Alkın Er 21902669

Kadir Metehan Çalışkan 21802129

Group 13

EEE485 Term Project – Final Report

1. Project and Dataset Description

Classification of images is a widely studied field in statistical and machine learning fields. As a term project, we wanted to work on this problem. Then, we have chosen our project is to do a binary classification between automobile and motorcycle images. To do this classification, we are using three different methods. We have decided to use Naïve-Bayes, k-Nearest Neighbors (k-NN) and artificial neural network (ANN) on the image data.

For our dataset, we have merged two different datasets from different sources. The car dataset is from Kaggle [1] and the motorcycle dataset is from image-cv [2]. The car dataset consists on different types of cars such as Hyundai, Toyota, etc. On the other hand, motorcycle dataset is more mixed and has many types of motorcycles. The car dataset contains total 1798 images which 1203 for training and 595 for test. In addition, the motorcycle dataset has 2080 images which 1374 for training and 706 for test. These two datasets are not labeled. Therefore, when we are loading them to our local, we have labeled them. For car images we labeled them as 0. On the other hand, motorcycle images labeled as 1.

2. Preprocessing & Validation

To use our methods, we needed to make some preprocess and extract features to work on. For Naïve Bayes and k-Nearest Neighbors, we have used resizing, color histogram, normalization and principal component analysis (PCA). Datasets that we have merged have images with different size. Thus, before starting to work on them, we have resized so that they are in same size.

Color histogram counts the times at which the same color occurs in the image [3]. Color histogram gives us a distribution of colors in an image. By using color histogram, we extract color features of the image to work on. Then, we normalize the color histogram since all images are not same in resolution or they have not same number of pixels.

By doing normalization, we can provide scale invariance for data. In addition, by normalizing, we are avoiding bias for higher resolution images to dominate the decisions. Since higher resolution images will have higher number of pixels, they will affect classifier and make it biased. To do color histogram, resizing and normalizing, we have utilized from PIL library in python.

As a last step of preprocessing, we implemented PCA on our dataset. PCA [4] can be used as a method for preprocess. Especially supervised learning ($\{x_i,y_i\}_1^n$), PCA is a good tool to reduce dimensions of the features. Since we are doing supervised learning, we think that PCA can be a method to well fit for our use. To implement PCA method, first we have normalized our data which is done after color histogram extraction. Then, covariance matrix of dataset is computed. According to this covariance matrix, corresponding eigenvectors are computed. Eigenvectors are in the same direction with where the data has most variance. These computed eigenvectors create eigenvalues and they are sorted. With these eigenvalues, principal components are created. The components contain crucial information about the data. For our

case mostly about which colors are most important for classifying cars and motorcycles. Then, we have used proportion of variance (PVE) to choose total number of principal components to use in our PCA function. By using that number of components enable us to reduce complexity of our classifier without losing significant amount of information. As it is mentioned above, we have use color histogram to extract features. Color histogram, count many types of color. Thus, at start we had many features. By using PCA, we have reduced these high dimensional feature set. When PCA is coded no external library is used.

PCA Method:

$\Sigma = X^T X$ where Σ is coverience matrix and X is data.
$\lambda_1 \geq \lambda_2 \geq \lambda_3 \ldots \geq \lambda_p$ where λ are eigenvalues from eigenvectors

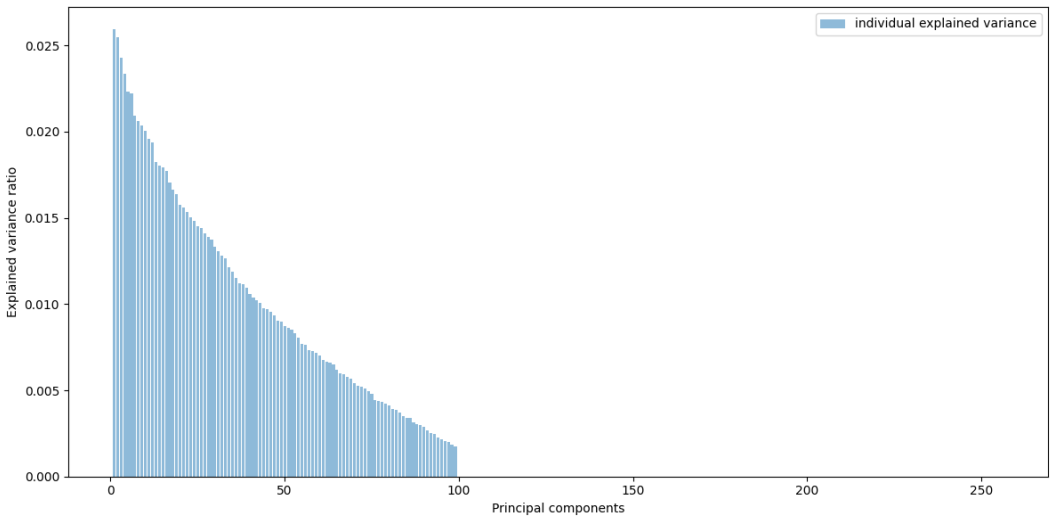

Fig. 1 PVE vs Principal Components

According to this plot, we have decided that we can use number of components as 20 for training.

K- fold cross validation:

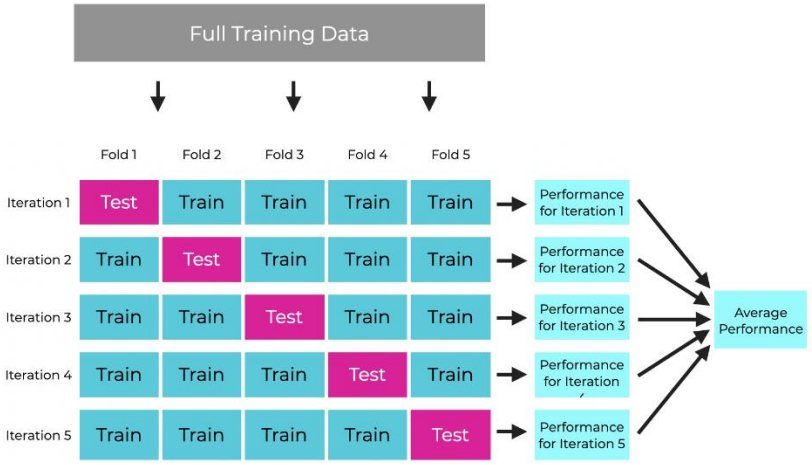

Fig. 2 K-fold cross validation visual [5]

To validate our result, we utilized k-fold cross validation and we compared our results with the test results which we trained our models with the full training data. To save time & resource, we have decided to take k as 5. We randomly split (since we do not use time series, we should shuffle and split), for each split we calculated accuracy and in the and we take the average of them to analyze overall performance of the model.

3. Used Methods and Results

3.1 Naïve Bayes Classifier

Naïve bayes classifier is a method that based on Bayes Rule. Naïve – Bayes method is a good way to do classifications. Even if it does not perform well in regression problems, it can be a good option for classification. Since our problem is a classification problem, we decided to use Naïve-Bayes classifier. In addition to them, surprisingly, it works fine in the practice.

Bayes Rule indicates likelihood of event based on from our prior knowledge of conditions which can be linked with the event itself. The formula is:

$$P(A\backslash B) = \frac{P(B\backslash A)\,P(A)}{P(B)}$$

P(A) is probability of A can occur

P(B) is probability of B can occur

P(A\B) is probability of A given that B is occurred

P(B\A) is probability of B given that A is occurred

Naïve Bayes algorithm assumes that features are conditionally are independent. Therefore, the probability of A occurs does not depend on B or any other feature [5]. This assumption can be a disadvantage of some dataset which features are strongly correlated each other. However, when they are not that highly correlated or dependent of each other. Naïve-Bayes algorithm simplifies the calculations. Therefore, it can increase computational efficiency as well. There are three types of naïve bayes classifier [6]:

1) Gaussian Naïve Bayes
2) Bernoulli Naïve Bayes
3) Multinominal Naïve Bayes

Up to now, we have assumed that our features are independent each other to use Naïve bayes. We have used Gaussian Naïve bayes in our model because Bernoulli is performing well for binary data which is consists of 1 and 0's and also multinomial is a good option for discreate cases such that problem has discreate type of data. For our case, we are using images for our data. If we use other types of bayes algorithm most likely we will have bad performing classifier.

The dataset is split 70% to 30% for training and test with number of components is 10 which is decided by the plot below (fig. 4). After training our model we have obtained 0.67 accuracy. When we use 20 components, we have obtained 0.64 accuracy with false negative rate (FNR) 0.36, true positive rate (TPR) 0.64, false positive rate (FPR) 0.36, F1 score 0.617 and true negative rate (TNR) 0.46. The training is quite fast due to computational efficiency of Naïve Bayes.

```
Confusion Matrix:
[[378 217]
 [252 454]]
```

Fig. 3 Confusion Matrix for number of components = 20

After this training, we have run several trainings for different number of principal components. Then, we plotted them.

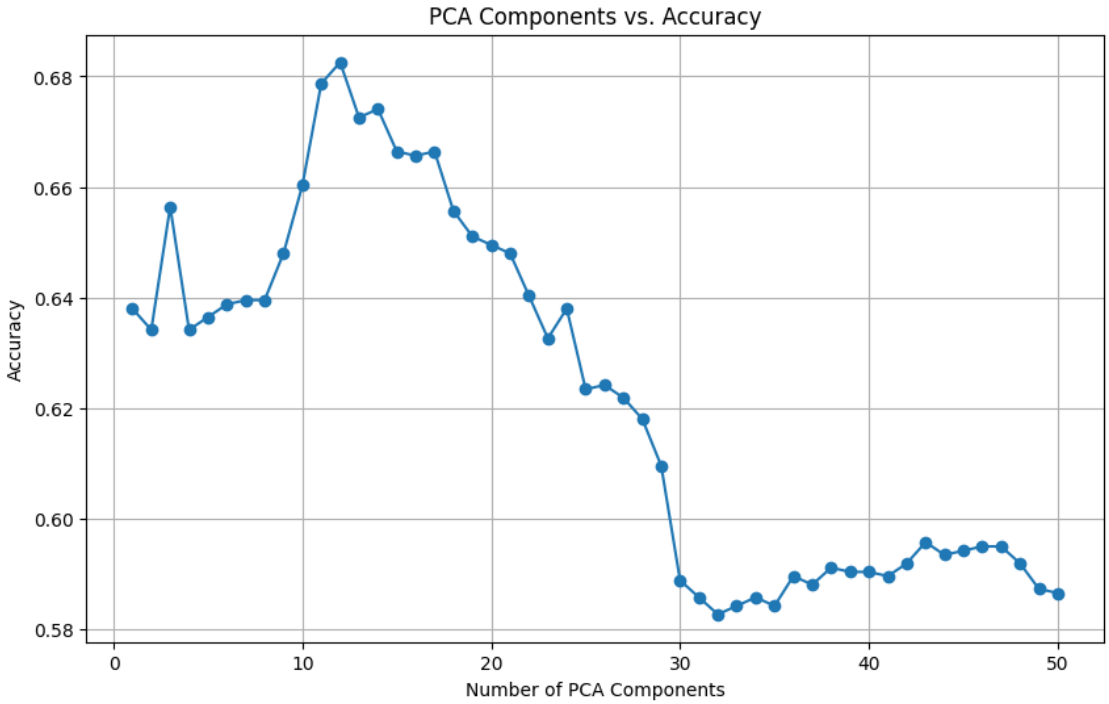

Fig. 4 Accuracy vs Number of PCA components

After this many runs, we have obtained that we have maximum accuracy where number of principal components are equal to 12. We have chosen 20 from PVE plot (Fig.1). Even we have seen that when we increase number of components our accuracy is dropped after 12. This can be a cause of naïve bayes independency assumption. When we increase number of components we take more features to train. However, these features can be more dependent each other than rest of them. Which means, when we take more features, we risk independency. They can create some correlation between each other. For example, feature 30 can be highly correlated with 9-10-11th features. This can create decrease in accuracy.

Then, we have completed k-fold cross validation to validate our model. As it is mentioned in validation part, we have conducted k-fold cross validation and obtained the results below:

Fold number	Accuracy
1. Fold	0.6611
2. Fold	0.6443
3. Fold	0.6353
4. Fold	0.6542
5. Fold	0.6671
Average Accuracy	0.6524

Table. 1 Cross Validation Accuracy Results for Naïve Bayes

As it can be interpreted from the results, there is not much difference between cross correlation and testing on whole dataset. This can be a result of the naïve bayes algorithm's basic approach to the classification. Since the model already captures related information reducing data size does not drop accuracy much.

3.2 k-Nearest Neighbors (KNN)

k-Nearest Neighbors (KNN) is a method that can be used for regression and classification. It is a widely used and simple & easy to understand method. It is a non-parametric supervised learning algorithm. It uses proximity of data points to make decisions [7]. It is a Since we are doing binary classification, we have decided that KNN is straightforward approach for solution. In addition, with the help of PCA method we can get good results.

KNN is an instance-based algorithm. This means that in training stage, it puts data in its memory and does not learn about data. All computations are made when classification is done. Therefore, it can be said that KNN algorithm is mainly memory-based learning method. It just memorizes the data instances. Then, with the calculations, it can decide the class of the data.

The algorithm work in structure as below:

1) Find closest k-trained data points to input
2) Classify it according to common class of k nearest data points

In KNN there is multiple ways to compute distance between data instances which are:

a) Euclidian Distance
b) Manhattan Distance
c) Minkowski Distance
d) Hamming Distance

Manhattan distance can be used for dataset which has many outliers. This reduces computational work of computer and increases efficiency. Moreover, Hamming distance can be used in cases like there is binary differences between each instance. Each distance method can provide good results according to used dataset. In our case, we have normalized our data instance and get rid of outliers. As a result, we have decided to use Euclidian Distance which is the most famous one among them. In final demo, we can try different distance to show that which one is performing best according to our dataset.

The formulation of Euclidian distance is given below:

$$d(x,y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

Fig. 5 Euclidian Distance Formula

There are some disadvantages of KNN algorithm. One of them is that KNN does not perform well in high dimensional data. Especially for after some K value, it does not get higher accuracies. Instead, it drops and makes more error in decisions. However, since we are using PCA to reduce dimension of features, we do not have such problem. We have used 20 principal components to train our model. In final report, we will show results with different principal components which will be obtained from PVE analysis in pre-processing part. Another disadvantage of KNN is that it can easily overfit. Lower K values can create overfit and also higher K values can also underfit. The solution of this problem is same. Using feature selection and dimension reduction helps to solve this issue [7].

As in part 3.1 Naïve Bayes Classification, same splitting is done. Then, as it mentioned above, we need to find correct k value such that it does not overfit or underfit. To prevent this, we have completed a hyperparameter search while taking k as hyperparameter. We have computed accuracies for k is between 1 and 60. The results are below.

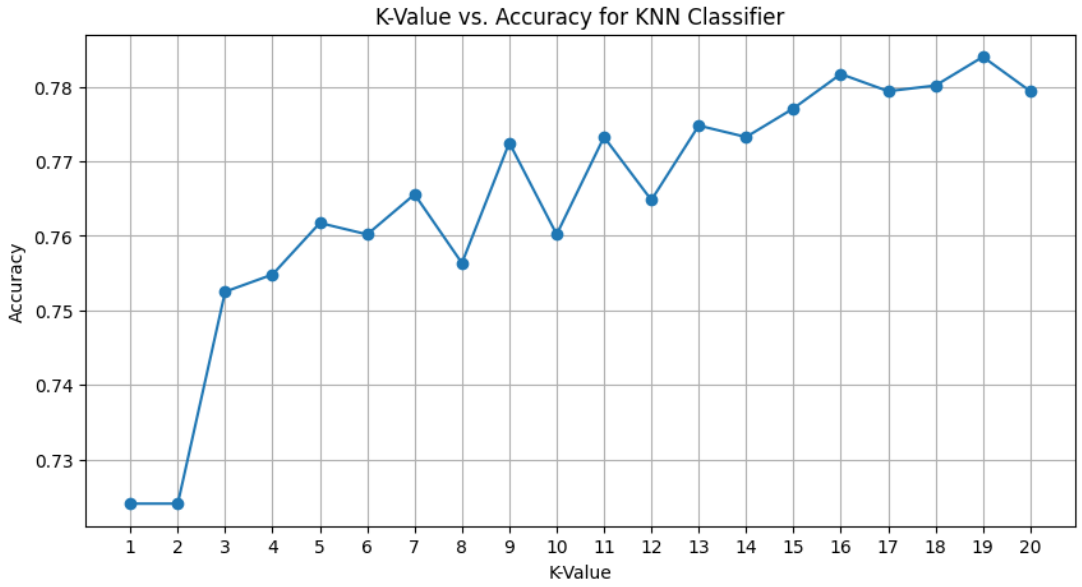

Fig. 6 Accuracy vs K value (1-20)

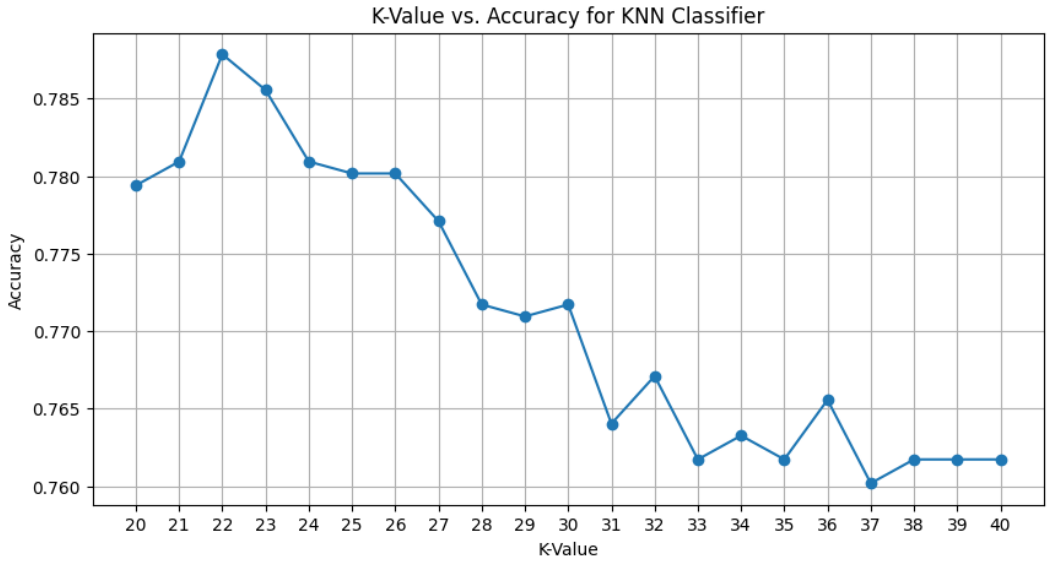

Fig. 7 Accuracy vs K value (20-40)

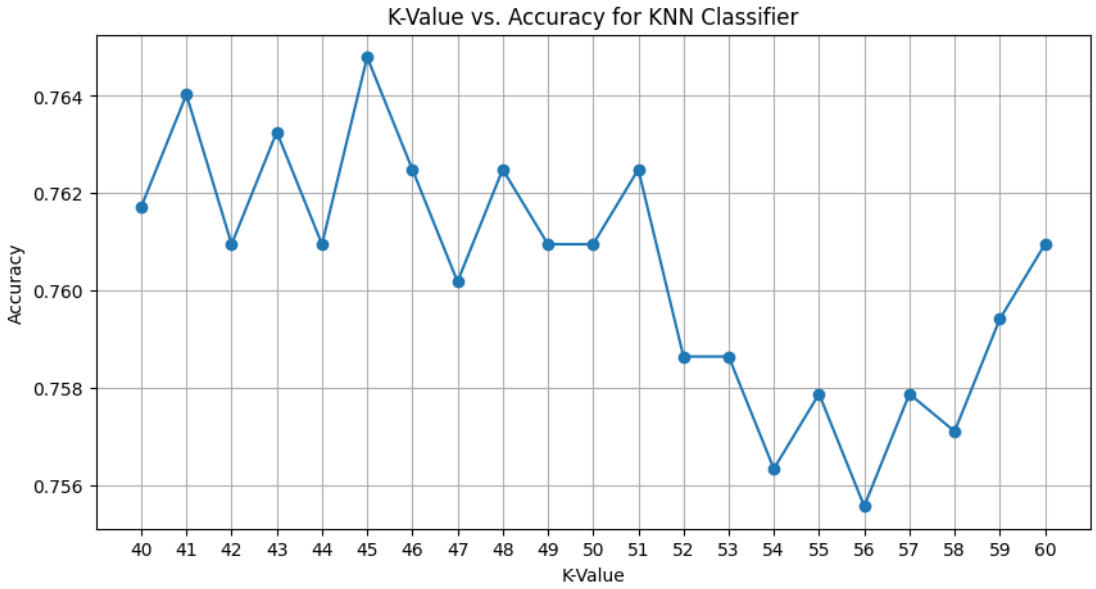

Fig. 8 Accuracy vs K value (40-60)

As it can be interpreted from the plots above, the highest accuracy is obtained when k =22. After k=22, the accuracy drops and it is inconsistent in terms of increase and decrease in the accuracy in different k values. In addition to them, increasing k value, increases model complexity as well. Thus, choosing k as 22 is quite feasible. As in the Naïve Bayes part, we have conducted a 5 fold cross – validation, and each fold we calculated our accuracy and in the end we looked at our average accuracy.

```
Confusion Matrix:
[[466 129]
 [147 559]]
```

Fig. 9 Confusion Matrix for k =22

As is can be seen from confusion matrix, we have obtained 78.8% accuracy with false negative rate (FNR) 0.24, true positive rate (TPR) 0.76, false positive rate (FPR) 0.18, F1 score 0.77 and true negative rate (TNR) 0.82 when k=22. It can be said that we have obtained good results from the model. In final, we have chosen this numbers both looking into cross validation results and hyperparameter search on K values which are plotted above.

Fold number	Accuracy
1. Fold	0.76
2. Fold	0.75
3. Fold	0.77
4. Fold	0.77
5. Fold	0.78
Average Accuracy	0.76

Table. 2 Cross Validation Accuracy Results for K-nearest neighbors

When we completed cross-validation, we obtained maximum 78% accuracy. We have faced similar outputs in naïve bayes as well. Our accuracy was higher when we use all of our dataset for training. This can be result of k-NN needs more data to classify better.

3.3 Artificial Neural Network (ANN)

For the last method, we have used artificial neural network. Artificial neural networks can be used for binary classification problem. It is well-suited for our purpose. In addition, we have wanted to gain some first-hand experience in coding neural network without using any ML related libraries. This have provided us insight and increased our knowledge about neural networks. As we did in other methods, we have labeled images as 0 and 1.

Before creating our neural network architecture, we have conducted some pre-processing. We have resized our images to put them into same size since in dataset, we have different sized image. To reduce complexity, we have converted them to greyscale. Then, we put them into arrays and flattened them to give pixels values between 0 and 1.

For network architecture, since our task is to do binary classification, we have used single neuron on output layer. As activation function, we have used sigmoid function after putting it into linear function to obtain values between [0,1] as output. For start, we have decided a threshold value as 0.5. This would indicate that the outputs which are below 0.5 are classified as cars and above 0.5 are classified as motorcycles. For the input layer, we have used 1024 neurons which are corresponding to flattened images which are 32x32.

We have started with one hidden layer with 10 neurons. However, the results were not acceptable. The accuracy was 53%. Therefore, we have though that the model is too simple for this classification. Then we have started to increase the neuron size. When we start to increase it, the training time is also increased. Since model becomes more complex. When, we increased it to 100 neurons, the accuracy level still was not acceptable even if it is better than 53%. In previous methods, we have already achieved 67% for Naïve-bayes and 78 % for k-NN. Since we can increase the complexity of ANN, we should have got much higher accuracy than k-NN and Naïve-bayes. Then we have added, second hidden layer with 50 neuron size. We have obtained 88% accuracy. Then, our model's training time significantly increased. We wanted to push little more with our architecture. To do that we wanted to avoid overfitting, therefore we added dropout layer as well. It also helps to capture robust features of the dataset. Dropout layer randomly set some of the neurons to zero. The remaining neurons are scaled up by (1/1-dropout_rate) to satisfy expected gain of the output. It should be noted that dropout is used in only for training part not testing. This is resulted our accuracy to be 88.9%.

For activation function in hidden layers, we have chosen ReLU function for simplicity in calculations of back propagation. In addition, it adds some randomness into data. Therefore, it increases the chance of reaching a global minimum. To update weights, we have utilized stochastic gradient descent to minimize the loss. As loss function, we have chosen residual sum of squares which is below:

$$E(W) = \sum_D 0.5(y - d)^2 \text{ where D is all data instances}$$

Before the starting to update weights and bias. We have initialized our weights as a normal distribution. Since the weights are updated according to previous layer, when we set them zero, they can get too small. To prevent this and provide stability to training we initialized them as a normal (Gaussian Distribution). Biases are initialized as 0 for start.

Back propagation formula is below:

$$w(n+1)^l = w(n)^l - \eta \nabla E(w(n)) \quad (1)$$

$$w(n+1)^l = w(n)^l - \eta \frac{\partial E(w(n))}{\partial w(n)} \quad (2)$$

$$w(n+1)^l = w(n)^l + \eta \delta j^l x^{l-1} \quad (3)$$

By using above architecture, we have trained our model. We have chosen our learning rate as 0.005. The learning rate is chosen accordingly trial and error. We tried to train it with different learning rates but accuracy significantly dropped. Since the training time is long, we could not do a search on learning rate. We have tried the values between [0.001, 0.1]. We have taken dropout rate as 0.2. We tried again with the vales [0,0.1.0.2,0.3] and the best performance is obtained when the dropout rate is equal to 0.2. For batch size, we have chosen 16. As in the previous parts, we have conducted a 5-fold cross validation to evaluate the overall performance of the model.

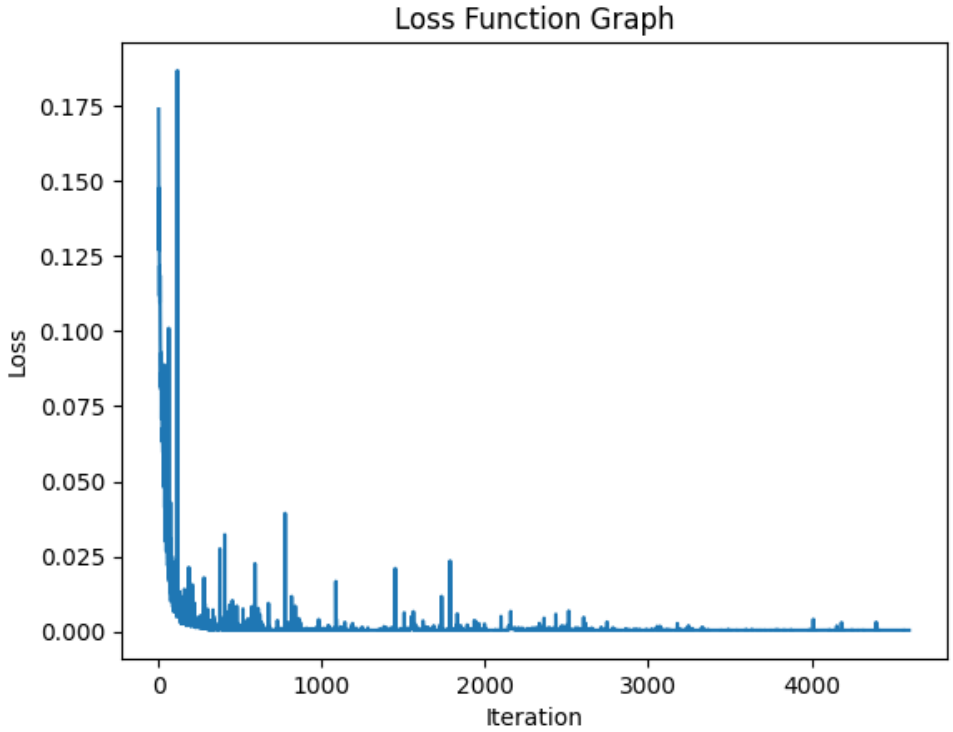

Fig. 10 Loss Plot for Training

When we looked at the loss graph, it is very fluctuating. To choose the best iteration number we have looked up the plot. We have chosen 1000 iteration where the loss is more stable. In addition, this is the point where the accuracy gets highest. After this point, model start to overfit and our test accuracy drops.

Then cross-validation results are below:

Fold number	Accuracy
1. Fold	0.833
2. Fold	0.95
3. Fold	0.85
4. Fold	0.813
5. Fold	0.79
Average Accuracy	0.86

Table. 3 Cross Validation Accuracy Results for ANN

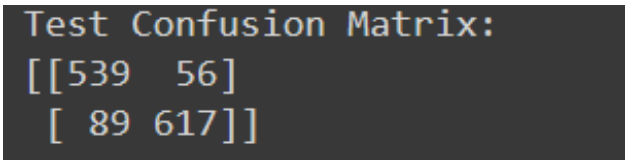

Fig. 11 Confusion Matrix for training all training data.

According to results, we have seen that the best performance is gained when the model is trained all over the training data which is similar to other two models. The cross-validation overall performance is 86% accuracy. However, when we train our model on all over training data without splitting for validation, we have obtained 88.9% accuracy with TPR = 0.858, FPR= 0.083, FNR=0.1417 and with precision = 0.9 and also F1 score= 0.88

4. Contribution of Members

The workload split as below:
Alkın: Naïve Bayes, KNN
Metehan: Neural Network, feature extraction

5. Conclusion

In conclusion, when we consider all results that have been shown above, we can say that the best performing model is neural network according to our performance metrics which are accuracy, f1 score, TPR, FPR and etc. This result is actually expected because neural network is the model which is the most complex one. The other two methods are more straightforward and computationally efficient methods. However, since we are working with image dataset and we want to classify these images, these two methods are performing poorly when it is compared to neural network. After progress demo, we have added cross validation to evaluate better first two methods but the cross validation did not help much about increasing the performance. When we split our training dataset for validation, we have seen that most of the time our accuracy drops. Therefore, we decided to use all training dataset for training for all methods that we have completed. Even if naïve bayes and k-NN are performing poorly when it is compared to ANN, we think that they are working great for this dataset and with their simplicity in the architecture. For example, in naïve bayes, we are losing the information about shapes of vehicles since we transform our pixels to color diagrams. We are changing our space to color space. This would indicate that we lose information about the shape, edges and sizes of vehicles. We are just classifying images according to their color diagrams. On the other hand, in neural network, we do not lose that information, we just use simple pre-processing techniques to do training. Therefore, the final results are not surprising. Moreover, it should be noted that our dataset contains some noisy images such as the images with texts on them. We have not deleted them because we think that in real life, we cannot get rid of noise that easily. Hence, we wanted to train our models with a bit noisy data to see results.

6. References

[1] K. Kumar, “Car images dataset,” Kaggle, https://www.kaggle.com/datasets/kshitij192/cars-image-dataset (accessed Apr. 21, 2024).
[2] “Image datasets for Computer Vision and machine learning,” Download Motorbike labeled image classifcation dataset labeled image dataset, https://images.cv/dataset/motorbike-image-classification-dataset (accessed Apr. 21, 2024).
[3] C. L. Novak and S. A. Shafer, “Anatomy of a color histogram,” Proceedings 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. doi:10.1109/cvpr.1992.223129
[4] S. Yazdani, J. Shanbehzadeh, and M. T. Manzuri Shalmani, “RPCA: A novel preprocessing method for PCA,” Advances in Artificial Intelligence, vol. 2012, pp. 1–7, Dec. 2012. doi:10.1155/2012/484595
[5] J. Ebner, “Cross validation, explained,” Sharp Sight, https://www.sharpsightlabs.com/blog/cross-validation-explained/ (accessed May 10, 2024).
[6] I. Rish, “An empirical study of the naive Bayes classifier,” 2001. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2825733f97124013e8841b3f8a0f5bd4ee4af88a
[7] “What is the K-nearest neighbors algorithm?,” IBM, https://www.ibm.com/topics/knn (accessed Apr. 21, 2024) [1] “What is the K-nearest neighbors algorithm?,” IBM, https://www.ibm.com/topics/knn (accessed Apr. 21, 2024).

APPENDIX

APPENDIX A – NAÏVE BAYES CLASSIFER

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

import os
import numpy as np
from PIL import Image, ImageFilter
import matplotlib.pyplot as plt

base_path = '/content/drive/MyDrive/Data_Project/'

def load_images(directory, label, size=(60, 60)):
    data = []
    for filename in os.listdir(directory):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            img_path = os.path.join(directory, filename)
            try:
                img = Image.open(img_path).convert('L')
                img = img.resize(size, Image.ANTIALIAS)
                data.append((img, label))
            except IOError:
                print(f"Cannot open image: {img_path}")
    return data

def to_grayscale(img):
    return img.convert('L')

def edge_detection(img):
    return img.filter(ImageFilter.FIND_EDGES)

def color_histogram(img):
    return img.histogram()

def normalize(features):
    return features / np.linalg.norm(features)


def pca(X, num_components):
    # Center the data
    X_meaned = X - np.mean(X, axis=0)
    # Calculate covariance matrix
    cov_mat = np.cov(X_meaned, rowvar=False)
    # Eigen decomposition
    eigen_values, eigen_vectors = np.linalg.eigh(cov_mat)
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvalues = eigen_values[sorted_index]
    sorted_eigenvectors = eigen_vectors[:, sorted_index]
    eigenvector_subset = sorted_eigenvectors[:, :num_components]
    total_variance = np.sum(sorted_eigenvalues)
    explained_variances = sorted_eigenvalues[:num_components] / total_variance
    X_reduced = np.dot(X_meaned, eigenvector_subset)
```

```
    return X_reduced, eigenvector_subset, explained_variances

class NaiveBayesClassifier:
    def __init__(self):
        self.model = {}

    def fit(self, features, labels):
        self.labels = np.unique(labels)
        for label in self.labels:
            indices = np.where(labels == label)
            self.model[label] = {
                'prior': len(indices[0]) / len(labels),
                'mean': np.mean(features[indices], axis=0),
                'var': np.var(features[indices], axis=0),
            }

    def predict(self, features):
        results = []
        for feature in features:
            posteriors = []
            for label in self.labels:
                prior = np.log(self.model[label]['prior'])
                likelihood = -0.5 * np.sum(np.log(2. * np.pi * self.model[label]['var']))
                likelihood -= 0.5 * np.sum(((feature - self.model[label]['mean']) ** 2) /
                                           self.model[label]['var'])
                posteriors.append(prior + likelihood)
            results.append(self.labels[np.argmax(posteriors)])
        return np.array(results)


train_autos = load_images(base_path + 'Cars Dataset/train', label=0)
train_motorcycles = load_images(base_path + 'Motorcycle Dataset/train/motorbike', label=1)
test_autos = load_images(base_path + 'Cars Dataset/test', label=0)
test_motorcycles = load_images(base_path + 'Motorcycle Dataset/test/motorbike', label=1)

# Prepare dataset
train_data = train_autos + train_motorcycles
test_data = test_autos + test_motorcycles

# Process training images
train_images, train_labels = zip(*train_data)
train_images = [to_grayscale(img) for img in train_images]
train_histograms = [color_histogram(img) for img in train_images]
train_features, train_components, train_explained_variances = pca(np.array([normalize(hist)
for hist in train_histograms]), 20)

# Process testing images
test_images, test_labels = zip(*test_data)
test_images = [to_grayscale(img) for img in test_images]
test_histograms = [color_histogram(img) for img in test_images]
test_features, test_components, test_explained_variances = pca(np.array([normalize(hist) for
hist in test_histograms]), 20)

# Train Naive Bayes Classifier
classifier = NaiveBayesClassifier()
```

```
classifier.fit(train_features, np.array(train_labels))

predictions = classifier.predict(test_features)

accuracy = np.mean(predictions == np.array(test_labels))
print(f'Accuracy: {accuracy}')

# Display confusion matrix
def confusion_matrix(true_labels, pred_labels, classes):
    matrix = np.zeros((classes, classes), int)
    for true, pred in zip(true_labels, pred_labels):
        matrix[true][pred] += 1
    return matrix

cm = confusion_matrix(test_labels, predictions, len(np.unique(train_labels)))
print("Confusion Matrix:")
print(cm)

train_labels = np.array(train_labels)
test_labels = np.array(test_labels)

print("Training labels:", np.bincount(train_labels))
print("Testing labels:", np.bincount(test_labels))

import matplotlib.pyplot as plt

def evaluate_pca_components(max_components, train_data, train_labels, test_data,
test_labels):
    accuracies = []
    components_range = range(1, max_components + 1)
    for num_components in components_range:
        # Apply PCA on training data
        train_features, _, _ = pca(np.array([normalize(hist) for hist in train_data]),
num_components)
        test_features, _, _ = pca(np.array([normalize(hist) for hist in test_data]),
num_components)

        # Train Naive Bayes Classifier
        classifier = NaiveBayesClassifier()
        classifier.fit(train_features, np.array(train_labels))

        # Predict on test data
        predictions = classifier.predict(test_features)

        # Calculate accuracy
        accuracy = np.mean(predictions == np.array(test_labels))
        accuracies.append(accuracy)
        print(f"Accuracy with {num_components} PCA components: {accuracy}")

    return components_range, accuracies

normalized_train_histograms = [normalize(hist) for hist in train_histograms]
normalized_test_histograms = [normalize(hist) for hist in test_histograms]
```

```
components_range, accuracies = evaluate_pca_components(50, normalized_train_histograms,
np.array(train_labels), normalized_test_histograms, np.array(test_labels))

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(components_range, accuracies, marker='o')
plt.title('PCA Components vs. Accuracy')
plt.xlabel('Number of PCA Components')
plt.ylabel('Accuracy')
plt.grid(True)
plt.show()


def pca_with_eig(X, num_components):
    X_meaned = X - np.mean(X, axis=0)
    cov_mat = np.cov(X_meaned, rowvar=False)
    eigen_values, eigen_vectors = np.linalg.eigh(cov_mat)
    sorted_index = np.argsort(eigen_values)[::-1]
    eigen_values = eigen_values[sorted_index]
    sorted_eigenvectors = eigen_vectors[:, sorted_index]
    eigenvector_subset = sorted_eigenvectors[:, 0:num_components]
    X_reduced = np.dot(eigenvector_subset.transpose(), X_meaned.transpose()).transpose()
    return X_reduced, eigenvector_subset, eigen_values

train_features, train_components, train_eigen_values =
pca_with_eig(np.array([normalize(hist) for hist in train_histograms]), 50)
test_features, _, _ = pca_with_eig(np.array([normalize(hist) for hist in test_histograms]), 50)


def plot_principal_components(components, eigen_values):
    total_variance = np.sum(eigen_values)
    explained_variances = eigen_values / total_variance
    plt.figure(figsize=(12, 6))
    plt.bar(range(1, len(explained_variances) + 1), explained_variances, alpha=0.5,
align='center', label='individual explained variance')
    plt.ylabel('Explained variance ratio')
    plt.xlabel('Principal components')
    plt.legend(loc='best')
    plt.tight_layout()

    num_components = components.shape[1]
    n_rows = (num_components + 9) // 10
    n_cols = 10 if num_components >= 10 else num_components

    plt.figure(figsize=(20, max(2, n_rows * 2
    for i in range(num_components):
        ax = plt.subplot(n_rows, n_cols, i + 1)
        plt.imshow(components[:, i].reshape(16, 16), cmap='gray'
        plt.title(f'PC {i+1}')
        plt.axis('off')
    plt.show()
```

APPENDIX- B k-NN Classifer

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

```
import numpy as np
from PIL import Image, ImageFilter
import matplotlib.pyplot as plt
from collections import Counter

# Define the path to your dataset
base_path = '/content/drive/MyDrive/Data_Project/'

def load_images(directory, label):
    data = []
    for filename in os.listdir(directory):
        if filename.lower().endswith(('.png', '.jpg', '.jpeg')):
            img_path = os.path.join(directory, filename)
            try:
                img = Image.open(img_path)
                data.append((img, label))
            except IOError:
                print(f"Cannot open image: {img_path}")
    return data

def to_grayscale(img):
    return img.convert('L')

def edge_detection(img):
    return img.filter(ImageFilter.FIND_EDGES)

def color_histogram(img):
    return img.histogram()

def normalize(features):
    return features / np.linalg.norm(features)

def pca(X, num_components):
    X_meaned = X - np.mean(X , axis = 0)
    cov_mat = np.cov(X_meaned , rowvar = False)
    eigen_values , eigen_vectors = np.linalg.eigh(cov_mat)
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvectors = eigen_vectors[:,sorted_index]
    eigenvector_subset = sorted_eigenvectors[:,0:num_components]
    X_reduced = np.dot(eigenvector_subset.transpose() , X_meaned.transpose() ).transpose()
    return X_reduced

class KNearestNeighbors:
    def __init__(self, k=3):
        self.k = k

    def fit(self, features, labels):
        self.train_features = features
        self.train_labels = labels

    def predict(self, features):
        predictions = []
        for point in features:
            distances = [np.sqrt(np.sum((point - other) ** 2)) for other in self.train_features]
            k_indices = np.argsort(distances)[:self.k]
```

```
            k_nearest_labels = [self.train_labels[i] for i in k_indices]
            most_common = Counter(k_nearest_labels).most_common(1)
            predictions.append(most_common[0][0])
        return np.array(predictions)

def find_best_k(train_features, train_labels, test_features, test_labels, max_k):
    accuracies = []
    for k in range(1, max_k + 1):
        knn = KNearestNeighbors(k=k)
        knn.fit(train_features, train_labels)
        predictions = knn.predict(test_features)
        accuracy = np.mean(predictions == test_labels)
        accuracies.append(accuracy)
        print(f'Accuracy for k={k}: {accuracy}')
    return range(1, max_k + 1), accuracies

train_autos = load_images(base_path + 'Cars Dataset/train', label=0)
train_motorcycles = load_images(base_path + 'Motorcycle Dataset/train/motorbike', label=1)
test_autos = load_images(base_path + 'Cars Dataset/test', label=0)
test_motorcycles = load_images(base_path + 'Motorcycle Dataset/test/motorbike', label=1)

train_data = train_autos + train_motorcycles
test_data = test_autos + test_motorcycles

train_images, train_labels = zip(*train_data)
train_images = [to_grayscale(img) for img in train_images]
train_edges = [edge_detection(img) for img in train_images]
train_histograms = [color_histogram(img) for img in train_images]
train_features = np.array([normalize(hist) for hist in train_histograms])
train_features = pca(train_features, 20)

test_images, test_labels = zip(*test_data)
test_images = [to_grayscale(img) for img in test_images]
test_edges = [edge_detection(img) for img in test_images]
test_histograms = [color_histogram(img) for img in test_images]
test_features = np.array([normalize(hist) for hist in test_histograms])
test_features = pca(test_features, 20)

max_k = 20
k_range, accuracies = find_best_k(train_features, train_labels, test_features, test_labels,
max_k)

plt.figure(figsize=(10, 5))
plt.plot(k_range, accuracies, marker='o')
plt.title('K-Value vs. Accuracy for KNN Classifier')
plt.xlabel('K-Value')
plt.ylabel('Accuracy')
plt.xticks(range(1, max_k + 1))
plt.grid(True)
plt.show()

knn = KNearestNeighbors()
knn.fit(train_features, train_labels)
predictions = knn.predict(test_features)
# Display confusion matrix
```

```
def confusion_matrix(true_labels, pred_labels, classes):
    matrix = np.zeros((classes, classes), int)
    for true, pred in zip(true_labels, pred_labels):
        matrix[true][pred] += 1
    return matrix

cm = confusion_matrix(test_labels, predictions, len(np.unique(train_labels)))
print("Confusion Matrix:")
print(cm)

def find_best_k(train_features, train_labels, test_features, test_labels, max_k):
    accuracies = []
    for k in range(20, max_k + 1):
        knn = KNearestNeig          hbors(k=k)
        knn.fit(train_features, train_labels)
        predictions = knn.predict(test_features)
        accuracy = np.mean(predictions == test_labels)
        accuracies.append(accuracy)
        print(f'Accuracy for k={k}: {accuracy}')
    return range(20, max_k + 1), accuracies


max_k = 40
k_range, accuracies = find_best_k(train_features, train_labels, test_features, test_labels,
max_k)

plt.figure(figsize=(10, 5))
plt.plot(k_range, accuracies, marker='o')
plt.title('K-Value vs. Accuracy for KNN Classifier')
plt.xlabel('K-Value')
plt.ylabel('Accuracy')
plt.xticks(range(20, max_k + 1))
plt.grid(True)
plt.show()

def find_best_k(train_features, train_labels, test_features, test_labels, max_k):
    accuracies = []
    for k in range(40, max_k + 1):
        knn = KNearestNeighbors(k=k)
        knn.fit(train_features, train_labels)
        predictions = knn.predict(test_features)
        accuracy = np.mean(predictions == test_labels)
        accuracies.append(accuracy)
        print(f'Accuracy for k={k}: {accuracy}')
    return range(40, max_k + 1), accuracies


max_k = 60
k_range, accuracies = find_best_k(train_features, train_labels, test_features, test_labels,
max_k)

plt.figure(figsize=(10, 5))
plt.plot(k_range, accuracies, marker='o')
plt.title('K-Value vs. Accuracy for KNN Classifier')
plt.xlabel('K-Value')
```

```
plt.ylabel('Accuracy')
plt.xticks(range(40, max_k + 1))
plt.grid(True)
plt.show()
```

APPENDIX C

ANN

```
from google.colab import drive
drive.mount('/content/drive')
import os
import numpy as np
from PIL import Image, ImageFilter
import matplotlib.pyplot as plt


base_path = '/content/drive/MyDrive/Data_Project/'

def load_images(base_path, label, img_size=(32, 32)):
    """Load images from the given path and return them with labels."""
    images = []
    labels = []
    valid_extensions = ('.jpg', '.jpeg', '.png', '.bmp', '.tiff', '.gif')
    for filename in os.listdir(base_path):
        if filename.endswith(valid_extensions):
            # Construct the full path to the image
            file_path = os.path.join(base_path, filename)
            # Open the image and resize it
            with Image.open(file_path) as img:
                img = img.resize(img_size)  # Resize image
                img = img.convert('L')  # Convert to grayscale
                img_array = np.array(img)  # Convert image to numpy array
                images.append(img_array.flatten())  # Flatten the image and append to list
                labels.append(label)
    return np.array(images), np.array(labels)



train_autos, train_auto_labels = load_images(base_path + 'Cars Dataset/train', label=0)
train_motorcycles, train_motorcycle_labels = load_images(base_path + 'Motorcycle
Dataset/train/motorbike', label=1)
test_autos, test_auto_labels = load_images(base_path + 'Cars Dataset/test', label=0)
test_motorcycles, test_motorcycle_labels = load_images(base_path + 'Motorcycle
Dataset/test/motorbike', label=1)


X_train = np.vstack((train_autos, train_motorcycles))
y_train = np.hstack((train_auto_labels, train_motorcycle_labels))
X_test = np.vstack((test_autos, test_motorcycles))
y_test = np.hstack((test_auto_labels, test_motorcycle_labels))
X_train = X_train / 255.0
X_test = X_test / 255.0


class SimpleNN:
```

```
    def __init__(self, layer_sizes, dropout_rate=0.5):
        self.layer_sizes = layer_sizes
        self.dropout_rate = dropout_rate
        self.weights, self.biases = self.initialize_weights()
        self.activations = []
        self.loss_history = []

    def initialize_weights(self):
        weights = []
        biases = []
        for i in range(len(self.layer_sizes) - 1):
            weights.append(np.random.randn(self.layer_sizes[i], self.layer_sizes[i + 1]) *
np.sqrt(2. / self.layer_sizes[i]))
            biases.append(np.zeros((1, self.layer_sizes[i + 1])))
        return weights, biases

    def feedforward(self, X, is_training=True):
        activation = X
        self.activations = [X]
        for i in range(len(self.weights)):
            z = np.dot(activation, self.weights[i]) + self.biases[i]
            activation = np.maximum(z, 0)  # ReLU activation
            if i < len(self.weights) - 1 and is_training:
                mask = (np.random.rand(*activation.shape) > self.dropout_rate) / (1.0 -
self.dropout_rate)
                activation *= mask
            self.activations.append(activation)
        return activation

    def backprop(self, X, y, learning_rate):
        activations = self.activations
        deltas = [activations[-1] - y[:, None]]

        for i in range(len(self.weights) - 1, 0, -1):
            delta = np.dot(deltas[0], self.weights[i].T) * (activations[i] > 0).astype(float)
            deltas.insert(0, delta)

        for i in range(len(self.weights)):
            d_weights =np.dot(activations[i].T, deltas[i])
            d_biases = np.sum(deltas[i], axis=0, keepdims=True)
            self.weights[i] -= learning_rate * d_weights
            self.biases[i] -= learning_rate * d_biases

    def compute_loss(self, X, y):
      predictions = self.feedforward(X, is_training=False)
      loss = np.mean((predictions - y[:, None]) ** 2)
      return loss

    def train(self, X, y, batch_size, iterations, learning_rate):
        for i in range(iterations):
            indices = np.random.permutation(len(X))
            for start_idx in range(0, len(X) - batch_size + 1, batch_size):
                excerpt = indices[start_idx:start_idx + batch_size]
                self.feedforward(X[excerpt], is_training=True)
                self.backprop(X[excerpt], y[excerpt], learning_rate)
```

```
            loss = self.compute_loss(X, y)
            self.loss_history.append(loss)

def evaluate_model(nn, X, y):
    predictions = nn.feedforward(X, is_training=False) >= 0.5
    accuracy = np.mean(predictions == y[:, None])
    return accuracy

def plot_loss(nn):
    plt.plot(nn.loss_history)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title('Loss Function Graph')
    plt.show()
# Parameters
input_size = 32 * 32
hidden_sizes = [100, 50]
output_size = 1
layer_sizes = [input_size] + hidden_sizes + [output_size]

nn = SimpleNN(layer_sizes, dropout_rate=0.2)
nn.train(X_train, y_train, 16, 4600, 0.005)
print(f"Training Accuracy: {evaluate_model(nn, X_train, y_train) * 100:.2f}%")
print(f"Test Accuracy: {evaluate_model(nn, X_test, y_test) * 100:.2f}%")



plot_loss(nn)

def cross_validate(nn_class, X, y, k_folds=5, batch_size=16, iterations=4700,
learning_rate=0.005):
    fold_size = len(X) // k_folds
    accuracies = []

    for i in range(k_folds):
        # Split the data into training and validation sets
        start, end = i * fold_size, (i + 1) * fold_size
        X_val, y_val = X[start:end], y[start:end]
        X_train = np.concatenate([X[:start], X[end:]])
        y_train = np.concatenate([y[:start], y[end:]])

        nn = nn_class(layer_sizes, dropout_rate=0.2)
        nn.train(X_train, y_train, batch_size, iterations, learning_rate)

        # Evaluate the model on the validation set
        val_accuracy = evaluate_model(nn, X_val, y_val)
        accuracies.append(val_accuracy)

        print(f"Fold {i + 1}/{k_folds} - Validation Accuracy: {val_accuracy * 100:.2f}%")

    # Calculate the average accuracy across all folds
    avg_accuracy = np.mean(accuracies)
    print(f"Average Cross-Validation Accuracy: {avg_accuracy * 100:.2f}%")
    return avg_accuracy
```

```
cross_validate(SimpleNN, X_train, y_train, k_folds=5)

def compute_confusion_matrix(y_true, y_pred, num_classes=2):
    """Compute the confusion matrix for a binary classification problem."""
    cm = np.zeros((num_classes, num_classes), dtype=int)
    for true, pred in zip(y_true, y_pred):
        cm[int(true), int(pred)] += 1
    return cm

# Generate predictions
train_predictions = nn.feedforward(X_train, is_training=False) >= 0.5
test_predictions = nn.feedforward(X_test, is_training=False) >= 0.5

train_predictions = train_predictions.flatten()
test_predictions = test_predictions.flatten()

# Compute confusion matrices
train_confusion_matrix = compute_confusion_matrix(y_train, train_predictions)
test_confusion_matrix = compute_confusion_matrix(y_test, test_predictions)

print("Train Confusion Matrix:")
print(train_confusion_matrix)

print("Test Confusion Matrix:")
print(test_confusion_matrix)
```