CIFAR 10 Image Classification

Karan Mudaliar

Northeastern University

DS5220 Supervised Machine Learning

1 INTRODUCTION

CIFAR-10 is an image classification dataset having 60,000 32x32 color images belonging to 10 classes. It is commonly used to benchmark the performance of image classification models. The classes are - airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

For this project, I will use neural networks to classify images in the CIFAR-10 dataset. Specifically, I will implement both a deep neural network (DNN) and a convolutional neural network (CNN) using the Keras library in Python. I will then compare the performance of the two models based on metrics such as accuracy, precision and recall to determine which model performs better for image classification.

2 METHODS

2.1 CNN:

For the first method I will be using a CNN. The architecture was picked by me after a few iterations of testing. All training and testing was done on google colab. I referred to a few resources and papers linked in the references to achieve on this model

The architecture is as follows:

- Conv2D (32 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- Conv2D (64 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- MaxPooling2D (2x2 window, stride=2)
- Dropout (0.2)
- Conv2D (128 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- Conv2D (128 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- MaxPooling2D (2x2 window, stride=2)
- Dropout (0.3)
- Conv2D (256 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- Conv2D (256 filters, 3x3 kernel, padding='same', activation='swish')

- BatchNormalization: normalizes the output of the previous layer.
- MaxPooling2D (2x2 window, stride=2)
- Dropout (0.4)
- Conv2D (512 filters, 3x3 kernel, padding='same', activation='swish')
- BatchNormalization: normalizes the output of the previous layer.
- MaxPooling2D (2x2 window, stride=2)
- Dropout (0.5): randomly drops out 50% of the neurons in the previous layer
- Flatten: Transforms the output of the previous layer into a 1 dimensional vector.
- Dense (1024 neurons, activation='swish'): Fully connected layer with 1024 neurons and a 'swish' activation function.
- Dropout (0.5): randomly drops out 50% of the neurons in the previous layer
- Dense (512 neurons, activation='swish'): another fully connected layer with 512 neurons and a 'swish' activation function.
- Dropout (0.5): randomly drops out 50% of the neurons in the previous layer
- Dense (10 neurons, activation='softmax'): the output layer with 10 neurons, each corresponding to a class label, using a 'softmax' activation function.

All the layers use the Swish activation function. This activation function was developed by researchers at Google Brain. In some cases it is better than ReLU function. In my testing both performed similarly but I opted for Swish for the marginal improvement.

For the optimiser I am using ADAM. It has an adaptive learning rate and momentum-based approach, which helps with a faster convergence of the optimization process. Some fully connected layers are also using dropout, which is a regularization technique that helps to prevent overfitting by randomly dropping out some neurons. The last layer uses softmax to classify and predict the outputs.

2.2 DNN:

For the second layer I am using a simple neural network with 3 fully connected layers and one soft max classifying layer. All layers are using Swish as the activation function. Just like the CNN, all layers here use Swish activation and the optimisation function used is ADAM.

Code for both the networks will be mentioned in the end.

3 RESULTS

3.1 Analysis:

For analysis we will be comparing the following:

- Accuracy on training and test data
- Loss
- Precision & Recall

3.1.1 Accuracy:

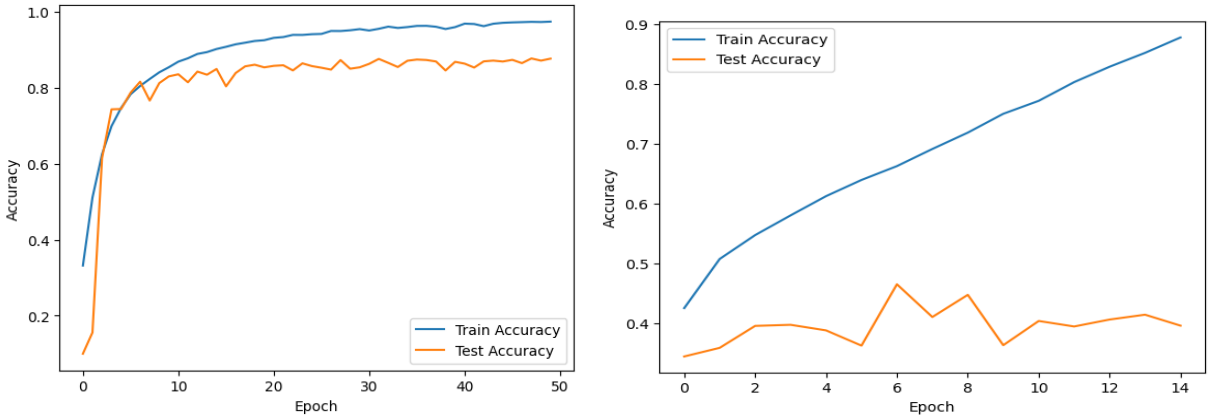

Figure 1 - CNN and DNN accuracy graph over epochs

The Final results of the CNN model gave 97% accuracy on training data and 87% accuracy on test data. The Final model for the DNN gave 87% accuracy on training data and 39% accuracy on testing data.

From the above graphs it can be clearly seen that CNN can explain the variance in the data pretty well. The accuracy on training and test both are gradually increasing every epoch suggesting that this is not an overfit. The performance on test data maxes out at around the 87% mark on the 50th epoch, beyond which the gains are minimal

The DNN on the other hand has an accuracy on the test set of 39%, Indicating poor performance. The performance on the training set is 87% indicating a clear overfit to the training set. Along with this, it can be noted that after the 9th epoch, the test performance decreases while the training performance keeps increasing, which further indicates an overfit. Hence it can be concluded that this approach is ineffective.

3.1.2 Loss Graphs:

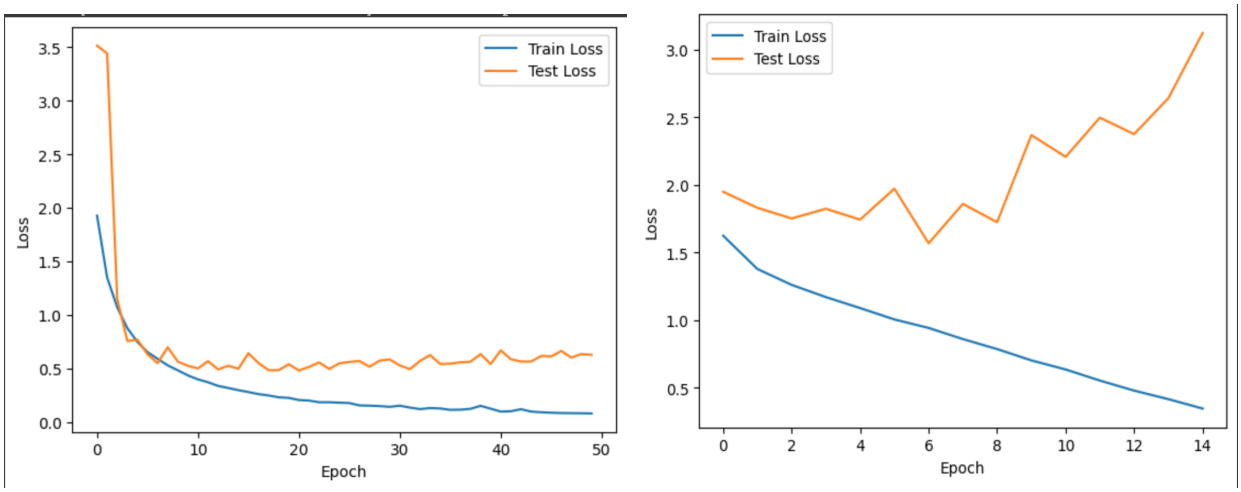

Figure 2 - Loss for CNN and DNN over the epochs

The loss charts show the loss for each epoch. Generally, if loss for training decreases but test increases, it indicates an over fit. In the above graphs we can see that for the CNN there has been a marginal increase in test loss after the 20th epoch while the training loss is decreasing. This can indicate a mild overfit.
However, in the case of the DNN we can see that the test loss is disproportionately increasing while training loss is decreasing. This implies a severe overfit, hence the model is not very useful for practical applications.

3.1.3 Precision and Recall:

Using SK learn, the precision and recall for both models was computed.
They are as follows:

- CNN : Precision - 87.733 & Recall - 87.660
- DNN : Precision - 49.826 & Recall: 39.629

The Scores for the CNN further imply that the model is practical and can be used in applications. The scores for the DNN on the other hand are poor and suggest the model should not be used.

3.2 Comparison:

3.2.1 Architecture:

There are fundamental differences between a CNN and a simple DNN. The convolutional and pooling layers of a CNN make it better at sensing features which in turn allow it to give better prediction results, something a simple DNN cannot capture.

The DNN used in this project has double the number of parameters compared to the CNN (8 million vs 4 million approximately). However, since the DNN cannot capture and pool features it is not as computationally intensive.

3.2.2 Convergence:

In terms of convergence, the DNN converges much faster as it's a simpler model with fewer layers and also since the computations are not as intensive as convoling. In this project I trained the model for 15 epochs but it started to converge around the 8th epoch before overfitting the training data. Each epoch took around 3 seconds to run on google colab using GPU acceleration.

The CNN takes much longer to converge as it has more layers and convolutions. This model was trained for 50 epochs as it was showing performance gains up to that point while I was testing it. Beyond 50 epochs there is a 1% gain on testing accuracy performance, although the testing loss also shows a marginal increase. Each epoch took 8 seconds to run on average on google colab using GPU acceleration.

4 CODE

Link for the colab notebook

Installing the necessary packages:

```
%pip install tensorflow
%pip install matplotlib
%pip install keras
```

Importing the Libraries:

```
import tensorflow as tf
from tensorflow.keras import layers, models, applications
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix, precision_score,
recall_score, accuracy_score
```

Defining the CNN architecture:

```
def cnn():
   model = models.Sequential()
   model.add(layers.Conv2D(32, kernel_size=3, padding='same',
activation='swish', input_shape=(32, 32, 3)))
   model.add(layers.BatchNormalization())
   model.add(layers.Conv2D(64, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
   model.add(layers.Dropout(0.2))

   model.add(layers.Conv2D(128, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.Conv2D(128, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
   model.add(layers.Dropout(0.3))

   model.add(layers.Conv2D(256, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.Conv2D(256, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
   model.add(layers.Dropout(0.4))

   model.add(layers.Conv2D(512, kernel_size=3, padding='same',
activation='swish'))
   model.add(layers.BatchNormalization())
   model.add(layers.MaxPooling2D(pool_size=(2, 2)))
   model.add(layers.Dropout(0.5))

   model.add(layers.Flatten())
   model.add(layers.Dense(1024, activation='swish'))
   model.add(layers.Dropout(0.5))
   model.add(layers.Dense(512, activation='swish'))
   model.add(layers.Dropout(0.5))
```

```
    model.add(layers.Dense(10, activation='softmax'))

    return model
```

Loading training and testing data:

```
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()

x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

Creating the model:

```
model = cnn()
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Training the model:

```
num_epochs = 100
history = model.fit(x_train, y_train, batch_size=256, epochs=num_epochs,
validation_data=(x_test, y_test), verbose=1)
```

Plotting test and training losses and accuracies:

```
plt.figure()
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.figure()
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Getting model summary:

```
CNN = model
CNN.summary()
```

Calculating Precision and Recall:

```
y_true = np.argmax(y_test, axis=-1)
cm = confusion_matrix(y_true, y_pred)
precision = precision_score(y_true, y_pred, average=None)
recall = recall_score(y_true, y_pred, average=None)
average_precision = np.mean(precision)
average_recall = np.mean(recall)
accuracy = accuracy_score(y_true, y_pred)

print("Average Precision:", average_precision*100)
print("Average Recall:", average_recall*100)
```

Defining a simple neural network:

```
def nn():
   model = models.Sequential()
   model.add(layers.Flatten(input_shape=(32, 32, 3)))

   model.add(layers.Dense(2048))
   model.add(layers.BatchNormalization())
   model.add(layers.Activation('swish'))

   model.add(layers.Dense(1024))
   model.add(layers.BatchNormalization())
   model.add(layers.Activation('swish'))

   model.add(layers.Dense(512))
   model.add(layers.BatchNormalization())
   model.add(layers.Activation('swish'))

   model.add(layers.Dense(10, activation='softmax'))

   return model
```

Creating and compiling model:

```
model = nn()
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Training the model:

```
num_epochs = 15
history = model.fit(x_train, y_train, batch_size=256, epochs=num_epochs,
validation_data=(x_test, y_test), verbose=1)
```

Plotting training and test losses and accuracy:

```
plt.figure()
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()


plt.figure()
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

5 REFERENCES

1. https://www.researchgate.net/figure/Well-tuned-CNN-architecture-for-CIFAR-10-dataset-Left-This-network-is-generated-by_fig3_326816043
2. https://www.kaggle.com/code/vassiliskrikonis/cifar-10-analysis-with-a-neural-network
3. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939
4. https://medium.com/@keonyonglee/the-bread-and-butter-from-deep-learning-by-andrew-ng-course-4-1-convolutional-neural-networks-fa8aa4a1dc32