IMAGE CLASSIFICATION ON CIFAR-10 DATASET.

Done by: Group 10
Members: Ataish Nehra (NUID: 002919899),
Sameer Prasad Koppolu (NUID: 002752920)

1. Overview:

Image Classification is the categorization of images into one of several classes or categories based on the object present in the images. It is implemented using Supervised Machine Learning models where the models are trained on a set of labeled images (training set) and then are used to predict the labels of the images that were not used in the training process but whose labels are known (test set). The predictions on this test set are then used to evaluate the model's performance.

In this context, we have implemented 4 Supervised Machine Learning Models namely, the Random Forest Classifier, the Support Vector Machine with Kernel, the Softmax Classifier, and the ResNet for image classification on the CIFAR-10 dataset, and their respective performances are discussed.

2. Data Description:

Collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton, the CIFAR-10 dataset is a subset of the '80 million tiny images' dataset. It consists of small 32x32 images with 3 channels (RGB channels), where each image is labeled with one of ten classes. The classes are 'Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', and 'Truck'. Each of these class values is represented as a number from 0 to 9 for each image where 0 corresponds to 'Airplane', 1 corresponds to 'Automobile', and so on and so forth. All of the classes are mutually exclusive in order to avoid confusion. This means that any image that is not a large truck is part of the 'Automobile' class and not the 'Truck' class. The 'Truck' class only has images of large trucks. Additionally, the number of images in each class is equal. This holds true in both the training set and the test set.

Let’s look at some of the images in the dataset.

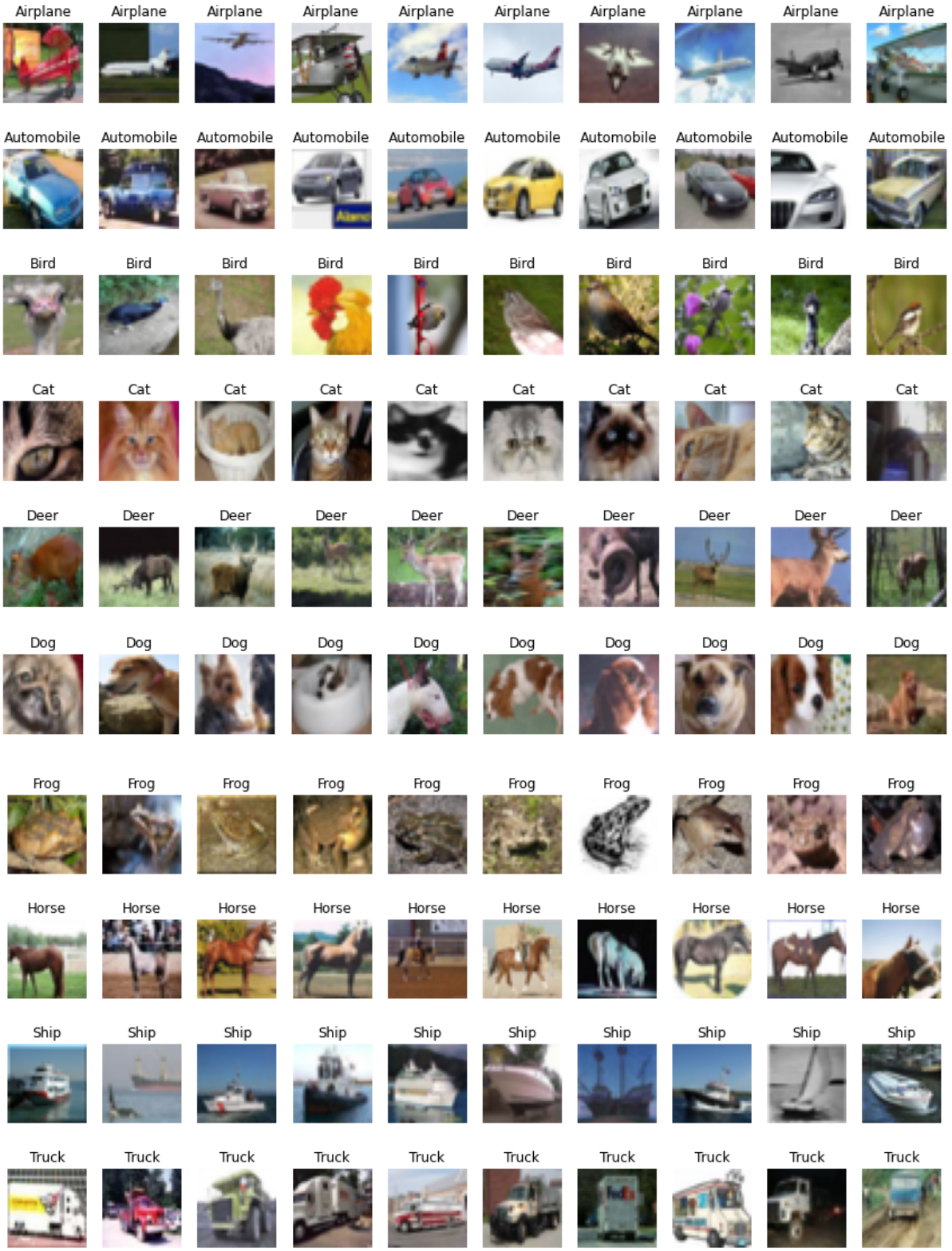


Fig 2.1: Some of the Images in the Dataset with their corresponding Class

Each pixel in each image has a value between 0 to 255. This value indicates the color intensity with respect to the channel that the pixel is present in (i.e. color intensity of

Red, Green, or Blue). These pixel values become the input features to the machine learning models in order to classify the image.

When the data is loaded, it is split into a Training Set of 50000 images and a Testing Set of 10000 images. The data in the training and test sets are represented as 3 Dimensional Arrays. As mentioned above, each image has a height and width of 32x32 with 3 channels which are the RGB channels. This indicates that each image is a 3-dimensional tensor with 32 rows, 32 columns, and 3 color channels.

Let's look at the count of images in each class in the Training Dataset and the Test Dataset.

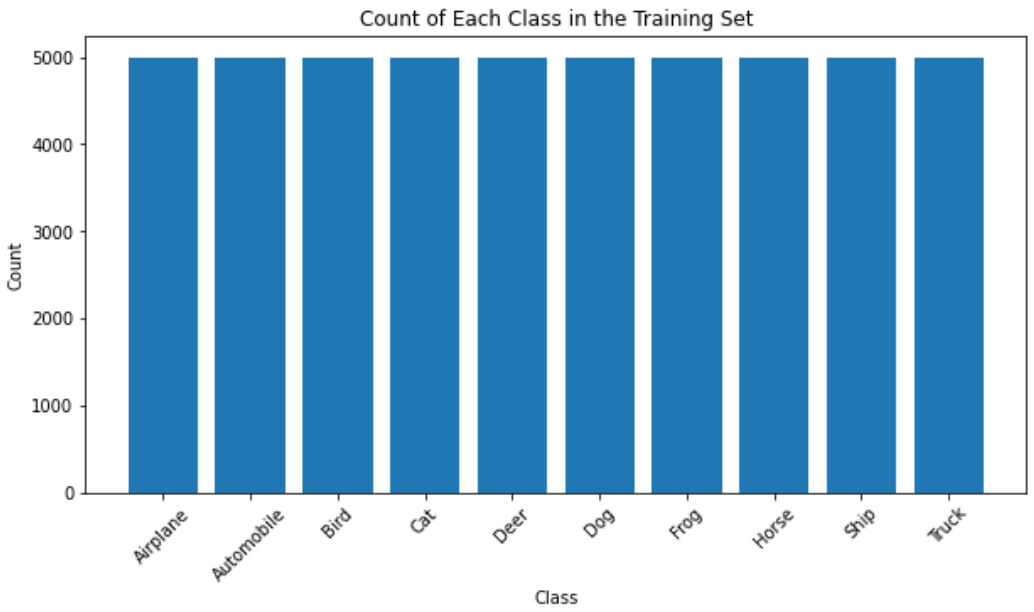


Fig 2.2: Count of images in each Class in the Training Set

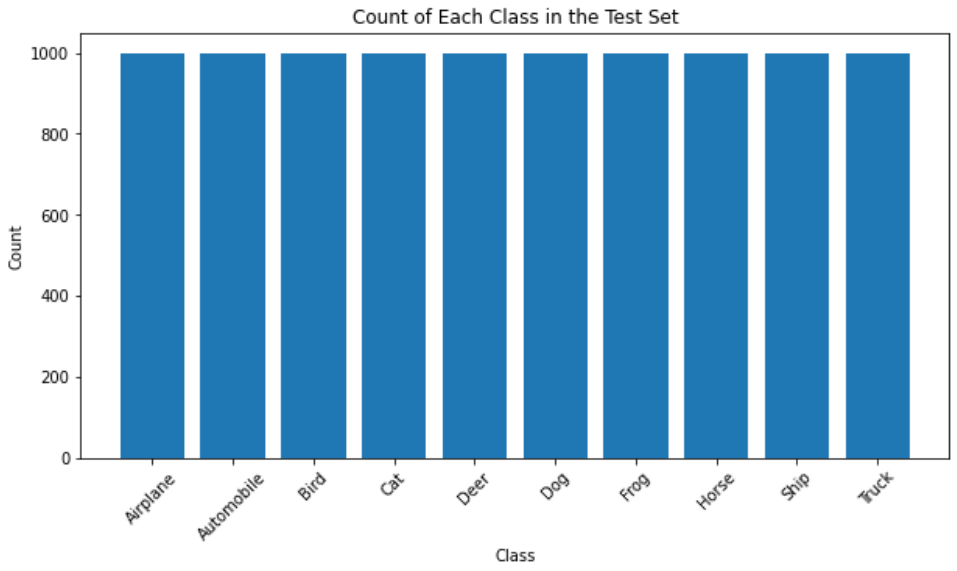


Fig 2.3: Count of images in each Class in the Training Set

3. Data Preprocessing:

A common step prior to training any of the models is to normalize the pixel values for each image in the Training Set and in the Test Set. To perform normalization, we simply divide each pixel value by 255. The reasons for this are as follows:

- The model performance improves as a result of normalization due to the decrease in scale and distribution differences of pixel values across the images.
- Normalization helps to prevent numerical instability which further helps in avoiding the vanishing gradient and exploding gradient problem.
- Additionally, Normalization ensures that convergence is achieved faster during training since the range of pixel values is between 0 to 1.

One additional step that is performed for the Random Forest Classifier, Support Vector Machine (SVM) with Kernel, and in the Softmax Classifier, is the flattening of the Training and Test datasets. Here each image gets converted to 2 dimensions with 3072 features (since 32*32*3 = 3072). Although flattening takes place in ResNet, it occurs much later in the neural network when we flatten the output feature map obtained from convolution that is to be used in the fully connected layers.

Additionally, we perform PCA in SVM with Kernel by reducing the number of features from 3072 to 200. This was done due to the fact that applying the Radial Basis Function (RBF) kernel on 3072 features turned out to be computationally expensive.

4. Models:

01. Softmax Classifier.

EXPLANATION:

The Softmax Classifier is a neural network algorithm that is used for multi-class classification. It is a variant of the logistic regression algorithm that is used for binary classification. Due to the presence of 10 different classes, the Softmax Classifier is used. Using the Softmax function, the classifier computes the probability distribution p for a given set of input features x and classes C. This is given as follows.

$$p(c|x) = \frac{e^{(w_c^T x)}}{\sum_j e^{(w_j^T x)}}$$

As stated earlier, we normalize the Training and Test sets that are to be used in the Softmax Classifier. The TensorFlow and Keras libraries are imported for building the neural network.

Next, the model is built using the Sequential() function and four dense layers. The first layer flattens the input image data into a 1-dimensional array, while the subsequent layers each have 256, 128, and 64 neurons respectively, with a rectified linear unit (ReLU) activation function at each layer. The output layer has 10 neurons, corresponding to the 10 classes in the CIFAR-10 dataset, and uses a softmax activation function to produce probabilities for each class for every image.

The model is then compiled using the compile() function with the Adam optimizer and sparse categorical cross-entropy loss function, which is suitable for multi-class classification problems with integer labels. The model is then trained on the normalized training set for 20 epochs, with a validation split of 0.2.

The model is evaluated on the normalized test set using the evaluate() function, which returns the test loss and accuracy. The model is then used to predict the labels of the test set using the predict() function, and the predictions are converted to class labels using np.argmax() function which is a part of the NumPy library.

The classification report, accuracy, and confusion matrix are printed to evaluate the performance of the model on the test set. The confusion matrix is displayed using matplotlib to show the number of correct and incorrect predictions for each class.

Finally, the change in training loss, training accuracy, validation loss, and validation accuracy across epochs is plotted using matplotlib. This helps to visualize how the model is learning during training and whether it is overfitting or underfitting.

Visualization of evaluation metrics:

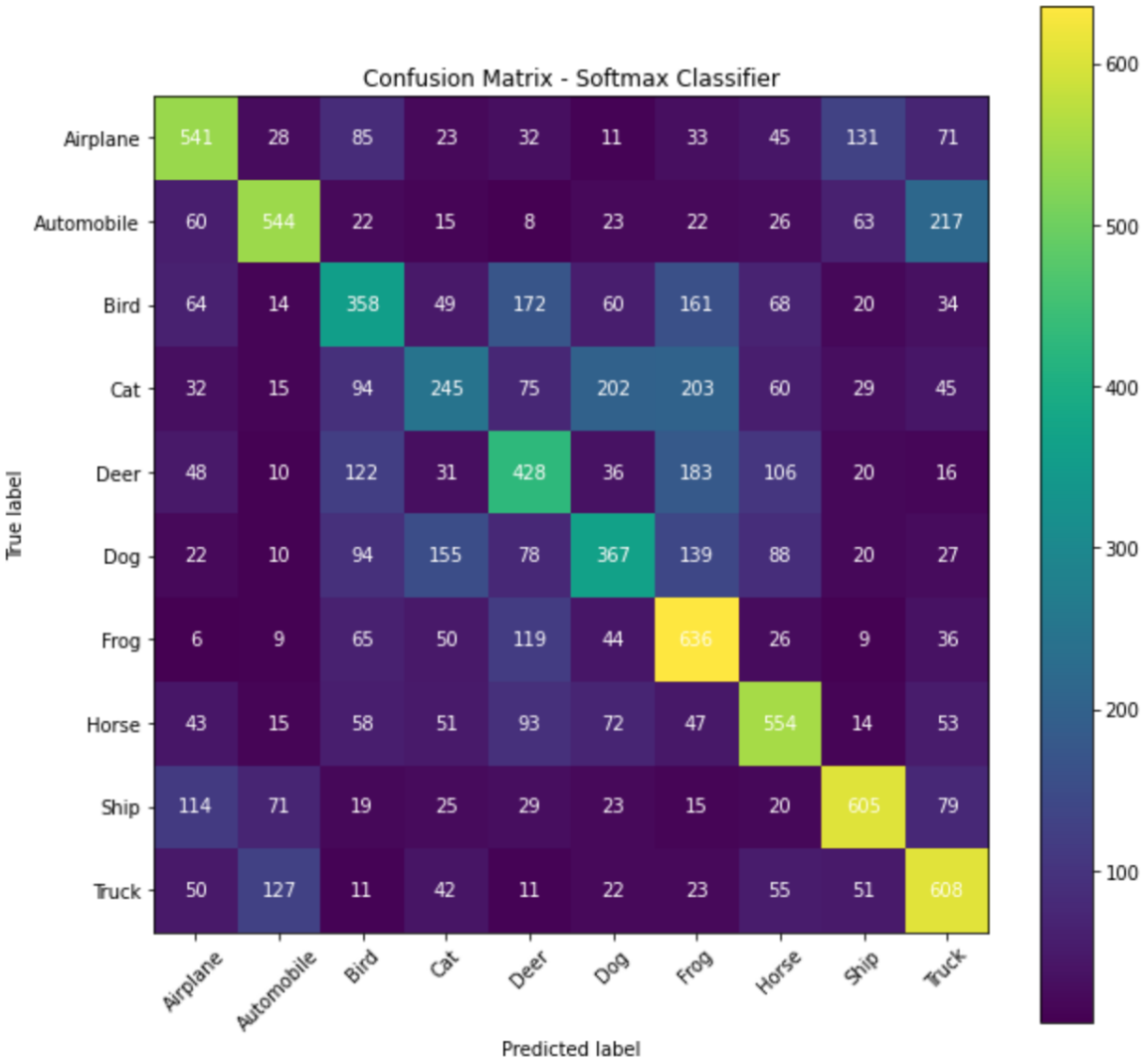


Fig 4.01.1: Confusion matrix for Softmax Classifier.

02. Random Forest.

EXPLANATION:

Widely used for Classification tasks such as the Image Classification task at hand, the Random Forest Classifier is an ensemble machine learning model that involves the working of many decision trees. Each of the decision trees performs classification on a bootstrapped dataset of the input while only using a subset of the features that are selected randomly. Once each tree performs classification on the observations in the input, the observation is classified into the class that the majority of the trees classify that observation into. In other words, the observation is assigned to the class that has the highest number of votes among all the trees.

In this project, the Random Forest Classifier on the CIFAR-10 dataset is implemented using the scikit-learn library. The code imports the necessary

libraries and loads the CIFAR-10 dataset. The labels are then converted to 1-D arrays, and the names of each label are stored in a list. The images are then normalized by dividing each pixel value by 255.0 and are also flattened to a vector of length 32x32x3=3072.

The Random Forest Classifier is then instantiated and trained on the training set. By default, the Random Forest consists of 100 trees. The model is then used to make predictions on the test set, and the classification report and accuracy score are printed. Additionally, a confusion matrix is generated and displayed using matplotlib.

Overall, the code loads the CIFAR-10 dataset, preprocesses the data, trains a Random Forest Classifier on the data, and evaluates the model's performance using the classification report, accuracy score, and confusion matrix.

Visualization of evaluation metrics:

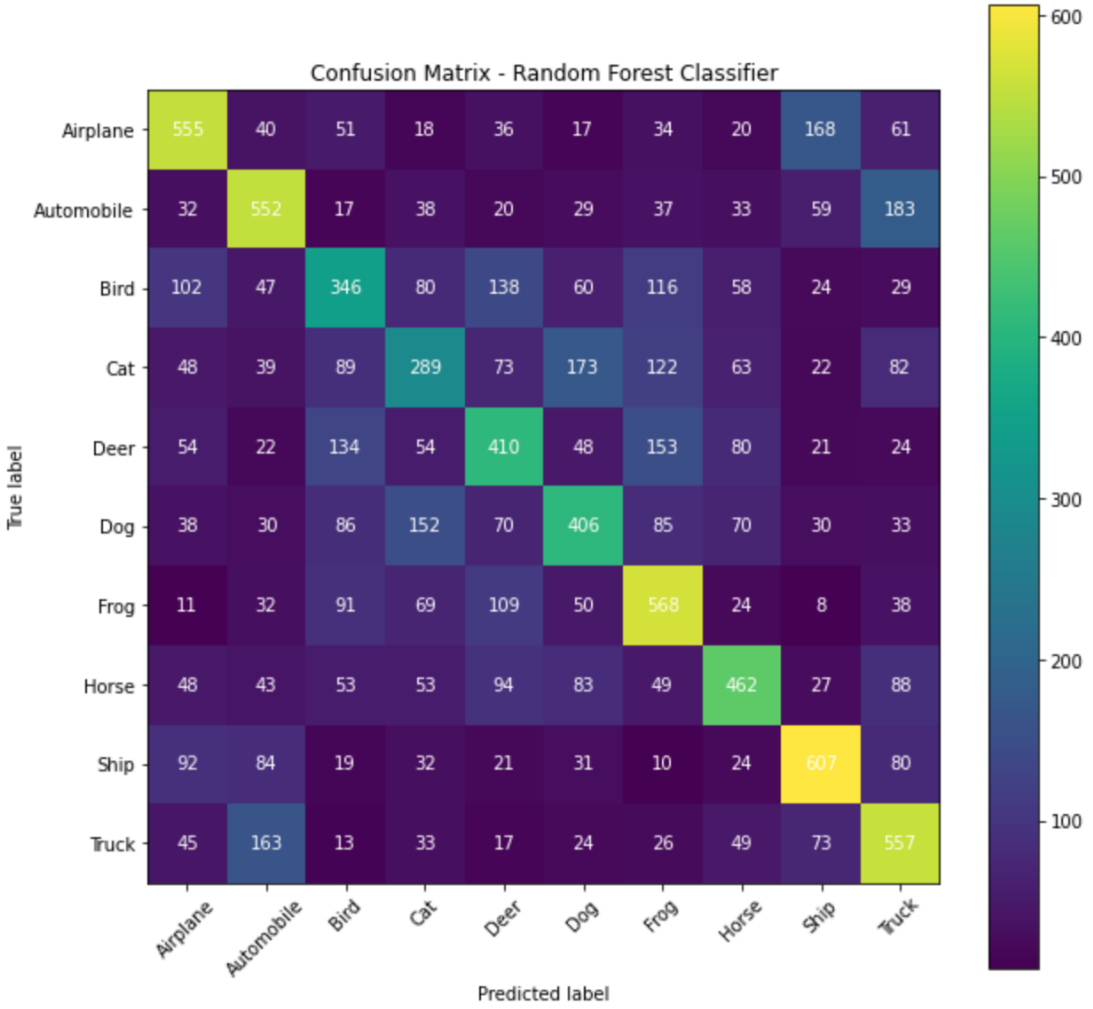


Fig 4.02.1: Confusion matrix for Random Forest.

03. SVM with Radial Basis Function Kernel.

EXPLANATION:

Support Vector Machines (SVM) is a machine learning algorithm used for classification and regression analysis. It is particularly useful in solving complex classification problems in datasets with multiple features. The algorithm works by finding the optimal hyperplane that separates the different classes of data points, which is done by maximizing the margin between the hyperplane and the closest data points, also known as support vectors. SVM is widely used in various fields, such as image classification, text classification, and bioinformatics.

For this project, we have implemented a kernel-based SVM, specifically the kernel used is the Radial Basis Function (RBF). In general, a kernel does a mapping of the input into a higher-dimensional feature space where a hyperplane can be drawn to classify the observations of the input. Kernel-based SVM is generally used when the input data is not linearly separable based on the classes. The RBF Kernel which is used in this project is a popularly used kernel for image classification tasks. The mapping function for the RBF kernel is based on a Gaussian distribution that is centered on each data point in the training set. Based on the Euclidean distance between any two data points corresponding representations in the feature space, the kernel function calculates how similar two data points in the feature space are.

The code provided applies the SVM algorithm to a dataset of images. First, the training and test sets are normalized and flattened into vectors of length 32*32*3 = 3072 to be used for SVM. However, due to computational issues with 3072 dimensions, Principal Component Analysis (PCA) is applied to reduce the number of dimensions to 200. This will help to avoid the curse of dimensionality, which refers to the increased complexity and difficulty of analysis that occurs when dealing with high-dimensional datasets. The RBF kernel is as follows.

$$K(x_i\ ,\ x_k)\ =\ e^{(-\gamma \sum_{j=1}^{p} (x_{ij}\ -\ x_{kj})^2\ )}$$

Here, gamma is a constant greater than zero and p is the number of features. Moreover, the 2 arguments to the kernel function are 2 observations in the input, and the kernel function is evaluated for all possible pairs of observations in the input.

After applying PCA, the SVM model is built and trained using the Radial Basis Function (RBF) kernel. The RBF kernel is a popular choice for SVM

because of its ability to model complex decision boundaries. The hyperparameters used in this code are C=10 and gamma='scale'. C is the regularization parameter, which determines the trade-off between the smoothness of the decision boundary and the number of misclassified training examples. Gamma is the kernel coefficient, which determines the shape of the decision boundary.

The trained model is then saved as a pickle file named "SVM.pickle". The pickle module in Python is used for serializing and de-serializing Python objects, allowing the SVM model to be saved and loaded for future use. The loaded model is used to make predictions on the test set, and the classification report is printed, along with the accuracy score on the test set. Finally, a confusion matrix is displayed to evaluate the performance of the SVM model. The confusion matrix shows the number of correct and incorrect predictions made by the model for each class, allowing for a more detailed analysis of the model's performance.

Visualization of evaluation metrics:

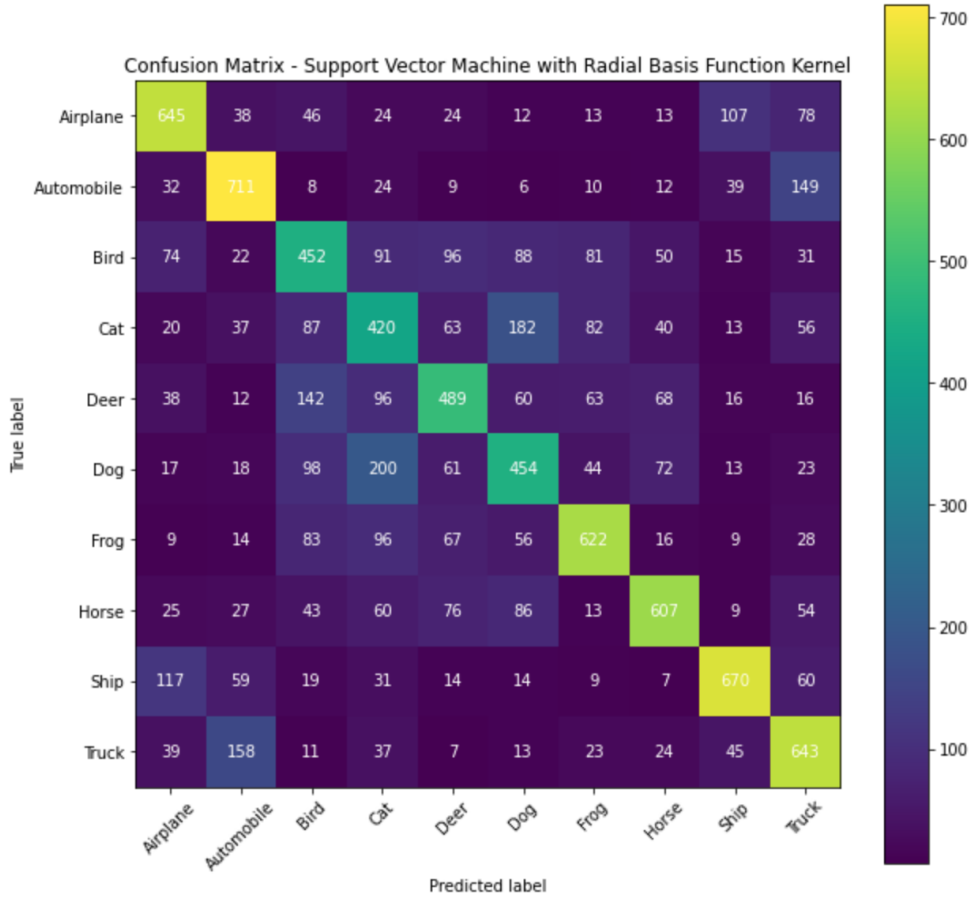


Fig 4.03.1: Confusion matrix for SVM with RBF Kernel.

04. ResNet (CNN).

EXPLANATION:

Our code implements a ResNet (Residual Network) model using Keras library for the CIFAR-10 dataset. The ResNet is a type of neural network architecture that is designed to overcome the degradation problem that arises as deep neural networks are trained. The code starts by importing the required libraries such as Keras, NumPy, Seaborn, and Matplotlib. The hyperparameters such as batch size, number of epochs, and number of classes are defined. The CIFAR-10 dataset is loaded and preprocessed by converting the class labels into one-hot encoded vectors and normalizing the pixel values to be between 0 and 1.

Next, the data is split into training, validation, and testing sets. The ResNet model architecture is defined using the residual block function which is responsible for adding the residual connection between the convolutional layers. The ResNet function is defined by stacking multiple residual blocks with different numbers of filters. Finally, the model is compiled with Adam optimizer and categorical cross-entropy loss.

To prevent overfitting, data augmentation is performed using the ImageDataGenerator class. The model is trained on the training set using the fit method, and the history of the training process is stored in the history variable. The accuracy and loss of the model are plotted for both training and validation sets.

Finally, the trained model is evaluated on the test set to calculate the test loss and accuracy. The confusion matrix and classification report are generated to visualize the performance of the model on each class. The confusion matrix is plotted using Seaborn's heatmap function.

Visualization of evaluation metrics:

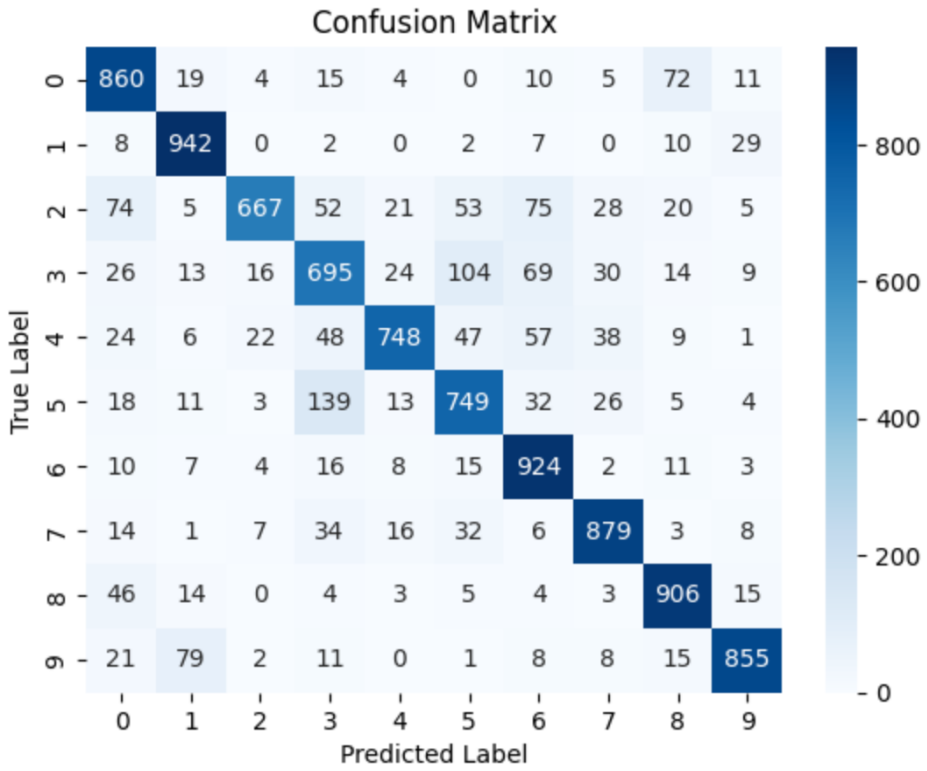


Fig 4.04.1: Confusion matrix for CNN using ResNet.

NOTE: For the above confusion matrix (Fig 4.1):

Label 0 - airplane
Label 1 - automobile
Label 2 - bird
Label 3 - cat
Label 4 - deer
Label 5 - dog
Label 6 - frog
Label 7 - horse
Label 8 - ship
Label 9 - truck

5. Models Comparison:

01. Softmax Classifier.

	precision	recall	f1-score	support
0	0.55	0.54	0.55	1000
1	0.65	0.54	0.59	1000
2	0.39	0.36	0.37	1000
3	0.36	0.24	0.29	1000
4	0.41	0.43	0.42	1000
5	0.43	0.37	0.39	1000
6	0.44	0.64	0.52	1000
7	0.53	0.55	0.54	1000
8	0.63	0.60	0.62	1000
9	0.51	0.61	0.56	1000
accuracy			0.49	10000
macro avg	0.49	0.49	0.48	10000
weighted avg	0.49	0.49	0.48	10000

Accuracy on Test Set is: 48.86 %

02. Random Forest.

	precision	recall	f1-score	support
0	0.54	0.56	0.55	1000
1	0.52	0.55	0.54	1000
2	0.38	0.35	0.36	1000
3	0.35	0.29	0.32	1000
4	0.41	0.41	0.41	1000
5	0.44	0.41	0.42	1000
6	0.47	0.57	0.52	1000
7	0.52	0.46	0.49	1000
8	0.58	0.61	0.60	1000
9	0.47	0.56	0.51	1000
accuracy			0.48	10000
macro avg	0.47	0.48	0.47	10000
weighted avg	0.47	0.48	0.47	10000

Accuracy on Test Set is: 47.52 %

03. SVM with Radial Basis Function Kernel.

	precision	recall	f1-score	support
0	0.63	0.65	0.64	1000
1	0.65	0.71	0.68	1000
2	0.46	0.45	0.45	1000
3	0.39	0.42	0.40	1000
4	0.54	0.49	0.51	1000
5	0.47	0.45	0.46	1000
6	0.65	0.62	0.63	1000
7	0.67	0.61	0.64	1000
8	0.72	0.67	0.69	1000
9	0.57	0.64	0.60	1000
accuracy			0.57	10000
macro avg	0.57	0.57	0.57	10000
weighted avg	0.57	0.57	0.57	10000

Accuracy on Test Set is: 57.13 %

04. ResNet (CNN).

	precision	recall	f1-score	support
airplane	0.78	0.86	0.82	1000
automobile	0.86	0.94	0.90	1000
bird	0.92	0.67	0.77	1000
cat	0.68	0.69	0.69	1000
deer	0.89	0.75	0.81	1000
dog	0.74	0.75	0.75	1000
frog	0.78	0.92	0.84	1000
horse	0.86	0.88	0.87	1000
ship	0.85	0.91	0.88	1000
truck	0.91	0.85	0.88	1000
accuracy			0.82	10000
macro avg	0.83	0.82	0.82	10000
weighted avg	0.83	0.82	0.82	10000

Accuracy on Test Set is: 82 %

ResNet outperforms all the other models with an accuracy of 82%. SVM comes in second with an accuracy of 57.13%, followed by Softmax with 48.86% and Random Forest with 47.52%. (Note, all of these accuracies are based on the predictions made on the test set).

However, accuracy alone is not the only metric to evaluate a model's performance. Other factors such as computational complexity, memory usage, training time, and model interpretability also play an important role in determining the suitability of a model for a particular task.

In terms of computational complexity and memory usage, ResNet is a relatively complex model compared to the other three models, and it requires significant computational resources to train and run. On the other hand, SVM and Random Forest are relatively simpler models that require less computational resources and memory.

In terms of training time, ResNet is a time-consuming model to train compared to the other models, and it may require specialized hardware such as GPUs to speed up the training process. On the other hand, SVM and Random Forest are relatively faster to train, and they can be trained on standard hardware.

Regarding interpretability, SVM and Random Forest are more interpretable than ResNet since they are based on decision boundaries and decision trees, respectively, which are easier to visualize and understand. Softmax is also relatively interpretable since it is a linear model with a single layer.

A major reason why ResNet displays the best performance among all of the other models is due to the fact it takes into account the benefits of Convolution that the other models ignore. Through convolution, the network is able to extract local characteristics in a hierarchical fashion from the input image. This is crucial for image classification since objects and features of interest might exist anywhere in a picture, and a convolutional layer can learn to recognize these features no matter where they are in the image.

In conclusion, while ResNet outperforms the other models in terms of accuracy, it is also more complex, time-consuming, and requires more computational resources to train and run. The choice of the best model depends on the specific requirements and constraints of the task at hand, and it may require a trade-off between accuracy, interpretability, computational complexity, and memory usage.

6. Code for Each Model:

01. Data Preprocessing and EDA.

Data Preprocessing

In [1]:
```
#Import the Libraries and the CIFAR-10 Dataset

from sklearn.metrics import classification_report, accuracy_score
from sklearn.metrics import confusion_matrix
from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt
import numpy as np
```

In [2]:
```
#Load the Data and Split into Training and Test Sets

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

In [3]:
```
#Shape of the dataset
#Number of images in training set = 50000
#Number of images in testing set = 10000
#Height and Width of each image = 32x32
#Number of channels in each image = 3

print("Shape of Training Images: ", X_train.shape)
print("Shape of Testing Images: ", X_test.shape)
```

```
Shape of Training Images:  (50000, 32, 32, 3)
Shape of Testing Images:  (10000, 32, 32, 3)
```

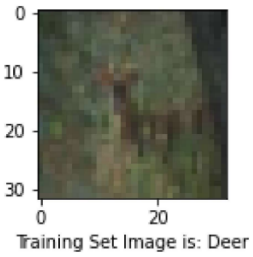
In [4]:
```
#Convert Labels to 1-D Array
y_train = y_train.reshape(-1,)
y_test = y_test.reshape(-1,)
```

In [5]:

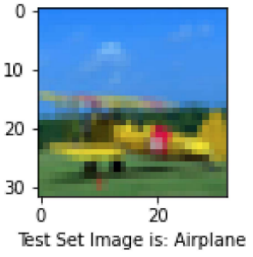
```
#Let's Store the name of each label in a List
labels = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
          'Dog', 'Frog', 'Horse', 'Ship', 'Truck']


#Function Definition to Display Image

def display_image(flag, i):
    #flag is a variable that allows one to display images from the training set or t
    #If flag = 1, Then display Image from Training Set
    #If flag != 1, Then display Image from Test Set
    #i is the index

    plt.figure(figsize = (10,2))

    if flag == 1:
        plt.imshow(X_train[i])
        plt.xlabel('Training Set Image is: ' + labels[y_train[i]])

    if flag != 1:


        plt.imshow(X_test[i])
        plt.xlabel('Test Set Image is: ' + labels[y_test[i]])
```

In [6]:

```
#Display Image from Training Set
display_image(1, 10)

#Display Image from Test Set
display_image(-1, 44)
```



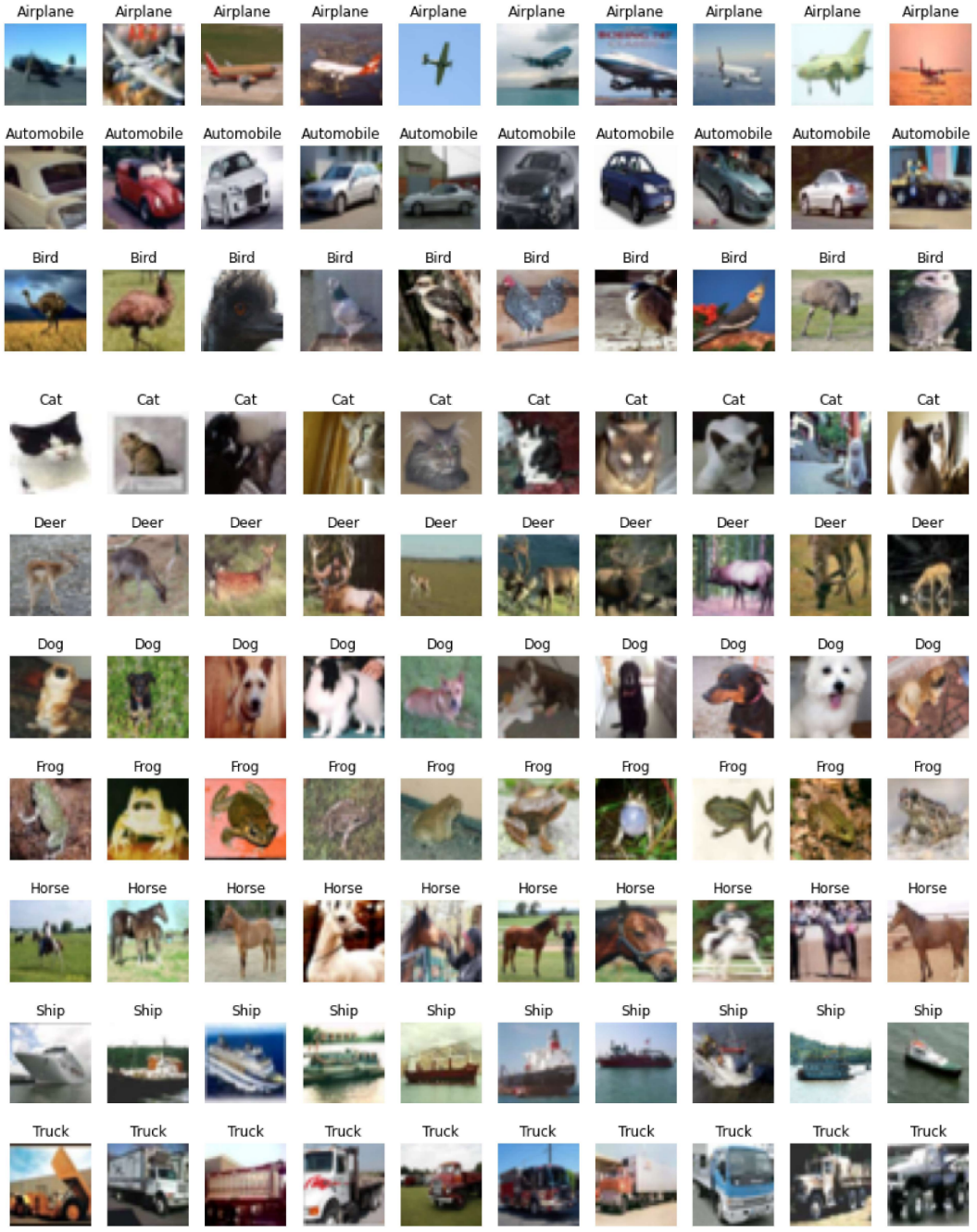
Training Set Image is: Deer



Test Set Image is: Airplane

In [8]:

```
#Generate 10 Random Images for Each Class

num_samples = 10
for i in range(len(labels)):
    images = X_train[y_train.flatten() == i]
    random_indices = np.random.choice(len(images), num_samples, replace=False)
    random_images = images[random_indices]

    # Plot the images
    fig, axs = plt.subplots(1, num_samples, figsize=(15, 15))
    for j in range(num_samples):
        axs[j].imshow(random_images[j])
        axs[j].axis('off')
        axs[j].set_title(labels[i])
    plt.show()
```



In [11]:

```
#Count of Each Class in the Training Set
class_counts_train = [0] * 10

for i in range(len(y_train)):
    class_counts_train[y_train[i]] += 1


plt.figure(figsize = (10,5))
plt.bar(labels, class_counts_train)

# Add title and axis labels
plt.title('Count of Each Class in the Training Set')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks(rotation = 45)
```

Out[11]:

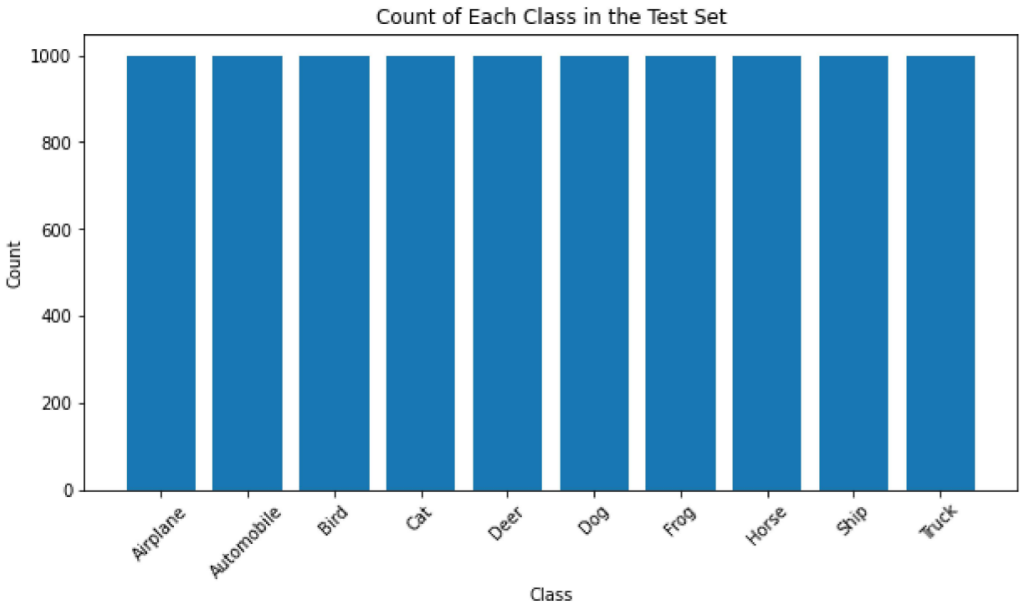
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```

In [12]:

```
#Count of Each Class in the Test Set
class_counts_test = [0] * 10

for i in range(len(y_test)):
    class_counts_test[y_test[i]] += 1


plt.figure(figsize = (10,5))
plt.bar(labels, class_counts_test)

# Add title and axis labels
plt.title('Count of Each Class in the Test Set')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks(rotation = 45)
```

Out[12]:

```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



In [11]:

```
#Normalize the Pixel Values

X_train_norm = X_train/255.0
X_test_norm = X_test/255.0
```

02. Softmax Classifier.

Softmax Classifier

In [44]:
```
#Create copies of the training and test sets to be used for the Softmax Classifier

import tensorflow as tf
from tensorflow.keras import layers, models

X_train_sc = X_train_norm.copy()
X_test_sc = X_test_norm.copy()

y_train_sc = y_train.copy()
y_test_sc = y_test.copy()
```

In [61]:
```
#Let's build the model

sc_model = tf.keras.Sequential()
sc_model.add(layers.Flatten(input_shape = (32, 32, 3)))

sc_model.add(layers.Dense(256, activation = 'relu'))
sc_model.add(layers.Dense(128, activation = 'relu'))
sc_model.add(layers.Dense(64, activation = 'relu'))
sc_model.add(layers.Dense(10, activation = 'softmax'))

#Compile the Model
sc_model.compile("adam", "sparse_categorical_crossentropy", metrics=["Accuracy"])

#Display Model Summary
sc_model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_3 (Flatten)         (None, 3072)              0

 dense_11 (Dense)            (None, 256)               786688

 dense_12 (Dense)            (None, 128)               32896

 dense_13 (Dense)            (None, 64)                8256

 dense_14 (Dense)            (None, 10)                650

=================================================================
Total params: 828,490
Trainable params: 828,490
Non-trainable params: 0
_________________________________________________________________
```

In [62]:
```
#Train the Model on the Training Set

history_sc = sc_model.fit(X_train_sc, y_train_sc, epochs = 20, validation_split = 0.
```

```
Epoch 1/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.8865 - Accuracy:
0.3154 - val_loss: 1.7710 - val_Accuracy: 0.3548
Epoch 2/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.7200 - Accuracy:
0.3807 - val_loss: 1.6949 - val_Accuracy: 0.3905
Epoch 3/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.6410 - Accuracy:
0.4090 - val_loss: 1.7174 - val_Accuracy: 0.3899
Epoch 4/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.5820 - Accuracy:
0.4300 - val_loss: 1.5880 - val_Accuracy: 0.4343
Epoch 5/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.5355 - Accuracy:
0.4526 - val_loss: 1.5869 - val_Accuracy: 0.4372
Epoch 6/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.5035 - Accurac
y: 0.4612 - val_loss: 1.5442 - val_Accuracy: 0.4473
Epoch 7/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.4781 - Accuracy:
0.4711 - val_loss: 1.5601 - val_Accuracy: 0.4462
Epoch 8/20
1250/1250 [==============================] - 14s 11ms/step - loss: 1.4554 - Accurac
y: 0.4801 - val_loss: 1.5350 - val_Accuracy: 0.4522
Epoch 9/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.4342 - Accuracy:
0.4899 - val_loss: 1.4887 - val_Accuracy: 0.4814
Epoch 10/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.4155 - Accuracy:
0.4936 - val_loss: 1.5166 - val_Accuracy: 0.4636
Epoch 11/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.4028 - Accurac
y: 0.4990 - val_loss: 1.5050 - val_Accuracy: 0.4712
```

```
Epoch 12/20
1250/1250 [==============================] - 11s 9ms/step - loss: 1.3834 - Accuracy:
0.5045 - val_loss: 1.5223 - val_Accuracy: 0.4642
Epoch 13/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.3641 - Accuracy:
0.5112 - val_loss: 1.4788 - val_Accuracy: 0.4825
Epoch 14/20
1250/1250 [==============================] - 14s 11ms/step - loss: 1.3548 - Accurac
y: 0.5137 - val_loss: 1.5173 - val_Accuracy: 0.4684
Epoch 15/20
1250/1250 [==============================] - 13s 10ms/step - loss: 1.3361 - Accurac
y: 0.5200 - val_loss: 1.5025 - val_Accuracy: 0.4699
Epoch 16/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.3226 - Accurac
y: 0.5260 - val_loss: 1.5140 - val_Accuracy: 0.4679
Epoch 17/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.3118 - Accuracy:
0.5315 - val_loss: 1.5524 - val_Accuracy: 0.4578
Epoch 18/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.2995 - Accurac
y: 0.5353 - val_loss: 1.5034 - val_Accuracy: 0.4769
Epoch 19/20
1250/1250 [==============================] - 12s 9ms/step - loss: 1.2867 - Accuracy:
0.5384 - val_loss: 1.5411 - val_Accuracy: 0.4663
Epoch 20/20
1250/1250 [==============================] - 12s 10ms/step - loss: 1.2780 - Accurac
y: 0.5414 - val_loss: 1.5054 - val_Accuracy: 0.4780
```

In [63]:

```
#Let's Save the Model
sc_model.save('Softmax_Classifier_CIFAR.h5')
print('Model Saved!')

#Let's Save the Model Weights
sc_model.save_weights('Softmax_Classifier_CIFAR_Weights')
print('Model Weights Saved!')

#Load the model
savedModel = models.load_model('Softmax_Classifier_CIFAR.h5')
savedModel.summary()

#Load the Model Weights
savedModel = sc_model.load_weights('Softmax_Classifier_CIFAR_Weights')
print('Model Loaded!')
```

```
Model Saved!
Model Weights Saved!
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_3 (Flatten)         (None, 3072)              0

 dense_11 (Dense)            (None, 256)               786688

 dense_12 (Dense)            (None, 128)               32896

 dense_13 (Dense)            (None, 64)                8256

 dense_14 (Dense)            (None, 10)                650

=================================================================
Total params: 828,490
Trainable params: 828,490
Non-trainable params: 0
_________________________________________________________________
Model Loaded!
```

In [64]:
```
#Evaluate the model on the Test Set

sc_model.evaluate(X_test_sc, y_test_sc)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 1.4760 - Accuracy: 0.
4886
```

Out[64]: [1.476041555404663, 0.4885999858379364]

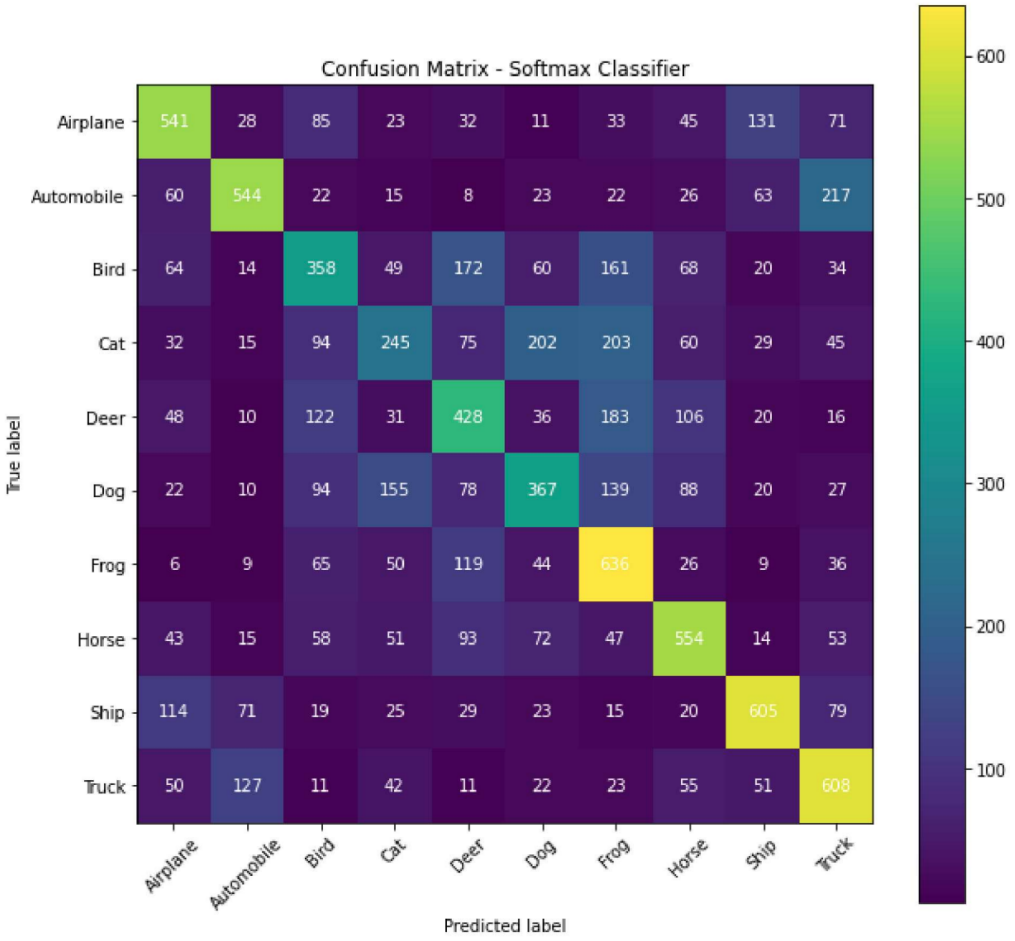
In [65]:
```
#Predict on the Test Set

label_pred_prob_sc = sc_model.predict(X_test_sc)

y_pred_test_sc = []

for i in label_pred_prob_sc:
    y_pred_test_sc.append(np.argmax(i))
```

```
313/313 [==============================] - 1s 2ms/step
```

In [66]:
```
#Let's Print the Classification Report of the Predictions on the Test Set

print(classification_report(y_test_sc, y_pred_test_sc))
print("Accuracy on Test Set is: ", accuracy_score(y_test_sc, y_pred_test_sc)*100.0,
```

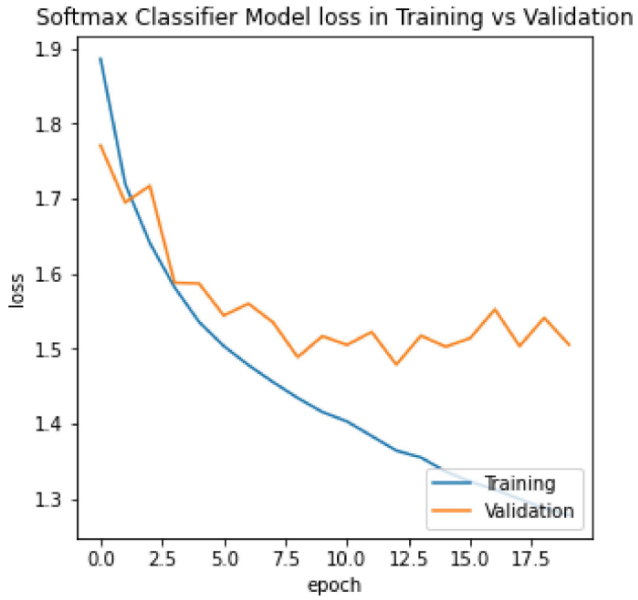
```
              precision    recall  f1-score   support

           0       0.55      0.54      0.55      1000
           1       0.65      0.54      0.59      1000
           2       0.39      0.36      0.37      1000
           3       0.36      0.24      0.29      1000
           4       0.41      0.43      0.42      1000
           5       0.43      0.37      0.39      1000
           6       0.44      0.64      0.52      1000
           7       0.53      0.55      0.54      1000
           8       0.63      0.60      0.62      1000
           9       0.51      0.61      0.56      1000

    accuracy                           0.49     10000
   macro avg       0.49      0.49      0.48     10000
weighted avg       0.49      0.49      0.48     10000

Accuracy on Test Set is:  48.86 %
```

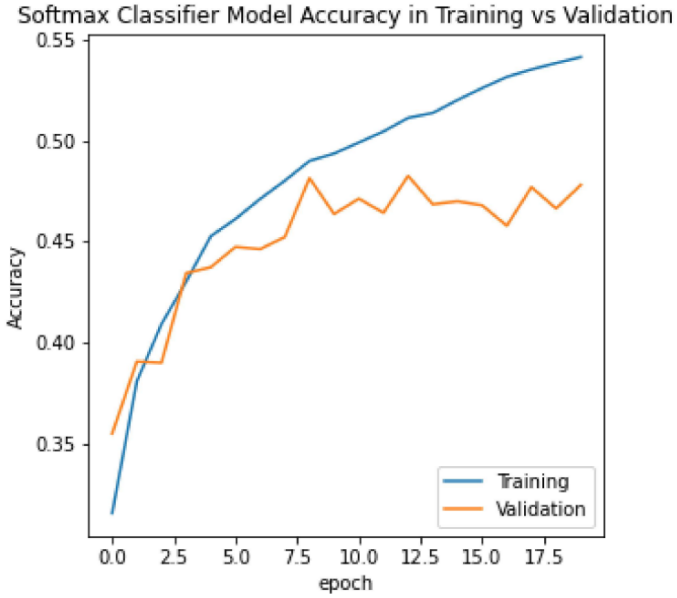
In [67]:
```
#Display the Confusion Matrix

cm = confusion_matrix(y_test_sc, y_pred_test_sc)
plt.figure(figsize=(10, 10))
plt.imshow(cm, cmap='viridis')
plt.title('Confusion Matrix - Softmax Classifier')
plt.colorbar()
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(10), labels, rotation = 45)
plt.yticks(np.arange(10), labels)
for i in range(10):
    for j in range(10):
        plt.text(j, i, cm[i, j], ha='center', va='center', color='white')
plt.show()
```



In [80]:

```
#Plot the change in Training Loss, Training Accuracy, Validation Loss, and Validatio

metrics = history_sc.history.keys()

for param in metrics:
    if 'val' in param:
        continue
    plt.figure(figsize=(5, 5))
    plt.plot(history_sc.history[param])
    plt.plot(history_sc.history['val_' + param])
    plt.title('Softmax Classifier Model ' + param.split('_')[0] +
              ' in Training vs Validation')
    plt.ylabel(param.split('_')[0])
    plt.xlabel('epoch')
    plt.legend(['Training', 'Validation'], loc = 'lower right')
    plt.show()
```


Softmax Classifier Model loss in Training vs Validation
loss
epoch
Training
Validation
1.9
1.8
1.7
1.6
1.5
1.4
1.3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Softmax Classifier Model Accuracy in Training vs Validation
Accuracy
epoch
Training
Validation
0.55
0.50
0.45
0.40
0.35
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

03. Random Forest.

Random Forest Classifier

In [17]:
```
#Create copies of the training and test sets to be used for the Random Forest Classi

X_train_rf = X_train_norm.copy()
X_test_rf = X_test_norm.copy()

y_train_rf = y_train.copy()
y_test_rf = y_test.copy()

#Flatten each image in the Training and Test Sets to a vector of length 32*32*3 = 30
X_train_rf = X_train_rf.reshape(X_train_rf.shape[0], -1)
X_test_rf = X_test_rf.reshape(X_test_rf.shape[0], -1)
```

In [18]:
```
#Import the Libraries and Build and Train the Model

from sklearn.ensemble import RandomForestClassifier

random_forest = RandomForestClassifier()

history_rf = random_forest.fit(X_train_rf, y_train_rf)
```

In [19]:
```
#Make the Predictions on the Test Set

y_pred_test_rf = random_forest.predict(X_test_rf)
```

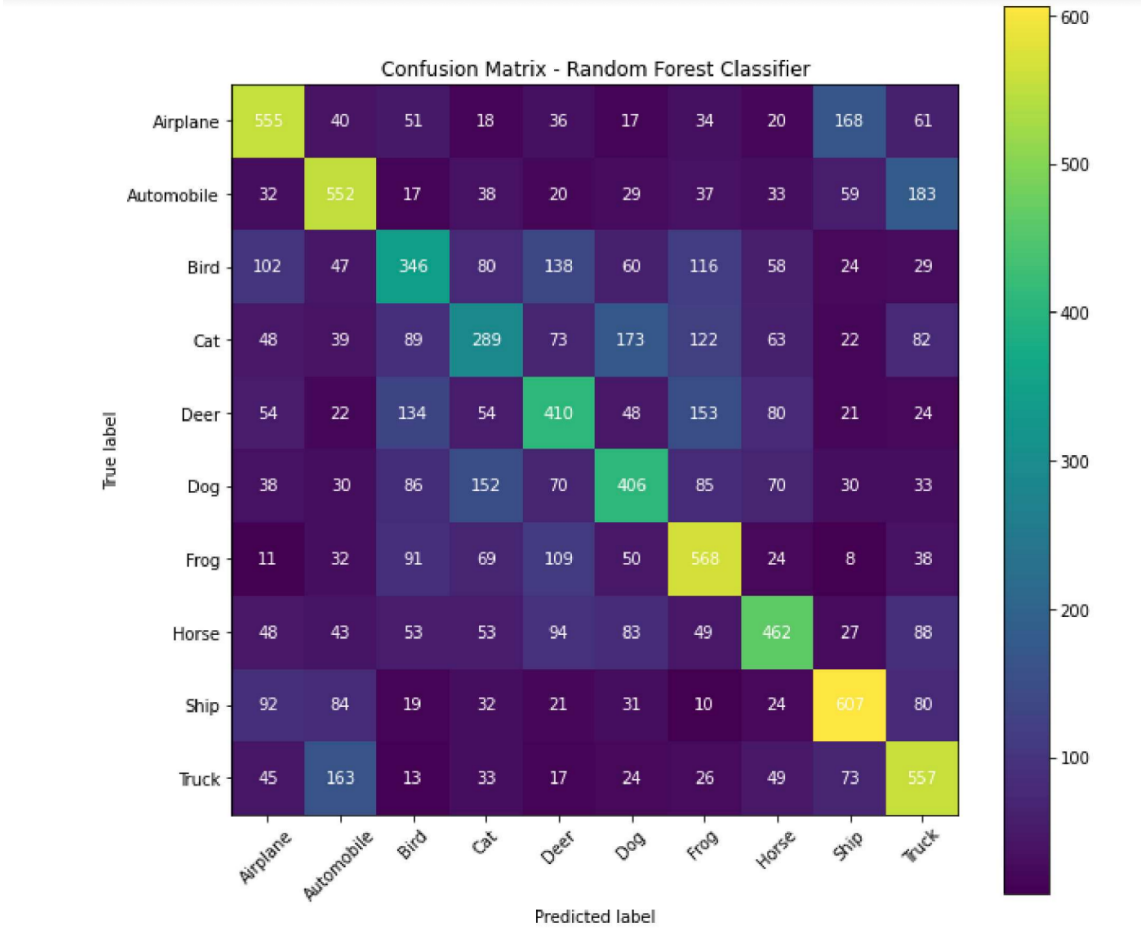
In [23]:
```
#Let's Print the Classification Report of the Predictions on the Test Set
```

```
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

print(classification_report(y_test_rf, y_pred_test_rf))
print("Accuracy on Test Set is: ", accuracy_score(y_test_rf, y_pred_test_rf)*100.0,
```

```
              precision    recall  f1-score   support

           0       0.54      0.56      0.55      1000
           1       0.52      0.55      0.54      1000
           2       0.38      0.35      0.36      1000
           3       0.35      0.29      0.32      1000
           4       0.41      0.41      0.41      1000
           5       0.44      0.41      0.42      1000
           6       0.47      0.57      0.52      1000
           7       0.52      0.46      0.49      1000
           8       0.58      0.61      0.60      1000
           9       0.47      0.56      0.51      1000

    accuracy                           0.48     10000
   macro avg       0.47      0.48      0.47     10000
weighted avg       0.47      0.48      0.47     10000

Accuracy on Test Set is:  47.52 %
```

In [24]:

```
#Display the Confusion Matrix

cm = confusion_matrix(y_test_rf, y_pred_test_rf)
plt.figure(figsize=(10, 10))
plt.imshow(cm, cmap='viridis')
plt.title('Confusion Matrix - Random Forest Classifier')
plt.colorbar()
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.xticks(np.arange(10), labels, rotation = 45)
plt.yticks(np.arange(10), labels)
for i in range(10):
    for j in range(10):
        plt.text(j, i, cm[i, j], ha='center', va='center', color='white')
plt.show()
```


Confusion Matrix - Random Forest Classifier
Airplane 555 40 51 18 36 17 34 20 168 61
Automobile 32 552 17 38 20 29 37 33 59 183
Bird 102 47 346 80 138 60 116 58 24 29
Cat 48 39 89 289 73 173 122 63 22 82
Deer 54 22 134 54 410 48 153 80 21 24
Dog 38 30 86 152 70 406 85 70 30 33
Frog 11 32 91 69 109 50 568 24 8 38
Horse 48 43 53 53 94 83 49 462 27 88
Ship 92 84 19 32 21 31 10 24 607 80
Truck 45 163 13 33 17 24 26 49 73 557
Airplane
Automobile
Bird
Cat
Deer
Dog
Frog
Horse
Ship
Truck
True label
Predicted label
600
500
400
300
200
100

04. SVM with Radial Basis Function Kernel.

Support Vector Machine (SVM) with Radial Basis Function Kernel

In [33]:
```
#Create copies of the training and test sets to be used for the Support Vector Machi

X_train_svm = X_train_norm.copy()
X_test_svm = X_test_norm.copy()

y_train_svm = y_train.copy()
y_test_svm = y_test.copy()

#Flatten each image in the Training and Test Sets to a vector of length 32*32*3 = 30
X_train_svm = X_train_svm.reshape(X_train_svm.shape[0], -1)
X_test_svm = X_test_svm.reshape(X_test_svm.shape[0], -1)
```

In [35]:
```
#We Will perform PCA to reduce the number of dimensions in the Training and Test Set
#This is due to computational issues with 3072 dimensions
#This will avoid the Curse of Dimensionality

from sklearn.decomposition import PCA

pca = PCA(n_components = 200, whiten = True)
pca.fit(X_train_svm)


X_train_svm = pca.transform(X_train_svm)
X_test_svm = pca.transform(X_test_svm)
```

In [39]:
```
#Let's build and Train the SVM model using Radial Basis Function

from sklearn.svm import SVC

clf = SVC(kernel = 'rbf', C = 10, gamma = 'scale')
clf.fit(X_train_svm, y_train_svm)
```

Out[39]:
```
SVC
SVC(C=10)
```

In [40]:
```
#Let's save the model

import pickle

filename = "SVM.pickle"
pickle.dump(clf, open(filename, "wb"))
```

In [41]:

```
#Load the Model and Predict on the Test Set

clf_loaded_model = pickle.load(open(filename, "rb"))

y_pred_test_svm = clf_loaded_model.predict(X_test_svm)
```

In [42]:

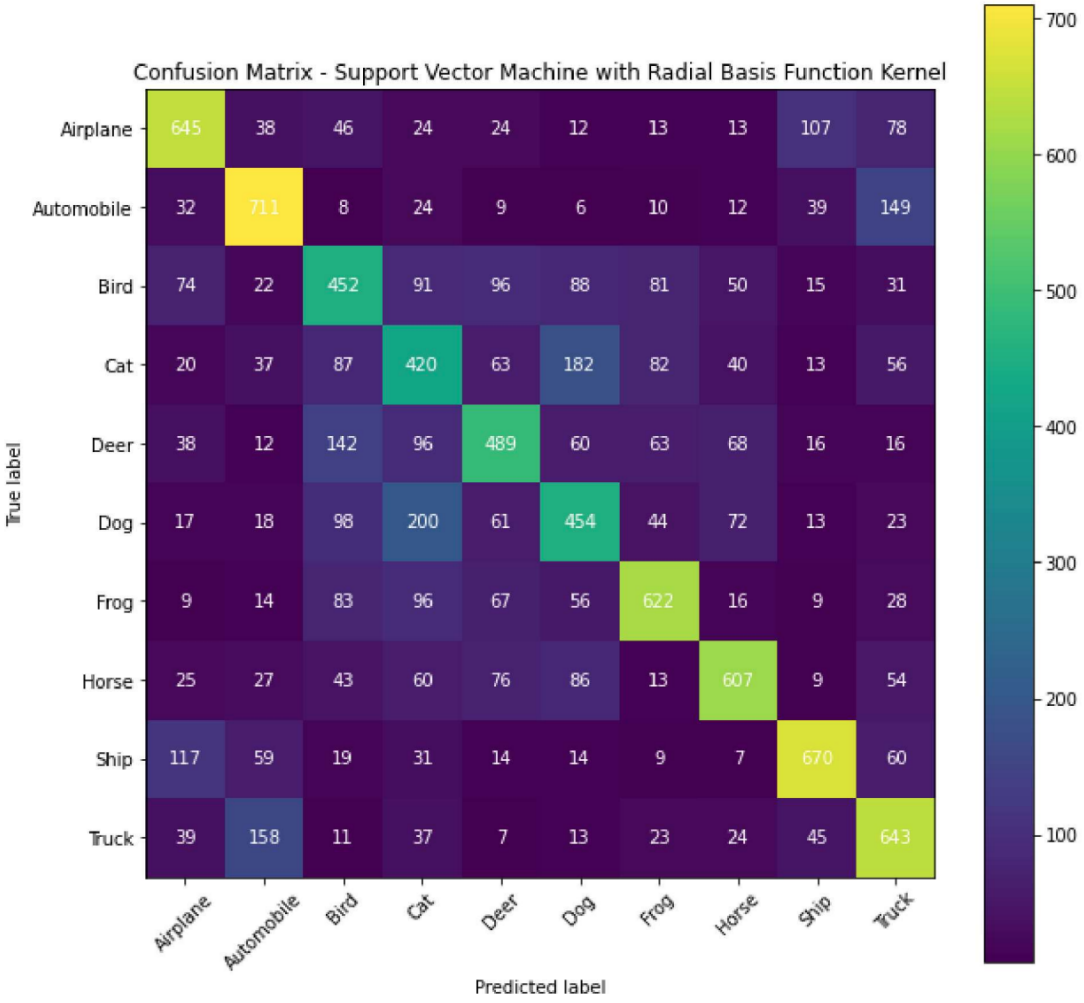
```
#Let's Print the Classification Report of the Predictions on the Test Set

print(classification_report(y_test_svm, y_pred_test_svm))
print("Accuracy on Test Set is: ", accuracy_score(y_test_svm, y_pred_test_svm)*100.0
```

```
              precision    recall  f1-score   support

           0       0.63      0.65      0.64      1000
           1       0.65      0.71      0.68      1000
           2       0.46      0.45      0.45      1000
           3       0.39      0.42      0.40      1000
           4       0.54      0.49      0.51      1000
           5       0.47      0.45      0.46      1000
           6       0.65      0.62      0.63      1000
           7       0.67      0.61      0.64      1000
           8       0.72      0.67      0.69      1000
           9       0.57      0.64      0.60      1000

    accuracy                           0.57     10000
   macro avg       0.57      0.57      0.57     10000
weighted avg       0.57      0.57      0.57     10000

Accuracy on Test Set is:  57.13 %
```

In [43]:

```
#Display the Confusion Matrix

cm = confusion_matrix(y_test_svm, y_pred_test_svm)
plt.figure(figsize=(10, 10))
plt.imshow(cm, cmap='viridis')
plt.title('Confusion Matrix - Support Vector Machine with Radial Basis Function Kern
plt.colorbar()
plt.xlabel('Predicted label')
```

```
plt.ylabel('True label')
plt.xticks(np.arange(10), labels, rotation = 45)
plt.yticks(np.arange(10), labels)
for i in range(10):
    for j in range(10):
        plt.text(j, i, cm[i, j], ha='center', va='center', color='white')
plt.show()
```


Confusion Matrix - Support Vector Machine with Radial Basis Function Kernel
Airplane 645 38 46 24 24 12 13 13 107 78
Automobile 32 711 8 24 9 6 10 12 39 149
Bird 74 22 452 91 96 88 81 50 15 31
Cat 20 37 87 420 63 182 82 40 13 56
Deer 38 12 142 96 489 60 63 68 16 16
Dog 17 18 98 200 61 454 44 72 13 23
Frog 9 14 83 96 67 56 622 16 9 28
Horse 25 27 43 60 76 86 13 607 9 54
Ship 117 59 19 31 14 14 9 7 670 60
Truck 39 158 11 37 7 13 23 24 45 643
True label
Airplane
Automobile
Bird
Cat
Deer
Dog
Frog
Horse
Ship
Truck
Predicted label
700
600
500
400
300
200
100

05. ResNet (CNN).

ResNet Model

In [1]:
```
import keras
from keras.datasets import cifar10
from keras.layers import Input, Conv2D, MaxPooling2D, Add, Flatten, Dense, Dropout
from keras.layers import Activation, BatchNormalization
from keras.models import Model
from keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [2]:
```
from keras.layers import Input, Conv2D, MaxPooling2D, Add, Flatten, Dense, Dropout
from keras.layers import Activation, BatchNormalization, AveragePooling2D
```

In [3]:
```
# define hyperparameters
batch_size = 128
epochs = 50
num_classes = 10

# load dataset
(x_train_all, y_train_all), (x_test, y_test) = cifar10.load_data()

# convert class vectors to binary class matrices
y_train_all = keras.utils.to_categorical(y_train_all, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

In [4]:
```
# normalize data
x_train_all = x_train_all.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# split data into training, validation and testing sets
val_ratio = 0.1
test_ratio = 0.1
num_val_samples = int(val_ratio * x_train_all.shape[0])
num_test_samples = int(test_ratio * x_train_all.shape[0])


x_val = x_train_all[num_test_samples:num_test_samples+num_val_samples]
y_val = y_train_all[num_test_samples:num_test_samples+num_val_samples]
x_train = x_train_all[num_test_samples+num_val_samples:]
y_train = y_train_all[num_test_samples+num_val_samples:]
```

In [5]:
```
# define the ResNet architecture
def residual_block(x, filters, kernel_size=3, stride=1, conv_shortcut=False):
    if conv_shortcut:
        shortcut = Conv2D(filters, kernel_size=1, strides=stride, padding='same',
                          use_bias=False)(x)
        shortcut = BatchNormalization()(shortcut)
    else:
        shortcut = x

    x = Conv2D(filters, kernel_size=kernel_size, strides=stride, padding='same',
               use_bias=False)(x)
```

```
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = Conv2D(filters, kernel_size=kernel_size, strides=1, padding='same',
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Add()([x, shortcut])
    x = Activation('relu')(x)
    return x
```

In [6]:

```
def resnet(input_shape, num_classes):
    inputs = Input(shape=input_shape)
    x = Conv2D(64, kernel_size=7, strides=2, padding='same', use_bias=False)(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = MaxPooling2D(pool_size=3, strides=2, padding='same')(x)

    filters = 64
    for i in range(3):
        x = residual_block(x, filters, conv_shortcut=True if i == 0 else False)
    filters *= 2
    for i in range(4):
        x = residual_block(x, filters, conv_shortcut=True if i == 0 else False)
    filters *= 2
    for i in range(6):
        x = residual_block(x, filters, conv_shortcut=True if i == 0 else False)

    x = AveragePooling2D(pool_size=7)(x)
    x = Flatten()(x)
    x = Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=inputs, outputs=x)
    return model
```

In [7]:

```
input_shape = x_train.shape[1:]
model = resnet(input_shape, num_classes)
model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================
==============
 input_1 (InputLayer)        [(None, 32, 32, 3)]       0         []

 conv2d (Conv2D)             (None, 16, 16, 64)        9408      ['input_1[0][0]']

 batch_normalization (BatchNorm  (None, 16, 16, 64)    256       ['conv2d[0][0]']
 alization)

 activation (Activation)     (None, 16, 16, 64)        0         ['batch_normalizati
on[0][0]']

 max_pooling2d (MaxPooling2D)   (None, 8, 8, 64)       0         ['activation[0]
[0]']

 conv2d_2 (Conv2D)           (None, 8, 8, 64)          36864     ['max_pooling2d[0]
[0]']

 batch_normalization_2 (BatchNo  (None, 8, 8, 64)      256       ['conv2d_2[0][0]']
 rmalization)

 activation_1 (Activation)   (None, 8, 8, 64)          0         ['batch_normalizati
```

```
on_2[0][0]']

 conv2d_3 (Conv2D)              (None, 8, 8, 64)     36864       ['activation_1[0]
[0]']

 conv2d_1 (Conv2D)              (None, 8, 8, 64)     4096        ['max_pooling2d[0]
[0]']

 batch_normalization_3 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_3[0][0]']
 rmalization)

 batch_normalization_1 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_1[0][0]']
 rmalization)

 add (Add)                      (None, 8, 8, 64)     0           ['batch_normalizati
on_3[0][0]',
                                                                  'batch_normalizati
on_1[0][0]']

 activation_2 (Activation)      (None, 8, 8, 64)     0           ['add[0][0]']

 conv2d_4 (Conv2D)              (None, 8, 8, 64)     36864       ['activation_2[0]
[0]']

 batch_normalization_4 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_4[0][0]']
 rmalization)

 activation_3 (Activation)      (None, 8, 8, 64)     0           ['batch_normalizati
on_4[0][0]']

 conv2d_5 (Conv2D)              (None, 8, 8, 64)     36864       ['activation_3[0]
[0]']

 batch_normalization_5 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_5[0][0]']
 rmalization)

 add_1 (Add)                    (None, 8, 8, 64)     0           ['batch_normalizati
on_5[0][0]',
                                                                  'activation_2[0]
[0]']
```

```
 activation_4 (Activation)      (None, 8, 8, 64)     0           ['add_1[0][0]']

 conv2d_6 (Conv2D)              (None, 8, 8, 64)     36864       ['activation_4[0]
[0]']

 batch_normalization_6 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_6[0][0]']
 rmalization)

 activation_5 (Activation)      (None, 8, 8, 64)     0           ['batch_normalizati
on_6[0][0]']

 conv2d_7 (Conv2D)              (None, 8, 8, 64)     36864       ['activation_5[0]
[0]']

 batch_normalization_7 (BatchNo  (None, 8, 8, 64)    256         ['conv2d_7[0][0]']
 rmalization)

 add_2 (Add)                    (None, 8, 8, 64)     0           ['batch_normalizati
on_7[0][0]',
                                                                  'activation_4[0]
[0]']

 activation_6 (Activation)      (None, 8, 8, 64)     0           ['add_2[0][0]']

 conv2d_9 (Conv2D)              (None, 8, 8, 128)    73728       ['activation_6[0]
[0]']

 batch_normalization_9 (BatchNo  (None, 8, 8, 128)   512         ['conv2d_9[0][0]']
```

```
 rmalization)

 activation_7 (Activation)      (None, 8, 8, 128)    0           ['batch_normalizati
on_9[0][0]']

 conv2d_10 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_7[0]
[0]']

 conv2d_8 (Conv2D)              (None, 8, 8, 128)    8192        ['activation_6[0]
[0]']

 batch_normalization_10 (BatchN  (None, 8, 8, 128)   512         ['conv2d_10[0][0]']
 ormalization)

 batch_normalization_8 (BatchNo  (None, 8, 8, 128)   512         ['conv2d_8[0][0]']
 rmalization)

 add_3 (Add)                    (None, 8, 8, 128)    0           ['batch_normalizati
on_10[0][0]',
                                                                  'batch_normalizati
on_8[0][0]']

 activation_8 (Activation)      (None, 8, 8, 128)    0           ['add_3[0][0]']

 conv2d_11 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_8[0]
[0]']

 batch_normalization_11 (BatchN  (None, 8, 8, 128)   512         ['conv2d_11[0][0]']
 ormalization)

 activation_9 (Activation)      (None, 8, 8, 128)    0           ['batch_normalizati
on_11[0][0]']

 conv2d_12 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_9[0]
[0]']

 batch_normalization_12 (BatchN  (None, 8, 8, 128)   512         ['conv2d_12[0][0]']
 ormalization)
```

```
 add_4 (Add)                    (None, 8, 8, 128)    0           ['batch_normalizati
on_12[0][0]',
                                                                  'activation_8[0]
[0]']

 activation_10 (Activation)     (None, 8, 8, 128)    0           ['add_4[0][0]']

 conv2d_13 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_10[0]
[0]']

 batch_normalization_13 (BatchN  (None, 8, 8, 128)   512         ['conv2d_13[0][0]']
 ormalization)

 activation_11 (Activation)     (None, 8, 8, 128)    0           ['batch_normalizati
on_13[0][0]']

 conv2d_14 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_11[0]
[0]']

 batch_normalization_14 (BatchN  (None, 8, 8, 128)   512         ['conv2d_14[0][0]']
 ormalization)

 add_5 (Add)                    (None, 8, 8, 128)    0           ['batch_normalizati
on_14[0][0]',
                                                                  'activation_10[0]
[0]']

 activation_12 (Activation)     (None, 8, 8, 128)    0           ['add_5[0][0]']

 conv2d_15 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_12[0]
```

```
[0]']

 batch_normalization_15 (BatchN  (None, 8, 8, 128)   512         ['conv2d_15[0][0]']
 ormalization)

 activation_13 (Activation)     (None, 8, 8, 128)    0           ['batch_normalizati
on_15[0][0]']

 conv2d_16 (Conv2D)             (None, 8, 8, 128)    147456      ['activation_13[0]
[0]']

 batch_normalization_16 (BatchN  (None, 8, 8, 128)   512         ['conv2d_16[0][0]']
 ormalization)

 add_6 (Add)                    (None, 8, 8, 128)    0           ['batch_normalizati
on_16[0][0]',
                                                                  'activation_12[0]
[0]']

 activation_14 (Activation)     (None, 8, 8, 128)    0           ['add_6[0][0]']

 conv2d_18 (Conv2D)             (None, 8, 8, 256)    294912      ['activation_14[0]
[0]']

 batch_normalization_18 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_18[0][0]']
 ormalization)

 activation_15 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_18[0][0]']

 conv2d_19 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_15[0]
[0]']

 conv2d_17 (Conv2D)             (None, 8, 8, 256)    32768       ['activation_14[0]
[0]']

 batch_normalization_19 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_19[0][0]']
 ormalization)

 batch_normalization_17 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_17[0][0]']
 ormalization)
```

```
 batch_normalization_17 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_17[0][0]']
 ormalization)

 add_7 (Add)                    (None, 8, 8, 256)    0           ['batch_normalizati
on_19[0][0]',
                                                                  'batch_normalizati
on_17[0][0]']

 activation_16 (Activation)     (None, 8, 8, 256)    0           ['add_7[0][0]']

 conv2d_20 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_16[0]
[0]']

 batch_normalization_20 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_20[0][0]']
 ormalization)

 activation_17 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_20[0][0]']

 conv2d_21 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_17[0]
[0]']

 batch_normalization_21 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_21[0][0]']
 ormalization)

 add_8 (Add)                    (None, 8, 8, 256)    0           ['batch_normalizati
on_21[0][0]',
                                                                  'activation_16[0]
[0]']
```

```
 activation_18 (Activation)     (None, 8, 8, 256)    0           ['add_8[0][0]']

 conv2d_22 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_18[0]
[0]']

 batch_normalization_22 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_22[0][0]']
 ormalization)

 activation_19 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_22[0][0]']

 conv2d_23 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_19[0]
[0]']

 batch_normalization_23 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_23[0][0]']
 ormalization)

 add_9 (Add)                    (None, 8, 8, 256)    0           ['batch_normalizati
on_23[0][0]',
                                                                  'activation_18[0]
[0]']

 activation_20 (Activation)     (None, 8, 8, 256)    0           ['add_9[0][0]']

 conv2d_24 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_20[0]
[0]']

 batch_normalization_24 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_24[0][0]']
 ormalization)

 activation_21 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_24[0][0]']

 conv2d_25 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_21[0]
[0]']

 batch_normalization_25 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_25[0][0]']
 ormalization)

 add_10 (Add)                   (None, 8, 8, 256)    0           ['batch_normalizati
on_25[0][0]',
                                                                  'activation_20[0]
[0]']

 activation_22 (Activation)     (None, 8, 8, 256)    0           ['add_10[0][0]']
```

```
 activation_22 (Activation)     (None, 8, 8, 256)    0           ['add_10[0][0]']

 conv2d_26 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_22[0]
[0]']

 batch_normalization_26 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_26[0][0]']
 ormalization)

 activation_23 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_26[0][0]']

 conv2d_27 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_23[0]
[0]']

 batch_normalization_27 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_27[0][0]']
 ormalization)

 add_11 (Add)                   (None, 8, 8, 256)    0           ['batch_normalizati
on_27[0][0]',
                                                                  'activation_22[0]
[0]']

 activation_24 (Activation)     (None, 8, 8, 256)    0           ['add_11[0][0]']

 conv2d_28 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_24[0]


 [0]']

  batch_normalization_28 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_28[0][0]']
  ormalization)

  activation_25 (Activation)     (None, 8, 8, 256)    0           ['batch_normalizati
on_28[0][0]']

  conv2d_29 (Conv2D)             (None, 8, 8, 256)    589824      ['activation_25[0]
[0]']

  batch_normalization_29 (BatchN  (None, 8, 8, 256)   1024        ['conv2d_29[0][0]']
  ormalization)

  add_12 (Add)                   (None, 8, 8, 256)    0           ['batch_normalizati
on_29[0][0]',
                                                                   'activation_24[0]
[0]']

  activation_26 (Activation)     (None, 8, 8, 256)    0           ['add_12[0][0]']

  average_pooling2d (AveragePool  (None, 1, 1, 256)   0           ['activation_26[0]
[0]']
  ing2D)

  flatten (Flatten)              (None, 256)          0           ['average_pooling2d
[0][0]']

  dense (Dense)                  (None, 10)           2570        ['flatten[0][0]']

==================================================================================
==============
Total params: 8,187,082
Trainable params: 8,177,098
Non-trainable params: 9,984
__________________________________________________________________________________
______________
```

In [8]:
```
optimizer = Adam(lr=0.001)
model.compile(loss='categorical_crossentropy',
optimizer=optimizer,
metrics=['accuracy'])
```

```
/usr/local/lib/python3.9/dist-packages/keras/optimizers/legacy/adam.py:117: UserWarn
ing: The `lr` argument is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

In [9]:
```
datagen_train = ImageDataGenerator(
width_shift_range=0.1,
height_shift_range=0.1,
horizontal_flip=True,
fill_mode='nearest')

datagen_train.fit(x_train)
```

In [10]:
```
history = model.fit(datagen_train.flow(x_train, y_train, batch_size=batch_size),
steps_per_epoch=len(x_train) / batch_size,
epochs=epochs,
validation_data=(x_val, y_val),
verbose=1)
```

```
Epoch 1/50
312/312 [==============================] - 29s 76ms/step - loss: 1.5831 - accuracy:
0.4423 - val_loss: 2.3939 - val_accuracy: 0.2628
```

```
Epoch 2/50
312/312 [==============================] - 20s 64ms/step - loss: 1.1809 - accuracy:
0.5816 - val_loss: 2.8911 - val_accuracy: 0.3492
Epoch 3/50
312/312 [==============================] - 20s 63ms/step - loss: 0.9956 - accuracy:
0.6497 - val_loss: 2.2074 - val_accuracy: 0.4442
Epoch 4/50
312/312 [==============================] - 20s 65ms/step - loss: 0.8747 - accuracy:
0.6926 - val_loss: 1.1842 - val_accuracy: 0.5888
Epoch 5/50
312/312 [==============================] - 20s 63ms/step - loss: 0.8079 - accuracy:
0.7188 - val_loss: 0.8403 - val_accuracy: 0.6960
Epoch 6/50
312/312 [==============================] - 20s 62ms/step - loss: 0.7387 - accuracy:
0.7413 - val_loss: 1.1716 - val_accuracy: 0.6122
Epoch 7/50
312/312 [==============================] - 20s 63ms/step - loss: 0.6945 - accuracy:
0.7582 - val_loss: 1.1770 - val_accuracy: 0.6152
Epoch 8/50
312/312 [==============================] - 20s 63ms/step - loss: 0.6508 - accuracy:
0.7726 - val_loss: 0.9201 - val_accuracy: 0.6850
Epoch 9/50
312/312 [==============================] - 20s 63ms/step - loss: 0.6115 - accuracy:
0.7876 - val_loss: 1.0254 - val_accuracy: 0.6654
Epoch 10/50
312/312 [==============================] - 20s 63ms/step - loss: 0.5776 - accuracy:
0.7974 - val_loss: 0.8397 - val_accuracy: 0.7228
Epoch 11/50
312/312 [==============================] - 20s 64ms/step - loss: 0.5502 - accuracy:
0.8081 - val_loss: 1.0612 - val_accuracy: 0.6648
Epoch 12/50
312/312 [==============================] - 20s 63ms/step - loss: 0.5221 - accuracy:
0.8185 - val_loss: 0.8152 - val_accuracy: 0.7228
Epoch 13/50
312/312 [==============================] - 20s 63ms/step - loss: 0.5010 - accuracy:
0.8231 - val_loss: 0.7274 - val_accuracy: 0.7612
Epoch 14/50
312/312 [==============================] - 20s 63ms/step - loss: 0.4766 - accuracy:
0.8334 - val_loss: 0.7680 - val_accuracy: 0.7514
Epoch 15/50
312/312 [==============================] - 20s 63ms/step - loss: 0.4614 - accuracy:
0.8386 - val_loss: 0.6858 - val_accuracy: 0.7804
```

```
Epoch 16/50
312/312 [==============================] - 19s 62ms/step - loss: 0.4334 - accuracy:
0.8473 - val_loss: 0.9452 - val_accuracy: 0.7050
Epoch 17/50
312/312 [==============================] - 20s 63ms/step - loss: 0.4188 - accuracy:
0.8540 - val_loss: 0.7773 - val_accuracy: 0.7434
Epoch 18/50
312/312 [==============================] - 20s 64ms/step - loss: 0.3994 - accuracy:
0.8605 - val_loss: 0.6393 - val_accuracy: 0.7854
Epoch 19/50
312/312 [==============================] - 20s 64ms/step - loss: 0.3861 - accuracy:
0.8654 - val_loss: 1.3173 - val_accuracy: 0.6506
Epoch 20/50
312/312 [==============================] - 20s 63ms/step - loss: 0.3673 - accuracy:
0.8704 - val_loss: 0.8061 - val_accuracy: 0.7564
Epoch 21/50
312/312 [==============================] - 20s 63ms/step - loss: 0.3459 - accuracy:
0.8777 - val_loss: 0.6337 - val_accuracy: 0.7960
Epoch 22/50
312/312 [==============================] - 20s 64ms/step - loss: 0.3388 - accuracy:
0.8803 - val_loss: 0.6771 - val_accuracy: 0.7924
Epoch 23/50
312/312 [==============================] - 20s 63ms/step - loss: 0.3230 - accuracy:
0.8868 - val_loss: 0.5978 - val_accuracy: 0.8090
Epoch 24/50
312/312 [==============================] - 20s 63ms/step - loss: 0.3136 - accuracy:
0.8881 - val_loss: 0.7250 - val_accuracy: 0.7688
```

```
Epoch 25/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2987 - accuracy:
0.8950 - val_loss: 0.6897 - val_accuracy: 0.7940
Epoch 26/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2898 - accuracy:
0.8990 - val_loss: 0.6365 - val_accuracy: 0.8018
Epoch 27/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2801 - accuracy:
0.9000 - val_loss: 0.6461 - val_accuracy: 0.7966
Epoch 28/50
312/312 [==============================] - 20s 64ms/step - loss: 0.2552 - accuracy:
0.9089 - val_loss: 0.6121 - val_accuracy: 0.8160
Epoch 29/50
312/312 [==============================] - 20s 62ms/step - loss: 0.2547 - accuracy:
0.9096 - val_loss: 0.6554 - val_accuracy: 0.8044
Epoch 30/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2445 - accuracy:
0.9131 - val_loss: 0.9471 - val_accuracy: 0.7592
Epoch 31/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2371 - accuracy:
0.9152 - val_loss: 0.7637 - val_accuracy: 0.7706
Epoch 32/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2265 - accuracy:
0.9195 - val_loss: 0.6881 - val_accuracy: 0.8060
Epoch 33/50
312/312 [==============================] - 20s 63ms/step - loss: 0.2204 - accuracy:
0.9205 - val_loss: 0.7539 - val_accuracy: 0.7920
Epoch 34/50
312/312 [==============================] - 20s 64ms/step - loss: 0.2094 - accuracy:
0.9253 - val_loss: 0.6254 - val_accuracy: 0.8144
Epoch 35/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1968 - accuracy:
0.9302 - val_loss: 0.7936 - val_accuracy: 0.7872
Epoch 36/50
312/312 [==============================] - 19s 62ms/step - loss: 0.1926 - accuracy:
0.9316 - val_loss: 0.6343 - val_accuracy: 0.8236
Epoch 37/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1803 - accuracy:
0.9352 - val_loss: 1.0552 - val_accuracy: 0.7332
Epoch 38/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1853 - accuracy:
0.9344 - val_loss: 0.9225 - val_accuracy: 0.7674
```

```
Epoch 39/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1730 - accuracy:
0.9380 - val_loss: 0.7373 - val_accuracy: 0.8092
Epoch 40/50
312/312 [==============================] - 20s 62ms/step - loss: 0.1612 - accuracy:
0.9430 - val_loss: 0.7448 - val_accuracy: 0.8062
Epoch 41/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1618 - accuracy:
0.9427 - val_loss: 0.7061 - val_accuracy: 0.8060
Epoch 42/50
312/312 [==============================] - 20s 64ms/step - loss: 0.1514 - accuracy:
0.9461 - val_loss: 0.7326 - val_accuracy: 0.8140
Epoch 43/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1467 - accuracy:
0.9478 - val_loss: 0.7206 - val_accuracy: 0.8218
Epoch 44/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1488 - accuracy:
0.9465 - val_loss: 0.7095 - val_accuracy: 0.8142
Epoch 45/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1422 - accuracy:
0.9494 - val_loss: 0.5987 - val_accuracy: 0.8412
Epoch 46/50
312/312 [==============================] - 19s 62ms/step - loss: 0.1356 - accuracy:
0.9513 - val_loss: 0.7552 - val_accuracy: 0.8112
Epoch 47/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1371 - accuracy:
0.9509 - val_loss: 0.8625 - val_accuracy: 0.8102

Epoch 48/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1282 - accuracy:
0.9538 - val_loss: 1.0274 - val_accuracy: 0.7706
Epoch 49/50
312/312 [==============================] - 20s 63ms/step - loss: 0.1232 - accuracy:
0.9561 - val_loss: 0.8216 - val_accuracy: 0.8072
Epoch 50/50
312/312 [==============================] - 20s 64ms/step - loss: 0.1238 - accuracy:
0.9558 - val_loss: 0.6921 - val_accuracy: 0.8278
```

In [11]:

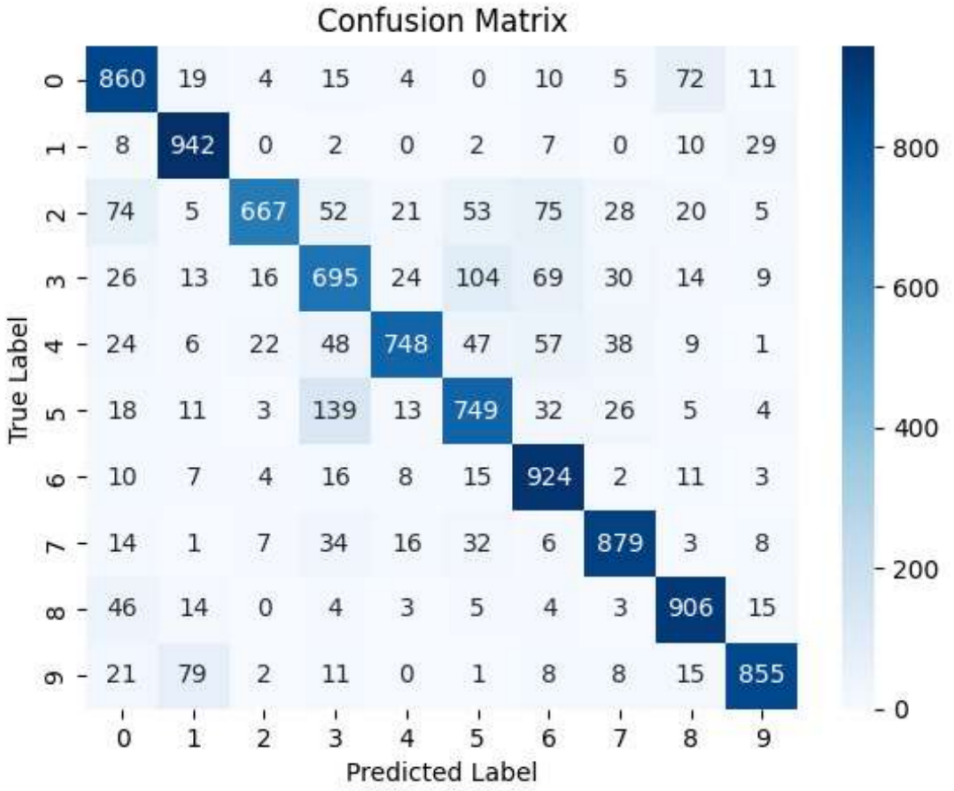
```
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
313/313 [==============================] - 2s 6ms/step - loss: 0.7243 - accuracy: 0.
8225
Test loss: 0.7243140339851379
Test accuracy: 0.8224999904632568
```

In [12]:

```
y_pred = model.predict(x_test)
y_pred = np.argmax(y_pred, axis=1)
y_test = np.argmax(y_test, axis=1)
cm = confusion_matrix(y_test, y_pred)
cr = classification_report(y_test, y_pred,
                           target_names=['airplane', 'automobile', 'bird', 'cat', 'd
                                         'dog', 'frog', 'horse', 'ship', 'truck'])
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
print(cr)
```

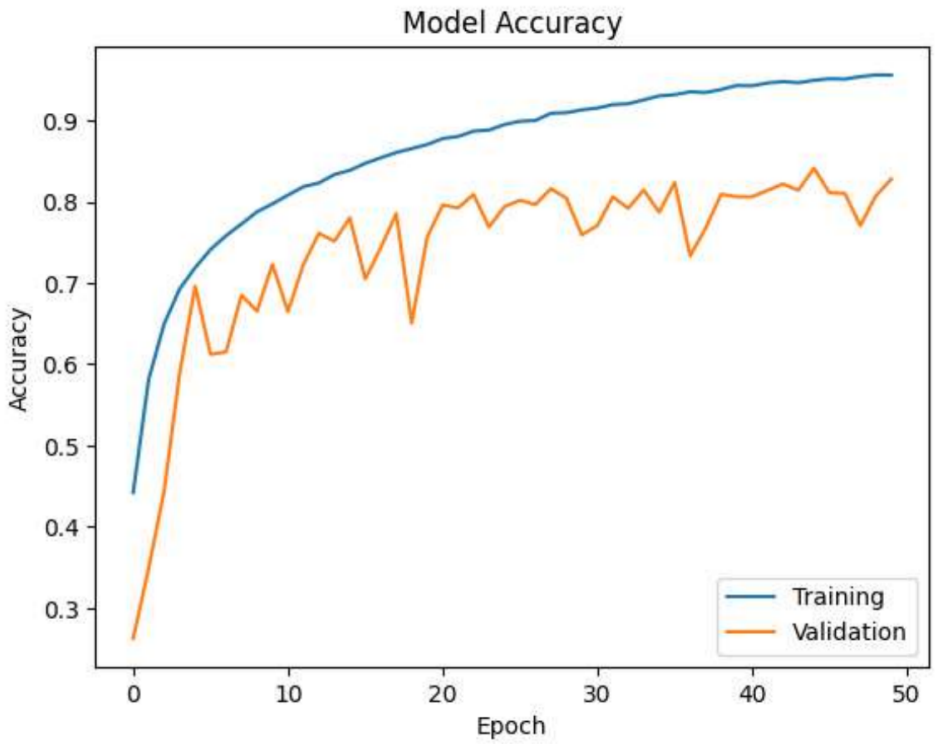
```
313/313 [==============================] - 2s 4ms/step
```



	precision	recall	f1-score	support
airplane	0.78	0.86	0.82	1000
automobile	0.86	0.94	0.90	1000
bird	0.92	0.67	0.77	1000
cat	0.68	0.69	0.69	1000
deer	0.89	0.75	0.81	1000
dog	0.74	0.75	0.75	1000
frog	0.78	0.92	0.84	1000
horse	0.86	0.88	0.87	1000
ship	0.85	0.91	0.88	1000
truck	0.91	0.85	0.88	1000
accuracy			0.82	10000
macro avg	0.83	0.82	0.82	10000
weighted avg	0.83	0.82	0.82	10000

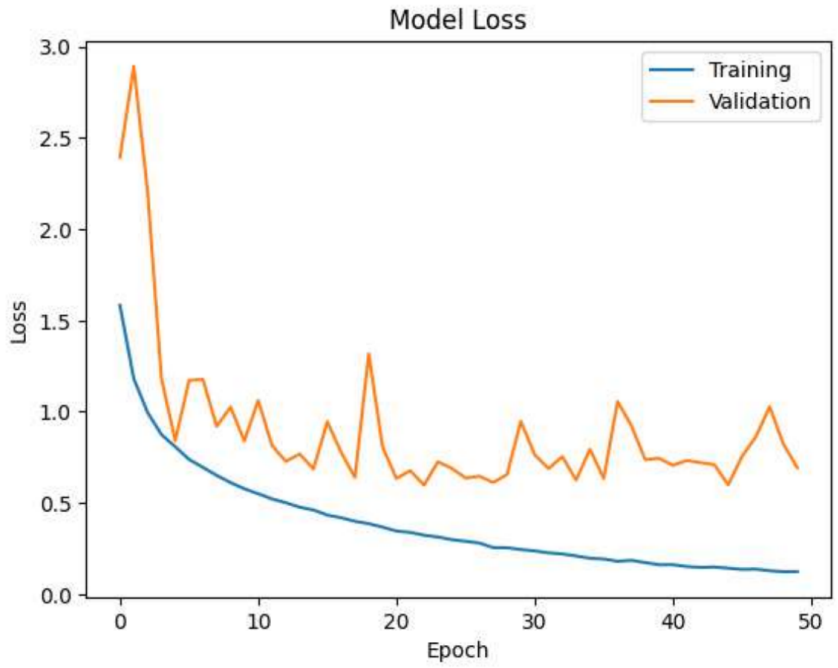
In [13]:

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.show()
```



In [14]:

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Training', 'Validation'], loc='upper right')
plt.show()
```



7. Conclusion:

ResNet (Residual Neural Network) is a deep learning algorithm that has demonstrated state-of-the-art performance on a variety of image recognition tasks, including CIFAR-10. ResNet uses skip connections to enable the training of very deep neural networks, which can improve the model's ability to learn complex features. Additionally, ResNet is able to significantly reduce the vanishing gradient problem, which is a common issue with very deep neural networks. Finally, a prominent reason for the high performance of ResNet compared to other models is due to the fact that the model uses Convolution. It is using Convolution, that the model is able to extract crucial characteristics of images. This is what the other models do not consider as they simply flatten the input prior to training. Therefore, ResNet performs better than SVM, Random Forest, and Softmax Classifier on the CIFAR-10 dataset.

8. Future Enhancements:

- Data Augmentation: One potential enhancement would be to use data augmentation techniques to artificially expand the dataset. This could involve applying techniques such as rotation, scaling, and flipping to the existing images, which can help improve the robustness of the model.
- Transfer Learning: Another potential enhancement would be to use transfer learning to leverage pre-trained models. Transfer learning involves using a pre-trained model on a different but related task, and then fine-tuning the model on the target task. This can help improve performance and reduce training time.
- Ensemble Learning: Ensemble learning is a technique that involves combining multiple models to improve performance. This could involve training multiple models using different algorithms or hyperparameters and then combining their predictions.
- Hyperparameter Tuning: The performance of machine learning models is often highly dependent on the choice of hyperparameters. One potential enhancement would be to perform an exhaustive search over the hyperparameter space to find the best combination of hyperparameters.