Name: Himanshu Kumar Assignment - 4
Roll no. : 2022215

Question 1

a) Forward pass of Convolutional Layer

1) Given
image dimension : $M \times N$ with P channels
kernel size : $K \times K$
Stride $S = 1$

No Padding

$\therefore$ The dimension of the resulting feature map $O_h \times O_w$ are given by

$$O_h = M - K + 1 \qquad O_w = N - K + 1$$

Thus output feature map dimension are:

$$\boxed{(M-K+1) \times (N-K+1)}$$

2) For each output pixel:

→ $K \times K$ weight for each of the P input channels, requiring $K^2 P$ multiplication.

→ The result are summed, requiring $K^2 \cdot P - 1$ addition

$\therefore$ total operation $= K^2 P + K^2 P - 1$
$= \boxed{2K^2 P - 1} \Rightarrow 2 \cdot K^2 P - 1$

For larg K, this simplied to
$$\boxed{O(K^2 \cdot P)}$$

3) with Q kernels, the total no. of operation is

Total operation $= Q \cdot (M-K+1)(N-K+1) \cdot (2K^2P-1)$

Big-O Complexity :

→ for General Case :

$$O(Q \cdot M \cdot N \cdot K^2 \cdot P)$$

→ Assuming min $(M, N) \geq> K$:

$$O(Q \cdot M \cdot N \cdot K^2 \cdot P)$$

b) 1) Assignment Step

→ Compute the distance of each data point to all k Centroids

→ Assign the data points to the nearest Centroid.

2) Update Step

→ Compute the mean of all points in each cluster.

→ Update the cluster centroid positions.

3) Determine the Optimal no. of Cluster. We will use Elbow method.

→ Compute the inertia for different value of k.

→ Plot inertia v/s k

→ optimal k: The elbow point where the decrease in inertia slow significantly.

4) Random initialization may lead to poor convergence:

→ K-mean minimize a non-convex objective function
→ May converge to local minimum

using K-Mean++ for initialization improve the likelihood of better results by ensuring centroids are well-seperated initially.

ML ASSIGNMENT-4

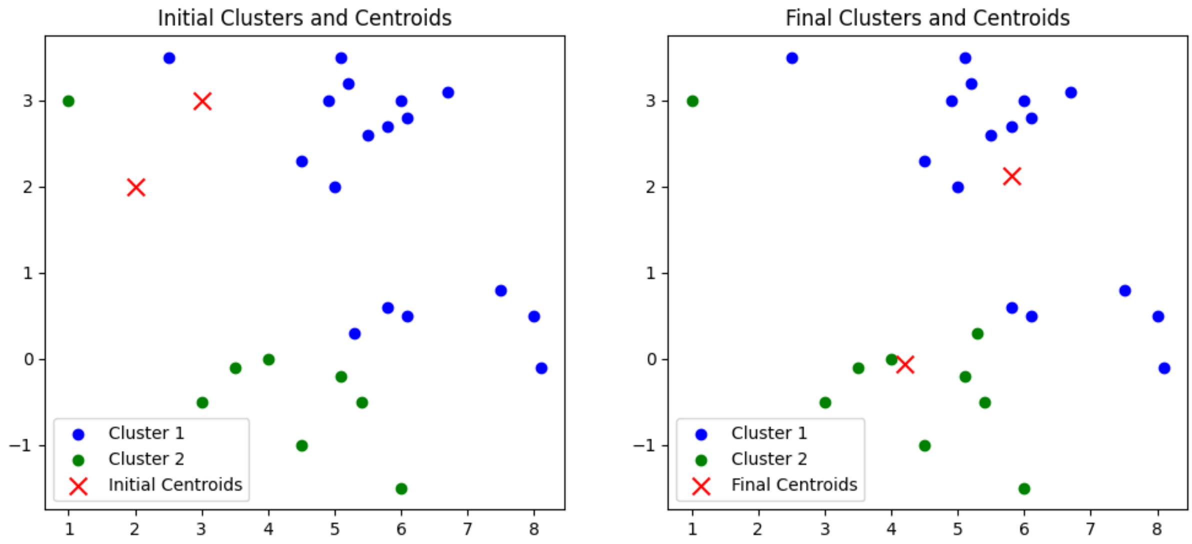
Question2

a)

```python
import numpy as np
import matplotlib.pyplot as plt

# Initial data points (X) and centroids
X = np.array([
    [5.1, 3.5], [4.9, 3.0], [5.8, 2.7], [6.0, 3.0], [6.7, 3.1], [4.5, 2.3], [6.1, 2.8],
    [5.2, 3.2], [5.5, 2.6], [5.0, 2.0],[8.0, 0.5], [7.5, 0.8], [8.1, -0.1], [2.5, 3.5],
    [1.0, 3.0], [4.5, -1.0], [3.0, -0.5], [5.1, -0.2], [6.0, -1.5], [3.5, -0.1],[4.0, 0.0],
    [6.1, 0.5], [5.4, -0.5], [5.3, 0.3], [5.8, 0.6]
])

# Initial centroids
initial_centroids = np.array([[3.0, 3.0], [2.0, 2.0]])
centroids = initial_centroids.copy()
k = centroids.shape[0]
max_iterations = 100
tolerance = 1e-4

def euclidean_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

def assign_clusters(X, centroids):
    clusters = []
    for x in X:
        distances = [euclidean_distance(x, centroid) for centroid in centroids]
        closest_centroid = np.argmin(distances)
        clusters.append(closest_centroid)
    return np.array(clusters)

def update_centroids(X, clusters, k):
    new_centroids = []
    for i in range(k):
        cluster_points = X[clusters == i]
        if len(cluster_points) > 0:
            new_centroid = np.mean(cluster_points, axis=0)
        else:
            new_centroid = centroids[i]
        new_centroids.append(new_centroid)
    return np.array(new_centroids)

def has_converged(old_centroids, centroids, tolerance):
    return np.all(np.linalg.norm(centroids - old_centroids, axis=1) < tolerance)

```

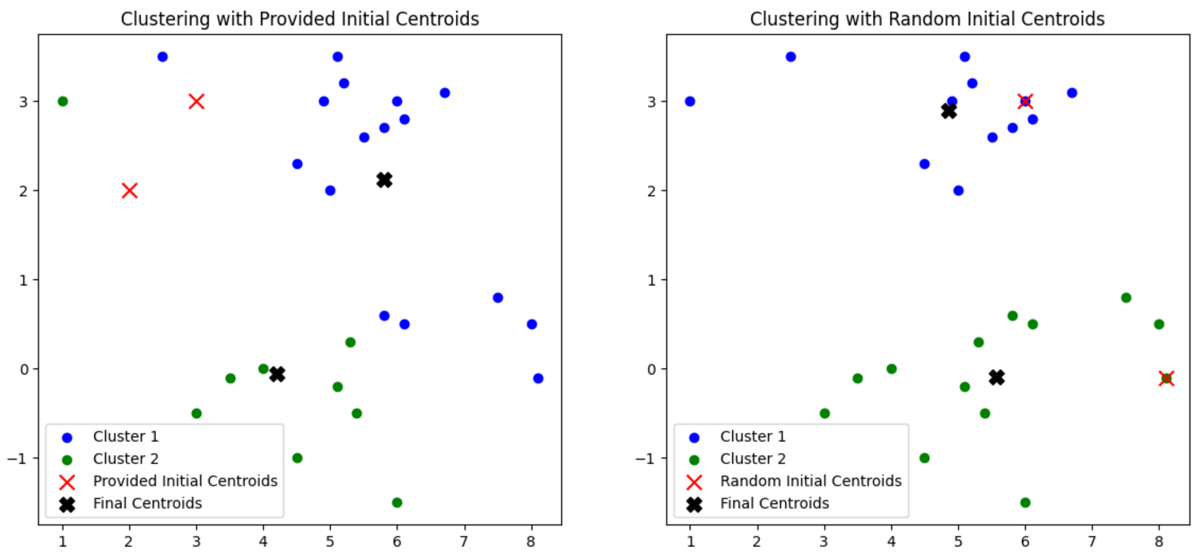
```python
# KMeans Algorithm
for iteration in range(max_iterations):
    # Step 1: Assignment Step
    clusters = assign_clusters(X, centroids)

    # Step 2: Update Step
    new_centroids = update_centroids(X, clusters, k)

    # Step 3: Convergence Check
    if has_converged(centroids, new_centroids, tolerance):
        print(f"Converged after {iteration + 1} iterations.")
        break

    # Update centroids for the next iteration
    centroids = new_centroids

# Plotting
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot initial clusters
ax1.scatter(X[initial_clusters == 0, 0], X[initial_clusters == 0, 1], color='blue', label='Cluster 1')
ax1.scatter(X[initial_clusters == 1, 0], X[initial_clusters == 1, 1], color='green', label='Cluster 2')
ax1.scatter(initial_centroids[:, 0], initial_centroids[:, 1], color='red', marker='x', s=100, label='Initial Centroids')
ax1.set_title('Initial Clusters and Centroids')
ax1.legend()

# Plot final clusters
ax2.scatter(X[clusters == 0, 0], X[clusters == 0, 1], color='blue', label='Cluster 1')
ax2.scatter(X[clusters == 1, 0], X[clusters == 1, 1], color='green', label='Cluster 2')
ax2.scatter(centroids[:, 0], centroids[:, 1], color='red', marker='x', s=100, label='Final Centroids')
ax2.set_title('Final Clusters and Centroids')
ax2.legend()

plt.show()

# Print final centroid values
print("Final centroids:\n", centroids)
```

b)



Converged after 3 iterations.
Final centroids:
[[5.8 2.125]
[4.2 -0.05555556]]

c)



Final centroids with provided initial centroids:

[[5.8   2.125   ]

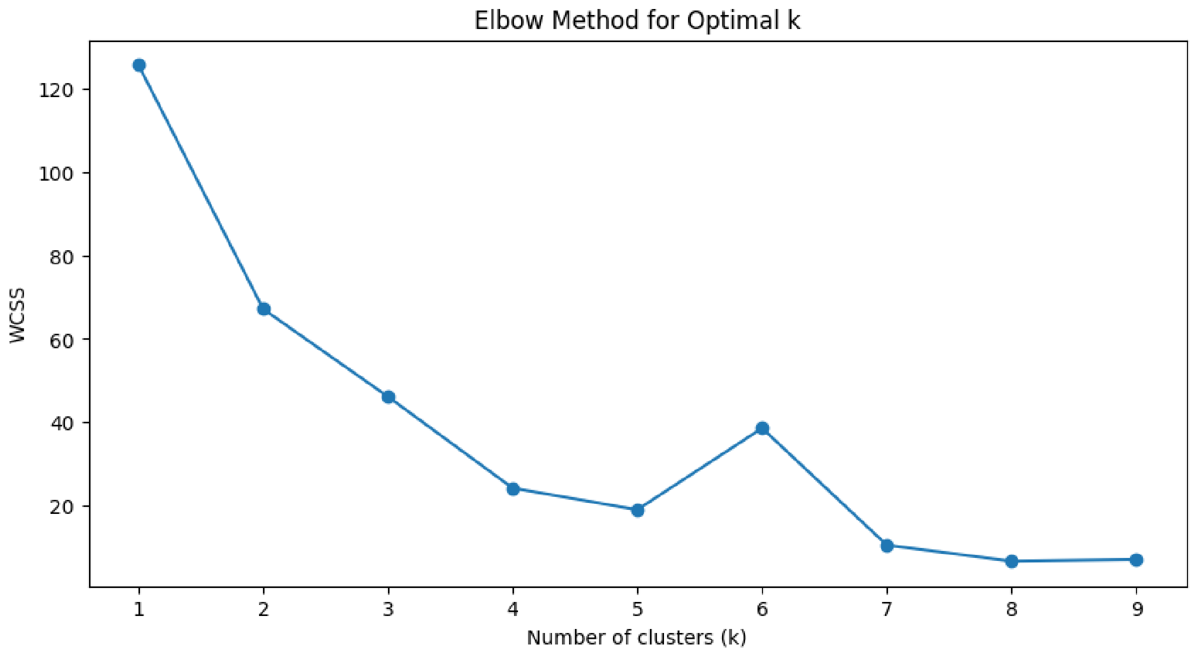
[4.2   -0.05555556]]

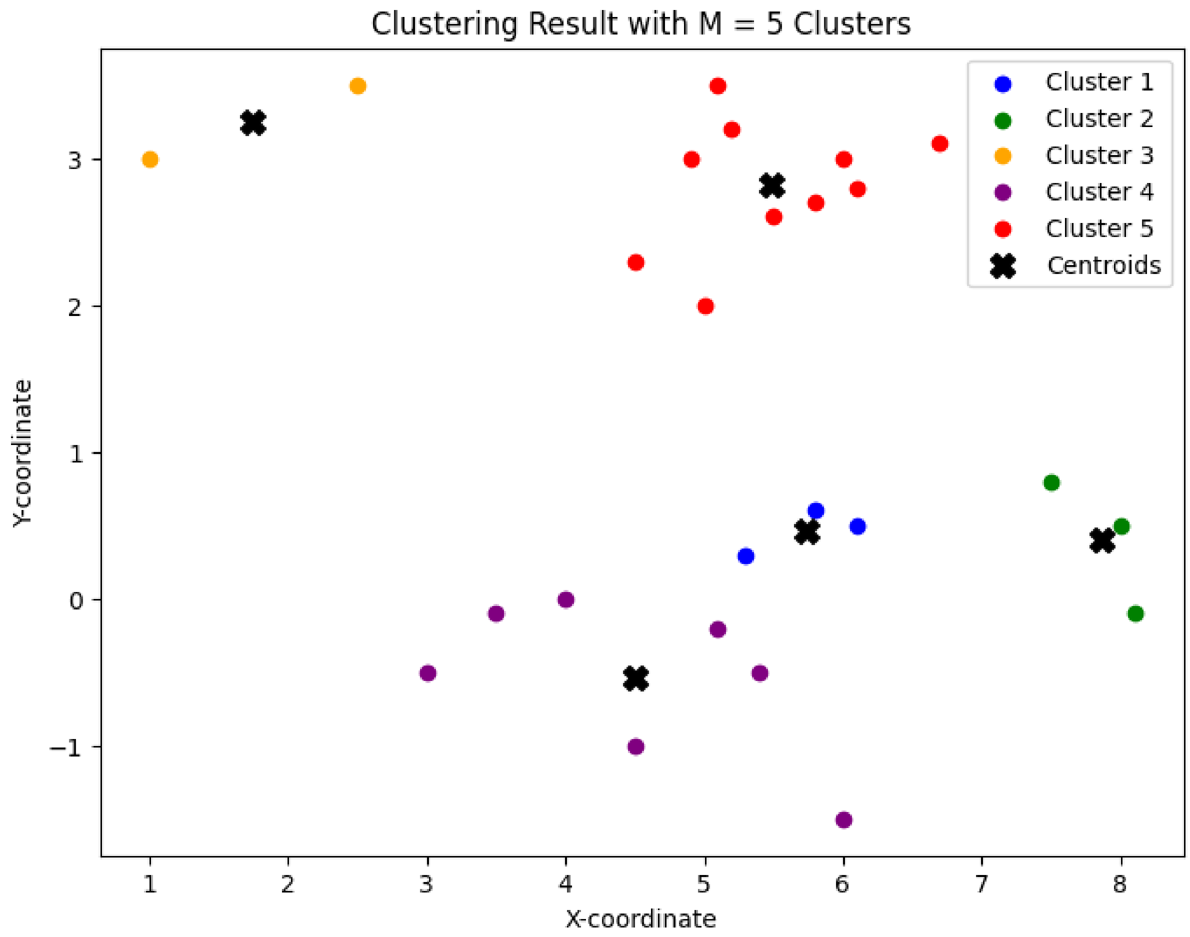
Final centroids with random initial centroids:

[[4.85833333 2.89166667]

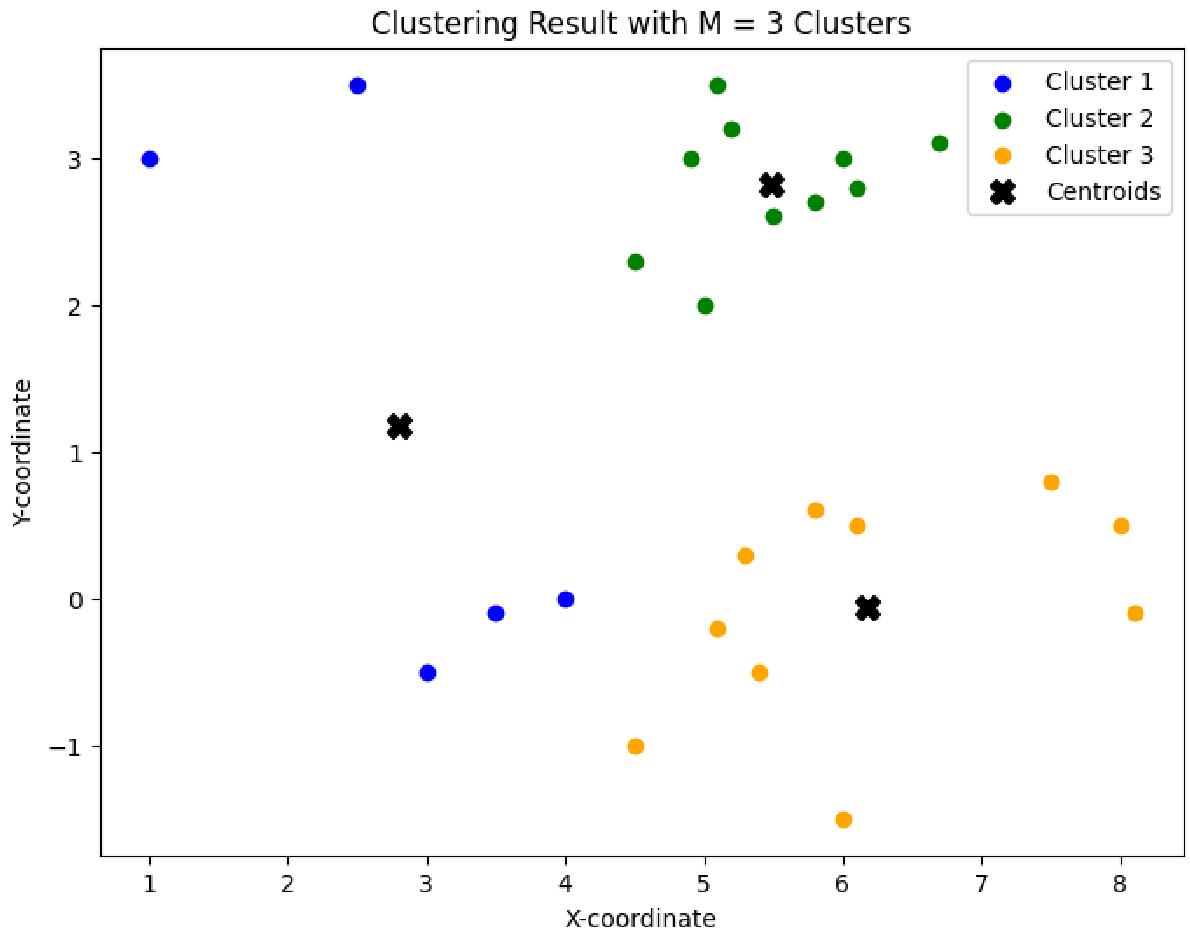
[5.56153846 -0.09230769]]

d)




Clustering Result with M = 5 Clusters
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Centroids
Y-coordinate
X-coordinate
3
2
1
0
−1
1
2
3
4
5
6
7
8

When m=5


Clustering Result with M = 3 Clusters
Cluster 1
Cluster 2
Cluster 3
Centroids
Y-coordinate
X-coordinate
3
2
1
0
−1
1
2
3
4
5
6
7
8

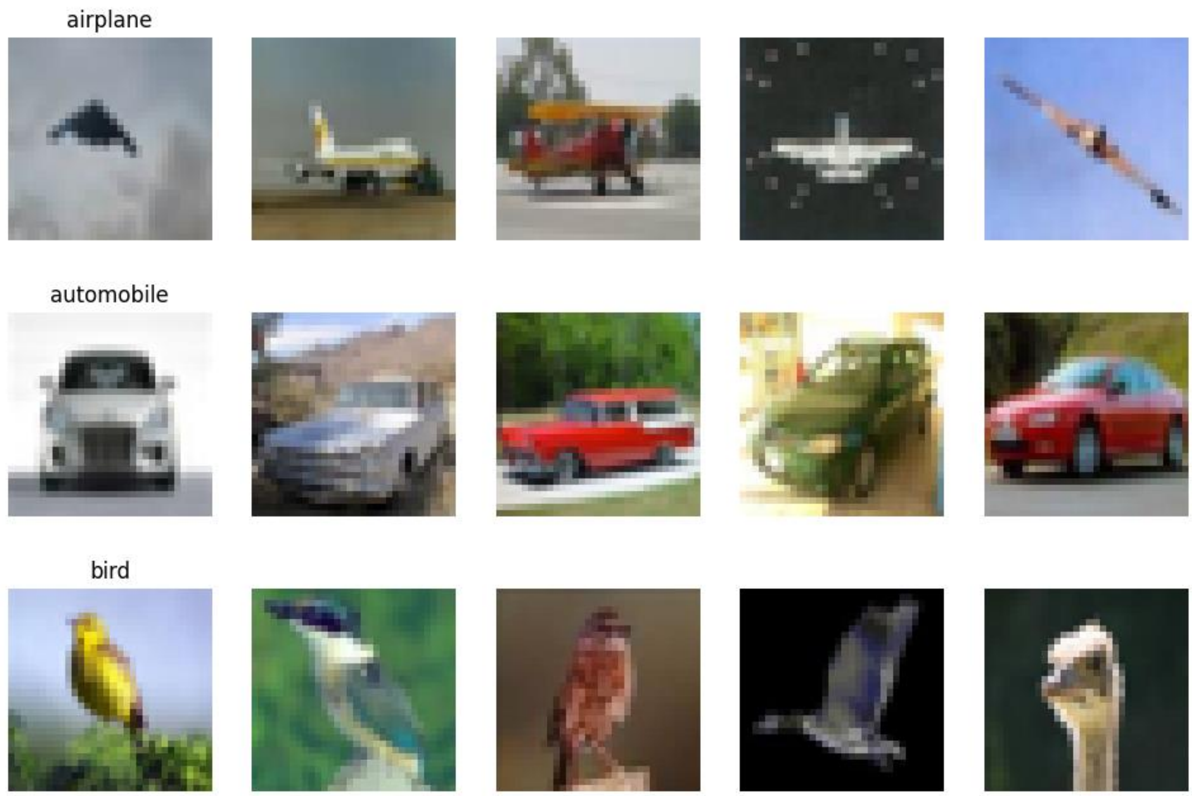
When m=3

Querstion-3

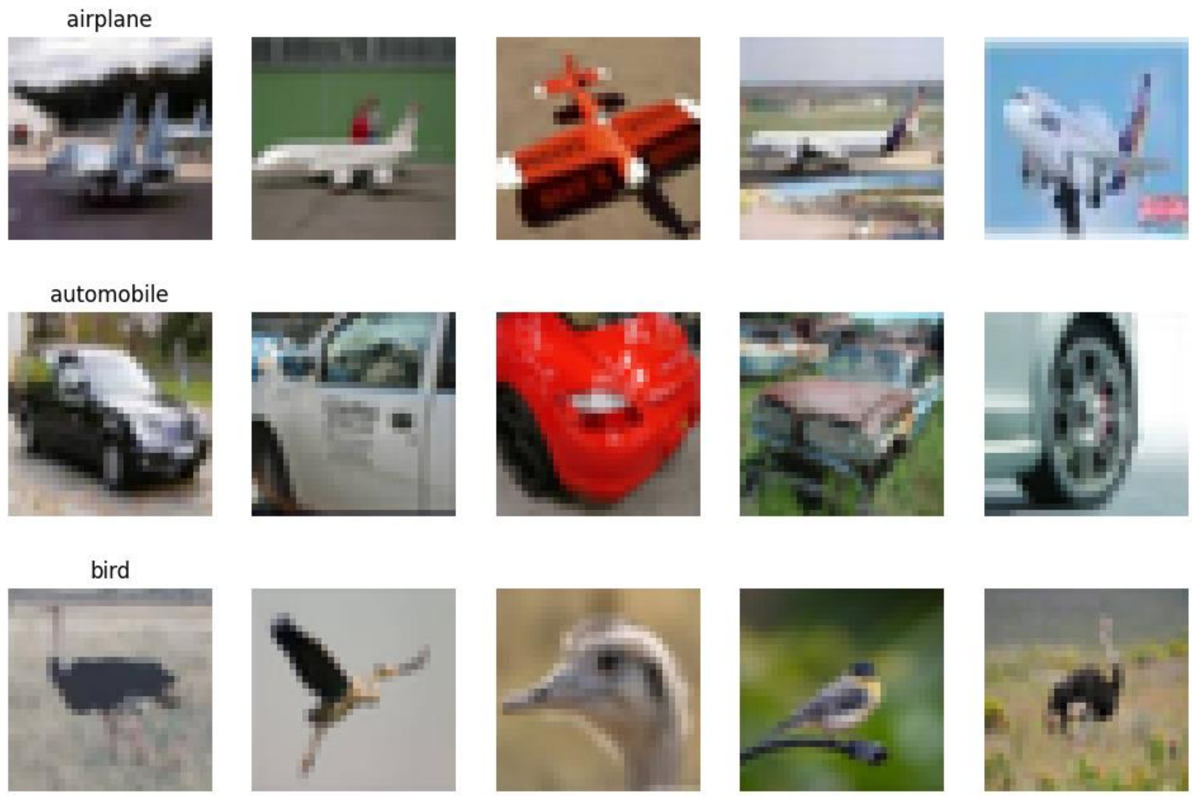
a)

```python
import torch
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
from sklearn.model_selection import train_test_split
import numpy as np

# Define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
dataset = datasets.CIFAR10(root='./data', train=True, transform=transform, download=True)

# Choose 3 classes (e.g., 0: Airplane, 1: Automobile, 2: Bird)
selected_classes = [0, 1, 2]
indices = [i for i, (_, label) in enumerate(dataset) if label in selected_classes]

# Split indices into train and test
train_indices, val_indices = train_test_split(indices, test_size=0.2, stratify=[dataset.targets[i] for i in indices])

# Define custom datasets
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, indices):
        self.dataset = dataset
        self.indices = indices

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        return self.dataset[self.indices[idx]]

train_dataset = CustomDataset(dataset, train_indices)
val_dataset = CustomDataset(dataset, val_indices)

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Test dataset (subset from original test data)
test_dataset = datasets.CIFAR10(root='./data', train=False, transform=transform, download=True)
test_indices = [i for i, (_, label) in enumerate(test_dataset) if label in selected_classes]
test_dataset = CustomDataset(test_dataset, test_indices)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

b) Training Dataset:



Validation Dataset:



c)

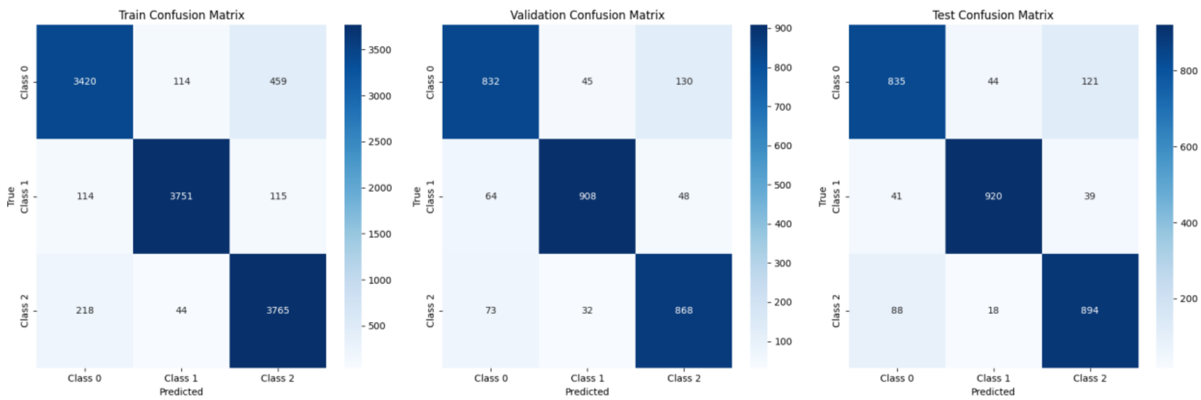
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=5, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=3)
        self.fc1 = nn.Linear(512, 16)
        self.fc2 = nn.Linear(16, 3)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)  # Flatten the output
        # print(x.shape)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

d)

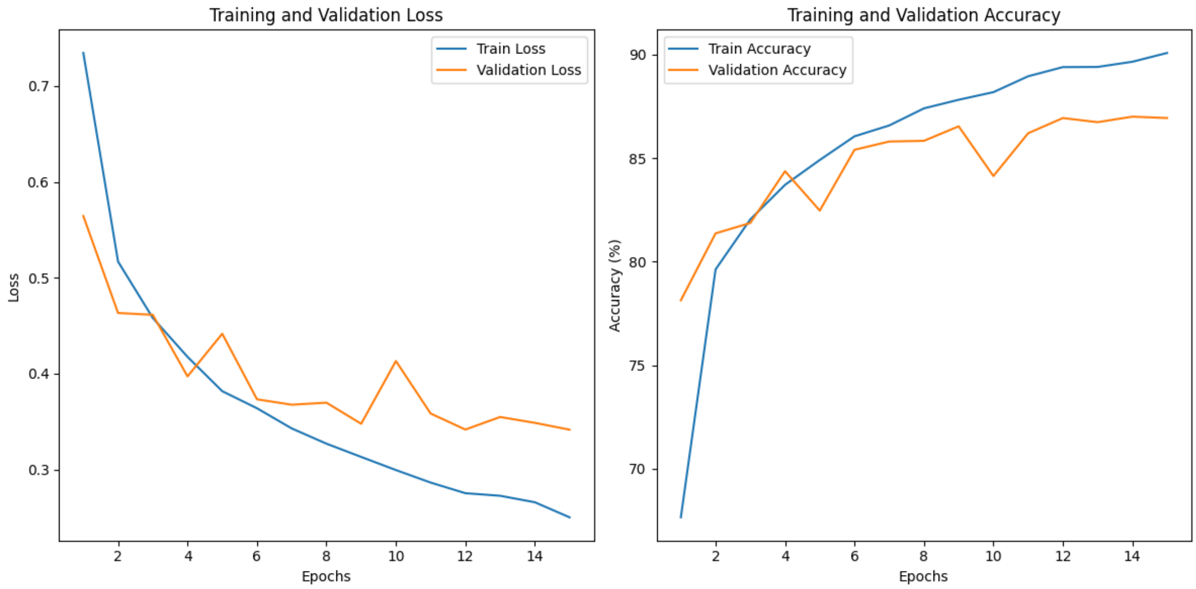
Epoch 1/15: Train Loss: 0.7344, Train Accuracy: 67.66% | Val Loss: 0.5643, Val Accuracy: 78.13%
Epoch 2/15: Train Loss: 0.5168, Train Accuracy: 79.62% | Val Loss: 0.4633, Val Accuracy: 81.37%
Epoch 3/15: Train Loss: 0.4581, Train Accuracy: 82.04% | Val Loss: 0.4613, Val Accuracy: 81.87%
Epoch 4/15: Train Loss: 0.4175, Train Accuracy: 83.71% | Val Loss: 0.3972, Val Accuracy: 84.37%
Epoch 5/15: Train Loss: 0.3817, Train Accuracy: 84.91% | Val Loss: 0.4417, Val Accuracy: 82.47%
Epoch 6/15: Train Loss: 0.3639, Train Accuracy: 86.05% | Val Loss: 0.3733, Val Accuracy: 85.40%

Epoch 7/15: Train Loss: 0.3429, Train Accuracy: 86.58% | Val Loss: 0.3676, Val Accuracy: 85.80%
Epoch 8/15: Train Loss: 0.3270, Train Accuracy: 87.40% | Val Loss: 0.3698, Val Accuracy: 85.83%
Epoch 9/15: Train Loss: 0.3132, Train Accuracy: 87.82% | Val Loss: 0.3477, Val Accuracy: 86.53%
Epoch 10/15: Train Loss: 0.2995, Train Accuracy: 88.18% | Val Loss: 0.4132, Val Accuracy: 84.13%
Epoch 11/15: Train Loss: 0.2864, Train Accuracy: 88.95% | Val Loss: 0.3584, Val Accuracy: 86.20%
Epoch 12/15: Train Loss: 0.2754, Train Accuracy: 89.39% | Val Loss: 0.3417, Val Accuracy: 86.93%
Epoch 13/15: Train Loss: 0.2727, Train Accuracy: 89.40% | Val Loss: 0.3548, Val Accuracy: 86.73%
Epoch 14/15: Train Loss: 0.2659, Train Accuracy: 89.65% | Val Loss: 0.3487, Val Accuracy: 87.00%
Epoch 15/15: Train Loss: 0.2502, Train Accuracy: 90.08% | Val Loss: 0.3416, Val Accuracy: 86.93%
Model saved to: model\cnn_model.pth

e)



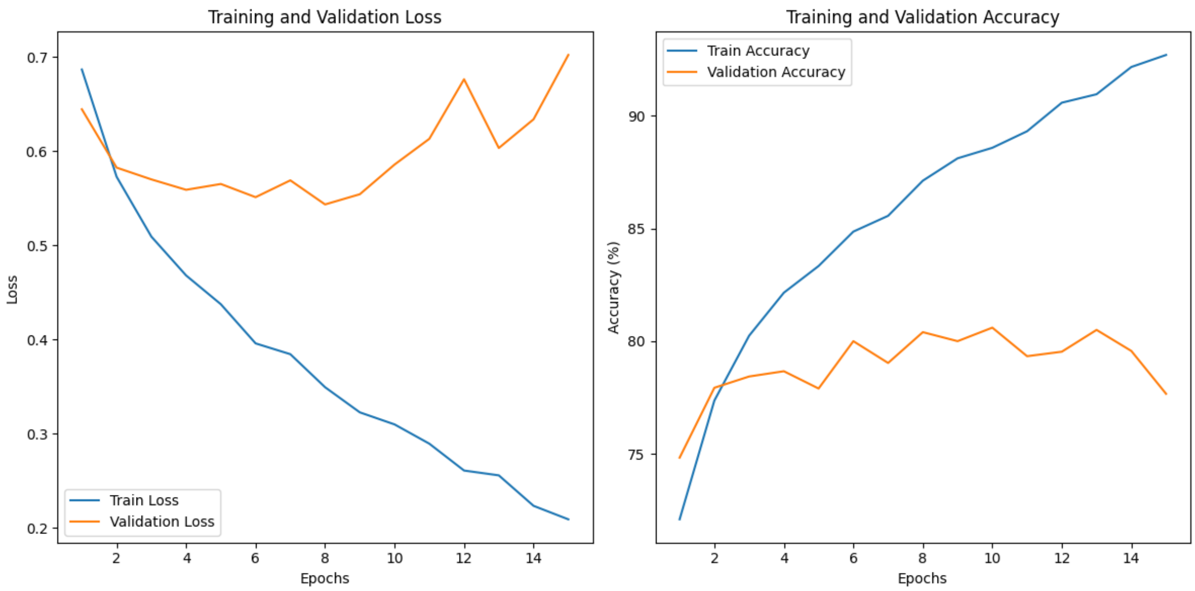
CNN Test Accuracy: 88.30%
CNN Test F1-Score: 0.8831



f)

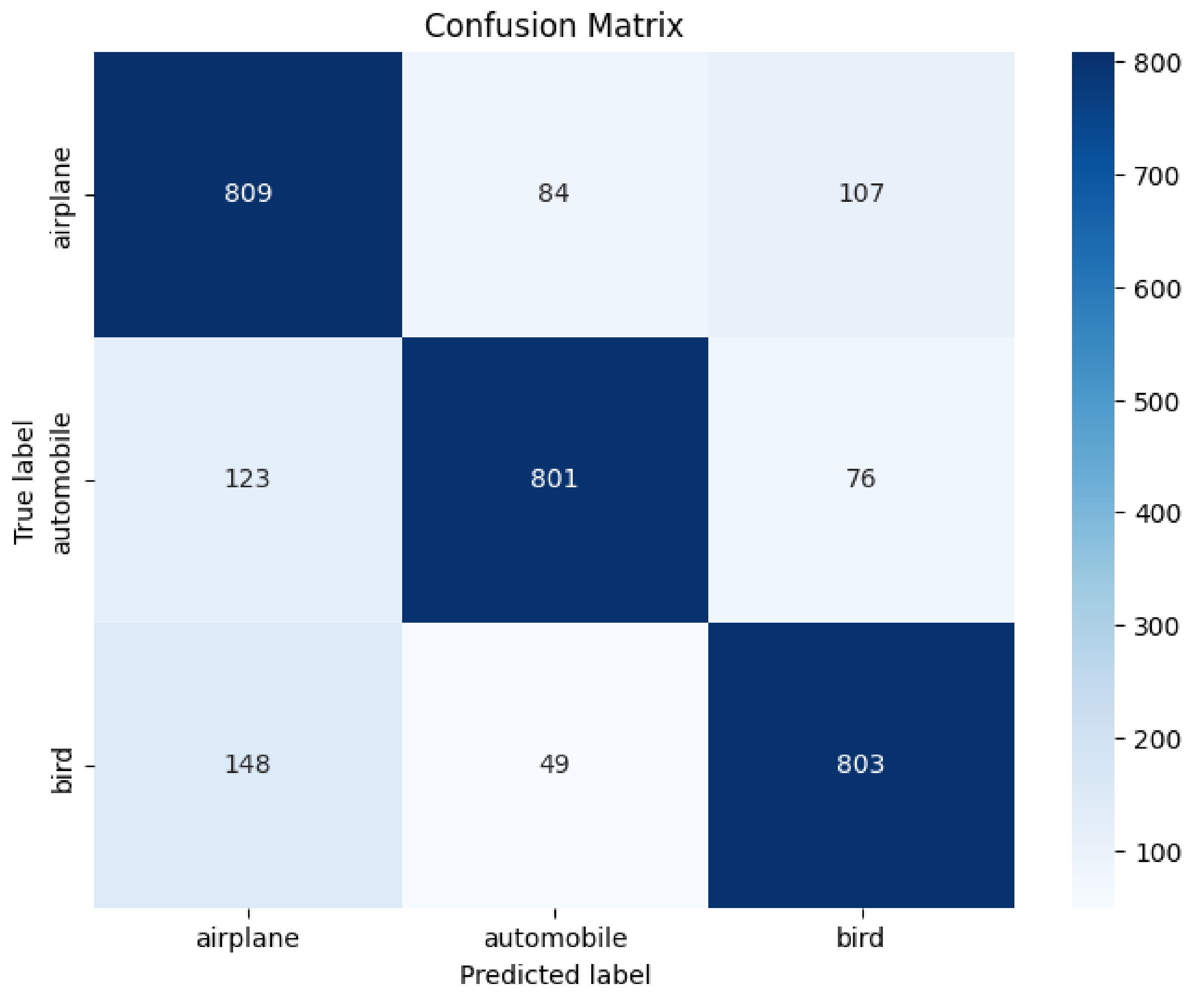
Epoch 1/15: Train Loss: 0.6868, Train Accuracy: 72.10% | Val Loss: 0.6446, Val Accuracy: 74.83%
Epoch 2/15: Train Loss: 0.5729, Train Accuracy: 77.37% | Val Loss: 0.5826, Val Accuracy: 77.93%
Epoch 3/15: Train Loss: 0.5094, Train Accuracy: 80.24% | Val Loss: 0.5701, Val Accuracy: 78.43%
Epoch 4/15: Train Loss: 0.4681, Train Accuracy: 82.15% | Val Loss: 0.5591, Val Accuracy: 78.67%
Epoch 5/15: Train Loss: 0.4375, Train Accuracy: 83.33% | Val Loss: 0.5653, Val Accuracy: 77.90%
Epoch 6/15: Train Loss: 0.3960, Train Accuracy: 84.86% | Val Loss: 0.5512, Val Accuracy: 80.00%
Epoch 7/15: Train Loss: 0.3845, Train Accuracy: 85.56% | Val Loss: 0.5691, Val Accuracy: 79.03%
Epoch 8/15: Train Loss: 0.3494, Train Accuracy: 87.12% | Val Loss: 0.5435, Val Accuracy: 80.40%
Epoch 9/15: Train Loss: 0.3227, Train Accuracy: 88.11% | Val Loss: 0.5544, Val Accuracy: 80.00%
Epoch 10/15: Train Loss: 0.3099, Train Accuracy: 88.58% | Val Loss: 0.5859, Val Accuracy: 80.60%
Epoch 11/15: Train Loss: 0.2895, Train Accuracy: 89.31% | Val Loss: 0.6132, Val Accuracy: 79.33%
Epoch 12/15: Train Loss: 0.2609, Train Accuracy: 90.58% | Val Loss: 0.6765, Val Accuracy: 79.53%
Epoch 13/15: Train Loss: 0.2558, Train Accuracy: 90.95% | Val Loss: 0.6034, Val Accuracy: 80.50%

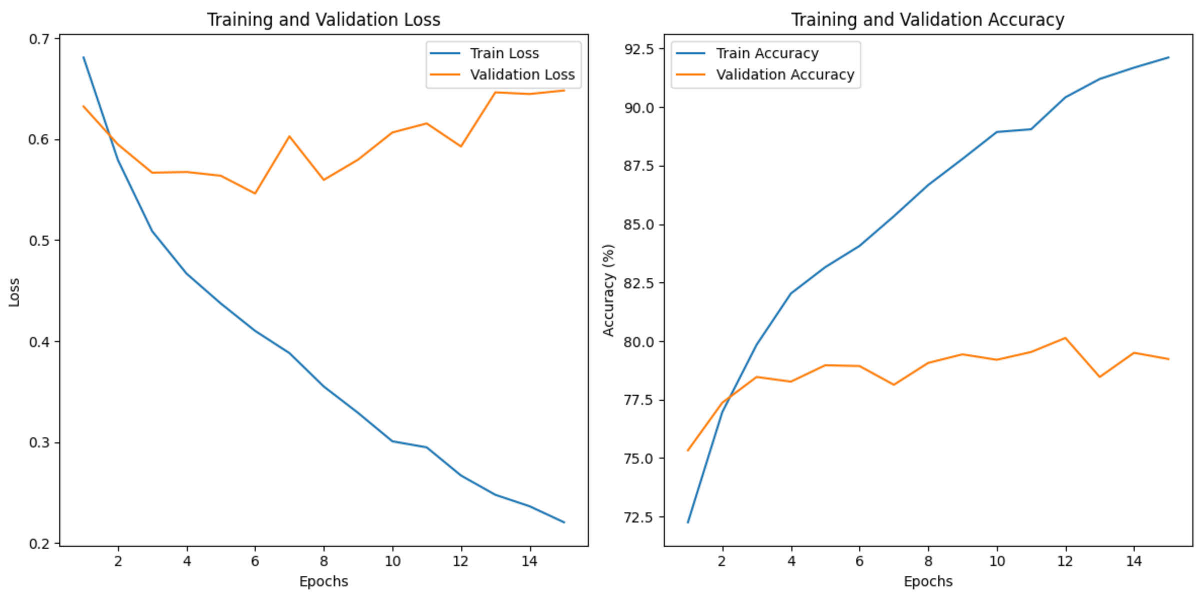
Epoch 14/15: Train Loss: 0.2234, Train Accuracy: 92.16% | Val Loss: 0.6340, Val Accuracy: 79.57%
Epoch 15/15: Train Loss: 0.2091, Train Accuracy: 92.69% | Val Loss: 0.7023, Val Accuracy: 77.67%
Model saved to: model\cnn_model.pth
Test Accuracy (MLP): 79.20%
Test F1-Score (MLP): 0.7927
Performance Comparison:
MLP Test Accuracy: 79.20%, MLP Test F1-Score: 0.7927



g)

Test Accuracy (MLP): 80.43%
Test F1-Score (MLP): 0.8050
Performance Comparison:
MLP Test Accuracy: 80.43%, MLP Test F1-Score: 0.8050





In test accuracy and the F1 score, the CNN model beats the MLP model, which suggests the fact that convolutional layers are essential for success with image data. The exclusive benefit of spatial relationship capture lies with the CNN-that is not the architecture of the MLP, which ignores them.

Nevertheless, MLP model was not bad in performance overall but is not comparable with CNN when image classification task is concerned. The performance differences again stress the need for different architectures relied upon different applications, which in this case is the CNN for image task-type applications.