, W2) + b2


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


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

    # 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:
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    ##  Part (d): Implement the CrossEntropyLoss
    logprobs = -np.log(probs[np.arange(N), y])
    loss = np.sum(logprobs) / N
    ##  Part (d): Implement the gradient of y wrt x
    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N

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


data_loss, dscore = softmax_loss(scores, y)

### ========== 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 ========== ###

loss = data_loss + reg_loss

grads = {}

### ========== TODO : START ========== ###
#  Compute backpropagation
#  Remember the loss contains two parts: cross-entropy and regularizati
## 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)
### ========== TODO : END ========== ###
```

```
        return loss, grads

    def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=1e-5, num_iters=100,
            batch_size=200, verbose=False):
        """
        Train this neural network using stochastic gradient descent.

        Inputs:
        - X: A numpy array of shape (N, D) giving training data.
        - y: A numpy array f shape (N,) giving training labels; y[i] = c means
          X[i] has label c, where 0 <= c < C.
        - X_val: A numpy array of shape (N_val, D) giving validation data.
        - y_val: A numpy array of shape (N_val,) giving validation labels.
        - learning_rate: Scalar giving learning rate for optimization.
        - learning_rate_decay: Scalar giving factor used to decay the learning
          after each epoch.
        - reg: Scalar giving regularization strength.
        - num_iters: Number of steps to take when optimizing.
        - batch_size: Number of training examples to use per step.
        - verbose: boolean; if true print progress during optimization.
        """
        num_train = X.shape[0]
        iterations_per_epoch = max(num_train / batch_size, 1)

        # Use SGD to optimize the parameters in self.model
        loss_history = []
        train_acc_history = []
        val_acc_history = []

        for it in np.arange(num_iters):
            X_batch = None
            y_batch = None

            #   Create a minibatch (X_batch, y_batch) by sampling batch_size
            #   samples randomly.

            b_index = np.random.choice(num_train, batch_size)
            X_batch = X[b_index]
            y_batch = y[b_index]

            # Compute loss and gradients using the current minibatch
            loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
            loss_history.append(loss)
```

```
            self.params['W1'] -= learning_rate * grads['W1']
            self.params['b1'] -= learning_rate * grads['b1']
            self.params['W2'] -= learning_rate * grads['W2']
            self.params['b2'] -= learning_rate * grads['b2']


            if verbose and it % 100 == 0:
                print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

            # Every epoch, check train and val accuracy and decay learning rate
            if it % iterations_per_epoch == 0:
                # Check accuracy
                train_acc = (self.predict(X_batch) == y_batch).mean()
                val_acc = (self.predict(X_val) == y_val).mean()
                train_acc_history.append(train_acc)
                val_acc_history.append(val_acc)

                # Decay learning rate
                learning_rate *= learning_rate_decay

        return {
          'loss_history': loss_history,
          'train_acc_history': train_acc_history,
          'val_acc_history': val_acc_history,
        }

    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 t
          classify.

        Returns:
        - y_pred: A numpy array of shape (N,) giving predicted labels for each
          the elements of X. For all i, y_pred[i] = c means that X[i] is predic
          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

        W1, b1 = self.params['W1'], self.params['b1']
```

```
        W2, b2 = self.params['W2'], self.params['b2']

        # Forward pass
        # First layer
        h1 = np.dot(X, W1) + b1
        a1 = np.maximum(0, h1)  # ReLU activation

        # Second layer
        scores = np.dot(a1, W2) + b2

        # Predicted class
        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')
```

```
learning_rate:  1e-05
iteration 0 / 1000: loss 0.6931649940072726
iteration 100 / 1000: loss 0.693154359458996
iteration 200 / 1000: loss 0.6931526464820942
iteration 300 / 1000: loss 0.6931347649910297
iteration 400 / 1000: loss 0.6931033439050472
iteration 500 / 1000: loss 0.6931010596462825
iteration 600 / 1000: loss 0.6930638379042646
iteration 700 / 1000: loss 0.6930082301083845
```

```
iteration 700 / 1000: loss 0.6930082301083845
iteration 800 / 1000: loss 0.6929382968535166
iteration 900 / 1000: loss 0.6929072867648536
Validation accuracy:  0.7899
Test accuracy (subopt_net):  0.7858


learning_rate:  0.0001
iteration 0 / 1000: loss 0.692801731114866
iteration 100 / 1000: loss 0.6719827463398109
iteration 200 / 1000: loss 0.5013737246731946
iteration 300 / 1000: loss 0.3241831299039786
iteration 400 / 1000: loss 0.36179172183527714
iteration 500 / 1000: loss 0.3308763959715828
iteration 600 / 1000: loss 0.3308903579575143
iteration 700 / 1000: loss 0.3145584885036203
iteration 800 / 1000: loss 0.3389736095308798
iteration 900 / 1000: loss 0.24493988512407197
Validation accuracy:  0.8845
Test accuracy (subopt_net):  0.8804


learning_rate:  0.001
iteration 0 / 1000: loss 0.3402363861952303
iteration 100 / 1000: loss 0.25044070293827736
iteration 200 / 1000: loss 0.14953351193130388
iteration 300 / 1000: loss 0.13545031903867275
iteration 400 / 1000: loss 0.09973515145836007
iteration 500 / 1000: loss 0.14104951597705032
iteration 600 / 1000: loss 0.11138129430291509
iteration 700 / 1000: loss 0.1143827758435182
iteration 800 / 1000: loss 0.15094791984968528
iteration 900 / 1000: loss 0.07903575570694399
Validation accuracy:  0.9733
Test accuracy (subopt_net):  0.9678


learning_rate:  0.005
iteration 0 / 1000: loss 0.10997640442696606
iteration 100 / 1000: loss 0.4957102099970299
iteration 200 / 1000: loss 0.46130670394323536
iteration 300 / 1000: loss 0.4059288527255275
iteration 400 / 1000: loss 0.3700568077290139
iteration 500 / 1000: loss 0.40863506773802044
iteration 600 / 1000: loss 0.3059037670573398
iteration 700 / 1000: loss 0.3225089167819696
iteration 800 / 1000: loss 0.2962714211639297
iteration 900 / 1000: loss 0.23732551453505904
Validation accuracy:  0.944
Test accuracy (subopt_net):  0.9458
```

# Problem 2: K-Means Algorithm

```
## 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


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

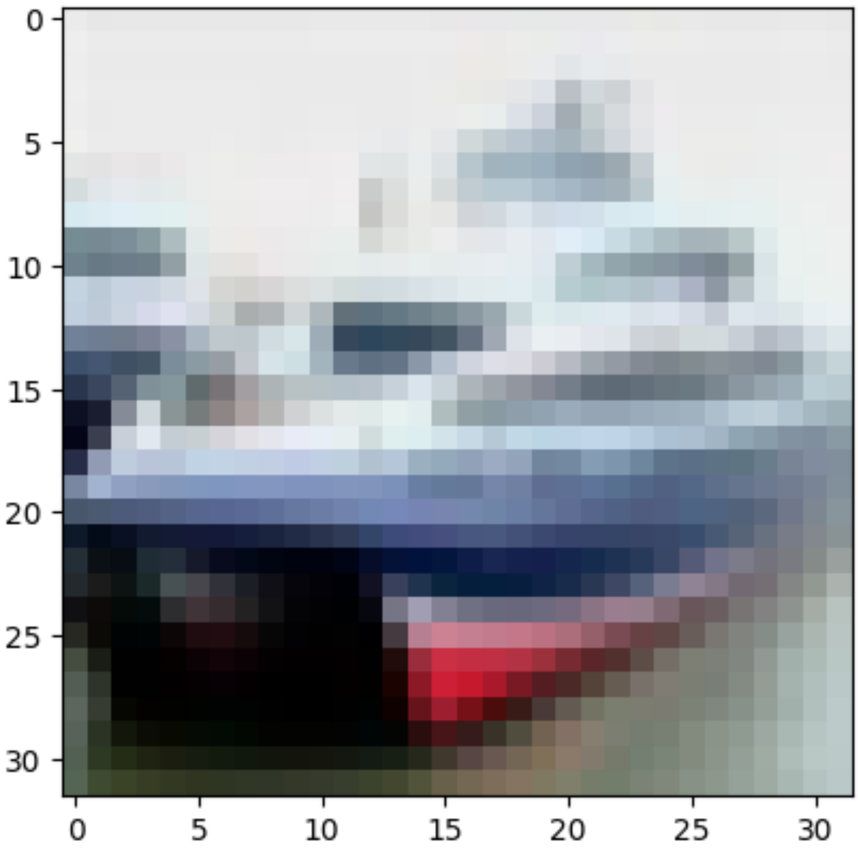
```
X, y = dataloader()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.
170498071/170498071 [==============================] - 3s 0us/step
```

```
# 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))
```

```
visualize(X, 1)
```

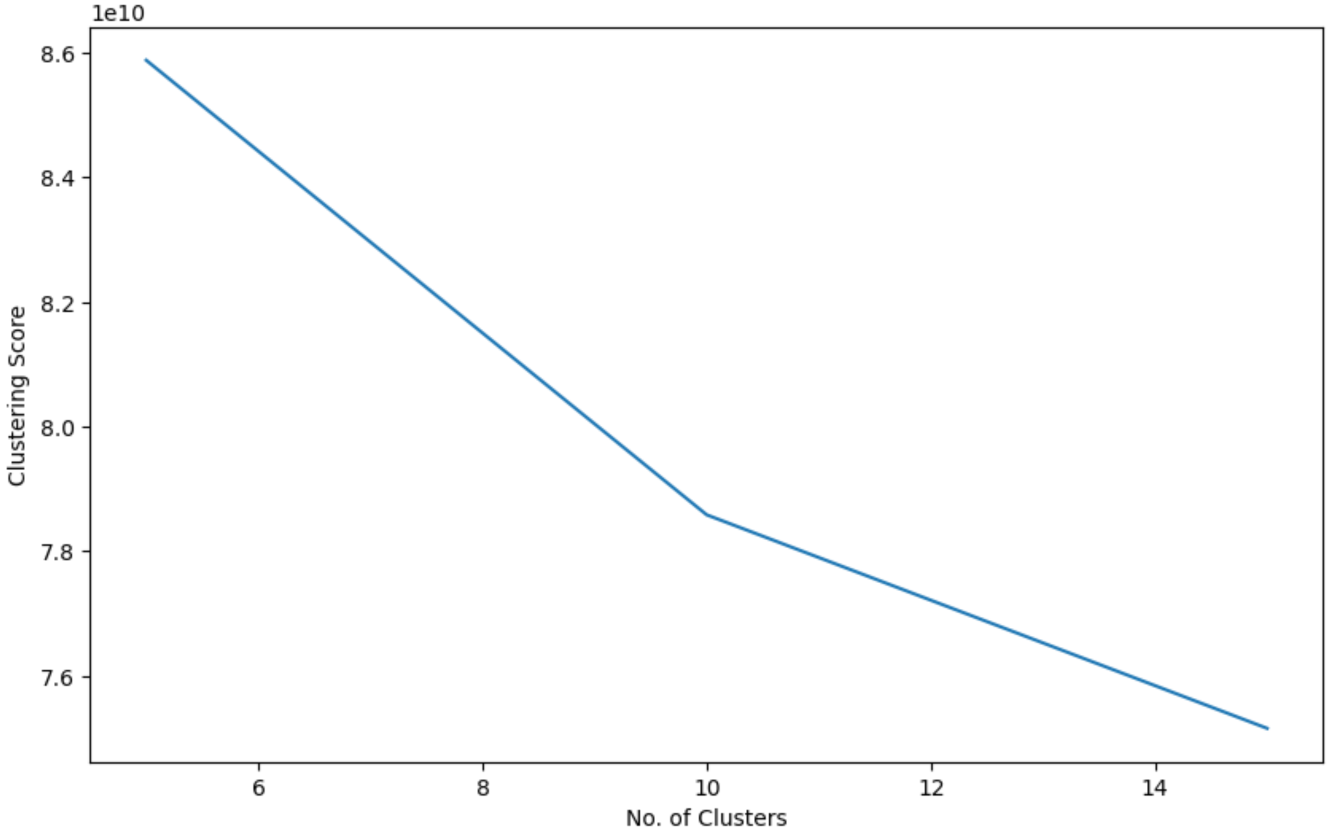



```
'''
  Implement this function to form a 10000 x N matrix
  from 10000 x 32 x 32 x 3 shape input.
'''
def reshape(X):
  '''
    Write one line of code here
  '''
  ### ========== TODO : START ========== ###
  # part (a)

  X = X.reshape(10000, -1)

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


X = reshape(X)


clustering_score = []
for i in tqdm(range(5, 20, 5)):
  score = 0
```

```
    for rs in tqdm(range(3)):
      kmeans = KMeans(n_clusters = i, init = 'random', random_state = rs)
      '''
        Write one line of code to fit the kMeans algorithm to the data
        Write another line of code to report the kMeans clustering score
        defined as sum of squared distances of samples to their closest
        cluster center, weighted by the sample weights if provided.
        Hint: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.K
      '''
      ### ========== TODO : START ========== ###
      # part (b)

      kmeans.fit(X)
      score += kmeans.inertia_

      ### ========== TODO : END ========== ###
    clustering_score.append(score/3) ## divide by 3 because 3 random states
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-package
  warnings.warn(

 33%|███▎      | 1/3 [00:58<01:57, 58.60s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

 67%|██████▋   | 2/3 [01:50<00:54, 54.84s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

100%|██████████| 3/3 [03:01<00:00, 60.60s/it]
 33%|███▎      | 1/3 [03:01<06:03, 181.82s/it]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-package
  warnings.warn(

 33%|███▎      | 1/3 [01:11<02:22, 71.12s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

 67%|██████▋   | 2/3 [02:15<01:06, 66.96s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

100%|██████████| 3/3 [03:21<00:00, 67.00s/it]
 67%|██████▋   | 2/3 [06:22<03:13, 193.11s/it]
  0%|          | 0/3 [00:00<?, ?it/s]/usr/local/lib/python3.10/dist-package
  warnings.warn(

 33%|███▎      | 1/3 [01:32<03:05, 92.81s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

 67%|██████▋   | 2/3 [02:58<01:28, 88.57s/it]/usr/local/lib/python3.10/dist
  warnings.warn(

100%|██████████| 3/3 [05:00<00:00, 100.11s/it]
100%|██████████| 3/3 [11:23<00:00, 227.73s/it]
```

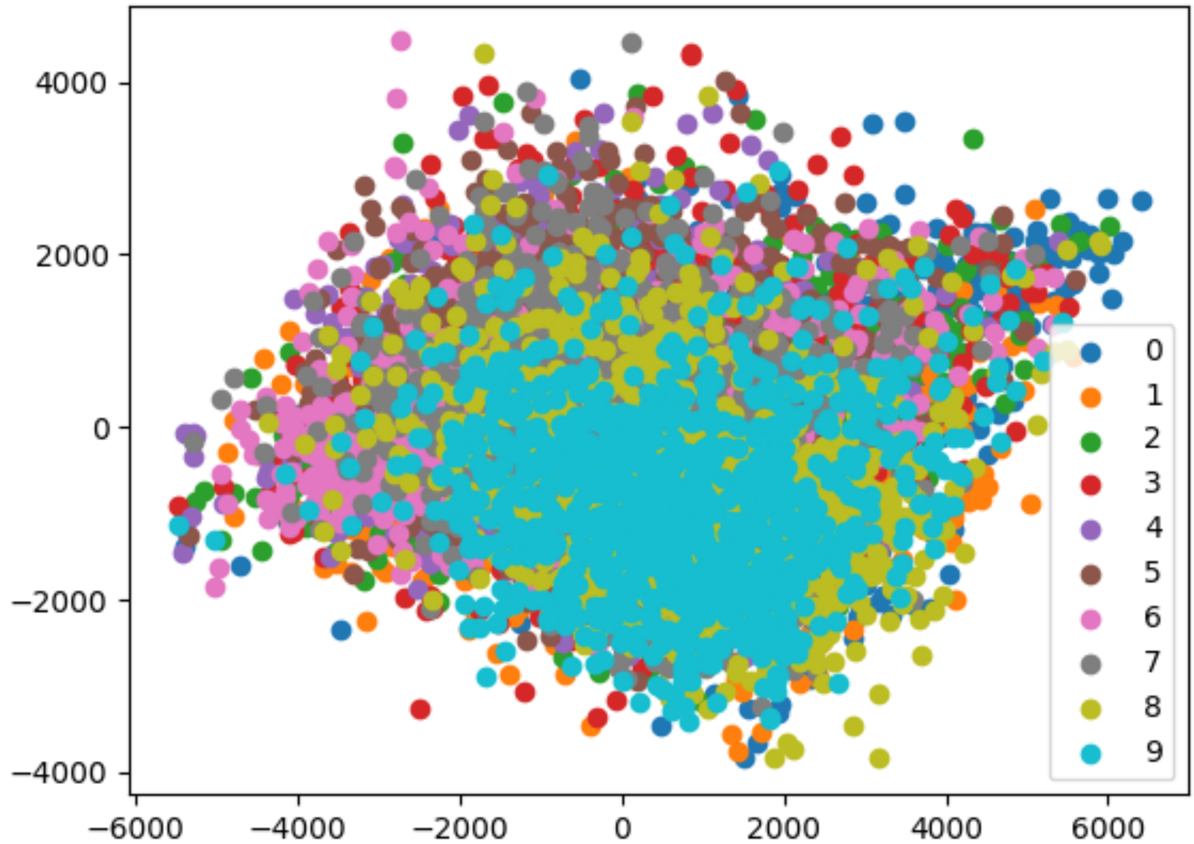
```
'''
  Submit the plot you get after running this piece of code in your solutions
'''
plt.figure(figsize=(10,6))
plt.plot(range(5, 20, 5), clustering_score)
plt.xlabel('No. of Clusters')
plt.ylabel('Clustering Score')
plt.show()
```




## Visualize K Clusters for K = 10 and random_state = 42

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


### 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()
```

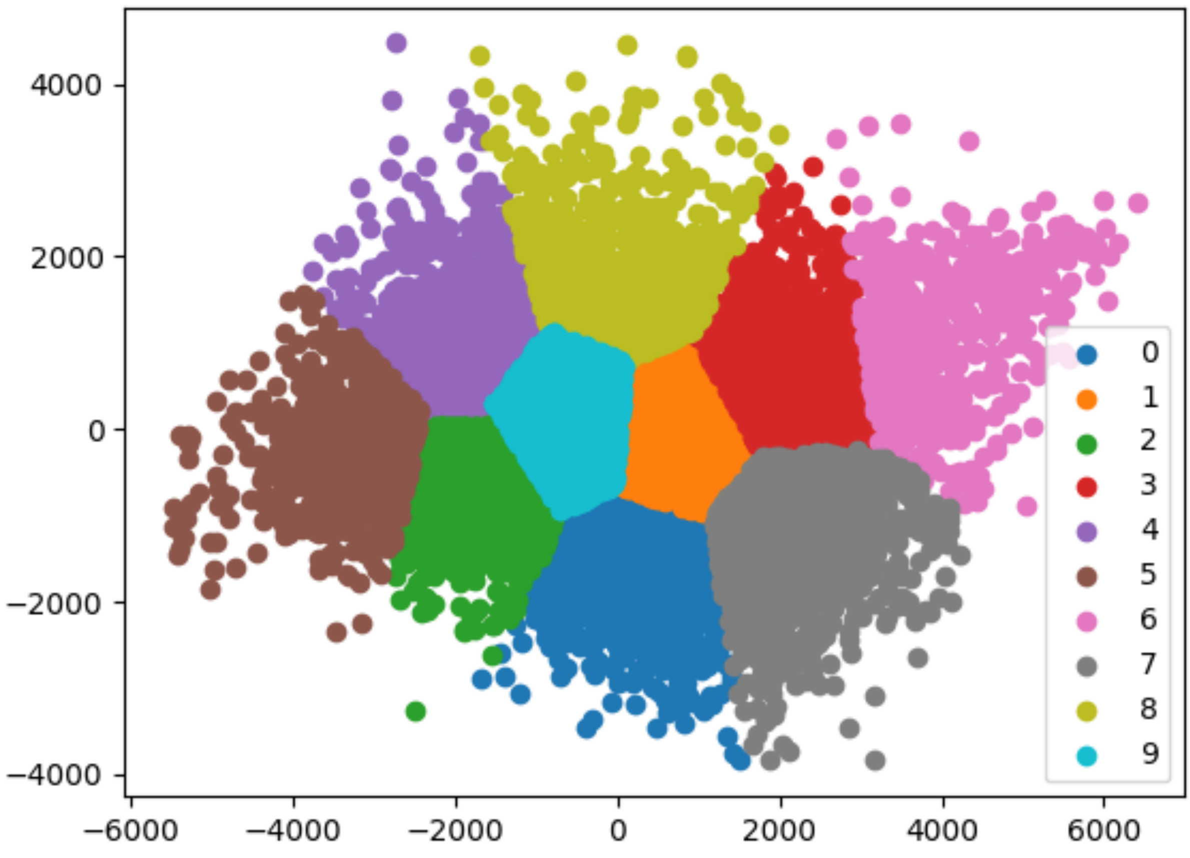
```
<matplotlib.legend.Legend at 0x7d900cbd91e0>
```




```
'''
  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(df)
labels = kmeans.labels_

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

u_labels = np.unique(labels)

#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[labels == i , 0], df[labels == 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(
```



Double-click (or enter) to edit