CIFAR-10 IMAGE CLASSIFICATION

Achyut Ramakrishna Kowshik
Khoury College of Computer Sciences

Introduction

Neural networks are adaptable models capable of learning almost any complicated pattern. These sophisticated models, which include multi-layer perceptrons, convolutional networks, sequence models, and many others, form the foundation of deep learning. In this short project, I will investigate the CIFAR-10 dataset and build a rudimentary neural network (multi-layer perceptron). Additionally, I will build a separate Convolutional Neural Network (CNN) to perform the same classification task. In the end, I will compare the two models on classification accuracy and compile my findings.

A neural network is basically a very simple notion. An activation function activates neurons within a layer in a neural network in the same way that neurons in the human brain do. This process produces output that is passed on to the next layer of the neural network, and the cycle is continued until the neural network is completed. This procedure is known as the forward pass, through which your data is sent forward through the network after weights and an activation function have been applied. The final output layer of your neural network will output one node for regression problems and numerous nodes for classification questions, depending on whether you are solving a regression or classification problem. In this project, I will be categorizing images belonging to one of ten categories, thus the neural network output will have one node for each class, which will be passed through the softmax function to achieve the final prediction.

The Dataset

The CIFAR-10 dataset, a well-established benchmark for image classification tasks in the fields of machine learning and computer vision, was developed by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton at the University of Toronto in 2009.

Comprising 60,000 color images with dimensions of 32x32 pixels, the CIFAR-10 dataset contains images from 10 distinct categories representing a variety of common objects. The 10 classes include airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck (Fig. 1). The dataset is evenly distributed, with each class consisting of 6,000 images. It is split into a training set containing 50,000 images and a test set with 10,000 images.

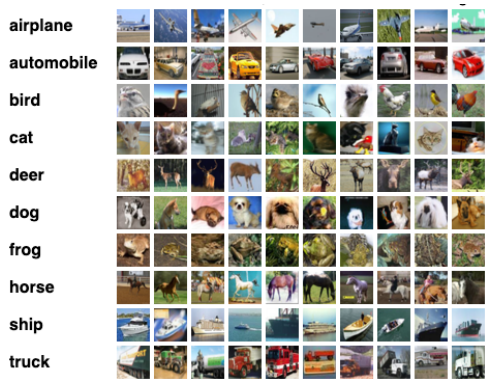

Fig. 1: Ten random images from each category

1. Data Preparation

In this section, we discuss the data preparation process for our image classification task. Proper data preparation is essential for achieving accurate and reliable results when training and evaluating machine learning models. The process involves several steps, including splitting the dataset, normalizing the images, flattening the data, and categorizing the labels.

1.1 Loading the dataset:

I will begin by importing the necessary modules and functions for loading the dataset. In my case, where I'll be using TensorFlow and Keras, I will import the cifar10 module from the tensorflow.keras.datasets package.

1.2 Splitting the data into train and test sets:

The first step in data preparation is dividing the dataset into separate training and testing sets. The training set is used to train the model, while the test set is used to evaluate the model's performance on unseen data. This separation helps to prevent overfitting, ensuring that the model generalizes well to new data. Typically, the dataset is split into 80% for training and 20% for testing, but our data is already split into a set number of images for training and testing sets. The training set consists of 50,000 images and the testing set consists of 10,000 images.

1.3 Normalizing the images:

Image normalization is a crucial step in data preprocessing, as it scales the pixel values to a common range, usually between 0 and 1. This process helps improve the convergence of the optimization algorithm during model training, leading to better performance and faster convergence, by ensuring that the input values are on a consistent scale. Normalization can be achieved by dividing the pixel values by the maximum value (typically 255 for 8-bit images like the ones in CIFAR-10).

1.4 Flattening the data (For Deep Neural Network only):

When working with a deep neural network (DNN) that does not handle 2D image data directly, it is necessary to flatten the images into one-dimensional arrays before inputting them into the model. This process involves converting each 32x32 pixel image into a single row of 1,024 (32 x 32) pixel values. The flattened data can then be fed into the DNN for training. However, I will skip this step for the Convolutional Neural Network (CNN), as we can specify the input dimensions in the CNN model.

1.5 Categorizing the labels:

The CIFAR-10 dataset contains labels as integers ranging from 0 to 9, representing the 10 different classes of objects. To train a neural network for multi-class classification, it is essential to convert these integer labels into one-hot encoded vectors. One-hot encoding involves representing each label as a binary vector of length equal to the number of classes, with a "1" in the position corresponding to the label's class and "0"s elsewhere. This encoding allows the model to output probabilities for each class and provides a more suitable target for training a multi-class classifier.

2. Methods

This section includes all the methods I have employed in the project, which includes the model architecture, model training, model evaluation, and testing.

2.1 Model Architecture

Deep Neural Network: I have implemented a DNN model for the CIFAR-10 image classification task, which is composed of five layers, including four fully connected (dense) layers with ReLU activation functions and one output layer with a softmax activation function for multi-class classification.

The architecture of the DNN model is as follows:

1. Input layer: A fully connected layer with 1024 units and a ReLU activation function, designed to accept input features of size 3072 (32x32 pixels image flattened).
2. Dropout layer with a dropout rate of 0.2, to prevent overfitting by randomly dropping out some of the neurons during training.
3. Second fully connected layer with 512 units and a ReLU activation function.
4. Another dropout layer with a dropout rate of 0.2.
5. Third fully connected layer with 256 units and a ReLU activation function.
6. Dropout layer with a dropout rate of 0.2.
7. Fourth fully connected layer with 128 units and a ReLU activation function.
8. Dropout layer with a dropout rate of 0.2.
9. Output layer: A fully connected layer with 10 units (one for each class) and a softmax activation function, which outputs the probability distribution over the 10 classes.

Convolutional Neural Network: My CNN model is composed of three convolutional blocks, followed by a fully connected layer and an output layer. Each convolutional block contains two convolutional layers with ReLU activation functions, batch normalization, and a max pooling layer.

The architecture of the CNN model is as follows:

1. First convolutional block:
    - Conv2D layer with 32 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same' to preserve the input dimensions.
    - BatchNormalization layer to stabilize training and improve convergence.
    - Another Conv2D layer with 32 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same'.
    - BatchNormalization layer.
    - MaxPooling2D layer with a pool size of (2,2) to reduce the spatial dimensions of the feature maps.
2. Second convolutional block:
    - Conv2D layer with 64 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same'.
    - BatchNormalization layer.
    - Another Conv2D layer with 64 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same'.
    - BatchNormalization layer.
    - MaxPooling2D layer with a pool size of (2,2).

3. Third convolutional block:
    - Conv2D layer with 128 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same'.
    - BatchNormalization layer.
    - Another Conv2D layer with 128 filters, a kernel size of (3,3), ReLU activation function, and padding set to 'same'.
    - BatchNormalization layer.
    - MaxPooling2D layer with a pool size of (2,2).
4. Flatten layer to convert the 3D feature maps into a 1D vector, suitable for input into the fully connected layers.
5. Dropout layer with a dropout rate of 0.2 to prevent overfitting by randomly dropping out some neurons during training.
6. Fully connected (dense) layer with 512 units and a ReLU activation function.
7. Dropout layer with a dropout rate of 0.2.
8. Output layer: A fully connected layer with 10 units (one for each class) and a softmax activation function, which outputs the probability distribution over the 10 classes.

2.2 Model Training

Both the DNN and the CNN models were compiled using the Adam optimizer with the default learning rate, the categorical cross-entropy loss function, and the accuracy metric for evaluation. Early stopping was employed to prevent overfitting and reduce training time, monitoring the validation loss with a patience of 5 epochs and a minimum delta of 0.001. The model's weights were restored to those of the best-performing epoch if the validation loss did not improve.

In addition, for the CNN model, I performed data augmentation on the training data to check if this step helped improving the model's accuracy. This can help improve the model's generalization by providing it with a more varied and robust set of training examples. For this task, I used TensorFlow's ImageDataGenerator library. The applied transformation includes width and height shifts of 10% and horizontal flipping. The augmented data was fed into the model in mini-batches of 64 images.

I did not perform augmentation for the DNN model, as it is typical that this step is used specifically in CNNs. They can naturally handle the spatial transformations applied to the images. The effectiveness of data augmentation for DNNs might not be as good as for CNNs for the same reason.

The models were trained for a maximum of 50 epochs for the original training data without augmentation, as well as the augmented training data, with the training process stopping early if the validation loss did not improve after 5 consecutive epochs.

2.3 Model Evaluation and Testing

The performance of both the DNN and CNN models was evaluated using the validation set (x_test, y_test) during training, and the early stopping mechanism monitored the validation loss to avoid overfitting. The models' accuracy and categorical cross-entropy loss were recorded for each epoch (Fig. 2.1, 2.2, 2.3, 2.4, 2.5, 2.6).

Prediction accuracy

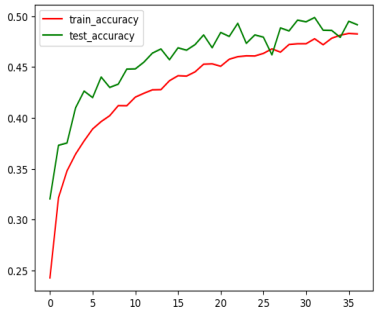

Fig. 2.1 Prediction accuracy of DNN

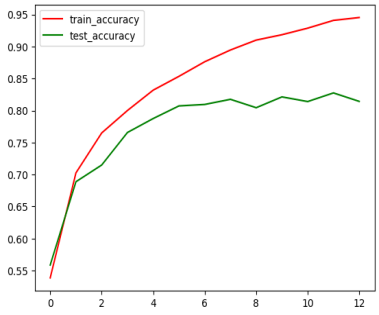

Fig. 2.2 Prediction accuracy of CNN (No data augmentation)

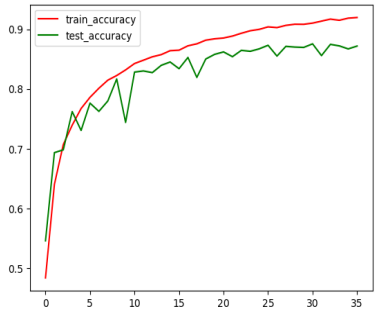

Fig. 2.3 Prediction accuracy of CNN (with data augmentation)

Cross-entropy loss

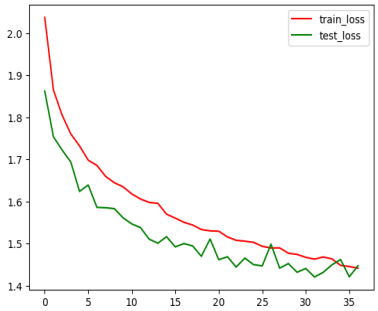

Fig. 2.4 Cross entropy loss of DNN

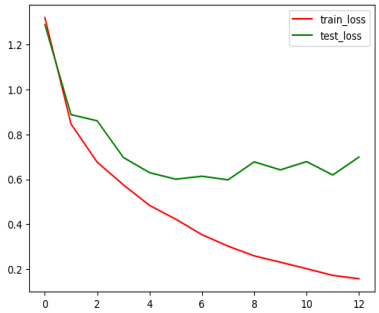

Fig. 2.5 Cross entropy loss of CNN (No data augmentation)

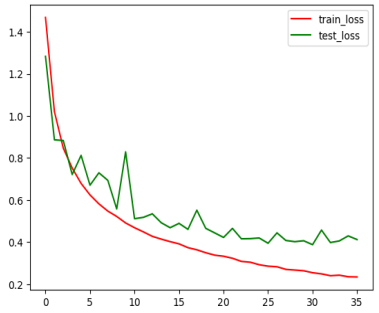

Fig. 2.6 Cross entropy loss of CNN (with data augmentation)

To further assess the model's performance, I computed the confusion matrix using the true labels and the model's predicted labels for the test set. I visualized the confusion matrices using a heatmap, which displayed the number of correctly and incorrectly classified samples for each class. This visualization allowed me to better understand the model's strengths and weaknesses in classifying the CIFAR-10 images and provided valuable information for further refining the model or identifying potential improvements. One of the noticeable aspects of the CNN model is that its predictive accuracy on certain classes, i.e. cats and dogs (~75%) is clearly lesser than that of the rest of the classes (~87%). The same pattern is observed even in the DNN model's accuracy. (Fig. 2.7, 2.8, 2.9).

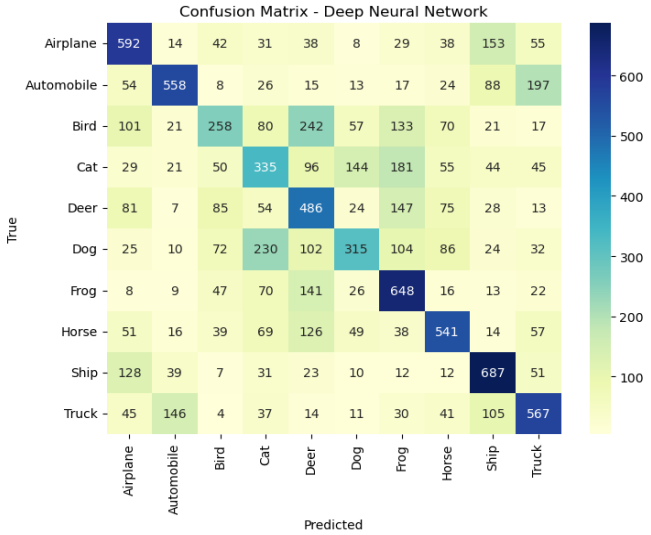

Fig. 2.7 Heatmap of DNN predictions

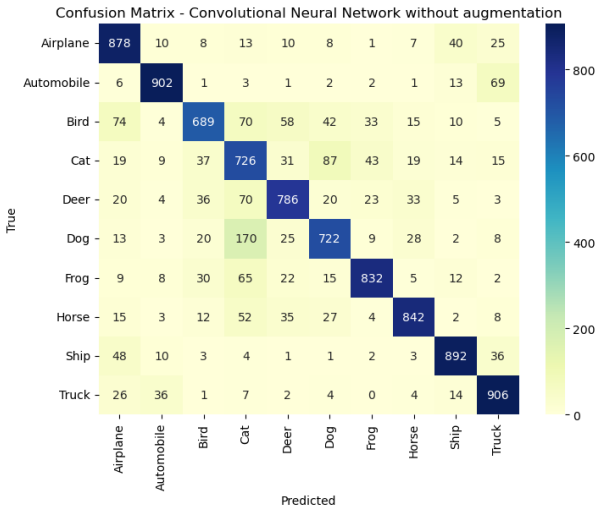

Fig. 2.8 Heatmap of CNN predictions (no data augmentation)

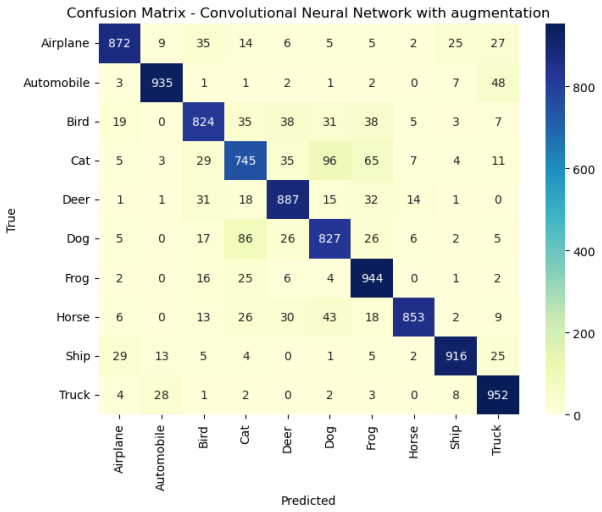

Fig. 2.9 Heatmap of CNN predictions (with data augmentation)

From the above three heatmaps, we can see that CNNs are significantly better at classifying images for all ten categories, regardless of whether we perform data augmentation or not. Additionally, we see that the prediction accuracy of CNN with data augmentation is similar to or better than the CNN without data augmentation.

It also makes sense to note that non-living things, such as car, truck, ship, etc. are classified with a better accuracy than living things like cat or dog.

3. Results

In this section, I will present the outcomes of my experiments with the Deep Neural Network (DNN) and Convolutional Neural Network (CNN) models for CIFAR-10 image classification. I will then analyse the models' performance based on their validation accuracy and loss, and the time it took to train these models.

Model	DNN	CNN (no data augmentation)	CNN (with data augmentation)
Validation (test) accuracy	49.87%	82.75%	87.55%
Validation loss	1.4206	0.5979	0.3878
Training time	296 seconds	1131 seconds	3240 seconds

The results of our experiments with the Deep Neural Network (DNN), Convolutional Neural Network (CNN) without data augmentation, and CNN with data augmentation are as follows:

1. DNN model achieved a validation (test) accuracy of 49.87% and a validation loss of 1.4206. The training time for this model was 296 seconds.
2. CNN model without data augmentation achieved a significantly higher validation (test) accuracy of 82.75% and a lower validation loss of 0.5979 compared to the DNN model. The training time for this model was 1131 seconds.
3. CNN model with data augmentation further improved the performance, achieving a validation (test) accuracy of 87.55% and a validation loss of 0.3878. The training time for this model was 3240 seconds, which is the longest among the three models.

In summary, the CNN models outperformed the DNN model in terms of validation accuracy and loss. The addition of data augmentation to the CNN model led to further improvements in performance but at the cost of a longer training time.

4. Conclusion

The observed outcomes can be attributed to the fundamental distinctions between Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs), as well as the influence of data augmentation.

1. DNN versus CNN: DNNs are entirely connected networks in which every neuron in one layer is linked to all neurons in the preceding and subsequent layers. Although this structure can identify intricate patterns, it does not consider the input images' spatial organization. On the other hand, CNNs are specifically tailored for image processing applications. They employ convolutional layers that can capture local patterns and spatial information within images, allowing CNNs to better learn features from images and achieve higher accuracy in image classification tasks compared to DNNs.

2. CNN without Data Augmentation versus CNN with Data Augmentation: Data augmentation is a strategy used to artificially expand the training dataset by generating new images via transformations such as rotation, scaling, and flipping. Augmenting the data exposes the model to a broader range of image variations, enhancing its ability to generalize. The CNN model with data augmentation outperformed the one without data augmentation because the expanded data helped the model learn more resilient features and generalize better to the test set's unseen images.

It is worth mentioning that the training time increased considerably when employing data augmentation, owing to the extra computational complexity introduced by the image transformations. Despite the extended training duration, the improved accuracy and reduced loss of the CNN model with data augmentation highlight the advantages of using this method in image classification tasks.

References

[1] Dataset- https://www.cs.toronto.edu/~kriz/cifar.html

[2] More on the CIFAR-10 dataset- https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[3] Theory on CNNs- https://yangxiaozhou.github.io/data/2020/09/24/intro-to-cnn.html#:~:text=While%20DNN%20uses%20many%20fully,a%20set%20of%20class%20probabilities.

Code

```
#!/usr/bin/env python
# coding: utf-8

# ## IMPORTING THE NECESSARY LIBRARIES

# In[1]:


import numpy as np
import tensorflow as tf
import keras
import random
from keras.datasets import cifar10
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, accuracy_score
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, Dropout, Activation
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping


# ## Classification using a DNN without convolution layers

# In[2]:


# Unpack and load the data
(X_train, Y_train), (x_test, y_test) = cifar10.load_data()


# In[3]:


categories = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog', 'Horse', 'Ship', 'Truck']


# In[4]:


# Normalize pixel values to be between 0 and 1
X_train, x_test = X_train / 255.0, x_test / 255.0

# Flatten the images
X_train = X_train.reshape(X_train.shape[0],-1)
x_test = x_test.reshape(x_test.shape[0],-1)

# One-hot encode the labels
Y_train = to_categorical(Y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)


# In[5]:


# Create the deep neural network model
dnn_model = Sequential([
    Dense(1024, activation='relu', input_shape=(3072,)),
```

```
    Dropout(0.2),
    Dense(512, activation='relu'),
    Dropout(0.2),
    Dense(256, activation='relu'),
    Dropout(0.2),
    Dense(128, activation='relu'),
    Dropout(0.2),
    Dense(10, activation='softmax')
])


# In[6]:


# View the summary of our DNN model
dnn_model.summary()


# In[7]:


# Compile the model
dnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# In[8]:


# Set up early stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=5, min_delta=0.001, restore_best_weights=True)


# In[9]:


# Train the model
DNN_M = dnn_model.fit(X_train, Y_train, epochs=50, batch_size=128, validation_data=(x_test, y_test),
          callbacks=[early_stopping])


# In[10]:


# Plot for training and testing accuracy of the DNN Model
plt.plot(DNN_M.history['accuracy'], label='train_accuracy', color='red')
plt.plot(DNN_M.history['val_accuracy'], label='test_accuracy', color='green')
plt.legend()


# In[46]:


# Plot for training and testing loss of DNN Model
plt.plot(DNN_M.history['loss'], label='train_loss', color='red')
plt.plot(DNN_M.history['val_loss'], label='test_loss', color='green')
plt.legend()


# In[48]:


#Testing loss, accuracy and training time of DNN model
loss, accuracy = min(DNN_M.history['val_loss']), max(DNN_M.history['val_accuracy'])
```

```
print(f"Test accuracy: {accuracy:.2%}")
print(f"Test loss: {loss:.4f}")
print(f"Training time: {37*8} seconds") #37 epochs, each epoch took an average of 8 seconds


# In[11]:


# Obtain the true labels from one-hot encoded test_labels
true_labels = np.argmax(y_test, axis=1)

# Get the predicted labels from the model
predicted_probabilities = dnn_model.predict(x_test)
predicted_labels = np.argmax(predicted_probabilities, axis=1)

# Compute the confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)
print("Confusion matrix:\n", conf_matrix)

# Calculate accuracy
accuracy = accuracy_score(true_labels, predicted_labels)
print(f"Accuracy: {accuracy:.4f}")


# In[49]:


# Visualizing the prediction accuracy using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="YlGnBu", xticklabels=categories, yticklabels=categories)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix- Deep Neural Network")
plt.show()


# ## END OF DNN MODEL CLASSIFICATION

# ##-----------------------------------------

# ## CIFAR-10 IMAGE CLASSIFICATION USING CNN

# In[13]:


# Unpack and load the data
(X_train, Y_train), (x_test, y_test) = cifar10.load_data()


# In[14]:


# Check the dimensions of our data
X_train.shape, x_test.shape


# In[15]:


# Check the shape of the labels
Y_train[:5]
```

In[16]:

```
# Normalize pixel values to be between 0 and 1
X_train, x_test = X_train/255.0, x_test/255.0

# Flatten the labels to reduce it from 2D to a 1D array
Y_train, y_test = Y_train.flatten(), y_test.flatten()
```

In[17]:

```
# Ensure the images are normalized
X_train[:5]
```

In[18]:

```
# Ensure the label shape is correct (1D array)
Y_train
```

In[19]:

```
# Building the first CNN model (for fitting without data augmentation)
cnn_model1 = Sequential([
    Conv2D(32, (3,3), activation='relu', padding = 'same', input_shape=(32, 32, 3)),
    BatchNormalization(),
    Conv2D(32, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Conv2D(64, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    Conv2D(64, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Conv2D(128, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    Conv2D(128, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Flatten(),
    Dropout(0.2),

    Dense(512, activation='relu'),
    Dropout(0.2),
    Dense(len(categories), activation='softmax')
])
```

In[20]:

```
# Building the second CNN model (for fitting with data augmentation)
cnn_model2 = Sequential([
    Conv2D(32, (3,3), activation='relu', padding = 'same', input_shape=(32, 32, 3)),
```

```
    BatchNormalization(),
    Conv2D(32, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Conv2D(64, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    Conv2D(64, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Conv2D(128, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    Conv2D(128, (3,3), activation='relu', padding = 'same'),
    BatchNormalization(),
    MaxPooling2D((2,2)),

    Flatten(),
    Dropout(0.2),

    Dense(512, activation='relu'),
    Dropout(0.2),
    Dense(len(categories), activation='softmax')
])


# In[21]:


# View the summary of our CNN model
cnn_model1.summary()


# In[22]:


# Compile the first model
cnn_model1.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])


# In[23]:


# Compile the second model
cnn_model2.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])


# In[24]:


# Fit the first model
CNN_M1 = cnn_model1.fit(X_train, Y_train, validation_data=(x_test, y_test),
                    epochs=50, batch_size=64, callbacks=[early_stopping])


# In[25]:


# Data augmentation for the second model
batch_size = 64
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
  width_shift_range=0.1, height_shift_range=0.1, horizontal_flip=True)
```

```
train_generator = datagen.flow(X_train, Y_train, batch_size)
steps_per_epoch = X_train.shape[0] // batch_size

# Fit the second model
CNN_M2 = cnn_model2.fit(train_generator, validation_data=(x_test, y_test),
          steps_per_epoch=steps_per_epoch, epochs=50, callbacks=[early_stopping])


# In[26]:


# Plot for training and testing accuracy of CNN Model 1
plt.plot(CNN_M1.history['accuracy'], label='train_accuracy', color='red')
plt.plot(CNN_M1.history['val_accuracy'], label='test_accuracy', color='green')
plt.legend()


# In[32]:


# Plot for training and testing loss of CNN Model 1
plt.plot(CNN_M1.history['loss'], label='train_loss', color='red')
plt.plot(CNN_M1.history['val_loss'], label='test_loss', color='green')
plt.legend()


# In[27]:


# Plot for training and testing accuracy of CNN Model 2
plt.plot(CNN_M2.history['accuracy'], label='train_accuracy', color='red')
plt.plot(CNN_M2.history['val_accuracy'], label='test_accuracy', color='green')
plt.legend()


# In[33]:


# Plot for training and testing loss of CNN Model 2
plt.plot(CNN_M2.history['loss'], label='train_loss', color='red')
plt.plot(CNN_M2.history['val_loss'], label='test_loss', color='green')
plt.legend()


# In[44]:


#Testing loss, accuracy and training time of CNN model 1 (without data augmentation)
test_loss, test_accuracy = min(CNN_M1.history['val_loss']), max(CNN_M1.history['val_accuracy'])

print(f"Test accuracy: {test_accuracy:.2%}")
print(f"Test loss: {test_loss:.4f}")
print(f"Training time: {13*87} seconds") #13 epochs, each epoch took an average of 87 seconds


# In[45]:


#Testing loss, accuracy and training time of CNN model 2 (with data augmentation)
loss, accuracy = min(CNN_M2.history['val_loss']), max(CNN_M2.history['val_accuracy'])

print(f"Test accuracy: {accuracy:.2%}")
print(f"Test loss: {loss:.4f}")
```

```
print(f"Training time: {36*90} seconds") #36 epochs, each epoch took an average of 90 seconds


# In[28]:


# One-hot encode the test labels
y_test = to_categorical(y_test, num_classes=10)


# In[29]:


# Obtain the true labels from one-hot encoded test_labels
true_labels = np.argmax(y_test, axis=1)

# Get the predicted labels from CNN model 1
predicted_probabilities1 = cnn_model1.predict(x_test)
predicted_labels1 = np.argmax(predicted_probabilities1, axis=1)

# Get the predicted labels from CNN model 2
predicted_probabilities2 = cnn_model2.predict(x_test)
predicted_labels2 = np.argmax(predicted_probabilities2, axis=1)

# Compute the confusion matrix for CNN model 1
conf_matrix1 = confusion_matrix(true_labels, predicted_labels1)
print("Confusion matrix:\n", conf_matrix1)

# Calculate accuracy for CNN model 1
accuracy1 = accuracy_score(true_labels, predicted_labels1)
print(f"Accuracy: {accuracy1:.4f}")

# Compute the confusion matrix for CNN model 2
conf_matrix2 = confusion_matrix(true_labels, predicted_labels2)
print("Confusion matrix:\n", conf_matrix2)

# Calculate accuracy for CNN model 2
accuracy2 = accuracy_score(true_labels, predicted_labels2)
print(f"Accuracy: {accuracy2:.4f}")


# In[30]:


# Visualizing the prediction accuracy using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix1, annot=True, fmt="d", cmap="YlGnBu", xticklabels=categories, yticklabels=categories)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix- Convolutional Neural Network without augmentation")
plt.show()

# In[31]:

# Visualizing the prediction accuracy using a heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix2, annot=True, fmt="d", cmap="YlGnBu", xticklabels=categories, yticklabels=categories)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix- Convolutional Neural Network with augmentation")
plt.show()

# ## END OF PROJECT
```