Neural Networks - Deep Learning First Assignment

Dimitrios Karatis 10775, *Electrical and Computer Engineering, AUTH*

***Abstract*—This document provides an overview of the neural network model training and evaluation process using the CIFAR-10 dataset. The focus is on leveraging convolutional neural networks (CNNs) to classify images into ten categories, such as airplanes, cars, and animals, making this a challenging yet foundational problem in computer vision.**

***Index Terms*—Neural networks, deep learning, CIFAR-10, image classification, computer vision.**

I. Introduction

TO effectively train and evaluate the neural network model presented in this work, the **CIFAR-10 Dataset** was selected as the primary data source (available at: CIFAR-10 Dataset). CIFAR-10 is a popular image classification dataset containing 60,000 32x32 color images divided into ten classes: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. This dataset serves as a benchmark in computer vision and deep learning for evaluating classification model performance.

II. Dataset Description

The CIFAR-10 dataset consists of 50,000 training images and 10,000 test images, each labeled with one of the ten classes. Due to its diversity, it presents a complex classification problem and provides a comprehensive platform for testing and training CNN architectures in the context of image recognition.

III. Interim Assignment

A. Loading and Preparing the Dataset

The archive of this dataset contains the files data_batch_1, data_batch_2, ..., data_batch_5, as well as a test_batch and a batches.meta file, which contains the label names in terms of string charachters, and which I didn't use. Each of these files is a Python "pickled" object produced with cPickle. To load the image data and labels found in the CIFAR's archive I used the ***unpickle*** function found in the official CIFAR's website. This function deserializes a binary file containing a pickled object and returns a dictionary with the corresponding image data and labels. This way, each of the batch files contains a dictionary with the following elements:

- **data** – a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.
- **labels** – a list of 10000 numbers in the range 0-9. The number at index i indicates the label of the ith image in the array data.

This document was prepared by Dimitrios Karatis as part of the first assignment for the Neural Networks course at Aristotle University of Thessaloniki.

Following that, the ***load_cifar10_data*** function reads and combines the five training batches from the specified directory, each containing images and corresponding labels. The images (x_train) and labels (y_train) from each batch are appended and then converted into arrays. It then loads a separate test batch for the test images (x_test) and labels (y_test). The ***preprocess*** matrix indicates whether or not normalization, augment by rotation and validation set (for Early Stopping) will be used or not. If the first element of the matrix is set to 1, it normalizes the pixel values of all the images to (0,1), likewise for the other elements. Finally, it returns the training, test and validation (if needed) image arrays along with the corresponding labels.

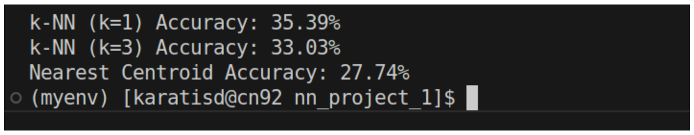
```python
# Function to load cifar-10 data
# preprocess is a matrxix indicating:
# [using normalize, using roatations, using
    validation set for EarlyStopping]
def load_cifar10_data(self, data_dir, preprocess
):
    x_train, y_train, x_test, y_test, x_val,
        y_val = [], [], [], [], [], []

    # Load training folder
    for i in range(1, 6):  # Load the 5 training
         batches
        batch = self.unpickle(f"{data_dir}/
            data_batch_{i}")
        x_train.append(batch[b'data'])
        y_train.extend(batch[b'labels'])
    x_train = np.concatenate(x_train)
    y_train = np.array(y_train)

    # Load test folder
    test_batch = self.unpickle(f"{data_dir}/
        test_batch")
    x_test = test_batch[b'data']
    y_test = np.array(test_batch[b'labels'])

    # Check if validation split is enabled
    if preprocess[2] == 1:
        # Split the training data into training
            and validation sets
        x_train, x_val, y_train, y_val =
            train_test_split(x_train, y_train,
            test_size=0.2, random_state=42)
        # Check if rotation augmentation is enabled
        if preprocess[1] == 1:
            x_train, y_train = self.custom_rotation(
                x_train, y_train)

        # Check if normalization is enabled
        if preprocess[0] == 1:
            x_train, x_test = self.normalize_pixels(
                x_train, x_test)
            if len(x_val) > 0:  # Normalize x_val if
                 it exists
                x_val = self.normalize_pixels(x_val)

        return x_train, y_train, x_test, y_test,
            x_val, y_val
```

B. Training and Evaluating Classifiers

After loading the dataset we the need to initializing the classifiers, and train every single one of them to a set of training images (img_train) with corresponding labels (label_train). After training, the classifiers are evaluated on a test set (img_test, label_test), where they predict labels for the test images and compute accuracy based on these predictions. The initialization, training and evaluation has been done by using the KNeighborsClassifier and NearestCentroid packages from sklearn.

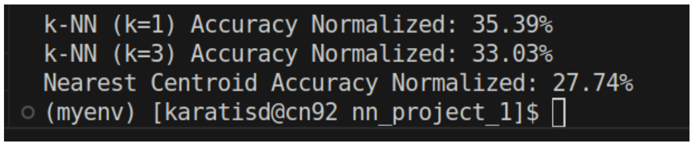
For each classifier, the accuracy is printed as a percentage, allowing a comparison of performance. Additionally, the function ***generate_conf_matrix*** generates a confusion matrix for each classifier, helping to visually assess where the models correctly or incorrectly classified the test images, and making it easier to analyze the strengths and weaknesses of each approach.

Results:



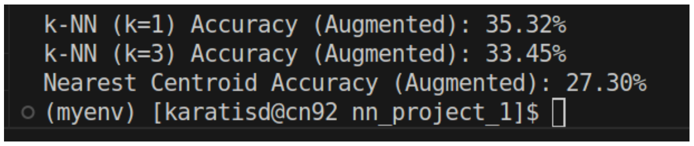
```
k-NN (k=1) Accuracy: 35.39%
k-NN (k=3) Accuracy: 33.03%
Nearest Centroid Accuracy: 27.74%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 1. Results for different classifiers (original dataset)



```
k-NN (k=1) Accuracy Normalized: 35.39%
k-NN (k=3) Accuracy Normalized: 33.03%
Nearest Centroid Accuracy Normalized: 27.74%
(myenv) [karatisd@cn92 nn_project_1]$
```

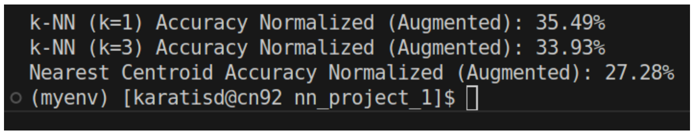
Fig. 2. Results for different classifiers (normalized dataset)



```
k-NN (k=1) Accuracy (Augmented): 35.32%
k-NN (k=3) Accuracy (Augmented): 33.45%
Nearest Centroid Accuracy (Augmented): 27.30%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 3. Results for different classifiers (augmented dataset)

As we can see, all of the classifiers yield roughly the same results, even if we perform normalization, rotation, augmentation or both on the dataset beforehand. By normalization,



```
k-NN (k=1) Accuracy Normalized (Augmented): 35.49%
k-NN (k=3) Accuracy Normalized (Augmented): 33.93%
Nearest Centroid Accuracy Normalized (Augmented): 27.28%
(myenv) [karatisd@cn92 nn_project_1]$
```

Fig. 4. Results for different classifiers (normalized and augmented dataset)

I mean dividing the pixel values of the data images by 255, so that their respective range now becomes from zero to one, while by rotation I mean rotating each image by a random number between -10 and 10 degrees, zero excluded. Also, by analyzing the results it is evident that there are limitations when it comes to using simple methods like k-Nearest Neighbors (k-NN) and Nearest Centroid for complex image data, since we can see that the accuracy on those methods is extremely low. More specifically, the k-NN classifier with k=1 performed best, with an accuracy of 35.39%, while the one with k=3 scored slightly lower at 33.03%. This means that looking at just the closest image gives slightly better results than averaging across more neighbors. The nearest centroid method did the worst, with an accuracy of only 27.74%, likely because averaging all images in a class loses important details. Normalizing the data didn't change the results, which suggests that pixel scaling alone isn't enough to improve these models, although there were some minuscule changes when applying the augmentation. Overall, these results suggest that CIFAR-10 is too complex for these simple classifiers, and more advanced methods like deep learning would likely perform better. Below we can see the Confusion Matrices in the case that we use normalization and augmentation for each one of the classifiers.

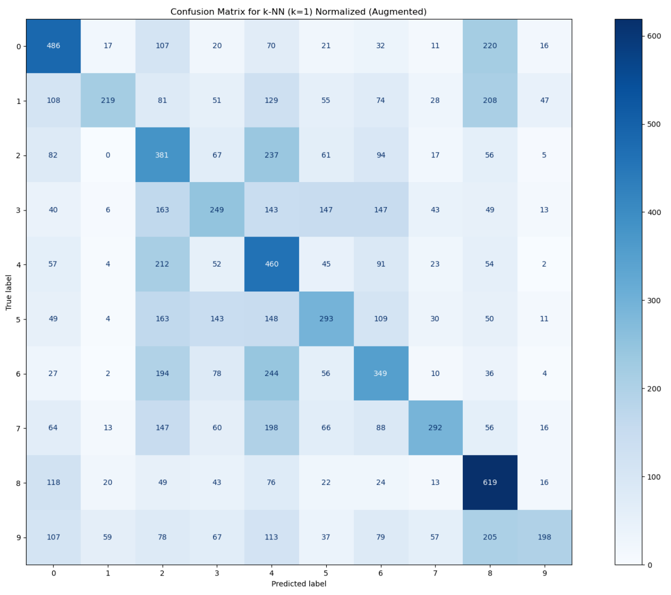


Fig. 5. k-NN (k=1) confusion matrix (normalized and augmented dataset)

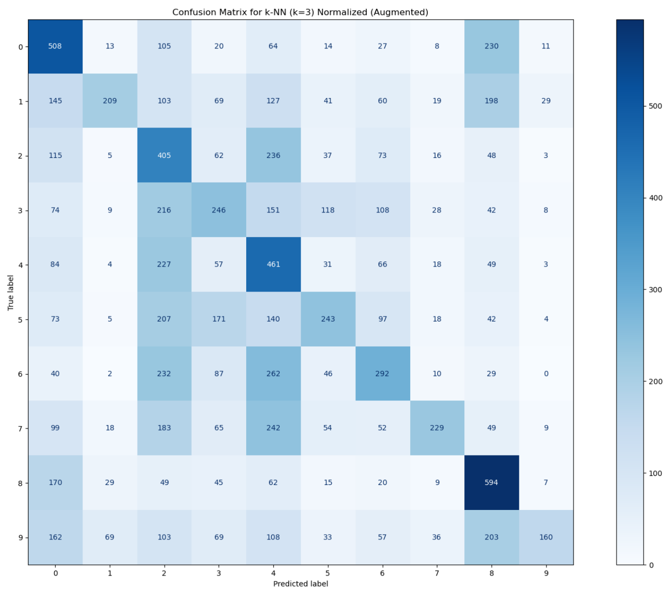


Fig. 6. k-NN (k=3) confusion matrix (normalized and augmented dataset)

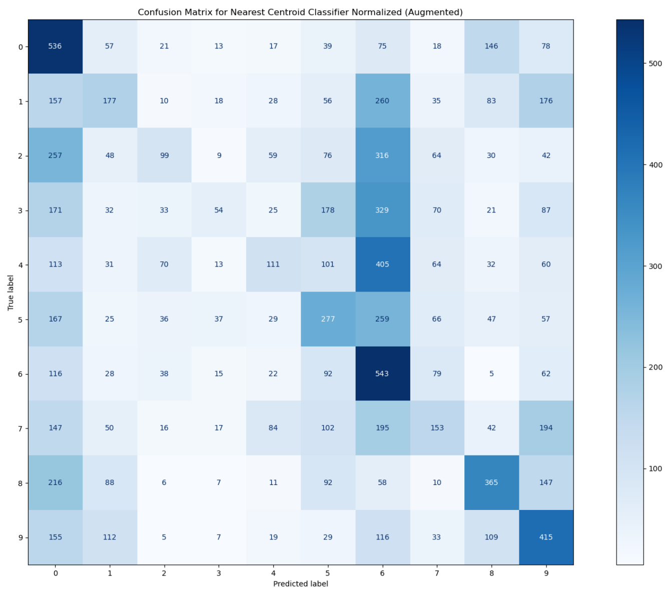


Fig. 7. Nearest Centroid confusion matrix (normalized and augmented dataset)

C. Training and Evaluating Multilayer Perceptron (MLP)

D. Training and Evaluating Convolutional Neural Network (CNN)