CO543: Image Processing
Lab 4

Ranage R.D.P.R. - E/19/310

Part 1

1. Using K-means algorithm Identify the different clusters of MNIST Handwritten Digits
a. Briefly describe the elbow method and the silhouette method
Elbow Method: The elbow approach is a technique used to ascertain the most suitable number of clusters in a dataset. The concept is founded on the notion that as the quantity of clusters rises, the dispersion within each cluster diminishes. The elbow technique graphically represents the variation (or alternative metric for assessing clustering effectiveness) in relation to the number of clusters. The inflection point on the line, where the rate of decrease abruptly changes, reveals the appropriate number of clusters. This position signifies a compromise between the decrease in variability and the rise in the quantity of clusters. The appropriate number of clusters is typically determined by identifying the point at which the rate of reduction slows down significantly, resulting in a plot that resembles the shape of a "elbow".

Silhouette Method: The silhouette approach is a methodology employed to assess the caliber of clusters generated by a clustering algorithm. Cohesion refers to the degree of similarity between an object and its own cluster, whereas separation refers to the degree of dissimilarity between an object and other clusters. The silhouette score is a numerical measure that falls within the range of -1 to 1. A higher value suggests that the object is well suited to its own cluster and not well suited to nearby clusters. A score around 0 signifies the presence of clusters that overlap. The silhouette score is computed for each individual sample and subsequently averaged to produce a comprehensive score for the clustering process. A higher silhouette score indicates superior clustering. The silhouette approach aids in identifying the most suitable number of clusters by evaluating silhouette scores for various cluster numbers. The ideal decision is determined by selecting the number of clusters that yields the highest average silhouette score.

b. Mention the criteria behind the way you define number of clusters
The principle for determining the number of clusters is the elbow approach. The elbow method is a heuristic technique employed to ascertain the most suitable number of clusters in a given dataset. The process is creating a graph that shows the number of clusters (k) plotted against a measure of clustering quality, such as the within-cluster sum of squares or inertia. The objective is to identify the point on the graph where there is a significant change in the slope, known as the "elbow" point. The elbow point signifies the point at which there is a compromise between minimizing error (or variance) and maximizing the number of clusters. The term "diminishing returns" refers to the point at which the addition of more clusters no longer leads to a meaningful improvement in the quality of clustering. Thus, the ideal value for k is frequently determined by selecting the number of clusters at the elbow point.

c. Visualize each cluster and justify the reasons for misclusted images(eg: 5 is in 8's cluster).

```
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf

#Loading datasets
df_train = pd.read_csv('train.csv')
df_test = pd.read_csv('test.csv')

df_train.head()
```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783
0	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	4	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 785 columns

```
df_test.head()
```

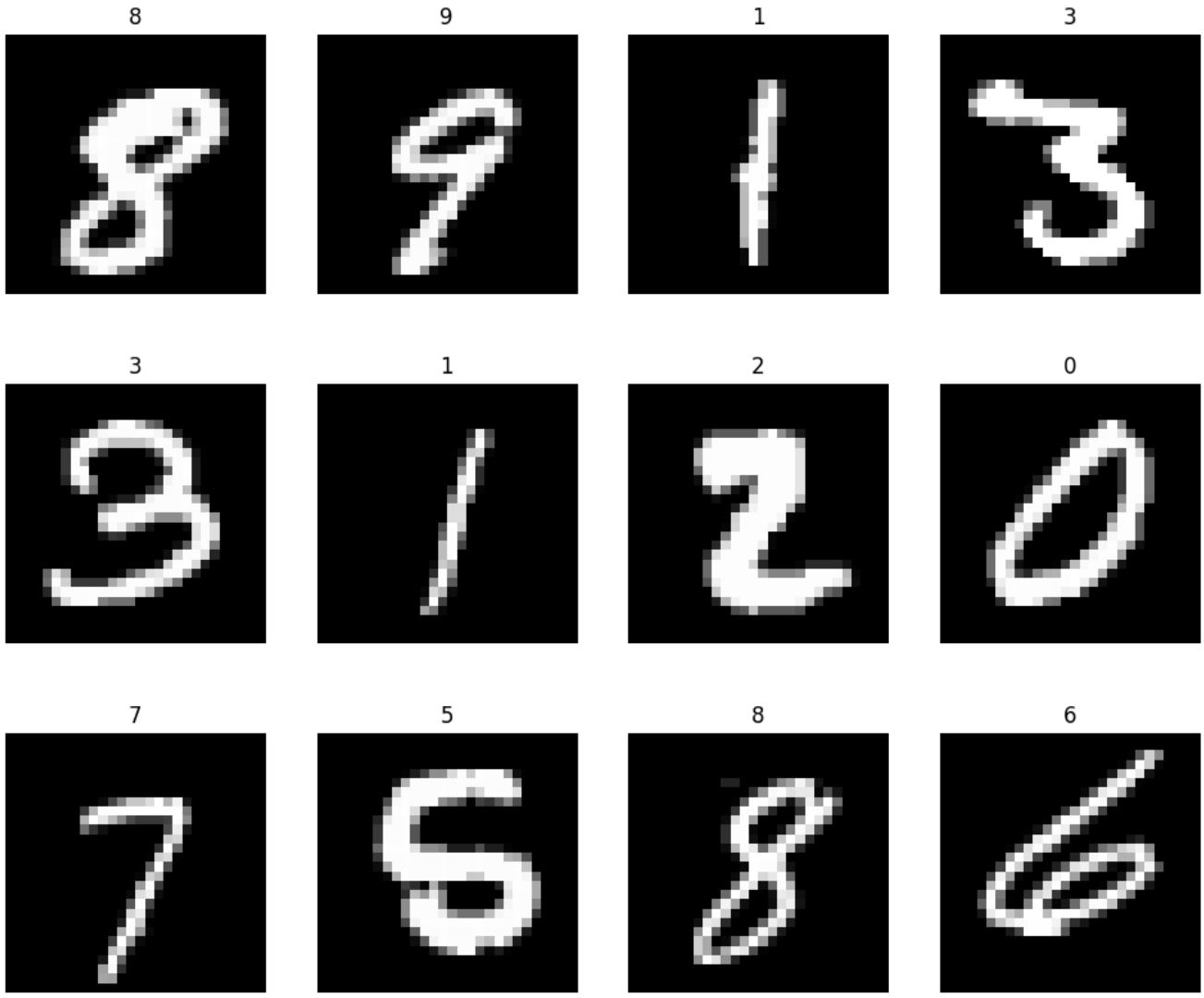
	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 784 columns

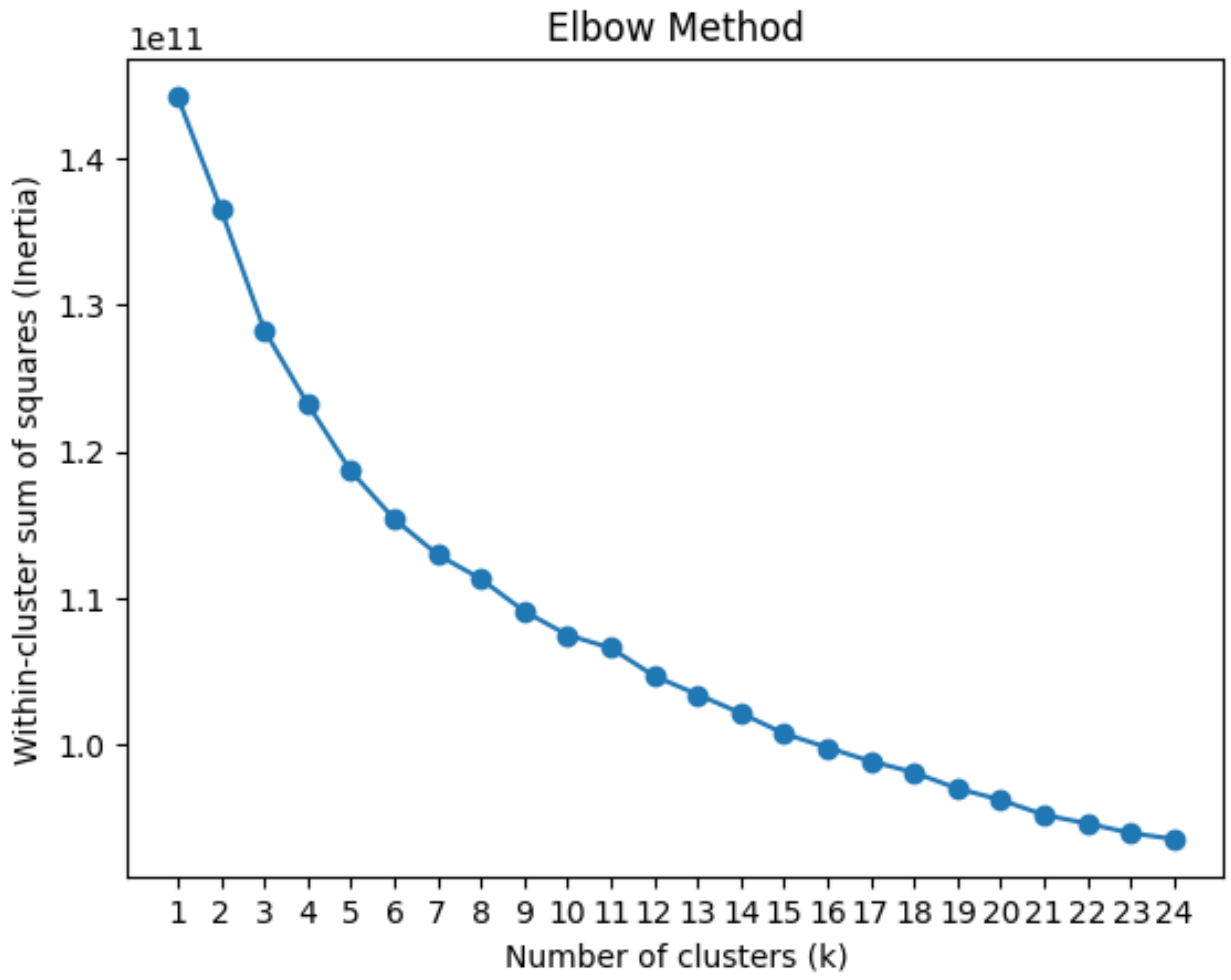
```
#since training data set has a label column, lets drop it
y_train = df_train['label']
df_train.drop(['label'],axis=1, inplace=True)

#defining the test and training dataframes
#training set
x_train = df_train.to_numpy()
y_train = y_train.to_numpy()
x_test = df_test.to_numpy()

print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
```

```
(42000, 784)
(42000,)
(28000, 784)
```

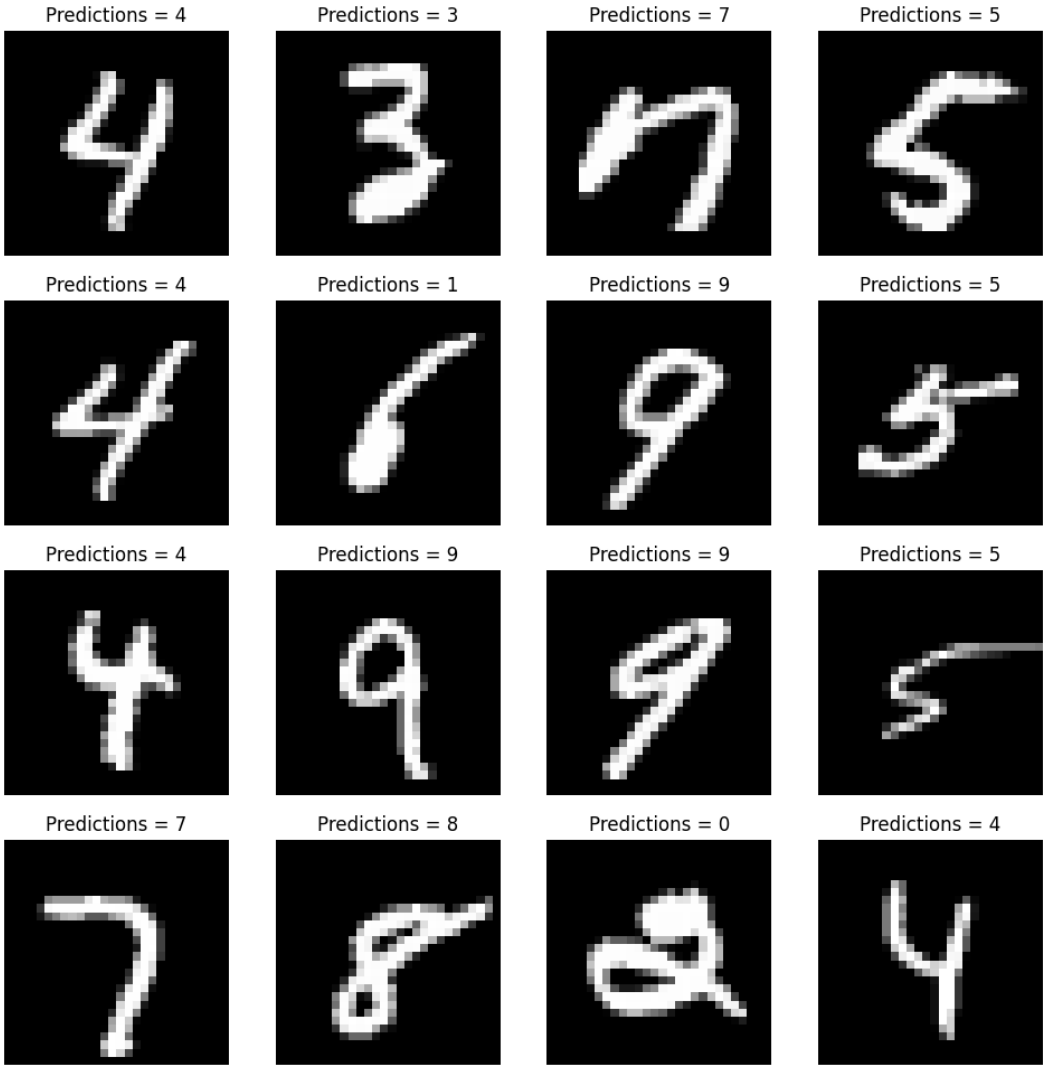
```python
#lets take a look at few images along with the label
plt.figure(figsize=(12, 10))
for i in range(12):
    plt.subplot(3, 4, i + 1)
    plt.imshow(x_train[i + 10].reshape(28, 28), cmap='gray')  # Reshape each image to
28x28
    plt.title(y_train[i + 10])  # Show the corresponding label
    plt.axis('off')  # Turn off axis
plt.show()
```



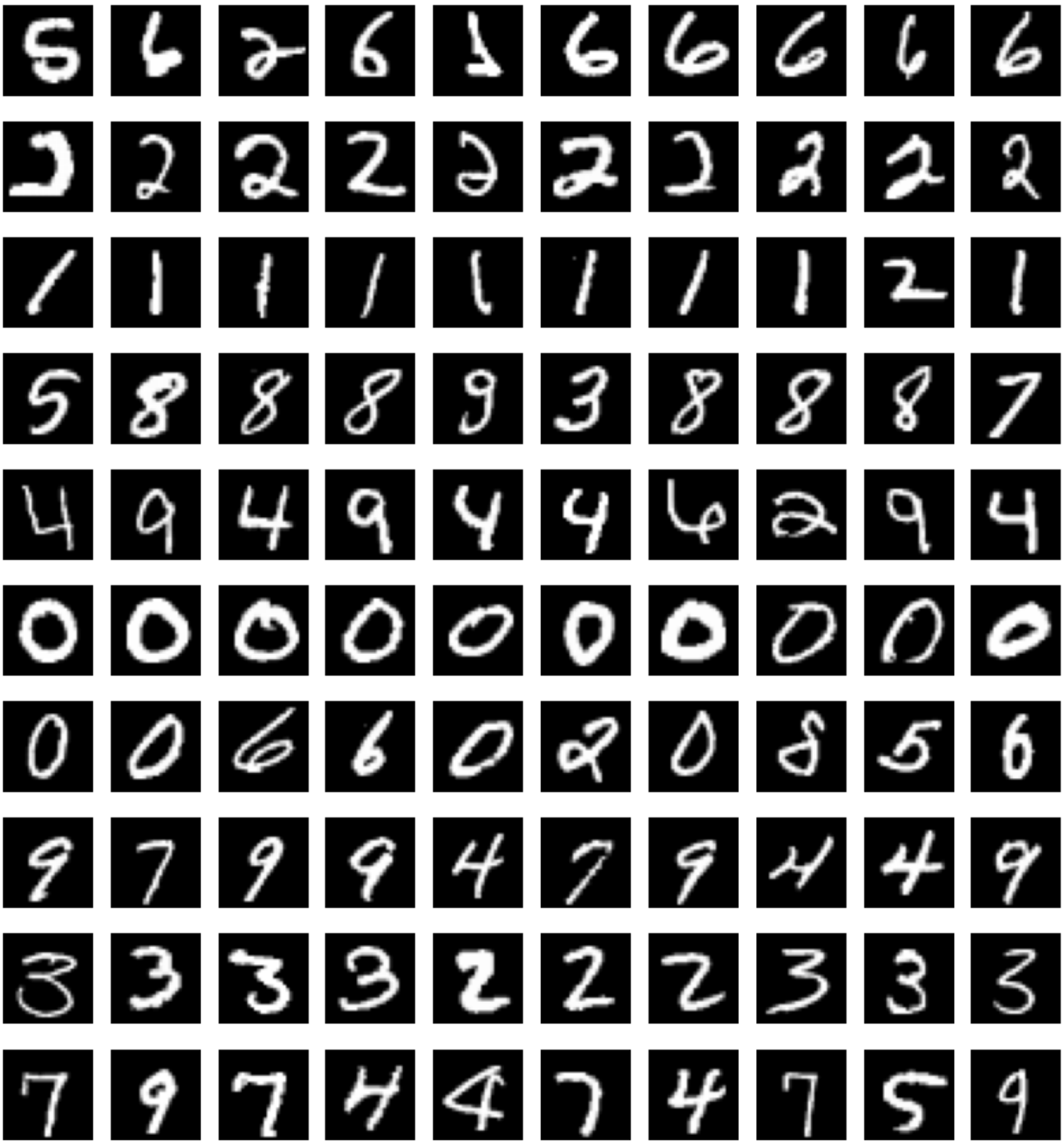
```python
from sklearn.cluster import KMeans
k_values = range(1, 25)  # Try different numbers of clusters from 1 to 10

inertia_values = []

# Iterate over each value of k
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=23)
    kmeans.fit(x_train)
    inertia_values.append(kmeans.inertia_)

plt.plot(k_values, inertia_values, marker='o')
plt.title('Elbow Method')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Within-cluster sum of squares (Inertia)')
plt.xticks(k_values)
plt.show()
```



```
from sklearn.neighbors import KNeighborsClassifier

KMeans_model = KNeighborsClassifier(n_neighbors=10)
KMeans_model.fit(x_train,y_train)

subset_x_test = x_test[1230:1246]
predictions = KMeans_model.predict(subset_x_test)
images = subset_x_test.reshape(16, 28, 28)

plt.figure(figsize=(12, 12))
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(images[i], cmap='gray')
    plt.title("Predictions = " + str(predictions[i]))
    plt.axis('off')
plt.show()
```



```
KMeans_model_2 = KMeans(n_clusters=10, random_state=0).fit(x_train)
cluster_list = {i : np.where(KMeans_model_2.labels_ == i) for i in range(15)}
for i in range(10):
  plt.figure(figsize=(10,10))
  for j in range(min(10, len(cluster_list[i][0]))):
    plt.subplot(1,10,j+1)
    plt.imshow(x_train[cluster_list[i][0][j]].reshape(28,28), cmap='gray')
    plt.axis('off')
```



Misclassifications in K-means clustering of MNIST images occur because the algorithm groups images based on pixel similarity, causing different digits with similar pixel patterns to be assigned to the same cluster. The centroid, representing an average of all images in a cluster, may not distinctly represent any single digit, leading to overlaps. Additionally, the high dimensionality of the data (784 pixels) makes distance measures less effective, resulting in further overlaps. For example, a faintly written '3' is classified as an '8' here becuase their pixel patterns are similar.

d. Suggest the ways to reduce the cluster errors.

In order to mitigate cluster errors in K-means clustering, particularly when dealing with high-dimensional data such as MNIST, there are numerous tactics that can be employed. Utilizing dimensionality reduction methods such as PCA or autoencoders can effectively emphasize important patterns. Advanced clustering methods, such as Gaussian Mixture Models (GMM) or DBSCAN, provide increased flexibility and resilience. Thorough preprocessing, which involves normalizing and removing noise, guarantees that all features make equal contributions. Utilizing more effective initialization techniques such as K-means++ and evaluating the quality of clusters using measures like the Silhouette Score aids in identifying the most ideal clusters. Utilizing data augmentation methods for photos and employing post-processing techniques such as hierarchical clustering can enhance and improve the accuracy of the outcomes. The integration of these methodologies improves the precision of clustering and minimizes errors.

Part 02

2. Using Artificial neural network and convolutional neural network Identify the different classes of MNIST Fashion dataset.
a. Initially train a classifier using artificial neural network while treating pixels as different features

```
# improting libraries
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras import layers,models
from keras.models import Sequential
from keras.layers import Flatten, Dense

# Importing the dataset
df_train = pd.read_csv('fashion-mnist_train.csv')
df_test = pd.read_csv('fashion-mnist_test.csv')

df_train.head()
```

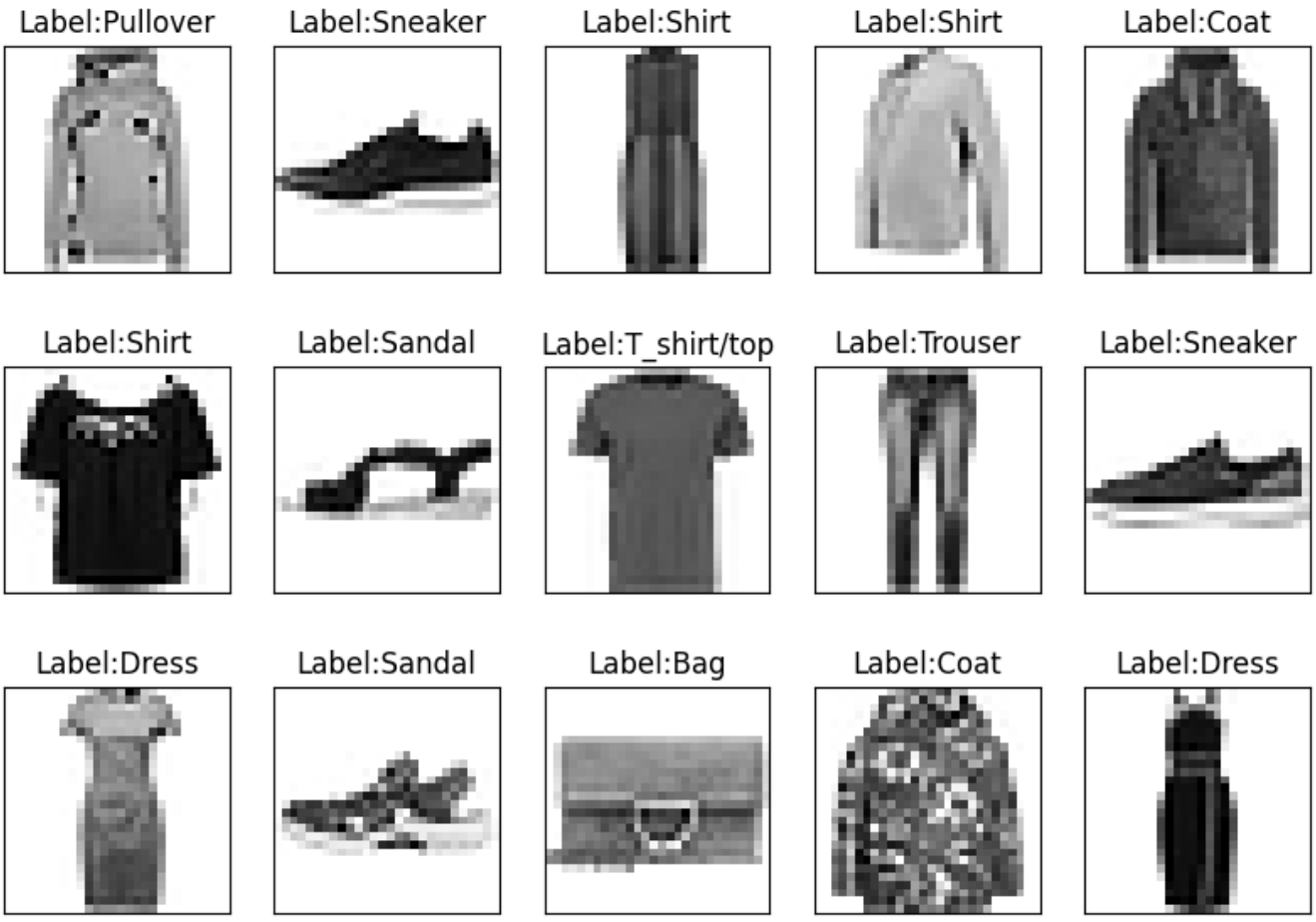
	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783	pixel784
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0	0	0
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

```
df_test.head()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782	pixel783	pixel784
0	0	0	0	0	0	0	0	0	9	8	...	103	87	56	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	...	34	0	0	0	0	0	0	0	0	0
2	2	0	0	0	0	0	0	14	53	99	...	0	0	0	0	63	53	31	0	0	0
3	2	0	0	0	0	0	0	0	0	0	...	137	126	140	0	133	224	222	56	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

```
y_train = df_train['label']
y_test = df_test['label']
df_train.drop(['label'],axis=1, inplace=True)
df_test.drop(['label'],axis=1, inplace=True)

x_train = df_train.to_numpy()
y_train = y_train.to_numpy()

x_test = df_test.to_numpy()
y_test = y_test.to_numpy()
```

```
x_train = x_train.reshape(x_train.shape[0], 28, 28).astype('float32')/255
x_test = x_test.reshape(x_test.shape[0], 28, 28).astype('float32')/255

class_names = ['T_shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

train_preview = x_train[85:100].reshape(15,28,28)
plt.figure(figsize=(10,7))
for i in range(15):
  plt.subplot(3,5,i+1)
  plt.xticks([])
  plt.yticks([])
  plt.imshow(train_preview[i],  cmap=plt.cm.binary)
  plt.title('Label:{}'.format(class_names[y_train[i+85]]))
```




```
model = Sequential()
model.add(Flatten(input_shape=((28,28))))
model.add(Dense(100,activation = "relu"))
model.add(Dense(10,activation="softmax"))
```

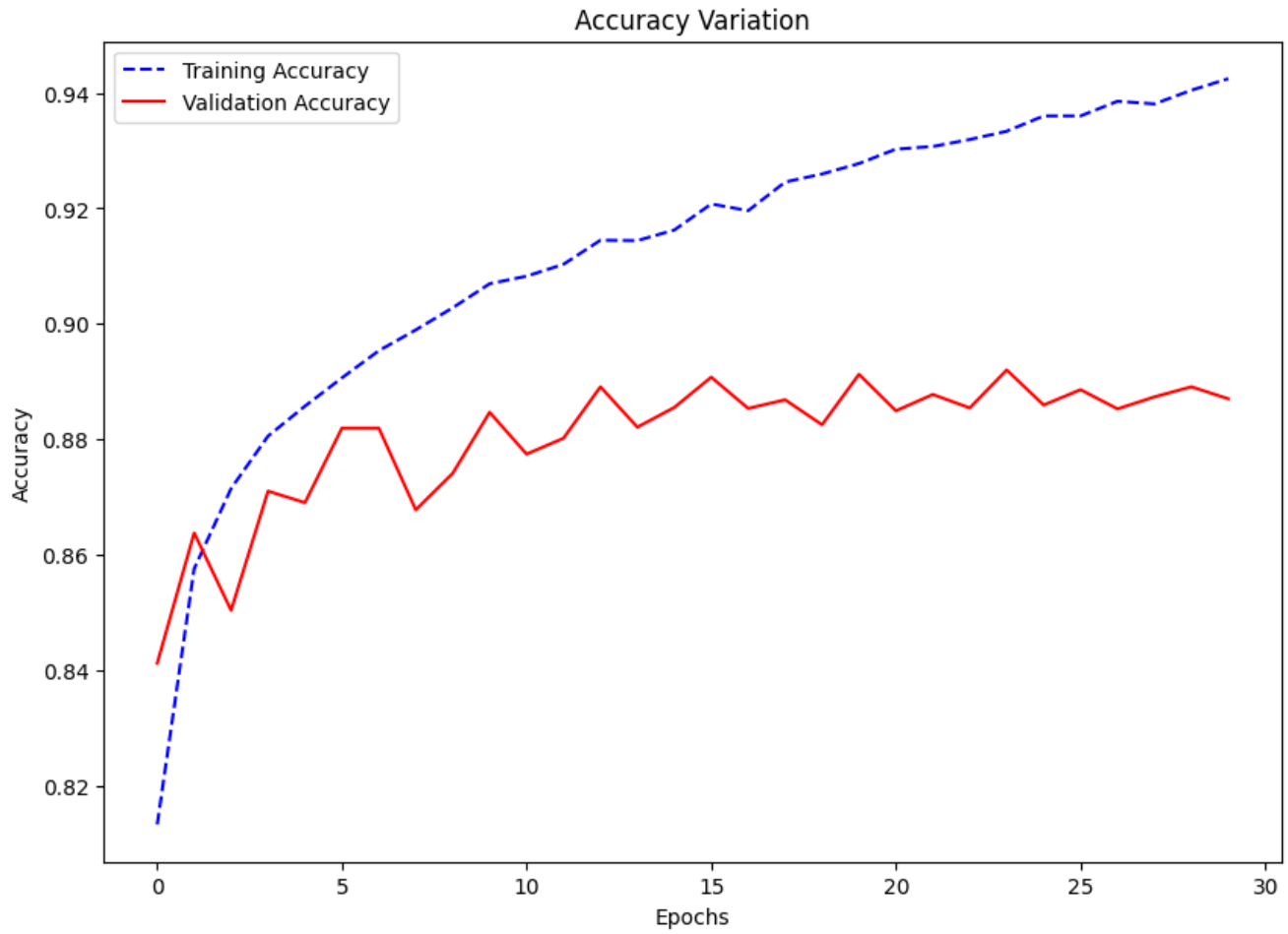
```
model.compile(optimizer =
"adam",loss="sparse_categorical_crossentropy",metrics=["accuracy"])

history = model.fit(x_train, y_train, epochs = 30, validation_split=0.2, verbose=1)
```

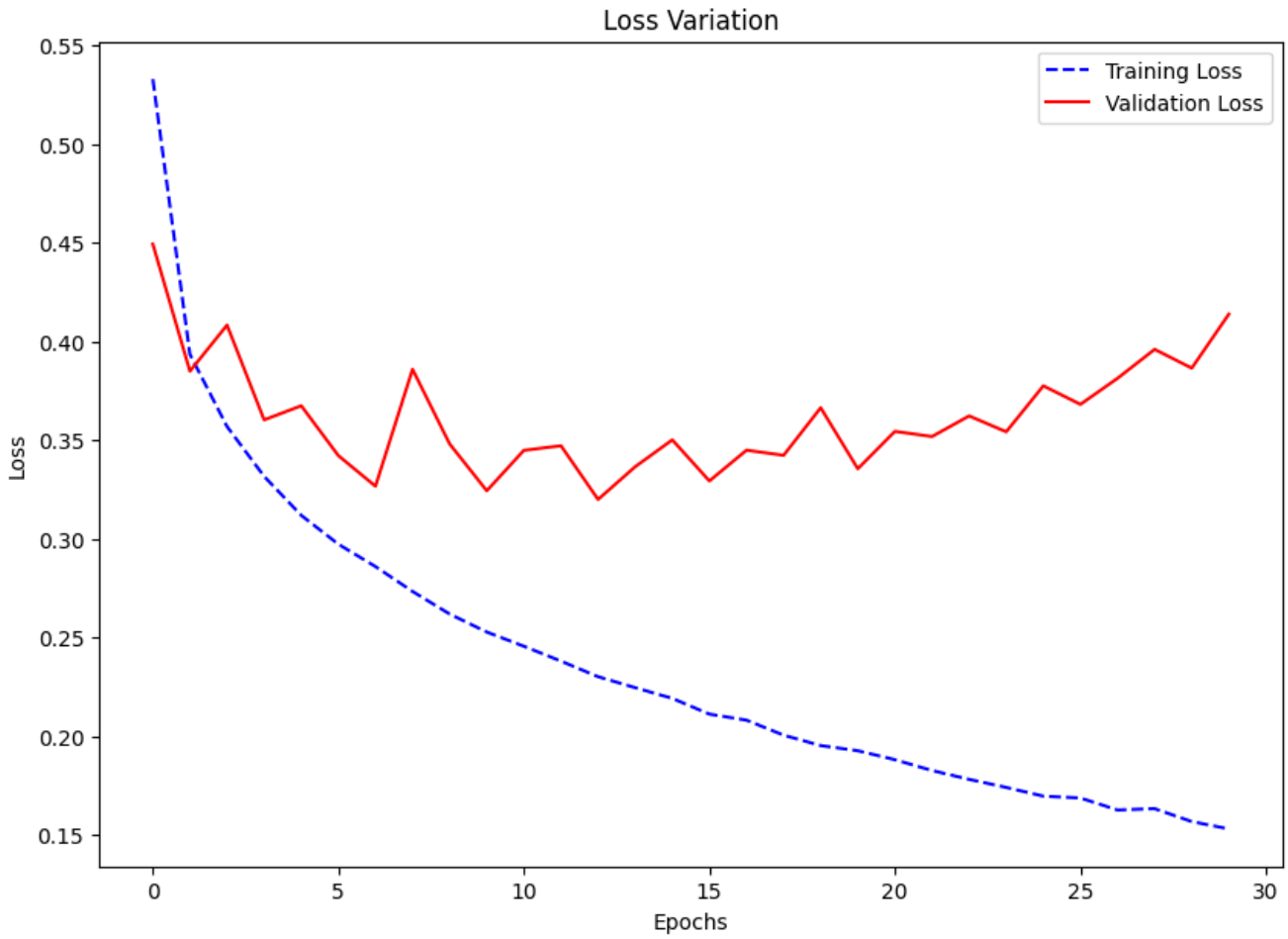
```
Epoch 1/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.7672 - loss: 0.6784 - val_accuracy: 0.8412 - val_loss: 0.4492
Epoch 2/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.8538 - loss: 0.4068 - val_accuracy: 0.8637 - val_loss: 0.3849
Epoch 3/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.8674 - loss: 0.3649 - val_accuracy: 0.8503 - val_loss: 0.4082
Epoch 4/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.8818 - loss: 0.3315 - val_accuracy: 0.8710 - val_loss: 0.3602
Epoch 5/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.8820 - loss: 0.3185 - val_accuracy: 0.8690 - val_loss: 0.3673
Epoch 6/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8869 - loss: 0.3012 - val_accuracy: 0.8819 - val_loss: 0.3421
Epoch 7/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8948 - loss: 0.2895 - val_accuracy: 0.8819 - val_loss: 0.3266
Epoch 8/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9016 - loss: 0.2685 - val_accuracy: 0.8677 - val_loss: 0.3858
Epoch 9/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9015 - loss: 0.2618 - val_accuracy: 0.8741 - val_loss: 0.3480
Epoch 10/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9094 - loss: 0.2478 - val_accuracy: 0.8847 - val_loss: 0.3243
Epoch 11/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9118 - loss: 0.2392 - val_accuracy: 0.8774 - val_loss: 0.3448
Epoch 12/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9082 - loss: 0.2374 - val_accuracy: 0.8802 - val_loss: 0.3471
Epoch 13/30
...
Epoch 29/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9439 - loss: 0.1511 - val_accuracy: 0.8891 - val_loss: 0.3864
Epoch 30/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9429 - loss: 0.1518 - val_accuracy: 0.8870 - val_loss: 0.4138
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
def eval_accuracy_loss(history):
    # Get parameters for the testing set
    accuracy_testing = history.history['accuracy']
    loss_testing = history.history['loss']
    number_of_epochs = range(len(accuracy_testing))

    # Get parameters for the validation set
    accuracy_validation = history.history['val_accuracy']
    loss_validation = history.history['val_loss']

    # Plotting the accuracy
    plt.figure(figsize=(10, 7))
    plt.plot(number_of_epochs, accuracy_testing, 'b--', label='Training Accuracy')
    plt.plot(number_of_epochs, accuracy_validation, 'r-', label='Validation
Accuracy')
    plt.title('Accuracy Variation')
    plt.xlabel('Epochs')
```

```
    plt.ylabel('Accuracy')
    plt.legend()

    # Plotting the loss
    plt.figure(figsize=(10, 7))
    plt.plot(number_of_epochs, loss_testing, 'b--', label='Training Loss')
    plt.plot(number_of_epochs, loss_validation, 'r-', label='Validation Loss')
    plt.title('Loss Variation')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

eval_accuracy_loss(history)
```



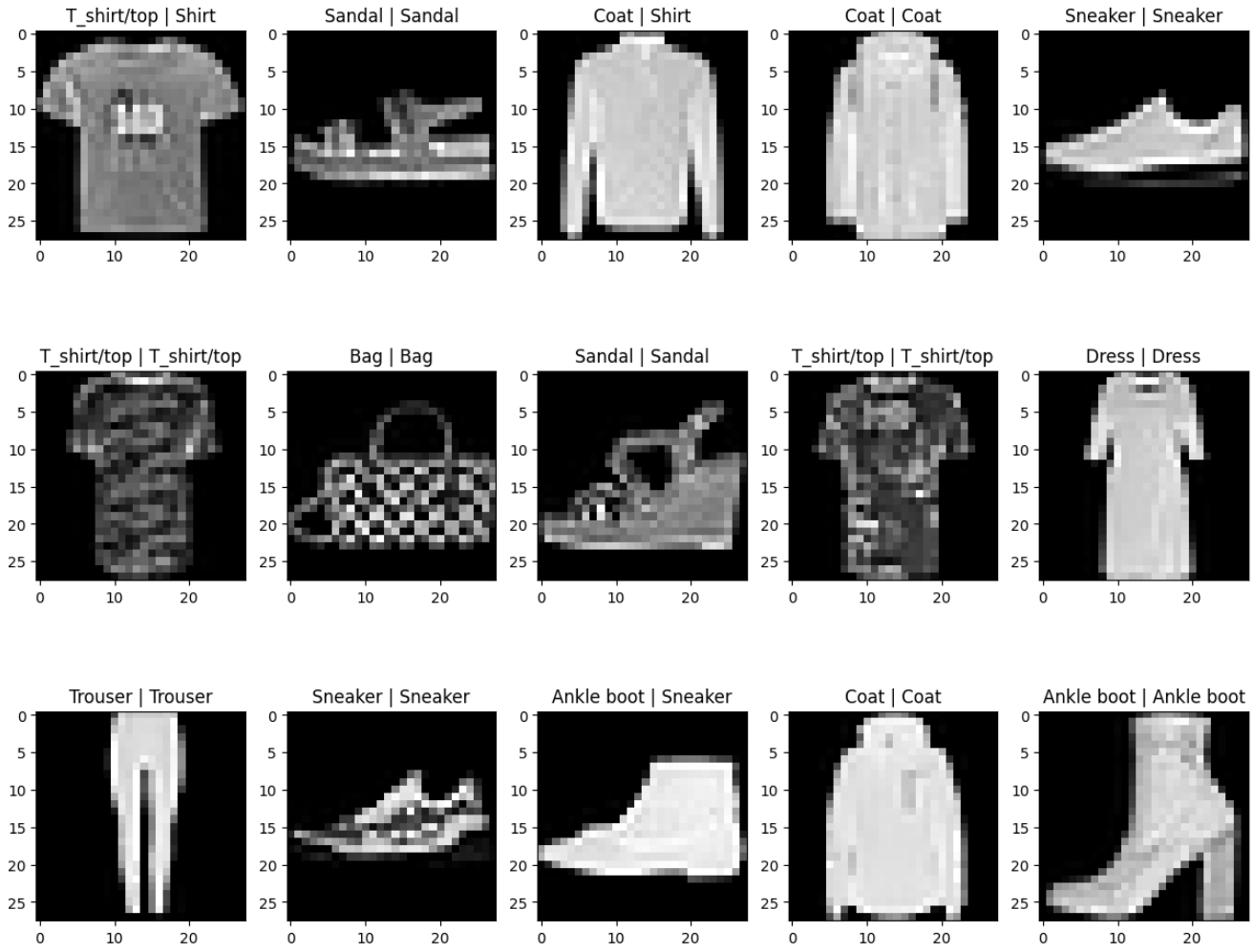



```
#test the model with test data
pred = np.argmax(model.predict(x_test[:20]), axis=1)
print(pred)
print(y_test[:20])
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step
[0 1 2 2 4 6 8 6 5 0 3 2 6 6 8 5 6 3 6 4]
[0 1 2 2 3 2 8 6 5 0 3 4 4 6 8 5 6 3 6 4]
```

```
def test_model(model):
    pred = model.predict(x_test[120:135])

    pred = np.argmax(pred, axis=1)

    plt.figure(figsize=(15, 12))
    for i in range(15):
        plt.subplot(3, 5, i + 1)
        plt.imshow(x_test[i+120], cmap="gray")
        plt.title(class_names[y_test[i+120]] + " | " + class_names[pred[i]])
```

```
test_model(model)
```



b. Train a Convolutional neural network(CNN) for the above data set considering data points as images.

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1),
padding='valid', activation='relu', input_shape=(28, 28, 1)))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(tf.keras.layers.Dropout(rate=0.25))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=128, activation='relu'))
model.add(tf.keras.layers.Dense(units=10, activation='softmax'))
model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy,
optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])

model.summary()
```

Model: "sequential_6"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
dropout_1 (Dropout)	(None, 13, 13, 32)	0
flatten_6 (Flatten)	(None, 5408)	0
dense_12 (Dense)	(None, 128)	692,352
dense_13 (Dense)	(None, 10)	1,290

Total params: 693,962 (2.65 MB)

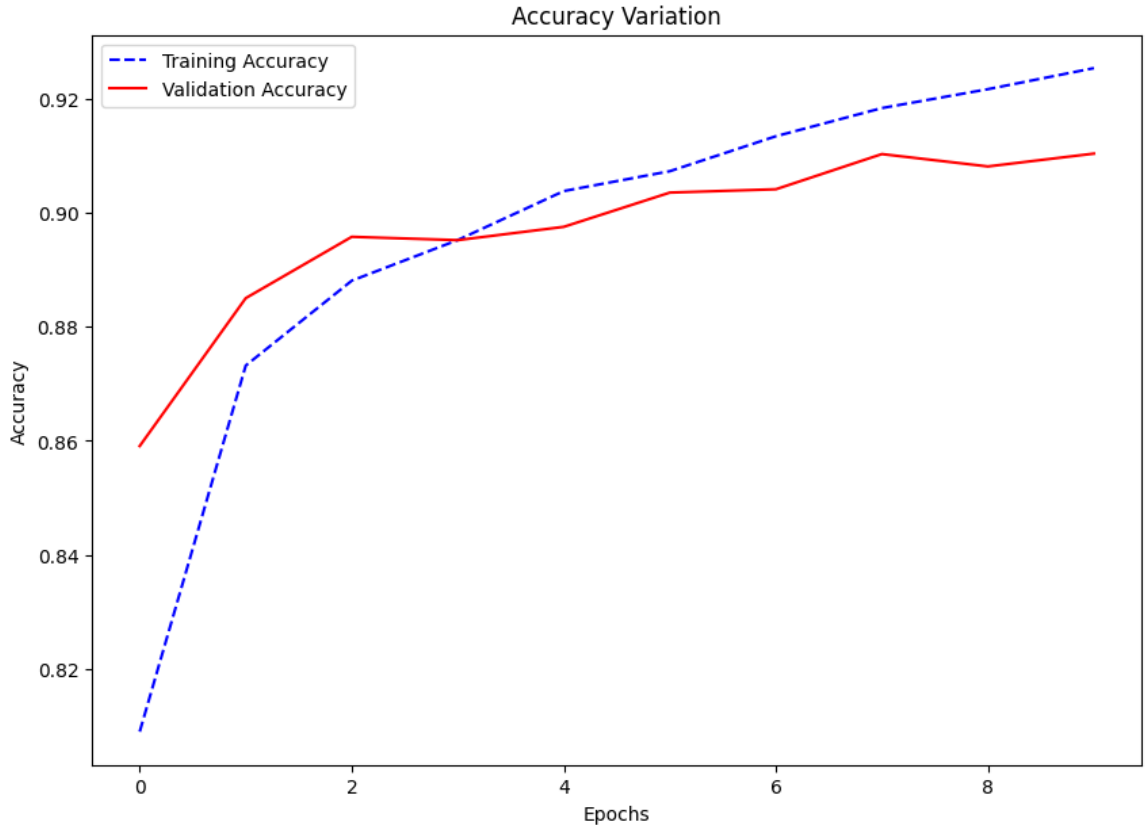
Trainable params: 693,962 (2.65 MB)

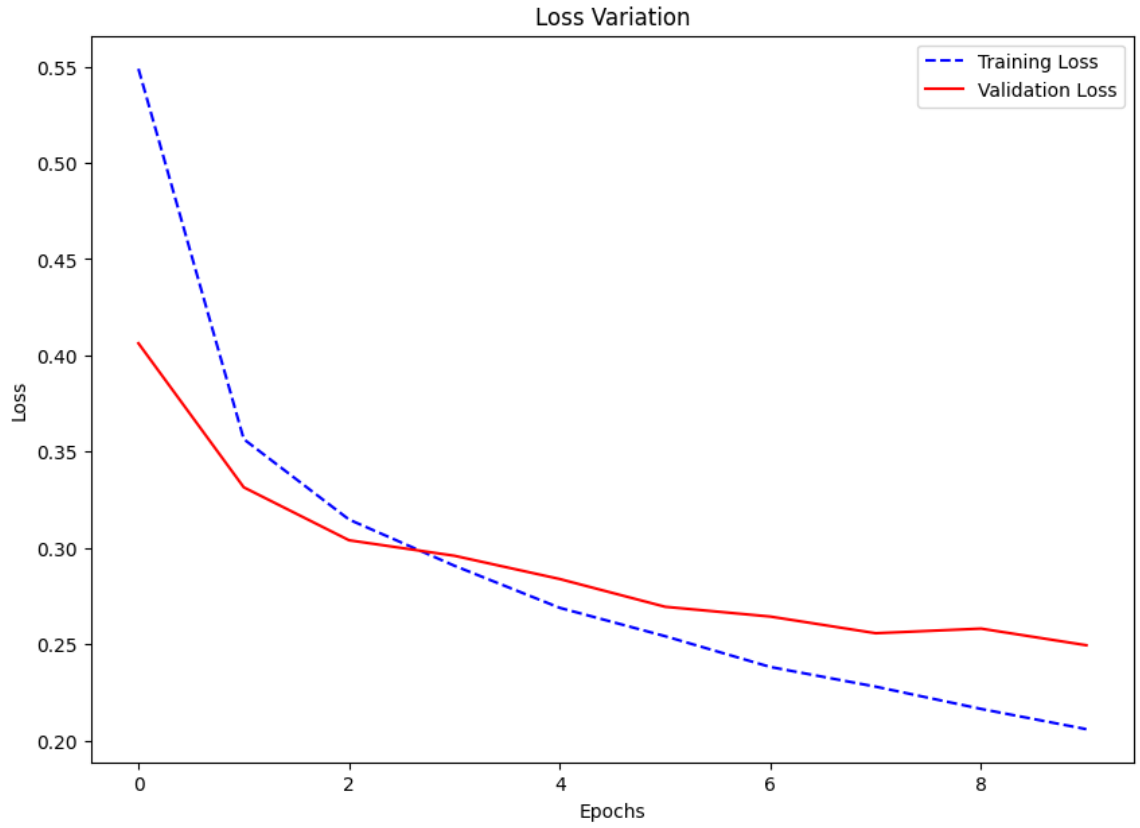
Non-trainable params: 0 (0.00 B)

```
history_cnn = model.fit(x_train, y_train,batch_size=256,epochs=10,
validation_split=0.2,verbose=1)
```

```
Epoch 1/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 74ms/step - accuracy: 0.7295 - loss: 0.7856 - val_accuracy: 0.8591 - val_loss: 0.4063
Epoch 2/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 13s 71ms/step - accuracy: 0.8649 - loss: 0.3777 - val_accuracy: 0.8850 - val_loss: 0.3315
Epoch 3/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 14s 74ms/step - accuracy: 0.8876 - loss: 0.3139 - val_accuracy: 0.8957 - val_loss: 0.3040
Epoch 4/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 77ms/step - accuracy: 0.8953 - loss: 0.2918 - val_accuracy: 0.8952 - val_loss: 0.2959
Epoch 5/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 82ms/step - accuracy: 0.9045 - loss: 0.2685 - val_accuracy: 0.8975 - val_loss: 0.2839
Epoch 6/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 78ms/step - accuracy: 0.9065 - loss: 0.2524 - val_accuracy: 0.9035 - val_loss: 0.2695
Epoch 7/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 78ms/step - accuracy: 0.9129 - loss: 0.2409 - val_accuracy: 0.9041 - val_loss: 0.2644
Epoch 8/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 78ms/step - accuracy: 0.9168 - loss: 0.2330 - val_accuracy: 0.9103 - val_loss: 0.2557
Epoch 9/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 77ms/step - accuracy: 0.9217 - loss: 0.2173 - val_accuracy: 0.9081 - val_loss: 0.2581
Epoch 10/10
188/188 ━━━━━━━━━━━━━━━━━━━━ 15s 80ms/step - accuracy: 0.9252 - loss: 0.2047 - val_accuracy: 0.9103 - val_loss: 0.2495
```

```
eval_accuracy_loss(history_cnn)
```

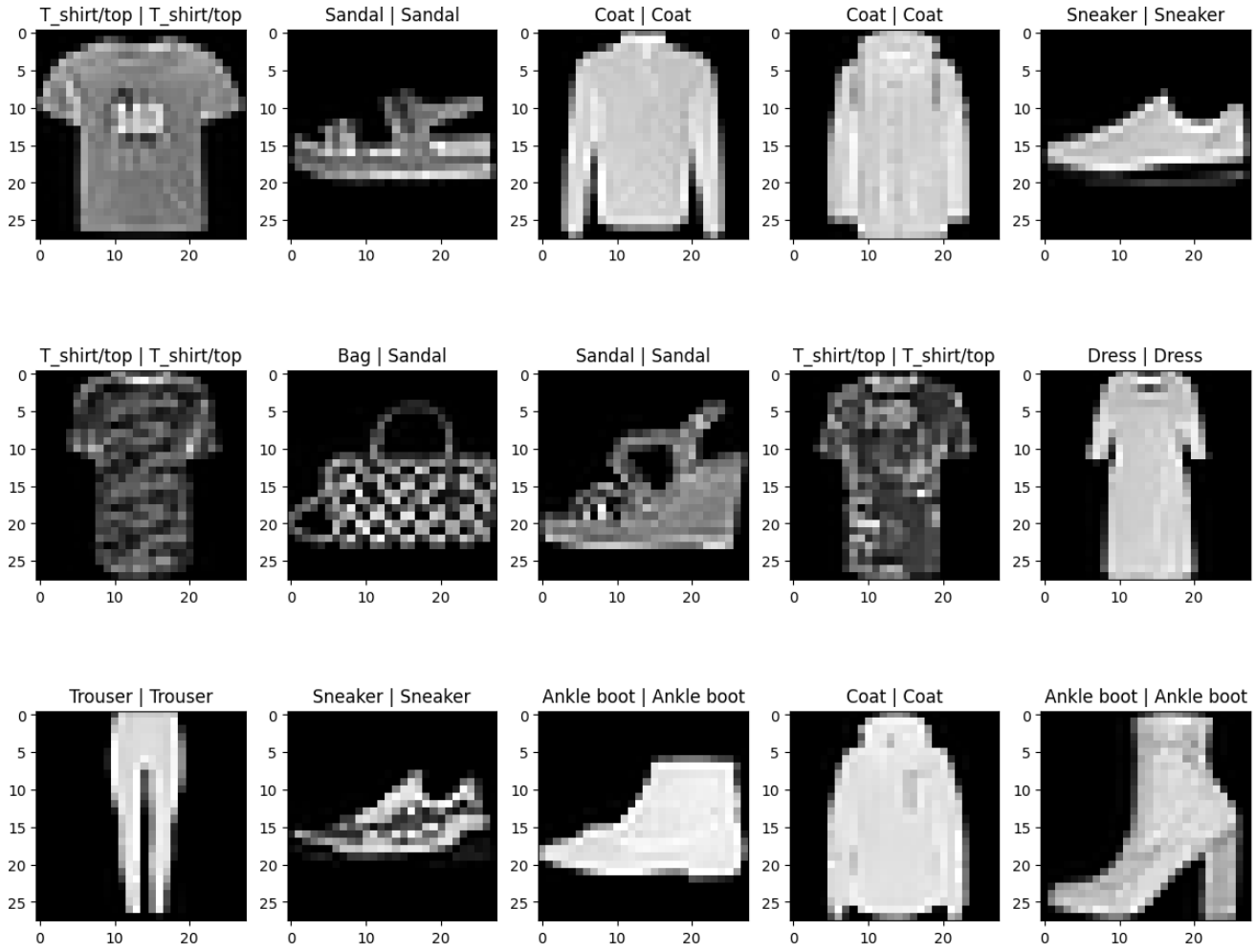




```
#test the model with test data
pred = np.argmax(model.predict(x_test[:20]), axis=1)
print(pred)
print(y_test[:20])
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step
[0 1 2 2 3 6 8 6 5 0 3 2 4 6 8 5 6 3 6 4]
[0 1 2 2 3 2 8 6 5 0 3 4 4 6 8 5 6 3 6 4]
```

```
test_model(model)
```



c. Identify the difference between above 2 models

Upon analysing the accuracy and loss curves of the ANN model, it becomes evident that the model starts to overfit the data during the training process after completing 3 epochs. This may be observed by analysing the validation and training curves after epoch=3, where it becomes evident that the disparity between them is progressively growing.

However, the CNN model does not follow this pattern. Since the accuracy and loss curves of both the training and validation datasets are consistently aligned and the discrepancy is not growing, it can be concluded that... Hence, the CNN model exhibits significantly higher accuracy in comparison to the ANN model.

f. Discuss having more or less nodes in a single layer and having a deep or a shallow network against the computational complexity.

Artificial Neural Networks (ANNs) are structured with an input layer, one or more hidden layers, and an output layer, each comprising neurons with associated weights and thresholds. Neurons activate when their output exceeds the threshold, transmitting data to subsequent layers. Adding layers increases neuron count and model complexity, potentially causing overfitting as some neurons fail to meet activation thresholds, leading to their removal. Dropout layers after hidden layers mitigate overfitting. ANNs may not suit large datasets due to explicit data point image detailing, demanding significant processing time. Contrastingly, Convolutional Neural Networks (CNNs) automatically extract image features, making them ideal for larger datasets. ANNs, favoring 1-dimensional vectors, exponentially increase parameters during training, raising memory and time complexities. Given CNNs' minimal human intervention and efficiency, they excel across various domains.

g. Discuss about the way you defined the optimum neural network architecture for the above problem.

An optimal strategy is to employ an equal number of nodes in each concealed layer.
Nevertheless, it is crucial to take into account the quantity of layers and nodes, as an excessive number of nodes can result in overfitting (when the model becomes excessively tailored to the training data), while a reduced number of nodes can lead to underfitting (when the model is too simplistic to capture patterns). Networks with a greater number of hidden layers (more than 2-3) can result in inaccurate models because to overfitting and a significant increase in computational complexity.