ML - Assignment 4

SECTION - A

a) Given, no padding, stride = 1

input img dim^n = $n \times n$ output img dim^n =

Size of Kernel = $K \times K$

⇒ Output Height = $(N \times 1) - K + 1 = N - K + 1$

Output width = $N - K + 1$

Output Dimension = $(N - K + 1)(N - K + 1)$

b) Each Kernel operation require $K \times K$ multiplication

Input channels = P

Total Multiplication = PK^2

Total Addition = They are summed across the channel so require 1 less = $PK^2 - 1$

Tot operation = Total Multiplication + Total Addition

$= PK^2 + PK^2 - 1 = \boxed{2PK^2 - 1}$

c) Operation / Kernel = $2PK^2 - 1$

As 2 kernel independtly calculate entire feature map

Total Operation = 2 × Operation / pixel × Total pixel × Q

$= 2 \times (PK^2 - 1) \times (N - k + 1)(N - k + 1) \times Q$

total op $= 2[mq - k^2q + 2kq - nq + pk^2q - 2pk^2q + pk^2q - mnq + kmp + knq - pk^2mq + pk^2nq - pk^2nq + pk^2nq + pk^2mnq - q]$

It's a 6th degree term for Big-O notation ←

as it needs highest degree term we can ignore other term except this

Comp complexity = $O(2PK^2MN) \xrightarrow[\text{constant}]{\text{Remains}} \boxed{O(PK^2MN)}$

for $\min(M,N) >> K \Rightarrow K^2 >> M,\ K^2 >> N$

So ignoring K^2 | Computation complexity $= O(2PMN) = O(PMN)$

B) Assignment Step

- Each & Every Data point is assigned to the nearest cluster based on euclidean distance to current clusters centroid

$$\text{Cluster}_j = \arg\min_j \|x_i - u_j\|^2$$

x_i: Data point
u_j: centroid of c_j

So each data point belong to cluster whose centroid is closest

Update Step

Once every point is assigned, the centroid of ~~nearest~~ cluster is recomputed based of ~~all~~ mean ~~of~~ all pt in that cluster

$$u_i = \frac{1}{|C_j|} \sum x_i$$: This step adjust cluster centroid to reflect centre of assigned point

OPTIMAL NUMBER OF CLUSTS

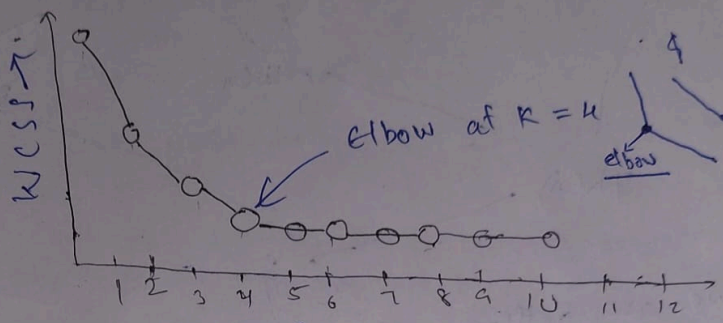
~~In~~ Elbow Method ⇒ Technique/Method used

In Elbow Method we vary K i.e no. of clusters from 1-10 for each value we are calculating WCSS (within-Clusters Sum of Square). WCSS is sum of squared distance between each point and centroid in cluster.

- As K increases sum of squared distance decreases b/c points are close to centroids
- At some point, rate of decrease slows down forming an elbow in plot, this elbow indicates optimal no. of clusts

$$W.C.S.S = \sum_j \sum_{x_i \in c_j} \|x_i - u_j\|^2$$

This method is subjective as elbow might not always be sharp, it may not be suitable for all dataset specially those with high dimensionality.



Though K-means is a converging Algorithm but it may get stuck on a local minima & not finding global minima Pf by Counter-example

1 x4 2 Here in update step c1 & c2 won't change but an optimal soln exist given below

1 xc_2 2

(1)

1 x 2

2 x 1

(2)

$WCSS_1 > WCSS_2$

Convergence proof

$SSE(c^{r+1}, u^{r+1}) < SSE(c^r, u^r)$ Since no. of poss clustering is finite (K^n)

$$SSE(c^{r+1}, u^r) = \sum_{i=1}^{n} \|x^i - u^r_{c^{r+1}(i)}\| \leq SSE(c^r, u^r)$$

$$SSE(c^{r+1}, u^{r+1}) = \sum_{i=1}^{n} \|x^i - u^{r+1}_{c^{r+1}}\|^2 \leq \sum_{k=1}^{K} \sum_{\substack{i \in \{1 \ldots n\} \\ c^{r+1}(i)=k}} \|x^i - u^{r+1}_{c^{r+1}(i)}\|^2$$

$$\leq SSE(c^{r+1}, u^r)$$

$\Rightarrow$ Hence K means converges

K-Means Clustering Algorithm:

K-means clustering is an iterative algorithm used to partition a dataset into kkk clusters. Each cluster is represented by its centroid, which is the mean of all data points assigned to that cluster. Here's how the algorithm works step-by-step:

(a) Initialization: Use the Given Centroids

In the initialization step, the centroids of the clusters are defined.

- If **predefined centroids** are given as input, we use those specific points as the initial centroids of the clusters. These centroids serve as starting points for the K-means algorithm.
- If no centroids are provided, the algorithm can randomly select kkk data points from the dataset to serve as the initial centroids.

```
u1: [3. 3.]
u2: [2. 2.]
Matrix X:
 [[ 5.1  3.5]
 [ 4.9  3. ]
 [ 5.8  2.7]
 [ 6.   3. ]
 [ 6.7  3.1]
 [ 4.5  2.3]
 [ 6.1  2.8]
 [ 5.2  3.2]
 [ 5.5  2.6]
 [ 5.   2. ]
 [ 8.   0.5]
 [ 7.5  0.8]
 [ 8.1 -0.1]
 [ 2.5  3.5]
 [ 1.   3. ]
 [ 4.5 -1. ]
 [ 3.  -0.5]
 [ 5.1 -0.2]
 [ 6.  -1.5]
 [ 3.5 -0.1]
 [ 4.   0. ]
 [ 6.1  0.5]
 [ 5.4 -0.5]
 [ 5.3  0.3]
 [ 5.8  0.6]]
```

(b) Assignment: Assign Each Data Point to the Nearest Centroid

In the assignment step, each data point in the dataset is assigned to the nearest centroid. This is done by calculating the **Euclidean distance** between the data point and each of the centroids, then assigning the data point to the centroid that has the smallest distance.

- **Euclidean Distance**: The Euclidean distance between two points is calculated as the square root of the sum of the squared differences between their coordinates. This is used to measure the closeness between a data point and a centroid.
- Each data point is assigned to the centroid with the smallest Euclidean distance. This forms the initial clusters for the algorithm.

```
for i, point in enumerate(X):
    print(f"Data point {point} is assigned to centroid {assignments[i] + 1}")
```

```
Data point [5.1 3.5] is assigned to centroid 1
Data point [4.9 3. ] is assigned to centroid 1
Data point [5.8 2.7] is assigned to centroid 1
Data point [6. 3.] is assigned to centroid 1
Data point [6.7 3.1] is assigned to centroid 1
Data point [4.5 2.3] is assigned to centroid 1
Data point [6.1 2.8] is assigned to centroid 1
Data point [5.2 3.2] is assigned to centroid 1
Data point [5.5 2.6] is assigned to centroid 1
Data point [5. 2.] is assigned to centroid 1
Data point [8.  0.5] is assigned to centroid 1
Data point [7.5 0.8] is assigned to centroid 1
Data point [ 8.1 -0.1] is assigned to centroid 1
Data point [2.5 3.5] is assigned to centroid 1
Data point [1. 3.] is assigned to centroid 2
Data point [ 4.5 -1. ] is assigned to centroid 2
Data point [ 3.  -0.5] is assigned to centroid 2
Data point [ 5.1 -0.2] is assigned to centroid 2
Data point [ 6.  -1.5] is assigned to centroid 2
Data point [ 3.5 -0.1] is assigned to centroid 2
Data point [4. 0.] is assigned to centroid 2
Data point [6.1 0.5] is assigned to centroid 1
Data point [ 5.4 -0.5] is assigned to centroid 2
Data point [5.3 0.3] is assigned to centroid 1
Data point [5.8 0.6] is assigned to centroid 1
```

(c) Update: Recalculate the Centroids

Once each data point is assigned to a cluster, the centroids are updated.

- The new position of each centroid is calculated as the **mean** of all the data points that have been assigned to that centroid (i.e., the average of all points in that cluster).
- After updating the centroids, the algorithm proceeds to the next iteration, where it will reassign points based on the updated centroids.

```
New centroids after the first update:
Centroid 1: [5.77058824 2.01764706]
Centroid 2: [ 4.0625 -0.1   ]
```

(d) Convergence Check

The algorithm repeats the assignment and update steps until convergence. Convergence means that the centroids do not change significantly between iterations.

- If the centroids do not change much (i.e., the change is smaller than a predefined threshold), the algorithm terminates, signaling that the clusters are stable.
- Alternatively, the algorithm can stop after a maximum number of iterations (e.g., 100 iterations) to avoid running indefinitely if convergence is slow.

The convergence threshold is typically a small value (e.g., 1×10−41 \times 10^{-4}1×10−4) that defines how much the centroids need to change before the algorithm is considered to have converged.

```
Converged at iteration 3
```

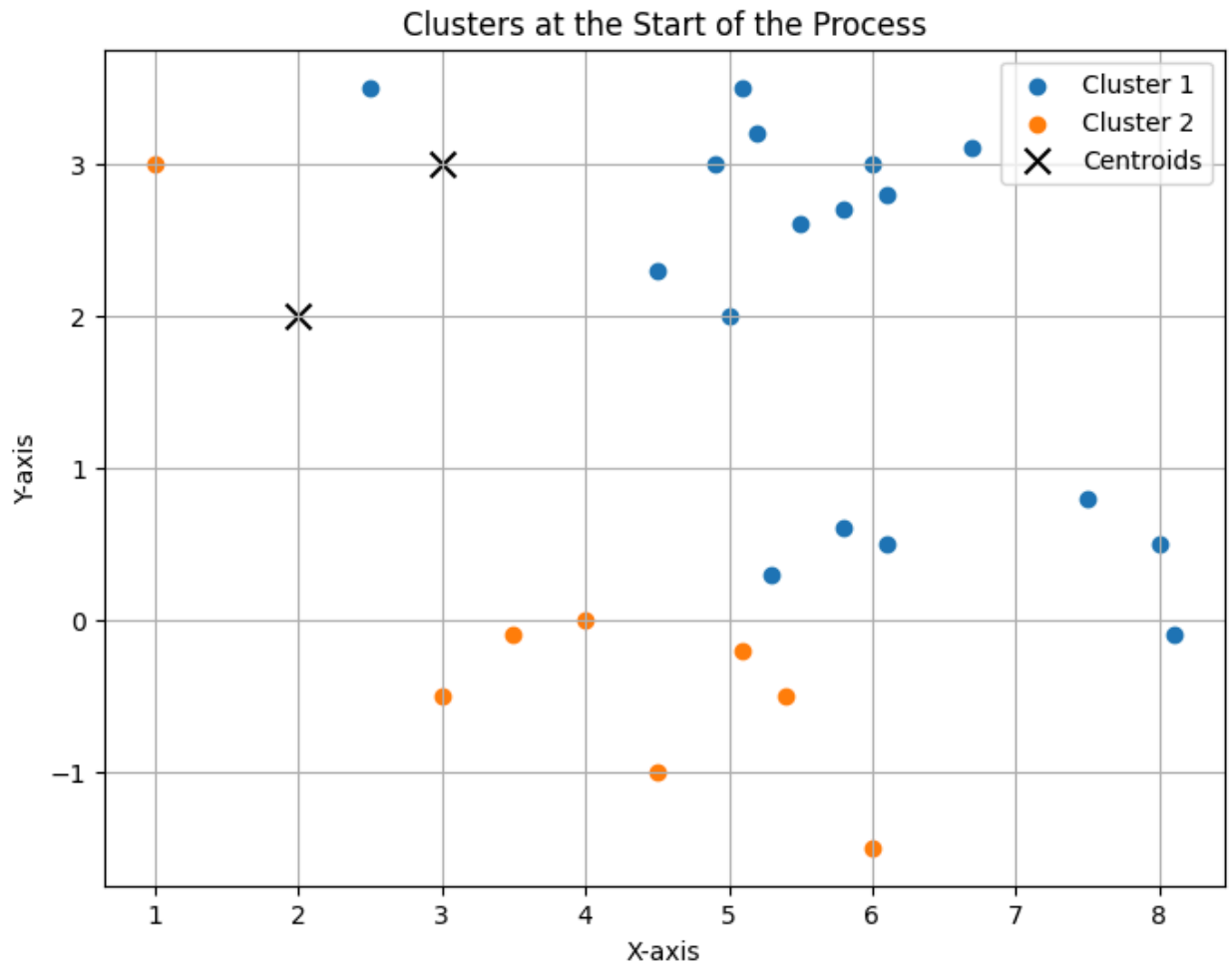
(b) Final Centroids after Convergence

Once the algorithm has converged, the final centroids represent the mean positions of all points in each cluster. These centroids do not change significantly in the last iteration.

- **Plotting the Clusters**:
    - The clusters can be visualized by plotting the data points, where each point is colored according to the cluster it belongs to.
    - The centroids are typically marked with a distinct symbol (e.g., a red 'X') to differentiate them from the data points.

```
Final centroids after convergence:
Centroid 1: [5.8   2.125]
Centroid 2: [ 4.2        -0.05555556]

Final assignments after convergence:
Data point [5.1 3.5] is assigned to centroid 1
Data point [4.9 3. ] is assigned to centroid 1
Data point [5.8 2.7] is assigned to centroid 1
Data point [6. 3.] is assigned to centroid 1
Data point [6.7 3.1] is assigned to centroid 1
Data point [4.5 2.3] is assigned to centroid 1
Data point [6.1 2.8] is assigned to centroid 1
Data point [5.2 3.2] is assigned to centroid 1
Data point [5.5 2.6] is assigned to centroid 1
Data point [5. 2.] is assigned to centroid 1
Data point [8.  0.5] is assigned to centroid 1
Data point [7.5 0.8] is assigned to centroid 1
Data point [ 8.1 -0.1] is assigned to centroid 1
Data point [2.5 3.5] is assigned to centroid 1
Data point [1. 3.] is assigned to centroid 2
Data point [ 4.5 -1. ] is assigned to centroid 2
Data point [ 3.  -0.5] is assigned to centroid 2
Data point [ 5.1 -0.2] is assigned to centroid 2
Data point [ 6.  -1.5] is assigned to centroid 2
Data point [ 3.5 -0.1] is assigned to centroid 2
Data point [4. 0.] is assigned to centroid 2
Data point [6.1 0.5] is assigned to centroid 1
Data point [ 5.4 -0.5] is assigned to centroid 2
Data point [5.3 0.3] is assigned to centroid 2
Data point [5.8 0.6] is assigned to centroid 1
```

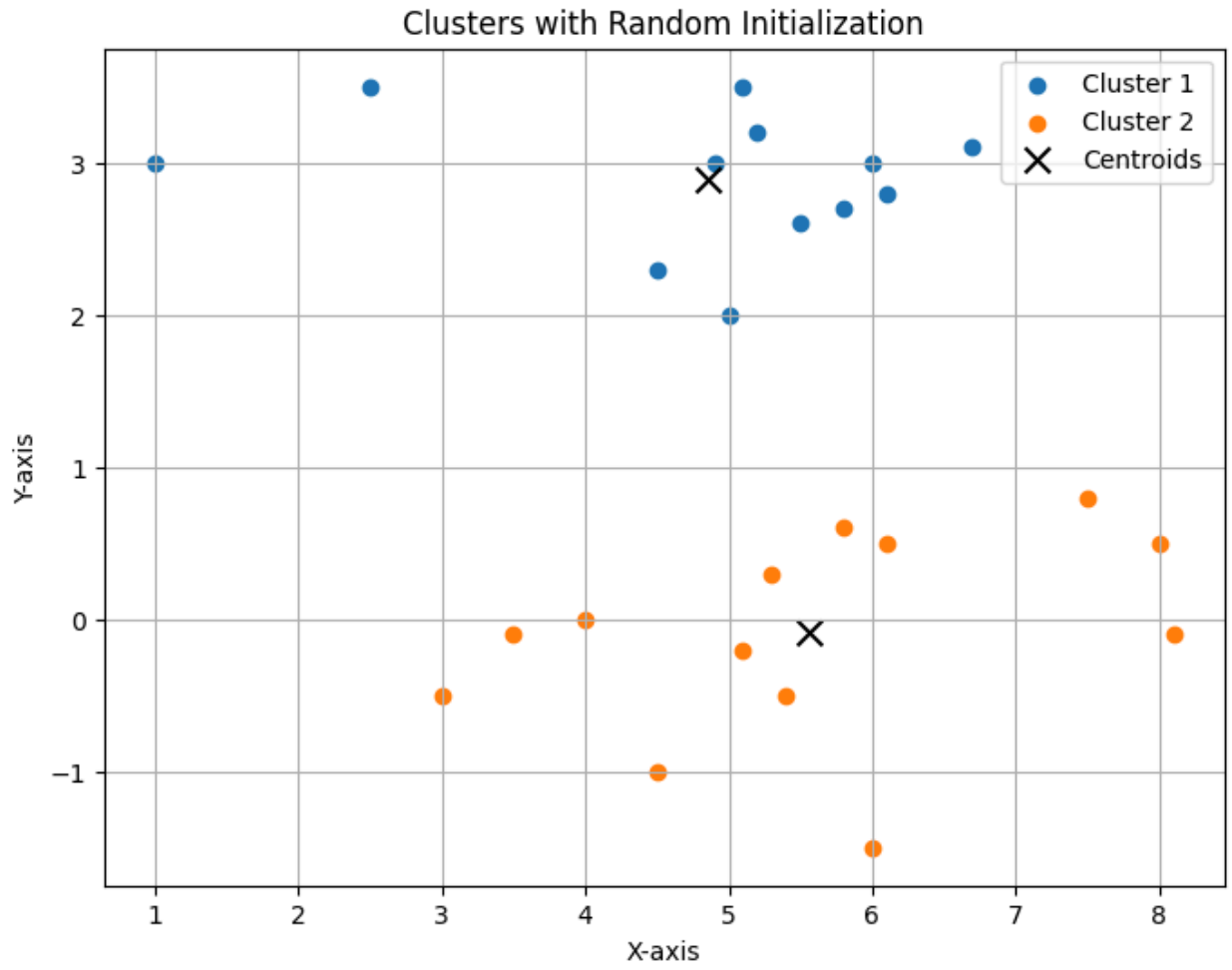

Clusters at the Start of the Process
Cluster 1
Cluster 2
Centroids
Y-axis
X-axis
3
2
1
0
−1
1
2
3
4
5
6
7
8

(c) Comparison: Given Centroids vs. Random Initialization

K-means clustering can produce different results depending on how the centroids are initialized.

1. **Given Centroids**:
    - When the centroids are provided, the algorithm begins with these predefined points. This often leads to faster convergence since the starting points are already close to the desired cluster centers.
    - The algorithm may be more stable because it is less likely to converge to poor local minima.
2. **Random Initialization**:
    - When centroids are initialized randomly, the starting points can vary each time the algorithm runs, leading to different results on different runs.
    - Poor initialization can result in the algorithm converging to suboptimal clusters (local minima), especially if the centroids are far from the true cluster centers.
    - To mitigate this, multiple runs of the algorithm with different initializations can be used, and the best result can be selected.

```
Converged at iteration 2
```




```
Final after convergence sum of square using random
67.15833333333333
Sum of square using random
94.08999999999999
Final after convergence sum of square using initial
83.67222222222222
sum of square using initial
301.75
```

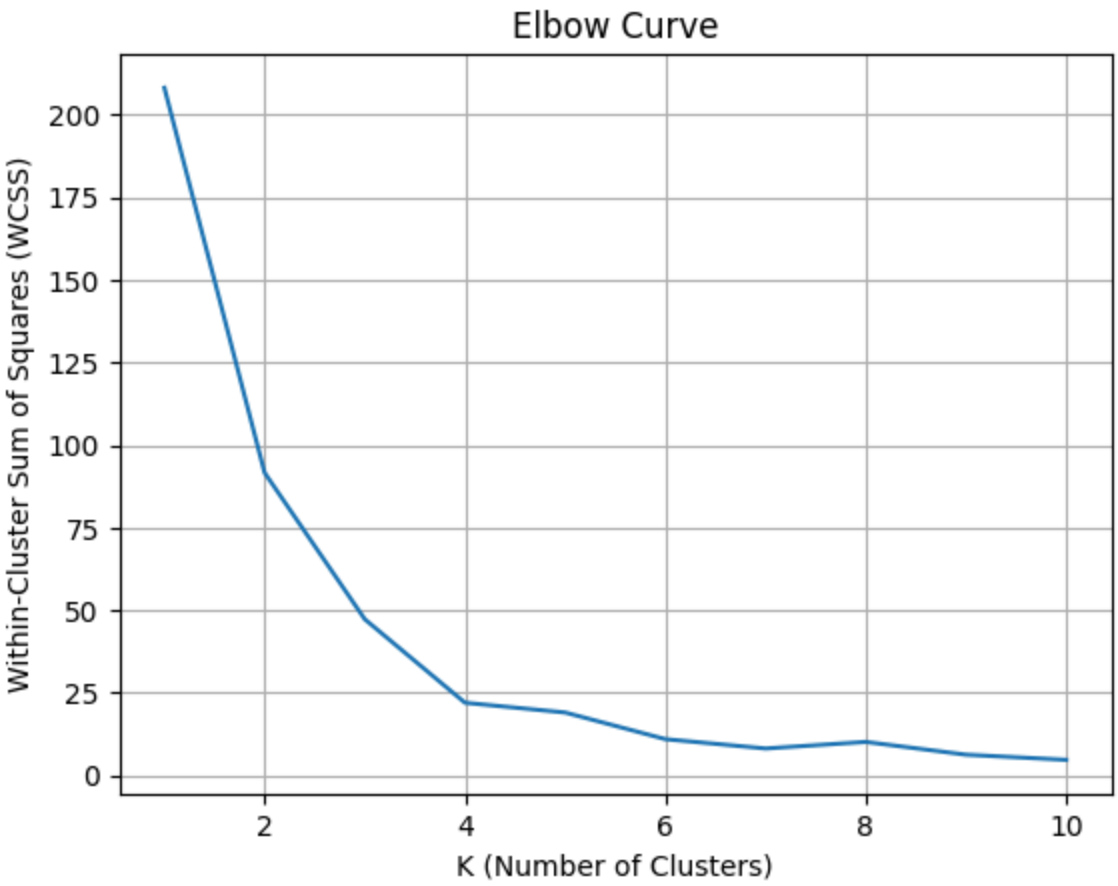
The Final submission shows random initialization works better this is purely luck based various earlier trials and experiment also showed otherwise.

(d) Elbow Method to Determine the Optimal Number of Clusters (k)

The **Elbow Method** is a heuristic used to determine the optimal number of clusters for the dataset. The idea is to find the point at which adding more clusters no longer significantly improves the fit of the model.

1. **Within-Cluster Sum of Squares (WCSS)**:
    - WCSS measures the compactness of the clusters. It is the sum of the squared distances between each data point and its assigned centroid.
    - As the number of clusters increases, the WCSS generally decreases because more clusters can fit the data better.
    - However, after a certain point, increasing the number of clusters results in only marginal improvements to WCSS.
2. **Finding the "Elbow"**:
    - Plot WCSS against the number of clusters kkk. The plot typically shows a steep decrease in WCSS initially, followed by a slower decrease as the number of clusters increases.
    - The "elbow" is the point where the rate of decrease in WCSS slows down. This point indicates the optimal number of clusters, as adding more clusters beyond this point provides only a small improvement.

By examining the WCSS plot, we can identify the optimal number of clusters kkk that balances the complexity of the model (number of clusters) and the fit to the data.


Elbow Curve
Within-Cluster Sum of Squares (WCSS)
200
175
150
125
100
75
50
25
0
2
4
6
8
10
K (Number of Clusters)

```
[[ 5.45714286 -0.25714286]
 [ 7.86666667  0.4       ]
 [ 5.20909091  2.88181818]
 [ 2.875       0.6       ]]
[[2.         0.39404959]
 [2.         0.10950413]
 [2.         0.3822314 ]
 [2.         0.63950413]
 [2.         2.27041322]
 [2.         0.84132231]
 [2.         0.80041322]
 [2.         0.10132231]
 [2.         0.16404959]
 [2.         0.82132231]
 [1.         0.02777778]
 [1.         0.29444444]
 [1.         0.30444444]
 [2.         7.72132231]
 [3.         9.275625  ]
 [0.         1.46795918]
 [3.         1.225625  ]
 [0.         0.13081633]
 [0.         1.83938776]
 [3.         0.880625  ]
 [3.         1.625625  ]
 [0.         0.98653061]
 [0.         0.0622449 ]
 [0.         0.33510204]
 [0.         0.8522449 ]]
```

```
print(a0,a1,a2,a3)
```

```
5.6742857142857135 0.6266666666666667 5.6742857142857135 5.6742857142857135
```

```
print(a0+a1+a2+a3)
```

```
17.649523809523807
```

Conclusion

K-means clustering is an unsupervised machine learning algorithm that partitions data into kkk clusters based on the following steps:

1. **Initialization**: Choose starting centroids (either predefined or randomly initialized).
2. **Assignment**: Assign each data point to the nearest centroid.
3. **Update**: Recalculate the centroids based on the mean of the assigned points.
4. **Convergence Check**: Repeat until the centroids no longer change or a maximum number of iterations is reached.
5. **Final Centroids**: After convergence, the final centroids represent the cluster centers.

The algorithm can be further evaluated by comparing the results with random vs. given initial centroids, and the optimal number of clusters can be determined using the Elbow Method, which analyzes the Within-Cluster Sum of Squares (WCSS) for various values of kkk.

This explanation provides a comprehensive overview of how K-means clustering works, including all the points you asked about: initialization, assignment, update, convergence, final centroids, and comparison with random initialization.

1. Data Preparation

In this task, I am preparing the CIFAR-10 dataset for training by selecting three classes and performing a stratified random split into training, validation, and test datasets.

- **Dataset Loading**: The CIFAR-10 dataset consists of 60,000 images, divided into 10 classes. Each class contains 6,000 images. For this task, I selected 3 classes from the CIFAR-10 dataset. These classes are:
    - Class 0: Cat
    - Class 1: Dog
    - Class 2: Horse
- This gives a total of 18,000 images (6,000 per class). I then split these images as follows:
    - **Training Dataset**: 15,000 images (80% of the curated data), consisting of 5,000 images from each of the 3 classes.
    - **Validation Dataset**: 3,000 images (20% of the curated data), with 1,000 images from each class.
    - **Test Dataset**: I retain 3,000 images (1,000 images per class) from the original test set in CIFAR-10.
- **Stratified Split**: To ensure that each class is proportionally represented in both the training and validation datasets, I perform a stratified random split. This means the class distribution is preserved in both the training and validation datasets, ensuring that each class has a similar number of examples in both splits.
- **Custom Dataset Class**: I created a custom Dataset class that handles loading, transforming, and splitting the images. This custom class:
    - Loads the images and their corresponding labels.
    - Applies necessary transformations such as resizing or normalizing images.
    - Manages the stratified split for the training, validation, and test datasets.
- **Data Loaders**: Using PyTorch's DataLoader, I created separate data loaders for the training, validation, and test datasets. The DataLoader is used to load the data in batches, which is useful for training models efficiently. The data loaders help manage the data and feed it to the model during training and testing.

```
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: ./data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
           )
Dataset CIFAR10
    Number of datapoints: 10000
    Root location: ./data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
               Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
           )
```

```
print(len(train_data))
print(len(test_data))
```

[8] ✓ 0.0s

```
15000
3000
```

```
for images, labels in train_loader:
    print(images.shape)
    print(labels.shape)
    break
```

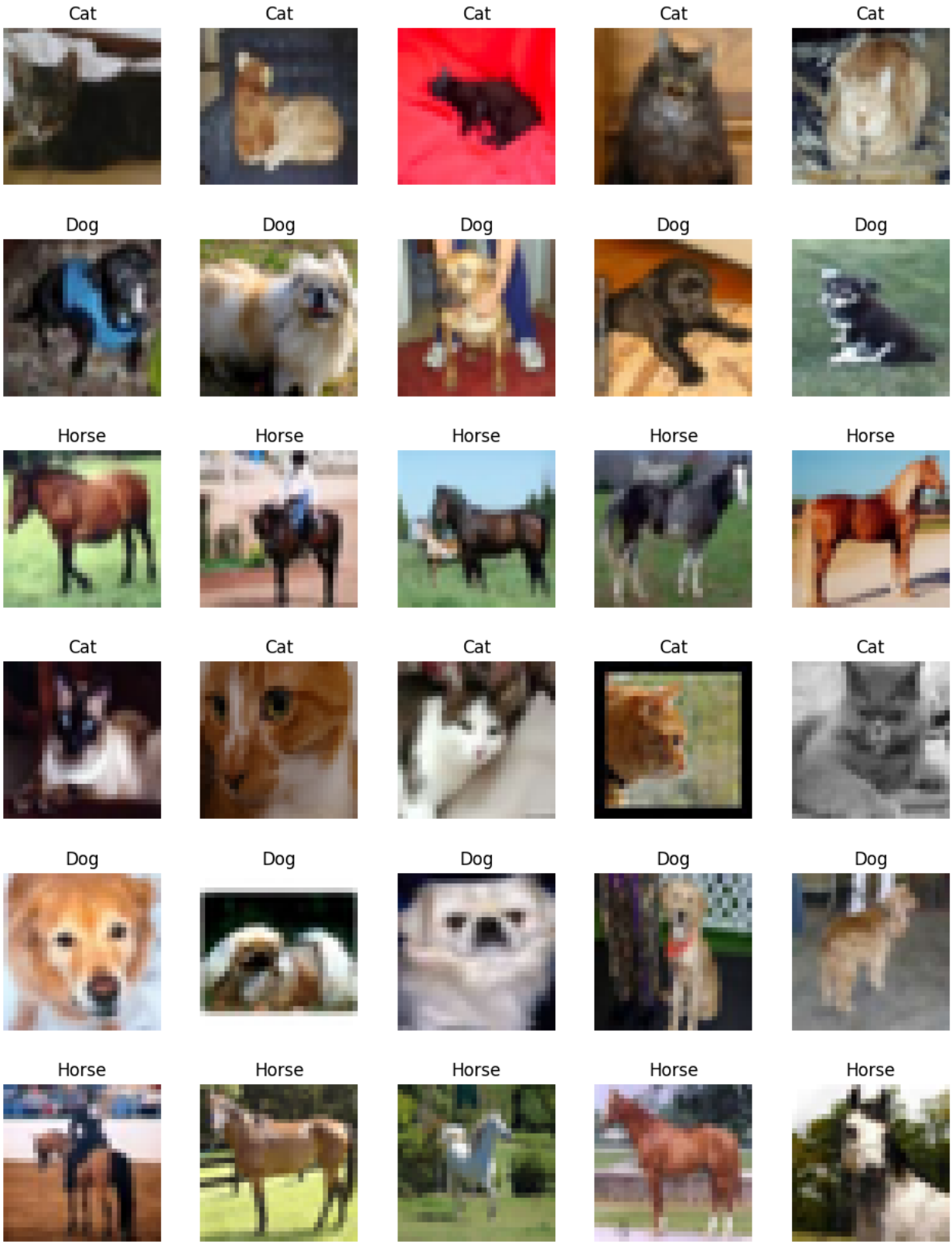
[14] ✓ 0.0s

```
torch.Size([64, 3, 32, 32])
torch.Size([64])
```

2. Visualization

In this step, I visualize the images from the training and validation datasets to ensure the dataset is loaded correctly and the classes are properly distributed.

- **Image Visualization**: I used matplotlib to display 5 images from each of the 3 selected classes in both the training and validation datasets. This helped me visually inspect whether the data is loaded correctly and if the images correspond to the correct labels. For example:
    - The first 5 images from the training set for class 'Airplane' are displayed.
    - The next 5 images from the training set for class 'Automobile' are shown, and so on.
- **Purpose**: The purpose of this visualization is to confirm that:
    - The data is loaded correctly.
    - The images are properly labeled.
    - The split between the training and validation datasets is correct.


Cat
Cat
Cat
Cat
Cat
Dog
Dog
Dog
Dog
Dog
Horse
Horse
Horse
Horse
Horse
Cat
Cat
Cat
Cat
Cat
Dog
Dog
Dog
Dog
Dog
Horse
Horse
Horse
Horse
Horse

3. CNN Implementation

For image classification, I implemented a Convolutional Neural Network (CNN). The architecture of the CNN is as follows:

- **Layer 1**:
  - A convolutional layer with a kernel size of 5x5.
  - The number of output channels (filters) is 16.
  - Padding is set to 1 to preserve the spatial dimensions.
  - Stride is set to 1.
- **Max-Pooling Layer 1**:
  - A max-pooling layer with a kernel size of 3x3 and a stride of 2 to reduce the spatial dimensions.
- **Layer 2**:
  - A convolutional layer with a kernel size of 3x3.
  - The number of output channels (filters) is 32.
  - Padding is set to 0, and stride is set to 1.
- **Max-Pooling Layer 2**:
  - A max-pooling layer with a kernel size of 3x3 and a stride of 3.
- **Fully Connected Layer (MLP)**:
  - After flattening the output from the previous layers, a fully connected layer is added with 16 neurons and ReLU activation.
  - The output layer is a classification head with 3 neurons (one for each class).
- **Activation Functions**:
  - I used the **ReLU activation function** after each convolutional and fully connected layer except for the final classification layer.
- **Purpose**: The CNN is designed to capture spatial patterns in images, such as edges and textures. By using convolutional layers, the network can automatically learn hierarchical features from the images. The fully connected layers at the end help to map these features to the class labels.

4. Training the Model (2.5 points)

After constructing the CNN, I trained it using the cross-entropy loss function and the Adam optimizer.

- **Loss Function**: I used **Cross-Entropy Loss** because this is a multi-class classification problem. This loss function measures the difference between the predicted class probabilities and the true labels.
- **Optimizer**: I used the **Adam Optimizer**, which adjusts the learning rate based on the gradient's momentum. Adam is widely used because it tends to perform well on a wide range of problems.

- **Training Procedure**: I trained the model for **15 epochs**, where each epoch corresponds to one pass through the entire training dataset. During each epoch:
    - I calculated the training loss and accuracy.
    - I calculated the validation loss and accuracy after each epoch to ensure the model is generalizing well to unseen data.
- **Metrics**: After each epoch, I logged:
    - **Training Loss** and **Training Accuracy**: To monitor the model's performance on the training data.
    - **Validation Loss** and **Validation Accuracy**: To monitor the model's performance on the validation data.
- **Saving the Model**: After training the model, I saved the trained model using torch.save(). This allows me to load and use the trained model later for testing or inference.

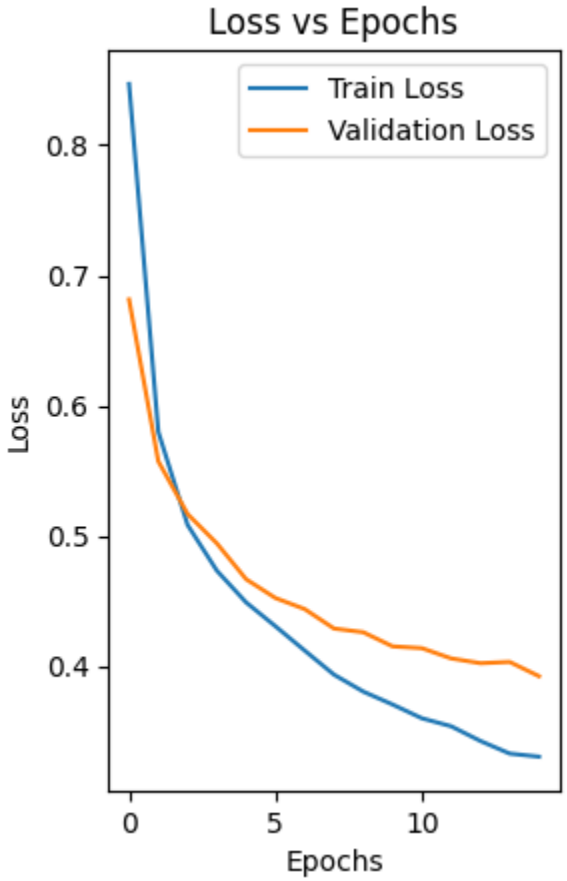
```
Epoch 1/15, Train Loss: 0.8469, Train Accuracy: 63.12%, Val Loss: 0.6815, Val Accuracy: 72.63%
Epoch 2/15, Train Loss: 0.5799, Train Accuracy: 77.33%, Val Loss: 0.5573, Val Accuracy: 78.33%
Epoch 3/15, Train Loss: 0.5084, Train Accuracy: 79.97%, Val Loss: 0.5168, Val Accuracy: 79.77%
Epoch 4/15, Train Loss: 0.4733, Train Accuracy: 81.31%, Val Loss: 0.4943, Val Accuracy: 81.23%
Epoch 5/15, Train Loss: 0.4489, Train Accuracy: 82.18%, Val Loss: 0.4667, Val Accuracy: 82.27%
Epoch 6/15, Train Loss: 0.4307, Train Accuracy: 83.22%, Val Loss: 0.4523, Val Accuracy: 82.67%
Epoch 7/15, Train Loss: 0.4120, Train Accuracy: 84.03%, Val Loss: 0.4438, Val Accuracy: 82.87%
Epoch 8/15, Train Loss: 0.3935, Train Accuracy: 84.63%, Val Loss: 0.4289, Val Accuracy: 83.87%
Epoch 9/15, Train Loss: 0.3805, Train Accuracy: 85.22%, Val Loss: 0.4261, Val Accuracy: 83.50%
Epoch 10/15, Train Loss: 0.3706, Train Accuracy: 85.62%, Val Loss: 0.4152, Val Accuracy: 84.17%
Epoch 11/15, Train Loss: 0.3600, Train Accuracy: 86.21%, Val Loss: 0.4138, Val Accuracy: 84.37%
Epoch 12/15, Train Loss: 0.3539, Train Accuracy: 86.32%, Val Loss: 0.4060, Val Accuracy: 84.67%
Epoch 13/15, Train Loss: 0.3426, Train Accuracy: 87.04%, Val Loss: 0.4023, Val Accuracy: 84.47%
Epoch 14/15, Train Loss: 0.3329, Train Accuracy: 87.19%, Val Loss: 0.4033, Val Accuracy: 84.80%
Epoch 15/15, Train Loss: 0.3306, Train Accuracy: 87.42%, Val Loss: 0.3923, Val Accuracy: 85.40%
```

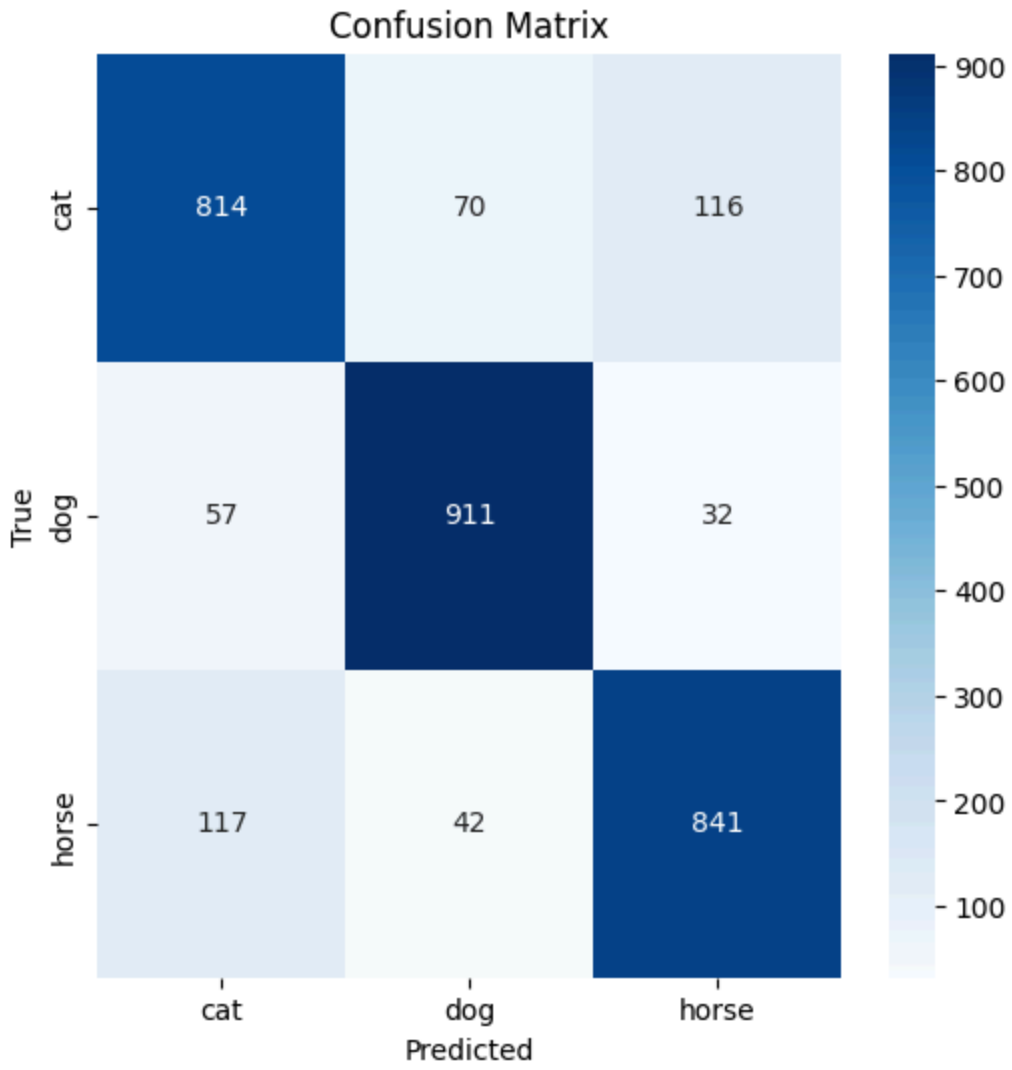
5. Testing (1.5 points)

After training the model, I evaluated its performance on the test dataset.

- **Training and Validation Plots**: I plotted the **training and validation loss** as well as **training and validation accuracy** over the 15 epochs. These plots allow me to observe:
    - If the model is improving over time.
    - If the model is overfitting (when validation accuracy decreases while training accuracy increases).
- **Performance Metrics**: I evaluated the model's performance on the test dataset by calculating:
    - **Accuracy**: This is the percentage of correct predictions over all the test images.
    - **F1-Score**: This combines precision and recall into a single metric, which is particularly useful for imbalanced classes.
- **Confusion Matrix**: I computed and visualized the **confusion matrix** for the training, validation, and test datasets. This matrix shows how well the model is classifying each of the three classes, allowing me to spot which classes are confused with each other.

```
Test Accuracy: 0.86
Test F1-Score: 0.86
```





6. Training an MLP (3 points)

In this section, I implemented and trained a Multilayer Perceptron (MLP) on the same dataset. The architecture of the MLP is as follows:

- **Flattening**:
  - Since MLPs require 1D input, I flattened the images from the original 32x32x3 shape into a 1D vector of length 3072 (32 * 32 * 3).
- **Fully Connected Layers**:
  - The first fully connected layer has 64 neurons, followed by a ReLU activation.
  - The second fully connected layer is the output layer (classification head), which has 3 neurons (for the 3 classes).
- **Loss Function**:
  - I used **Cross-Entropy Loss** for the classification task.
- **Optimizer**:

    - The model is trained using the **Adam Optimizer** to adapt the learning rate during training.
- **Training Procedure**:
    - The MLP model was trained for **15 epochs**.
    - I logged **training and validation loss** and **accuracy** after each epoch.
- **Saving the Model**:
    - After training the MLP, I saved the trained model using torch.save() for later evaluation.

```
Epoch 1/15, Train Loss: 0.6921, Train Accuracy: 71.62%, Val Loss: 0.6163, Val Accuracy: 74.90%
Epoch 2/15, Train Loss: 0.5717, Train Accuracy: 77.35%, Val Loss: 0.6163, Val Accuracy: 75.50%
Epoch 3/15, Train Loss: 0.5115, Train Accuracy: 79.86%, Val Loss: 0.5524, Val Accuracy: 78.93%
Epoch 4/15, Train Loss: 0.4722, Train Accuracy: 81.41%, Val Loss: 0.5512, Val Accuracy: 78.80%
Epoch 5/15, Train Loss: 0.4361, Train Accuracy: 83.29%, Val Loss: 0.5767, Val Accuracy: 77.77%
Epoch 6/15, Train Loss: 0.4123, Train Accuracy: 84.35%, Val Loss: 0.5849, Val Accuracy: 78.60%
Epoch 7/15, Train Loss: 0.3645, Train Accuracy: 86.19%, Val Loss: 0.5378, Val Accuracy: 80.13%
Epoch 8/15, Train Loss: 0.3544, Train Accuracy: 86.82%, Val Loss: 0.5652, Val Accuracy: 79.27%
Epoch 9/15, Train Loss: 0.3249, Train Accuracy: 87.97%, Val Loss: 0.6039, Val Accuracy: 79.43%
Epoch 10/15, Train Loss: 0.2966, Train Accuracy: 88.80%, Val Loss: 0.5906, Val Accuracy: 79.57%
Epoch 11/15, Train Loss: 0.2803, Train Accuracy: 89.76%, Val Loss: 0.5945, Val Accuracy: 79.23%
Epoch 12/15, Train Loss: 0.2607, Train Accuracy: 90.51%, Val Loss: 0.5977, Val Accuracy: 80.03%
Epoch 13/15, Train Loss: 0.2411, Train Accuracy: 91.38%, Val Loss: 0.6423, Val Accuracy: 79.83%
Epoch 14/15, Train Loss: 0.2185, Train Accuracy: 92.09%, Val Loss: 0.6588, Val Accuracy: 77.90%
Epoch 15/15, Train Loss: 0.2116, Train Accuracy: 92.47%, Val Loss: 0.6594, Val Accuracy: 79.43%
```

7. Infer and Compare (2 points)

In this section, I will evaluate the performance of both the **Multilayer Perceptron (MLP)** and **Convolutional Neural Network (CNN)** models on the test dataset. I will also compare their performance, specifically focusing on accuracy, F1-score, and the trends observed in the training and validation performance over epochs.

MLP Model Evaluation:

The **Multilayer Perceptron (MLP)** model, despite being a simpler architecture, was trained and evaluated on the CIFAR-10 dataset.

- **Test Accuracy**:
    - The MLP achieved a test accuracy of **0.80**. This indicates that the model correctly classified 80% of the images in the test set. Given that the MLP is a fully connected model that treats the image as a 1D vector (losing spatial relationships), this performance is relatively good, though it is expected that a more advanced model like CNN would perform better.
- **Test F1-Score**:
    - The F1-score for the MLP was **0.8041**, which is a good metric because it balances precision and recall. This suggests that, while the model performs fairly well, it is still struggling with certain classes more than others (as reflected by the test accuracy and the F1-score being below 1.0). This may point to the model's difficulty in handling class imbalance or distinguishing between certain classes.

- **Training vs Validation Performance**:
    - **Training Accuracy**: In the training phase, the accuracy was steadily increasing. However, this growth was **not mirrored in the validation set**.
    - **Validation Accuracy**: The validation accuracy remained relatively constant throughout the training process, showing little to no improvement over the epochs.
        - **Possible Reasons for This Discrepancy**:
            - **Overfitting**: The increasing training accuracy without corresponding validation improvement is a strong indication of overfitting. The MLP may have learned the training data too well, but this has led to poor generalization to the validation data.
            - **Model Simplicity**: Since the MLP architecture does not have the spatial feature extraction capability that a CNN possesses, it may struggle more with the complexity of image data, leading to poorer performance on the validation set.
            - **Lack of Regularization**: The MLP may also suffer from a lack of techniques like dropout or early stopping, which could help mitigate overfitting.

CNN Model Evaluation:

The **Convolutional Neural Network (CNN)** model, on the other hand, is designed to capture spatial patterns in images, and this is expected to provide a better performance on image classification tasks.

- **Test Accuracy**:
    - The CNN model achieved a test accuracy of **0.86**, which is higher than the MLP model. This reflects the CNN's ability to automatically learn hierarchical features from the raw image data, capturing important spatial relationships that the MLP model cannot.
- **Test F1-Score**:
    - The CNN model achieved an F1-score of **0.86** on the test dataset, which indicates that the model has a balanced performance in terms of precision and recall across the different classes. This suggests that the CNN model is more robust and better at generalizing to new, unseen data.
- **Training vs Validation Performance**:
    - **Training Accuracy**: The accuracy of the CNN model increased steadily during training.
    - **Validation Accuracy**: Unlike the MLP, the CNN's validation accuracy also increased in parallel with training accuracy. This suggests that the CNN model was able to generalize well to unseen data and avoid overfitting.
        - **Reason for This Better Performance**:
            - **Better Generalization**: The CNN, with its convolutional and pooling layers, is better at learning spatial features from the

images. It can generalize better to unseen data, which is why both training and validation accuracy improved simultaneously.

- **Reduction of Overfitting**: The CNN is less prone to overfitting compared to the MLP due to its more complex architecture, which learns hierarchical patterns and reduces reliance on the raw pixel values. Additionally, the convolutional layers and pooling layers reduce the dimensionality and help the model learn more abstract features.

Comparison and Key Observations:

1. **Model Performance**:
   - **CNN** outperforms **MLP** in both test accuracy (0.86 vs 0.80) and F1-score (0.86 vs 0.8041). This is expected, as CNNs are specifically designed for image data and excel at capturing spatial features that are crucial for image classification tasks.
   - The **MLP**, while achieving good accuracy (80%), struggles to generalize effectively to the validation set, which may indicate overfitting.
2. **Training Dynamics**:
   - The **MLP** shows increasing training accuracy but stagnant validation accuracy, pointing to potential overfitting. The model is likely memorizing the training data but failing to generalize to unseen data.
   - The **CNN**, on the other hand, improves both training and validation accuracy. This suggests that the CNN is learning relevant features and generalizing better to unseen data, which is consistent with the typical performance of CNNs on image datasets.
3. **Impact of Model Architecture**:
   - The key difference between the two models lies in their architectures. The **CNN** uses convolutional layers, which allow it to capture local spatial features like edges, textures, and shapes in the image. These features are critical for image classification tasks, and the CNN can combine them hierarchically through deeper layers.
   - The **MLP**, being a fully connected neural network, does not capture spatial relationships and treats the image as a flattened vector. While it may work for simpler classification tasks, it is not ideal for image classification, where spatial structures play an important role.
4. **Generalization**:
   - The **CNN** demonstrates better generalization, as evidenced by the similar trends in both training and validation accuracy. This indicates that the model is learning from the data in a way that allows it to generalize well to new examples.
   - The **MLP**, with its increasing training accuracy but stagnant validation accuracy, shows clear signs of **overfitting**. This means that the model is not able to

generalize well to unseen data, which is a common issue with fully connected layers when applied to complex image data.

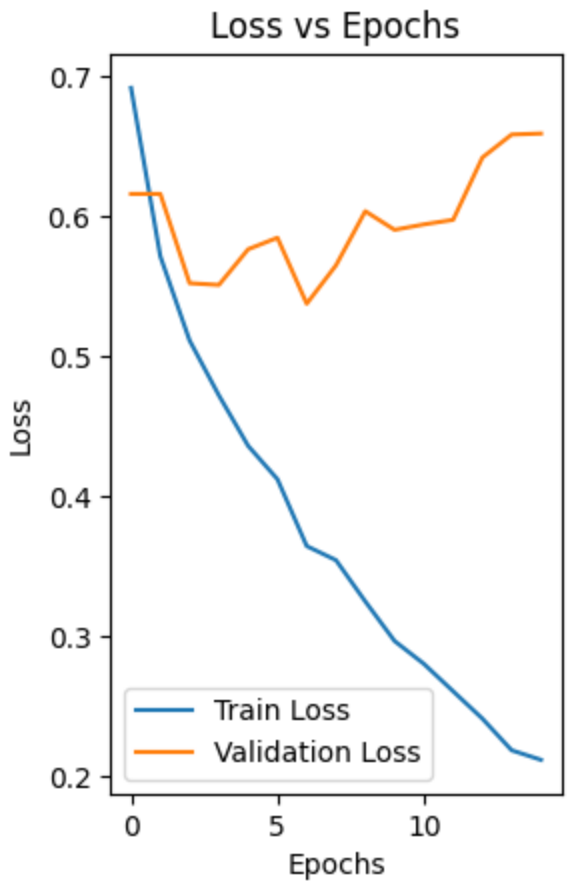
Conclusion:

The **CNN** model outperforms the **MLP** in all key metrics, including accuracy, F1-score, and validation generalization. The CNN's ability to capture spatial patterns through convolutional layers allows it to learn relevant features from the images, making it a better fit for the image classification task on the CIFAR-10 dataset.

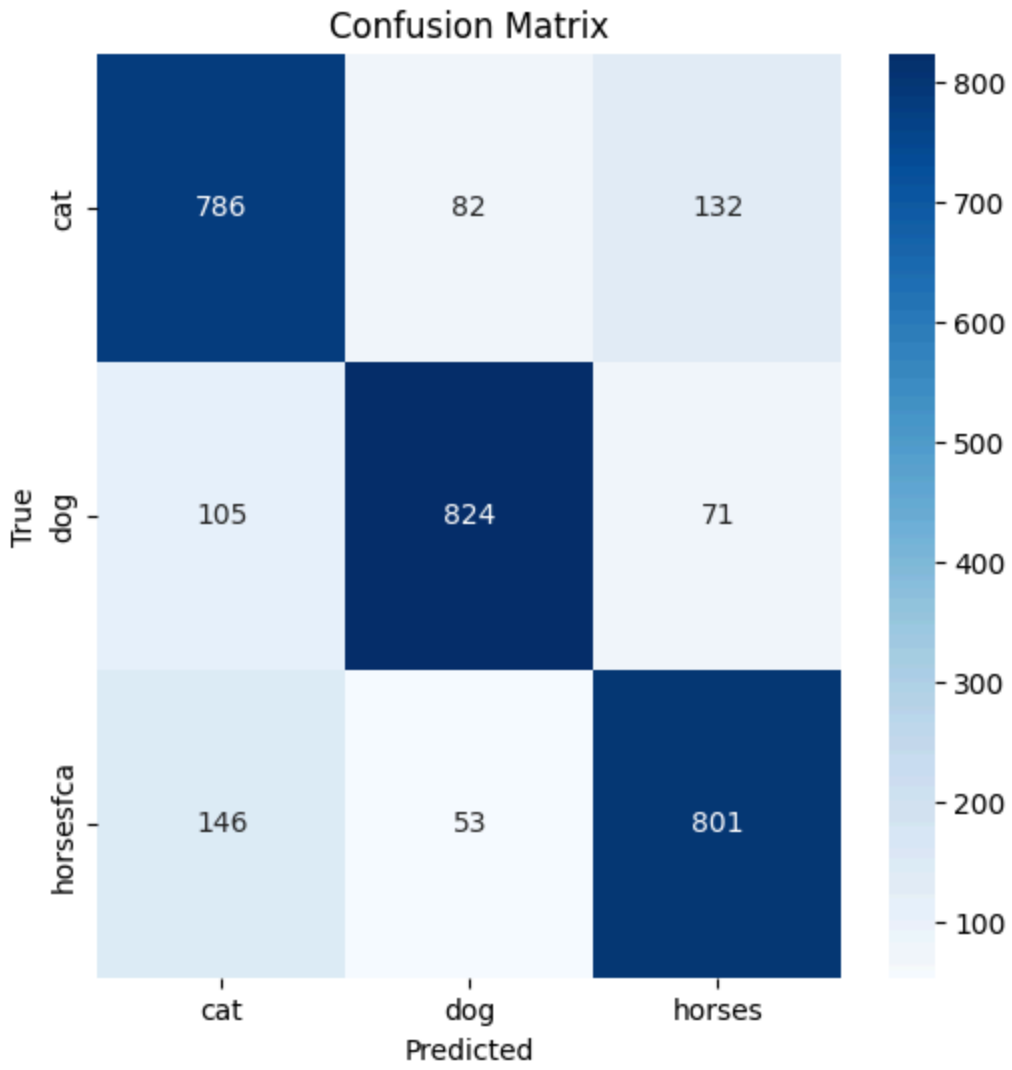
The **MLP**, although it achieved a respectable accuracy of 80%, struggled with generalization and was prone to overfitting. While it can still be useful for simpler tasks, it is clear that for image classification, particularly for datasets like CIFAR-10, **CNNs** provide a superior solution.

In future experiments, techniques such as data augmentation, dropout, or regularization methods could help improve the performance of the MLP model, but even then, CNNs would likely still outperform MLPs for this type of task.

-




```
Test Accuracy: 0.80
f1score 0.8041232844262677
```



Conclusion:

Through this process, I learned the importance of data preparation, model architecture design, and the evaluation of performance metrics like accuracy, F1-score, and confusion matrices. The CNN model outperformed the MLP model due to its ability to learn spatial features from images. The entire workflow, from data loading to model evaluation, helped me understand how to efficiently train and evaluate deep learning models for image classification tasks.