ID: 23035392

Image Classification with Convolutional Neural Networks: An Exploration with the CIFAR-10 Dataset

Table of Contents

1. Introduction

Convolutional Neural Networks (CNNs) have become a cornerstone of modern deep learning, particularly in the realm of computer vision. CNNs are meant to self-observe and self-learn the spatial hierarchies of features from large-scale Image data; therefore they are commonly applied in performing tasks such as image classification, object detection, facial recognition, and more. These abilities of processing the data with grid-like topology like the images, have placed the CNNs far ahead of the traditional Machine Learning models of data analysis.

One of the key reasons CNNs are so effective for image data is their ability to leverage local spatial correlations through convolutional layers. These layers place several filters upon the input image and are responsible for identifying low-level features such as edges, textures, and patterns. These learned features are progressively abstract through the addition of convolutional and pooling layers which result in high-level abstraction that discriminates complex objects or scenes. For instance, pooling layers are used to accomplish down sampling to minimize redundancy and increase efficiency, in addition to eliminating translation invariance.

A CNN model is composed of several types of layers: There are convolutional layers that learn features from images; the pooling layers that reduce image dimensions, and fully connected layers, which integrate the learned features for classification or regression tasks. Applying the architecture of successively more complex representations of the image data, the model is capable of surpassing many classical machine learning applications.

In this report, we explore the implementation of a CNN for the task of image classification using the CIFAR-10 dataset (online). We will walk through key stages including data preprocessing, model creation and assessment, and comparability analysis with other machine learning techniques such as Logistics Regression and k-Nearest Neighbors (k-NN). The goal is to demonstrate how CNNs effectively handle image data and how various techniques can be applied to optimize their performance.

2. Code Implementation

Step 1: Importing Libraries

We import libraries for data handling, CNN building, training, evaluation, and visualisation.

Python Script:

```
import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

import matplotlib.pyplot as plt

import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

print("TensorFlow Version:", tf.__version__)
```

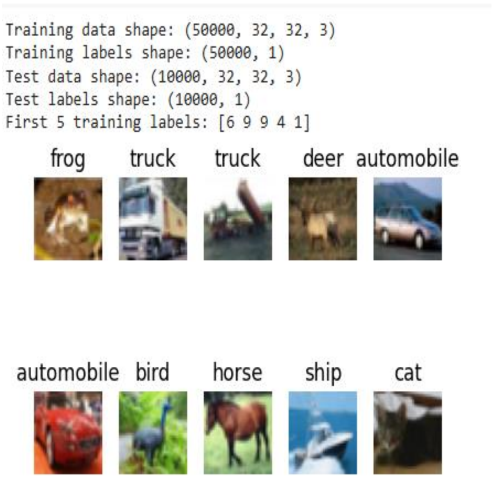
```
TensorFlow Version: 2.17.1
```

Step 2: Load and Explore the Dataset

In this step, I'll load a dataset suitable for training a CNN. I'll use the CIFAR-10 dataset, which contains 60,000 32x32 colour images across 10 classes (e.g., aeroplanes, cars, birds, etc.).

Python Script:

```
# Load the CIFAR-10 dataset from Keras datasets
from tensorflow.keras.datasets import cifar10
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

# Print dataset shapes
print("Training data shape:", train_images.shape)
print("Training labels shape:", train_labels.shape)
print("Test data shape:", test_images.shape)
print("Test labels shape:", test_labels.shape)

# Display the first 5 labels
print("First 5 training labels:", train_labels[:5].flatten())
# Display a few sample images with their labels
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(5, 5))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(train_images[i])
    plt.title(class_names[train_labels[i][0]])
    plt.axis('off')
plt.tight_layout()
plt.show()
```



```
Training data shape: (50000, 32, 32, 3)
Training labels shape: (50000, 1)
Test data shape: (10000, 32, 32, 3)
Test labels shape: (10000, 1)
First 5 training labels: [6 9 9 4 1]
```

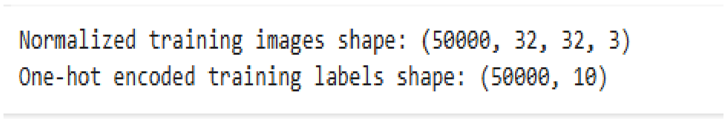
Step 3: Preprocess the Data

In this step, I'll prepare the data for input into the CNN. Preprocessing ensures that the data is in a suitable format for training and can improve model performance.

Python Script:

```
# Normalize pixel values to range 0-1
train_images = train_images / 255.0
test_images = test_images / 255.0

# Convert labels to one-hot encoding
```

```
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels, num_classes=10)
test_labels = to_categorical(test_labels, num_classes=10)

# Print shapes after preprocessing
print("Normalized training images shape:", train_images.shape)
print("One-hot encoded training labels shape:", train_labels.shape)
```



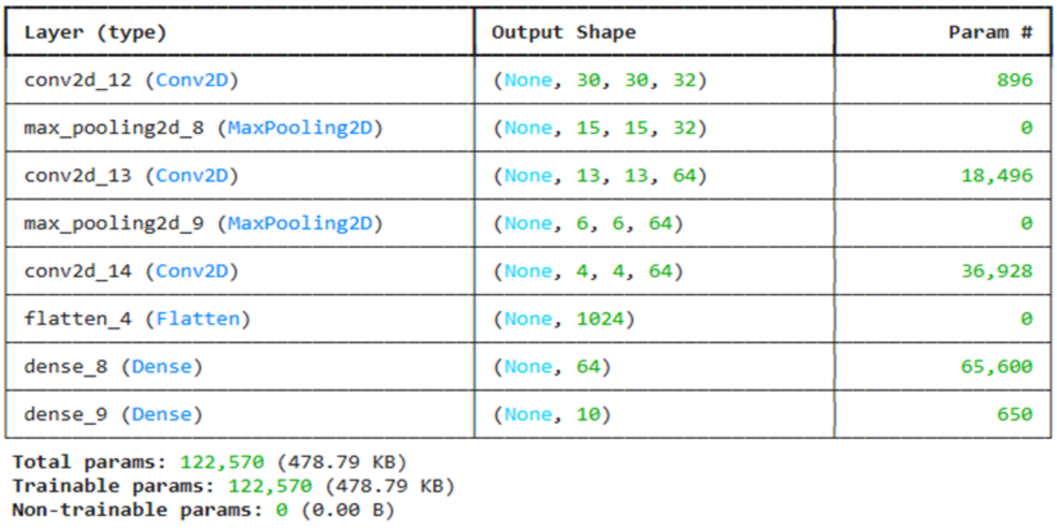
```
Normalized training images shape: (50000, 32, 32, 3)
One-hot encoded training labels shape: (50000, 10)
```

Step 4: Build the CNN Model

In this step, we'll construct the architecture of the Convolutional Neural Network (CNN). The model will include convolutional layers, pooling layers, and fully connected (dense) layers to process and classify the image data.

Python Script:

```
from tensorflow.keras import layers, models
# Initialize a Sequential model
model = models.Sequential()

# Add convolutional layers followed by pooling layers
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Flatten the output from the convolutional layers
model.add(layers.Flatten())
# Add dense (fully connected) layers
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))  # Output layer for 10 classes
model.summary()
```



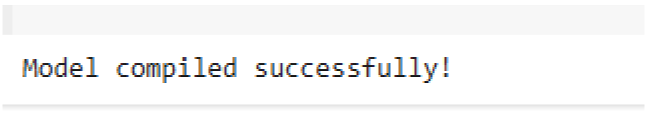
Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_8 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_13 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_9 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_14 (Conv2D)	(None, 4, 4, 64)	36,928
flatten_4 (Flatten)	(None, 1024)	0
dense_8 (Dense)	(None, 64)	65,600
dense_9 (Dense)	(None, 10)	650

```
Total params: 122,570 (478.79 KB)
Trainable params: 122,570 (478.79 KB)
Non-trainable params: 0 (0.00 B)
```

Step 5: Compile the Model

After building the CNN model, the next step is to compile it. Compilation specifies the optimizer, loss function, and evaluation metrics that the model will use during training.
Python Script:

```
# Compile the model

model.compile(

    optimizer='adam',  # Optimizer for updating weights

    loss='categorical_crossentropy',  # Loss function for multi-class classification

    metrics=['accuracy']  # Metric to evaluate during training and testing

)

# Display a confirmation message

print("Model compiled successfully!")
```



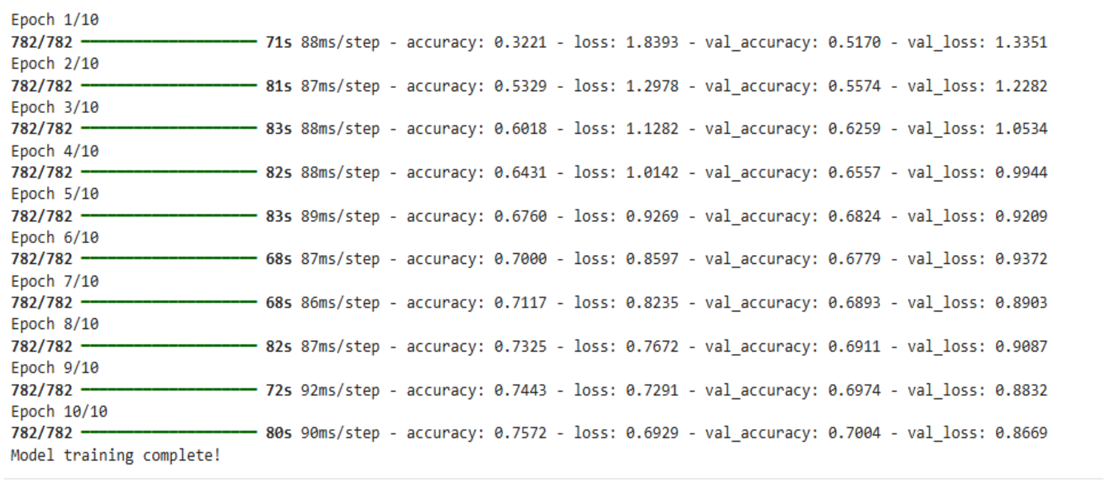
```
Model compiled successfully!
```

Step 6: Train the Model

Now that the CNN model is compiled, the next step is to train it using the training dataset. During training, the model will adjust its weights to minimize the loss function, improving its ability to classify images.

Python script:

```
# Train the model

history = model.fit(

    train_images,          # Training images
    train_labels,           # Training labels (one-hot encoded)
    epochs=10,              # Number of training epochs
    validation_data=(test_images, test_labels),  # Validation data
    batch_size=64,          # Batch size for gradient updates
    verbose=1               # Verbosity level for training output
)
# Display a confirmation message
print("Model training complete!")
```



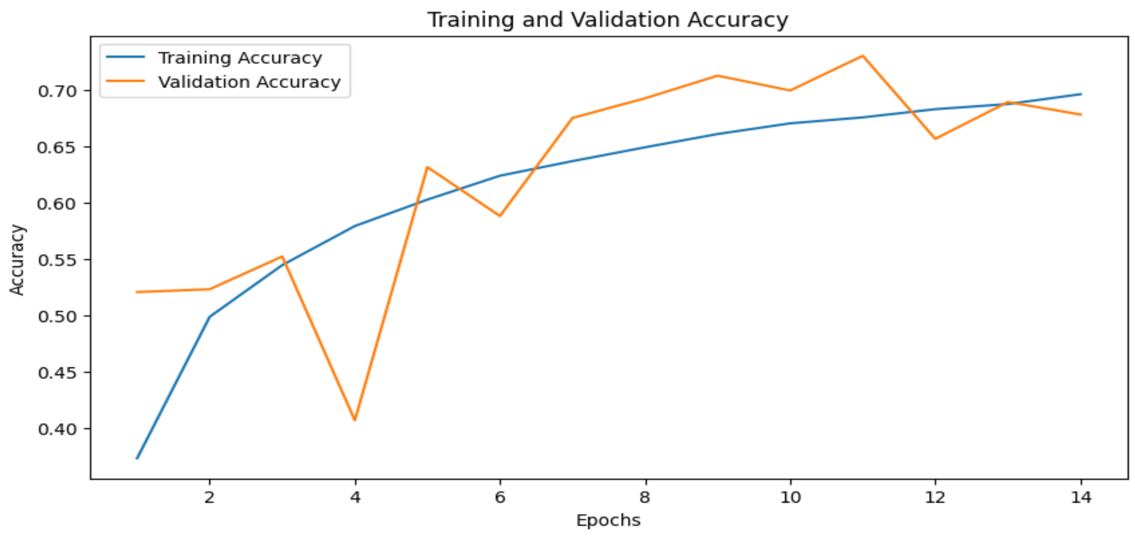
```
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 71s 88ms/step - accuracy: 0.3221 - loss: 1.8393 - val_accuracy: 0.5170 - val_loss: 1.3351
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 81s 87ms/step - accuracy: 0.5329 - loss: 1.2978 - val_accuracy: 0.5574 - val_loss: 1.2282
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 88ms/step - accuracy: 0.6018 - loss: 1.1282 - val_accuracy: 0.6259 - val_loss: 1.0534
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 88ms/step - accuracy: 0.6431 - loss: 1.0142 - val_accuracy: 0.6557 - val_loss: 0.9944
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 89ms/step - accuracy: 0.6760 - loss: 0.9269 - val_accuracy: 0.6824 - val_loss: 0.9209
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 68s 87ms/step - accuracy: 0.7000 - loss: 0.8597 - val_accuracy: 0.6779 - val_loss: 0.9372
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 68s 86ms/step - accuracy: 0.7117 - loss: 0.8235 - val_accuracy: 0.6893 - val_loss: 0.8903
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 87ms/step - accuracy: 0.7325 - loss: 0.7672 - val_accuracy: 0.6911 - val_loss: 0.9087
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 72s 92ms/step - accuracy: 0.7443 - loss: 0.7291 - val_accuracy: 0.6974 - val_loss: 0.8832
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 80s 90ms/step - accuracy: 0.7572 - loss: 0.6929 - val_accuracy: 0.7004 - val_loss: 0.8669
Model training complete!
```

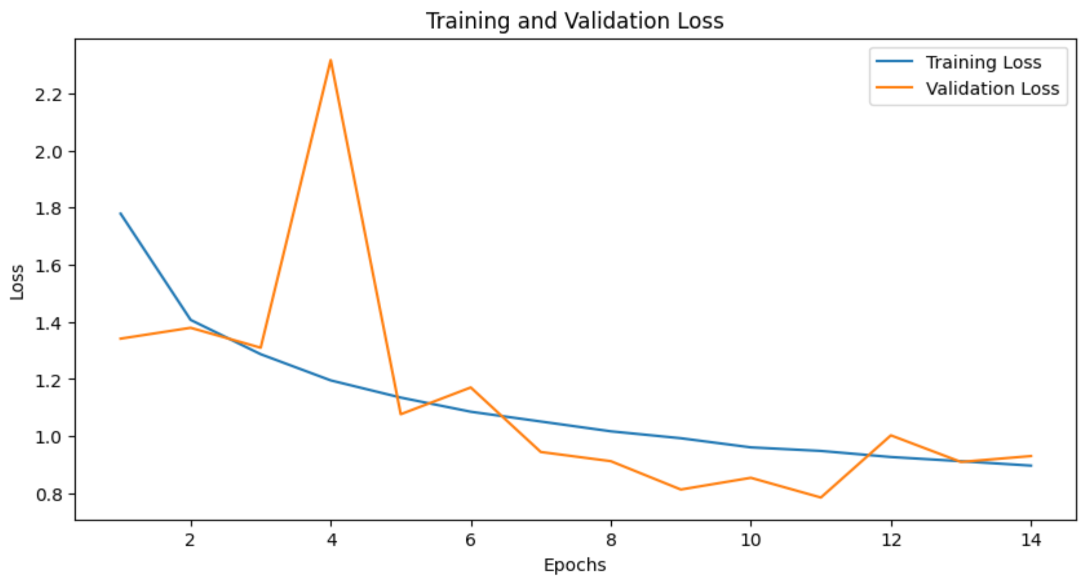
Step 7: Visualizing Training Results

After training the model, it's important to visualize the training and validation accuracy and loss. This helps us understand how well the model is learning and identify potential issues like overfitting or underfitting.

Python script:

```
import matplotlib.pyplot as plt
# Extract accuracy and loss from the training history
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(accuracy) + 1)

# Plot Training and Validation Accuracy
plt.figure(figsize=(10, 5))
plt.plot(epochs, accuracy, label='Training Accuracy')
plt.plot(epochs, val_accuracy, label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot Training and Validation Loss
plt.figure(figsize=(10, 5))
plt.plot(epochs, loss, label='Training Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



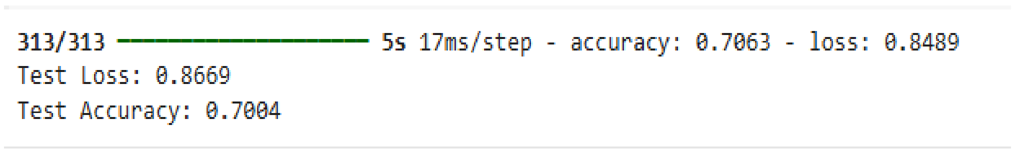


Step 8: Evaluate Model Performance

Now that the model is trained, we need to evaluate its performance on the test dataset to see how well it generalizes to unseen data. This involves calculating the loss and accuracy of the test set and interpreting the results.

Python Script:

```
# Evaluate the model on the test dataset

test_loss, test_accuracy = model.evaluate(test_images, test_labels, verbose=1)

# Print the test results

print(f"Test Loss: {test_loss:.4f}")

print(f"Test Accuracy: {test_accuracy:.4f}")
```

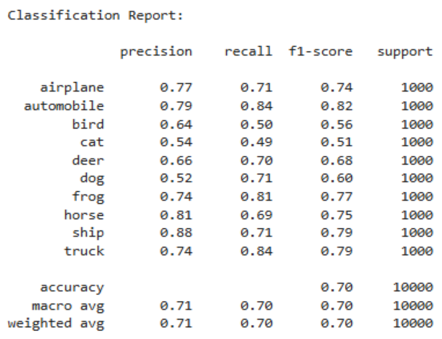


```
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - accuracy: 0.7063 - loss: 0.8489
Test Loss: 0.8669
Test Accuracy: 0.7004
```

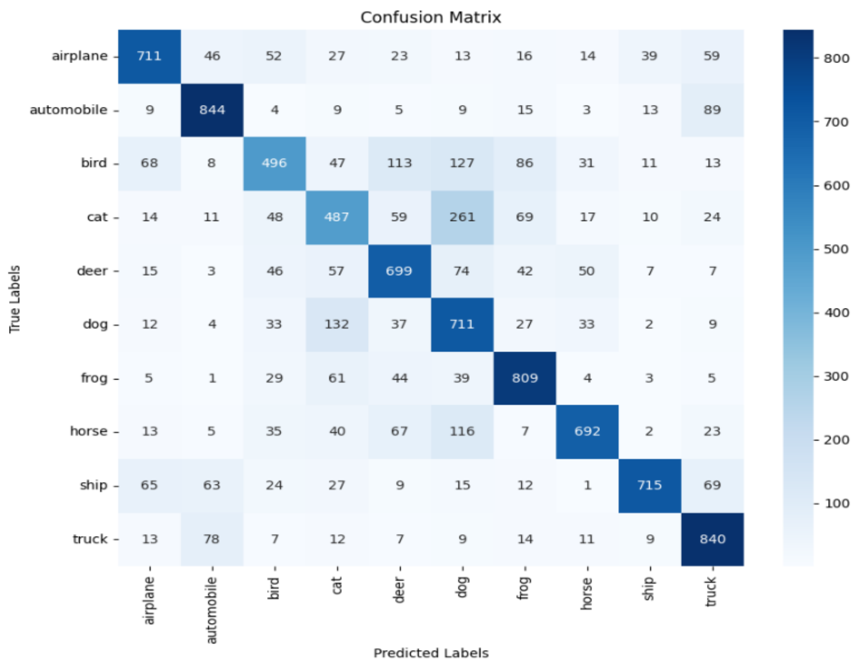
Classification Report and Confusion Matrix

```
from sklearn.metrics import classification_report, confusion_matrix

import seaborn as sns

# Predict the labels for the test dataset

predictions = model.predict(test_images)

predicted_labels = np.argmax(predictions, axis=1)

true_labels = np.argmax(test_labels, axis=1)
```

```
# Generate the classification report
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

print("Classification Report:\n")
print(classification_report(true_labels, predicted_labels, target_names=class_names))

# Generate the confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels)

# Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```



```
Classification Report:

              precision    recall  f1-score   support

    airplane       0.77      0.71      0.74      1000
  automobile       0.79      0.84      0.82      1000
        bird       0.64      0.50      0.56      1000
         cat       0.54      0.49      0.51      1000
        deer       0.66      0.70      0.68      1000
         dog       0.52      0.71      0.60      1000
        frog       0.74      0.81      0.77      1000
       horse       0.81      0.69      0.75      1000
        ship       0.88      0.71      0.79      1000
       truck       0.74      0.84      0.79      1000

    accuracy                           0.70     10000
   macro avg       0.71      0.70      0.70     10000
weighted avg       0.71      0.70      0.70     10000
```



Step 9: Visualizing Model Predictions

Visualizing the model's predictions helps us understand how it performs on individual test samples. We'll display a few test images along with their predicted and true labels.

Python Script:

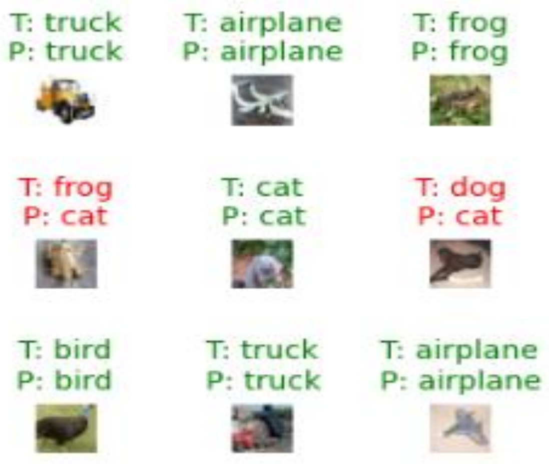
```
import matplotlib.pyplot as plt

import numpy as np


# Predict the labels for the test dataset

predictions = model.predict(test_images)

predicted_labels = np.argmax(predictions, axis=1)

true_labels = np.argmax(test_labels, axis=1)


# Class names for CIFAR-10

class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',

               'dog', 'frog', 'horse', 'ship', 'truck']


# Select random test samples to visualize

num_samples = 9

indices = np.random.choice(len(test_images), num_samples, replace=False)

selected_images = test_images[indices]

selected_true_labels = true_labels[indices]

selected_predicted_labels = predicted_labels[indices]


# Plot the selected test images with predictions

plt.figure(figsize=(3, 3))

for i in range(num_samples):

    plt.subplot(3, 3, i + 1)

    plt.imshow(selected_images[i], interpolation='nearest')


    # Shorter titles with correct/incorrect coloring

    color = 'green' if selected_true_labels[i] == selected_predicted_labels[i] else 'red'

    plt.title(f"T: {class_names[selected_true_labels[i]]}\nP:
{class_names[selected_predicted_labels[i]]}",

              fontsize=10, color=color)
```

Note

T: True label
P: Predicted label

Green: Correct predictions.
Red: Incorrect predictions.

```
    plt.axis('off')  # Turn off axes for better visibility


plt.tight_layout(pad=2.0)

plt.show()
```



Step 10: Model Tuning and Optimization

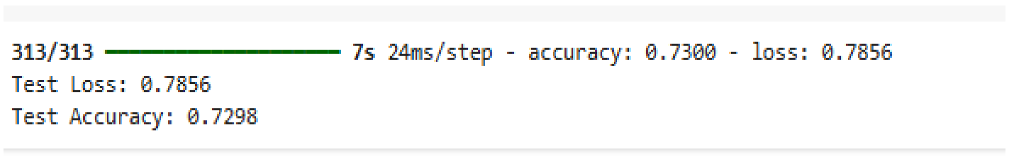
To improve the model's performance, we can apply tuning and optimization techniques. These techniques include adjusting the model architecture, hyperparameters, and training strategies. The combined approach already integrates the key techniques (Dropout, Batch Normalization, Data Augmentation, and Early Stopping).

Python Script:

```
from tensorflow.keras import layers, models

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.preprocessing.image import ImageDataGenerator


# Define the CNN Model with Batch Normalization and Dropout

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),

    layers.BatchNormalization(),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.BatchNormalization(),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.BatchNormalization(),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.5),  # Dropout to prevent overfitting
    layers.Dense(10, activation='softmax')  # Output layer
])

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

# Define Early Stopping
early_stopping = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Set Up Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(train_images)

# Train the Model with Data Augmentation and Early Stopping
history = model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),
    epochs=20,  # Early stopping will halt if no improvement
    validation_data=(test_images, test_labels),
    callbacks=[early_stopping]
)
```

And then

```
test_loss, test_accuracy = model.evaluate(test_images, test_labels)
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_accuracy:.4f}")
```



```
313/313 ━━━━━━━━━━━━━━━━━━━━ 7s 24ms/step - accuracy: 0.7300 - loss: 0.7856
Test Loss: 0.7856
Test Accuracy: 0.7298
```

- **Previous Test Loss:** 0.8489
- **Previous Test Accuracy:** 0.7004
- **New Test Loss:** 0.7856
- **New Test Accuracy:** 0.7298

This indicates that the optimizations (Dropout, Batch Normalization, Data Augmentation, and Early Stopping) have helped the model generalize better.

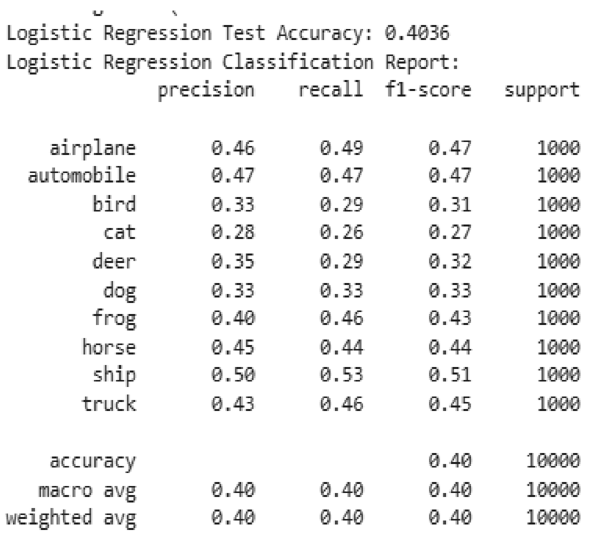
Step 11: Compare CNN Performance with Other Models

To understand how well our CNN performs, it's useful to compare it with simpler models, such as Logistic Regression, k-Nearest Neighbors (k-NN), or Decision Trees. This step provides insights into whether CNN's complexity is justified based on the dataset and task.

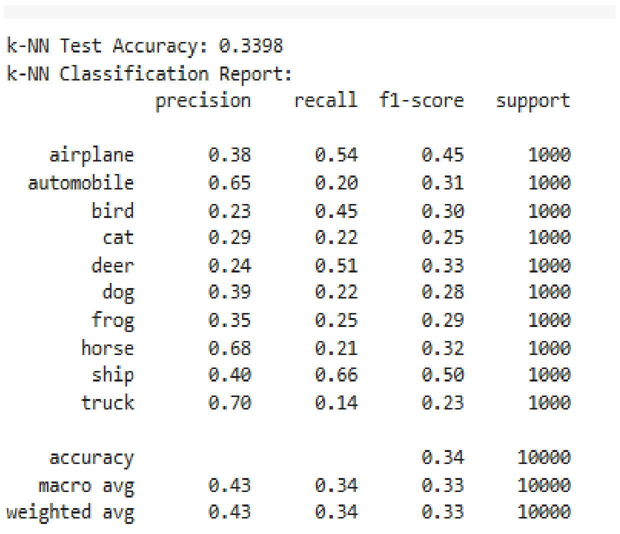
Python Script:

```
#Preprocess Data for Simpler Models

# Flatten the images for Logistic Regression and k-NN

train_images_flat = train_images.reshape(train_images.shape[0], -1)

test_images_flat = test_images.reshape(test_images.shape[0], -1)

print("Flattened training data shape:", train_images_flat.shape)

print("Flattened test data shape:", test_images_flat.shape)


#Logistic Regression

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score, classification_report

# Train Logistic Regression

logistic_model = LogisticRegression(max_iter=100, solver='saga', multi_class='multinomial')

logistic_model.fit(train_images_flat, np.argmax(train_labels, axis=1))

# Predict and evaluate

logistic_predictions = logistic_model.predict(test_images_flat)

logistic_accuracy = accuracy_score(np.argmax(test_labels, axis=1), logistic_predictions)

print(f"Logistic Regression Test Accuracy: {logistic_accuracy:.4f}")

print("Logistic Regression Classification Report:")

print(classification_report(np.argmax(test_labels, axis=1), logistic_predictions,
target_names=class_names))
```

```
#k-Nearest Neighbors (k-NN)

from sklearn.neighbors import KNeighborsClassifier

# Train k-NN

knn_model = KNeighborsClassifier(n_neighbors=5)  # You can experiment with different k values

knn_model.fit(train_images_flat, np.argmax(train_labels, axis=1))

# Predict and evaluate

knn_predictions = knn_model.predict(test_images_flat)

knn_accuracy = accuracy_score(np.argmax(test_labels, axis=1), knn_predictions)

print(f"k-NN Test Accuracy: {knn_accuracy:.4f}")

print("k-NN Classification Report:")

print(classification_report(np.argmax(test_labels, axis=1), knn_predictions,
target_names=class_names))
```



```
Logistic Regression Test Accuracy: 0.4036
Logistic Regression Classification Report:
              precision    recall  f1-score   support

    airplane       0.46      0.49      0.47      1000
  automobile       0.47      0.47      0.47      1000
        bird       0.33      0.29      0.31      1000
         cat       0.28      0.26      0.27      1000
        deer       0.35      0.29      0.32      1000
         dog       0.33      0.33      0.33      1000
        frog       0.40      0.46      0.43      1000
       horse       0.45      0.44      0.44      1000
        ship       0.50      0.53      0.51      1000
       truck       0.43      0.46      0.45      1000

    accuracy                           0.40     10000
   macro avg       0.40      0.40      0.40     10000
weighted avg       0.40      0.40      0.40     10000
```



```
k-NN Test Accuracy: 0.3398
k-NN Classification Report:
              precision    recall  f1-score   support

    airplane       0.38      0.54      0.45      1000
  automobile       0.65      0.20      0.31      1000
        bird       0.23      0.45      0.30      1000
         cat       0.29      0.22      0.25      1000
        deer       0.24      0.51      0.33      1000
         dog       0.39      0.22      0.28      1000
        frog       0.35      0.25      0.29      1000
       horse       0.68      0.21      0.32      1000
        ship       0.40      0.66      0.50      1000
       truck       0.70      0.14      0.23      1000

    accuracy                           0.34     10000
   macro avg       0.43      0.34      0.33     10000
weighted avg       0.43      0.34      0.33     10000
```

Model	**Test Accuracy**
Convolutional Neural Network (CNN)	72.98%
Logistic Regression	40.36%
k-Nearest Neighbors (k-NN)	33.98%

Note: The CNN outperforms both Logistic Regression (40.36%) and k-NN (33.98%) with an accuracy of 72.98%, highlighting its ability to effectively learn spatial features from images. Logistic Regression and k-NN struggle with high-dimensional data, making CNN the most suitable model for CIFAR-10 classification tasks.

3. Conclusion

This project presents a detailed study of CNNs for image classification using the CIFAR-10 dataset and an evaluation of their performance against other machine learning techniques such as Logistic

Regression and k-NN. We started by importing essential libraries and preparing the dataset, followed by data preprocessing and normalization, ensuring the images were ready for model training. We were able to train and design a CNN model inclusive of numerous convolutional and pooling layers sufficient to work spatial features of images and fine-tuning by the Adam optimizer. During training, we also assessed accuracy and loss, visualized the training, tested performance and model test using test accuracy, classification report, and confusion matrix. We then compared CNN performance with other models. Logistic Regression and k-NN achieved comparatively low accuracies of 40.36% and 33.98%, respectively, but the test accuracy of CNN was 72.98%. This demonstrated the advantages of CNNs in handling high-dimensional, image-based data. Thus, CNNs are very effective in image classification; moreover, the capacity to learn required features directly from the pixel data gives a huge advantage to CNNs compared to traditional machine learning algorithms. This project highlights the importance of model selection and optimization in achieving high performance in image classification tasks.

4. Key Take-Aways

- Convolutional Neural Networks (CNNs) excel in image classification tasks by learning spatial and hierarchical features, achieving 72.98% accuracy on the CIFAR-10 dataset.
- Techniques like Dropout, Data Augmentation, Batch Normalization, and Early Stopping improved CNN performance and generalization.
- Logistic Regression (40.36%) and k-Nearest Neighbors (33.98%) were outperformed by CNN due to their inability to handle high-dimensional image data effectively.
- Metrics such as accuracy, loss curves, classification reports, and confusion matrices were essential in assessing model performance.
- The results underscore the importance of selecting the right model and applying optimizations for complex datasets like CIFAR-10.

5. References

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Available at: https://www.cs.toronto.edu/~kriz/cifar.html (Accessed: 5 December 2024).

Analytics Vidhya (2020). *Learn Image Classification using CNN (Convolutional Neural Networks) – 3 Datasets*. Available at: https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/ (Accessed: 5 December 2024).

Kaggle (2023). *Image Classification using CNN | 94% Accuracy*. Available at: https://www.kaggle.com/code/arbazkhan971/image-classification-using-cnn-94-accuracy (Accessed: 5 December 2024).

Fast.ai (2019). *Determining When You Are Overfitting, Underfitting, or Just Right*. Available at: https://forums.fast.ai/t/determining-when-you-are-overfitting-underfitting-or-just-right/7732 (Accessed: 5 December 2024).

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. Available at: https://www.deeplearningbook.org/ (Accessed: 5 December 2024).

Chollet, F. (2017). *Deep Learning with Python*. Manning Publications. Available at: https://www.manning.com/books/deep-learning-with-python (Accessed: 5 December 2024).

Keras Documentation (2024). *Convolutional Neural Networks (CNNs).* Available at: https://keras.io/guides/understanding_convnets/ (Accessed: 5 December 2024).

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer. Available at: https://web.stanford.edu/~hastie/ElemStatLearn/ (Accessed: 5 December 2024).

TechTarget (n.d.). *Convolutional Neural Network (CNN)*. Available at: https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network (Accessed: 5 December 2024).

Scikit-learn Documentation (2024*). k-Nearest Neighbors*. Available at: https://scikit-learn.org/stable/modules/neighbors.html (Accessed: 5 December 2024).

GitHub Repository:

https://github.com/Ibrahim22331/CNN-Model-Comparison-Report.git

Dataset link:

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images*. Available at: https://www.cs.toronto.edu/~kriz/cifar.html (Accessed: 5 December 2024)