Jayawardena H.D.N.S.
E/19/167

1. MINST handwritten digits dataset.

- Import libraries

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import seaborn as sns
from tensorflow.keras.utils import to_categorical
```

- Mount the google drive and import datasets

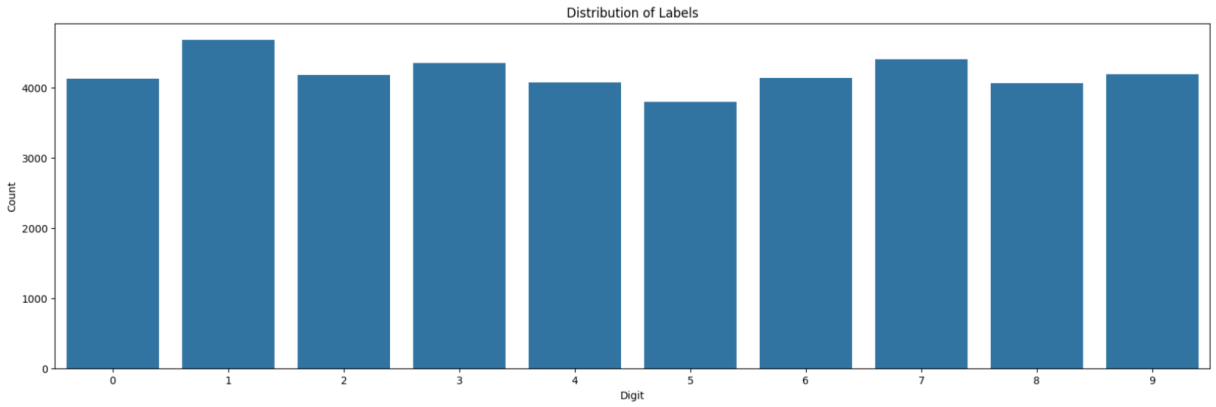
```
from google.colab import drive
drive.mount('/content/drive/')
```

```
train_set= pd.read_csv('/content/drive/MyDrive/IP/digit_recognizer/train.csv')
test_set = pd.read_csv('/content/drive/MyDrive/IP/digit_recognizer/test.csv')
```

- Histogram of data distribution

```
# # Convert labels to one-hot encoding
y_train = to_categorical(y_train, num_classes=10)
y_val = to_categorical(y_val, num_classes=10)


# # Visualize the distribution of the labels
plt.figure(figsize=(20, 6))
# # count plot on single categorical variable
sns.countplot(x ='label', data = train_set)
plt.title('Distribution of Labels')
plt.xlabel('Digit')
plt.ylabel('Count')
plt.show()
```

Output

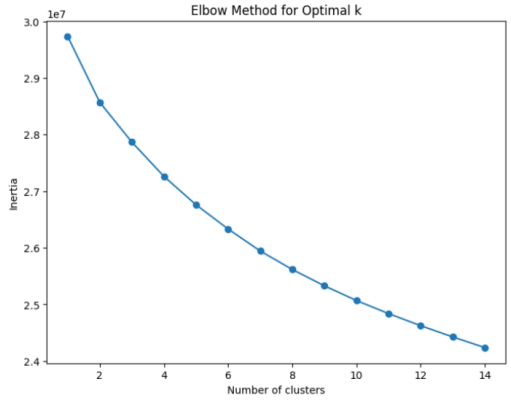


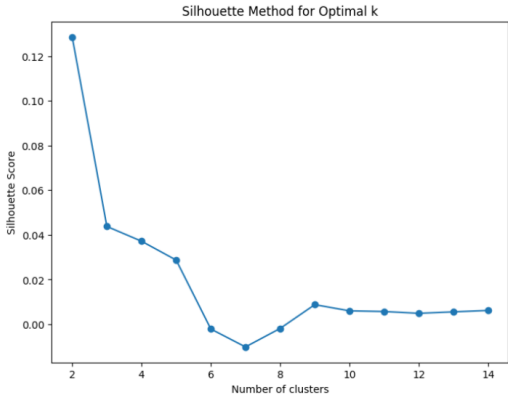
- K means clustering algorithm

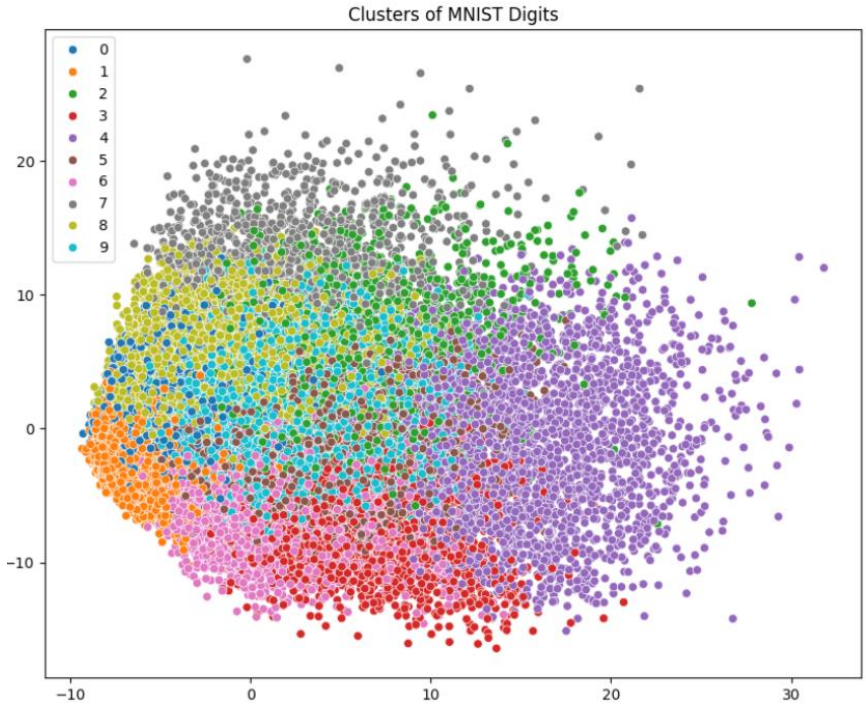
```
# Find the optimal number of clusters using the Elbow method
inertia = []
for k in range(1, 15):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    inertia.append(kmeans.inertia_)

# Plotting the Elbow curve
plt.figure(figsize=(8, 6))
plt.plot(range(1, 15), inertia, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.title('Elbow Method for Optimal k')
plt.show()

# Silhouette method
silhouette_scores = []
for k in range(2, 15):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    score = silhouette_score(X_scaled, kmeans.labels_)
    silhouette_scores.append(score)

# Plotting the Silhouette scores
plt.figure(figsize=(8, 6))
plt.plot(range(2, 15), silhouette_scores, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.title('Silhouette Method for Optimal k')
plt.show()

# Based on the above plots, choose an appropriate number of clusters
optimal_k = 10  # for example

# Apply K-means with the chosen number of clusters
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
clusters = kmeans.fit_predict(X_scaled)
```

```
# Visualize the clusters using PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

plt.figure(figsize=(10, 8))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=clusters, palette='tab10', legend='full')
plt.title('Clusters of MNIST Digits')
plt.show()
```

Output







a.

Elbow Method: The elbow method involves plotting the sum of squared distances from each point to its assigned cluster center (inertia) as a function of the number of clusters. As the number of clusters increases, the inertia decreases. The "elbow point," where the rate of decrease sharply slows down, suggests a suitable number of clusters.

Silhouette Method: The silhouette method measures how similar a point is to its own cluster compared to other clusters. The silhouette score ranges from -1 to 1: A score close to 1 indicates that the point is well matched to its own cluster and poorly matched to neighboring clusters.A score near 0 means the point is on or very close to the decision boundary between two neighboring clusters. A negative score indicates that the point might have been assigned to the wrong cluster.

b. The number of clusters is chosen based on the elbow point in the elbow method or the maximum average silhouette score. Both methods provide insights into the appropriate number of clusters for the dataset.

c.

- **Ambiguous Images**

  Some digits are inherently ambiguous and can be easily confused with others. For instance:

  - 5 and 8: The digit '5' might sometimes look like '8' if the loop of '5' is closed.
  - 1 and 7: A '1' with a serif or a slanted '1' might resemble '7'.
  - 4 and 9**:** Some styles of writing '4' might look like '9'.

- **Poor Image Quality**
  - Noise: Images with noise or poor quality can make it harder for the model to correctly identify the digit.
  - Blurry Images: Blurred images might lose essential details, leading to incorrect classification.
  - Handwriting Variations: Variability in handwriting styles can cause the model to misinterpret digits.
- **Model Limitations**
  - Insufficient Training Data: If the model hasn't seen enough examples of certain styles of digits during training, it might not generalize well to new, unseen styles.
  - Model Complexity: A too simple model might not capture all the nuances of the digit shapes, while an overly complex model might overfit to the training data and fail to generalize.
  - Feature Representation: The features extracted from the images might not be sufficient to differentiate between similar digits.
- **Suboptimal Hyperparameters**
  - Learning Rate: A learning rate that is too high might cause the model to converge to a suboptimal solution, while a too-low learning rate might make the training process too slow and might not reach the optimal solution.
  - Number of Clusters (for K-means): If the number of clusters kk is not chosen properly, digits might be grouped incorrectly.
- **Mislabeling in Training Data**
  - Incorrect Labels: Errors in the training data labels can confuse the model during training, leading to incorrect classifications.
- **Overfitting**
  - Training vs. Validation Performance: A model that performs very well on the training data but poorly on validation data is likely overfitting. This means it is not generalizing well to new, unseen data.

d. Ways to reduce cluster errors:
   - Feature Engineering: Improve the quality of features by including more informative ones or by using techniques like Gabor filters for texture analysis.
   - Dimensionality Reduction: Apply advanced dimensionality reduction techniques such as t-SNE before clustering.

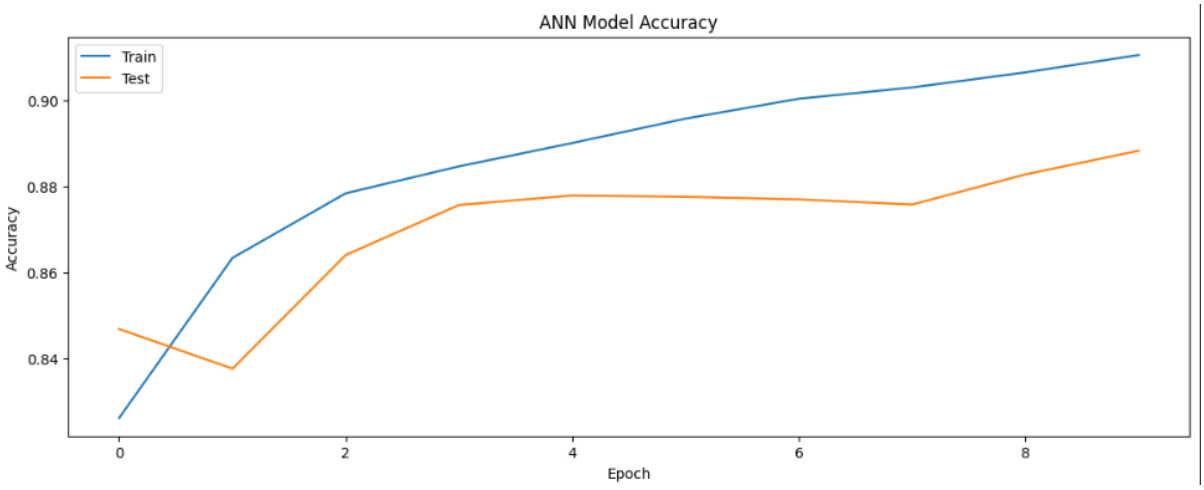
- Advanced Clustering Algorithms: Use clustering algorithms like DBSCAN or Gaussian Mixture Models that might handle the variability in handwritten digits better.
- Semi-supervised Learning: Incorporate some label information to guide the clustering process.
- Post-processing: Apply refinement techniques like spectral clustering on the initial K-means clusters to improve separation.

2. MINST fashion dataset
    a. ANN model
    Code

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Load the dataset
fashion_mnist = tf.keras.datasets.fashion_mnist
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()

# Preprocess the data
X_train = X_train / 255.0
X_test = X_test / 255.0

# One-hot encoding the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Split the training data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# Define the ANN model
ann_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(128, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
ann_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history_ann = ann_model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20, batch_size=32)
```

Output

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
Epoch 1/20
1500/1500 [==============================] - 11s 6ms/step - loss: 0.5151 - accuracy: 0.8159 - val_loss: 0.4055 - val_accuracy: 0.8533
Epoch 2/20
1500/1500 [==============================] - 13s 9ms/step - loss: 0.3799 - accuracy: 0.8609 - val_loss: 0.3839 - val_accuracy: 0.8594
Epoch 3/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.3433 - accuracy: 0.8727 - val_loss: 0.3473 - val_accuracy: 0.8758
Epoch 4/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.3164 - accuracy: 0.8831 - val_loss: 0.3617 - val_accuracy: 0.8708
Epoch 5/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.2966 - accuracy: 0.8901 - val_loss: 0.3370 - val_accuracy: 0.8754
Epoch 6/20
1500/1500 [==============================] - 7s 5ms/step - loss: 0.2822 - accuracy: 0.8941 - val_loss: 0.3359 - val_accuracy: 0.8792
Epoch 7/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.2720 - accuracy: 0.8975 - val_loss: 0.3180 - val_accuracy: 0.8873
Epoch 8/20
1500/1500 [==============================] - 7s 5ms/step - loss: 0.2561 - accuracy: 0.9042 - val_loss: 0.3162 - val_accuracy: 0.8873
Epoch 9/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.2491 - accuracy: 0.9059 - val_loss: 0.3259 - val_accuracy: 0.8863
Epoch 10/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.2391 - accuracy: 0.9095 - val_loss: 0.3547 - val_accuracy: 0.8792
Epoch 11/20
1500/1500 [==============================] - 10s 7ms/step - loss: 0.2325 - accuracy: 0.9115 - val_loss: 0.3114 - val_accuracy: 0.8907
Epoch 12/20
1500/1500 [==============================] - 6s 4ms/step - loss: 0.2237 - accuracy: 0.9160 - val_loss: 0.3280 - val_accuracy: 0.8882
Epoch 13/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.2158 - accuracy: 0.9180 - val_loss: 0.3346 - val_accuracy: 0.8859
Epoch 14/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.2098 - accuracy: 0.9200 - val_loss: 0.3345 - val_accuracy: 0.8887
Epoch 15/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.2027 - accuracy: 0.9226 - val_loss: 0.3261 - val_accuracy: 0.8875
Epoch 16/20
1500/1500 [==============================] - 7s 5ms/step - loss: 0.1964 - accuracy: 0.9251 - val_loss: 0.3297 - val_accuracy: 0.8928
Epoch 17/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1905 - accuracy: 0.9282 - val_loss: 0.3277 - val_accuracy: 0.8951
Epoch 18/20
1500/1500 [==============================] - 7s 5ms/step - loss: 0.1874 - accuracy: 0.9288 - val_loss: 0.3398 - val_accuracy: 0.8906
Epoch 19/20
1500/1500 [==============================] - 9s 6ms/step - loss: 0.1804 - accuracy: 0.9315 - val_loss: 0.3294 - val_accuracy: 0.8931
Epoch 20/20
1500/1500 [==============================] - 8s 5ms/step - loss: 0.1761 - accuracy: 0.9339 - val_loss: 0.3429 - val_accuracy: 0.8903
```



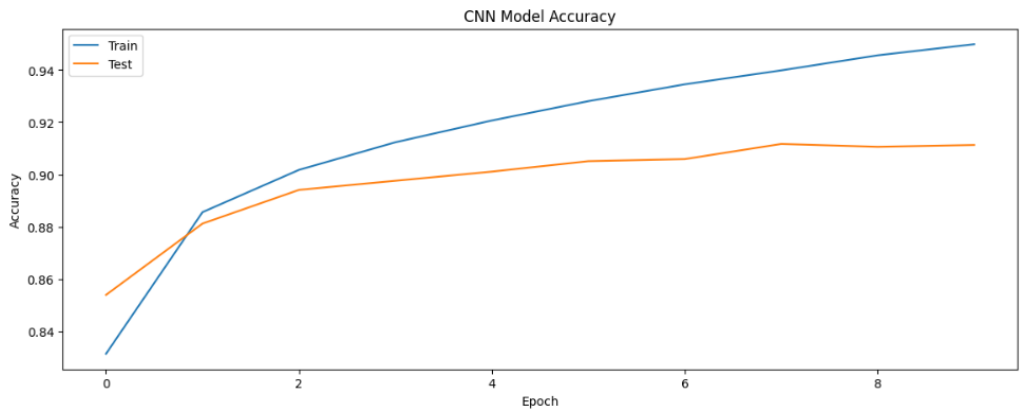
b. CNN model

Code

```python
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout

# Reshape the data for CNN
X_train_cnn = X_train.reshape(-1, 28, 28, 1)
X_val_cnn = X_val.reshape(-1, 28, 28, 1)
X_test_cnn = X_test.reshape(-1, 28, 28, 1)

# Define the CNN model
cnn_model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

# Compile the model
cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
history_cnn = cnn_model.fit(X_train_cnn, y_train, validation_data=(X_val_cnn, y_val), epochs=20, batch_size=32)
```

Output

```
Epoch 1/20
1500/1500 [==============================] - 65s 41ms/step - loss: 0.6001 - accuracy: 0.7826 - val_loss: 0.3944 - val_accuracy: 0.8578
Epoch 2/20
1500/1500 [==============================] - 56s 37ms/step - loss: 0.4033 - accuracy: 0.8559 - val_loss: 0.3332 - val_accuracy: 0.8749
Epoch 3/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.3492 - accuracy: 0.8753 - val_loss: 0.2902 - val_accuracy: 0.8893
Epoch 4/20
1500/1500 [==============================] - 55s 37ms/step - loss: 0.3168 - accuracy: 0.8857 - val_loss: 0.2692 - val_accuracy: 0.8989
Epoch 5/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.2876 - accuracy: 0.8955 - val_loss: 0.2579 - val_accuracy: 0.9021
Epoch 6/20
1500/1500 [==============================] - 52s 34ms/step - loss: 0.2657 - accuracy: 0.9018 - val_loss: 0.2492 - val_accuracy: 0.9053
Epoch 7/20
1500/1500 [==============================] - 50s 33ms/step - loss: 0.2481 - accuracy: 0.9089 - val_loss: 0.2561 - val_accuracy: 0.9053
Epoch 8/20
1500/1500 [==============================] - 54s 36ms/step - loss: 0.2378 - accuracy: 0.9116 - val_loss: 0.2422 - val_accuracy: 0.9107
Epoch 9/20
1500/1500 [==============================] - 50s 33ms/step - loss: 0.2226 - accuracy: 0.9183 - val_loss: 0.2645 - val_accuracy: 0.9048
Epoch 10/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.2108 - accuracy: 0.9211 - val_loss: 0.2373 - val_accuracy: 0.9152
Epoch 11/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.2024 - accuracy: 0.9252 - val_loss: 0.2403 - val_accuracy: 0.9116
Epoch 12/20
1500/1500 [==============================] - 50s 34ms/step - loss: 0.1915 - accuracy: 0.9288 - val_loss: 0.2342 - val_accuracy: 0.9122
Epoch 13/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.1813 - accuracy: 0.9316 - val_loss: 0.2434 - val_accuracy: 0.9172
Epoch 14/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.1758 - accuracy: 0.9331 - val_loss: 0.2440 - val_accuracy: 0.9166
Epoch 15/20
1500/1500 [==============================] - 48s 32ms/step - loss: 0.1673 - accuracy: 0.9348 - val_loss: 0.2642 - val_accuracy: 0.9103
Epoch 16/20
1500/1500 [==============================] - 49s 33ms/step - loss: 0.1592 - accuracy: 0.9382 - val_loss: 0.2509 - val_accuracy: 0.9186
Epoch 17/20
1500/1500 [==============================] - 51s 34ms/step - loss: 0.1528 - accuracy: 0.9419 - val_loss: 0.2589 - val_accuracy: 0.9119
Epoch 18/20
1500/1500 [==============================] - 50s 33ms/step - loss: 0.1492 - accuracy: 0.9413 - val_loss: 0.2663 - val_accuracy: 0.9153
Epoch 19/20
1500/1500 [==============================] - 53s 35ms/step - loss: 0.1425 - accuracy: 0.9448 - val_loss: 0.2663 - val_accuracy: 0.9178
Epoch 20/20
1500/1500 [==============================] - 50s 34ms/step - loss: 0.1368 - accuracy: 0.9476 - val_loss: 0.2639 - val_accuracy: 0.9175
```



c. Identify the difference between ANN and CNN models

ANN:

- Treats each pixel as an independent feature.
- Less computationally intensive.
- Less effective in capturing spatial relationships in images.

CNN:

- Uses convolutional layers to capture spatial hierarchies in images.
- More computationally intensive due to convolution operations.
- Typically performs better on image data by recognizing patterns such as edges, textures, and more complex features.

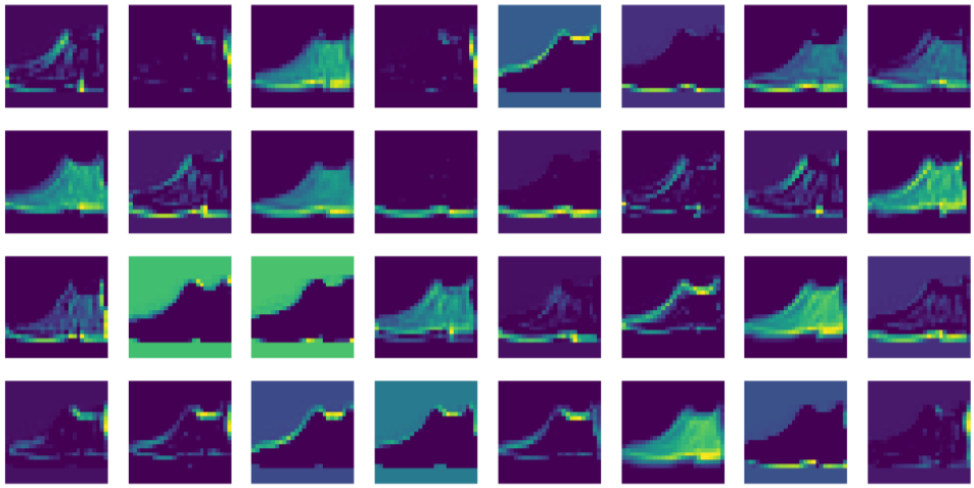
d. Visualize different layers in the

Code

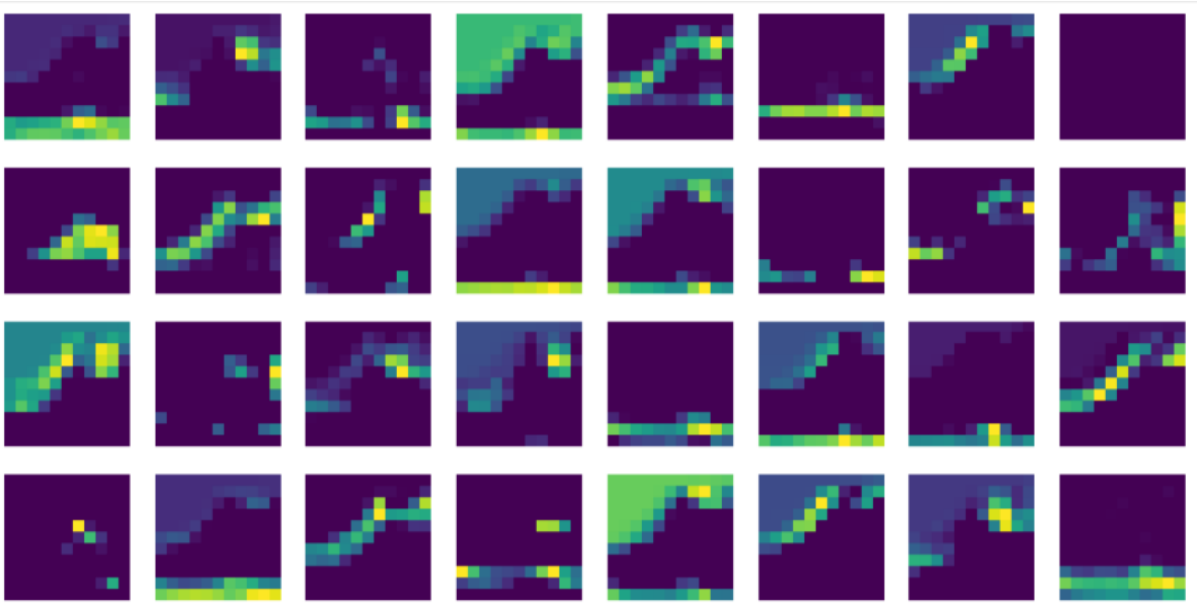
```
from tensorflow.keras.models import Model
import numpy as np

# Define a model that outputs the activations from each layer
layer_outputs = [layer.output for layer in cnn_model.layers]
activation_model = Model(inputs=cnn_model.input, outputs=layer_outputs)

# Get the activations for a sample input
sample_image = X_test_cnn[0].reshape(1, 28, 28, 1)
activations = activation_model.predict(sample_image)

# Plot the activations of the first convolutional layer
first_layer_activation = activations[0]

fig, axes = plt.subplots(4, 8, figsize=(12, 6))
for i in range(32):
    ax = axes[i // 8, i % 8]
    ax.matshow(first_layer_activation[0, :, :, i], cmap='viridis')
    ax.axis('off')

plt.show()

second_layer_activation = activations[2]

fig, axes = plt.subplots(4, 8, figsize=(12, 6))
for i in range(32):
    ax = axes[i // 8, i % 8]
    ax.matshow(second_layer_activation[0, :, :, i], cmap='viridis')
    ax.axis('off')

plt.show()
```

Output

- First layer activation



- Second layer activation



As we move deeper into the network, the complexity of patterns recognized by the layers increases:

- Initial Layers: Recognize simple features such as edges and corners.
- Intermediate Layers: Capture more complex patterns such as textures and shapes.
- Deeper Layers: Identify high-level patterns and object parts, contributing to the final classification.

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

More Nodes in a Single Layer:

- Increases model capacity, potentially capturing more complex patterns.
- Higher risk of overfitting.
- Increases computational complexity and training time.

Deep Network (More Layers):

- Allows the model to learn hierarchical features.
- Can capture complex patterns effectively.
- Increases computational complexity.
- Requires careful tuning to avoid vanishing/exploding gradient problems.

Shallow Network (Fewer Layers):

- Easier to train.
- Lower computational complexity.May fail to capture complex patterns and relationships in the data.

g. Defining the Optimal Neural Network Architecture

The optimal neural network architecture is defined by balancing model complexity and performance. This involves:

- Hyperparameter Tuning: Experimenting with different numbers of layers, nodes, learning rates, and batch sizes.
- Cross-Validation: Using techniques like k-fold cross-validation to assess model performance.
- Regularization Techniques: Implementing dropout, L2 regularization, and batch normalization to prevent overfitting.
- Performance Monitoring: Monitoring metrics such as loss, accuracy, precision, recall, and F1 score to guide adjustments to the architecture.