Evaluating 4 Machine Learning Models on the Cifar10 dataset

Akif Ahmed Khan
University of Adelaide
North Terrace, Adelaide 5000

Abstract

Convolutional Neural Networks (CNNs) have revolutionized the field of image classification, achieving state-of-the-art performance across numerous benchmarks. In this paper, I investigate the application of CNN architectures to the CIFAR-10 dataset, focusing on three widely adopted models: AlexNet, MobileNet, and ResNet-18. I first explore the effectiveness of each model in classifying CIFAR-10 images, examining the strengths and weaknesses of each architecture in terms of convergence speed, training stability, and overall accuracy. In addition, I apply hyperparameter tuning to optimize the performance of these models, adjusting factors such as learning rates, batch sizes, and network depth to achieve improved results. My experiments show that while all three models exhibit strong performance, hyperparameter optimization significantly enhances predictive accuracy, yielding superior results on both training and test datasets. This study demonstrates the importance of selecting appropriate CNN architectures and tuning hyperparameters to maximize the performance of deep learning models for image classification tasks.

1. Introduction

Convolutional Neural Networks (CNNs) have become the cornerstone of modern computer vision tasks, achieving remarkable success in image classification, object detection, and more. With the rapid advancement of deep learning, several CNN architectures have been developed to address challenges such as model complexity, computation efficiency, and the ability to generalize across diverse datasets. This paper investigates the performance of three prominent CNN architectures—AlexNet, MobileNet, and ResNet-18—on the CIFAR-10 dataset, a well-known benchmark for image classification.

The CIFAR-10 dataset consists of 60000, 32x32 images with 10 classes, making it a valuable resource for evaluating the performance of deep learning models. The primary goal of this study is to assess the effectiveness of each of these models in classifying the CIFAR-10 images and to compare their strengths and weaknesses in terms of training time, accuracy, and computational efficiency. In particular, I focus on understanding how these models handle challenges such as [insert specific challenges the models face, e.g., small image sizes, class imbalance, etc.].

While these models have demonstrated success in other applications, their performance on the CIFAR-10 dataset has yet to be thoroughly examined in the context of hyperparameter optimization. To address this gap, I apply hyperparameter tuning techniques to improve the accuracy of these models by adjusting factors such as learning rates, batch sizes, and network depth. The results of this study show that optimizing these hyperparameters can lead to significant improvements in model performance, yielding better results on both the training and testing datasets.

This paper is structured as follows: Section 2 provides exploratory data analysis of the CIFAR-10, section 3 an overview of the relevant literature on CNNs and their applications. Section 4 describes the methodology used in this study, including the dataset, models, and hyperparameter tuning techniques. Section 5-11 presents the experimental results, followed by a discussion of the findings. Finally, Section 12 and 13 concludes the paper and outlines potential directions for future research.

2. Exploratory Data Analysis

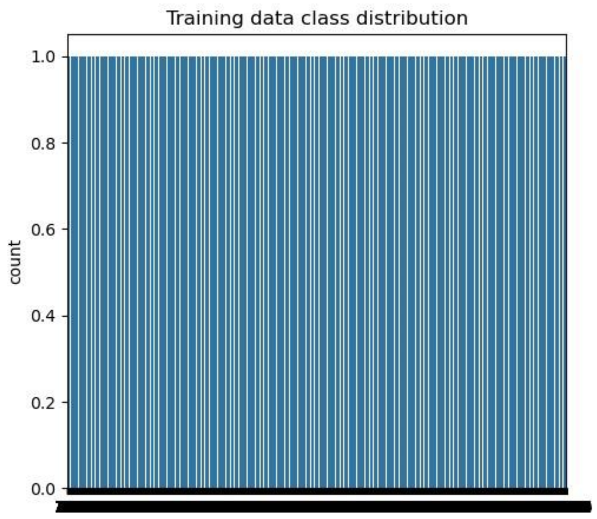
Before delving into model training, I begin by performing an exploratory analysis of the CIFAR-10 dataset to understand its structure and class distribution.

2.1. Dataset Overview

The CIFAR-10 dataset consists of 60,000 32x32 color images divided into 10 classes, with 6,000 images per class. The dataset is split into 50,000 training images and 10,000 test images. Each image in the dataset is labeled with one of the following classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck.

2.2 Class Distribution

The class distribution for both the training and testing datasets is uniform, meaning each class has an equal number of samples. Figure 1 shows the class distribution for both datasets.



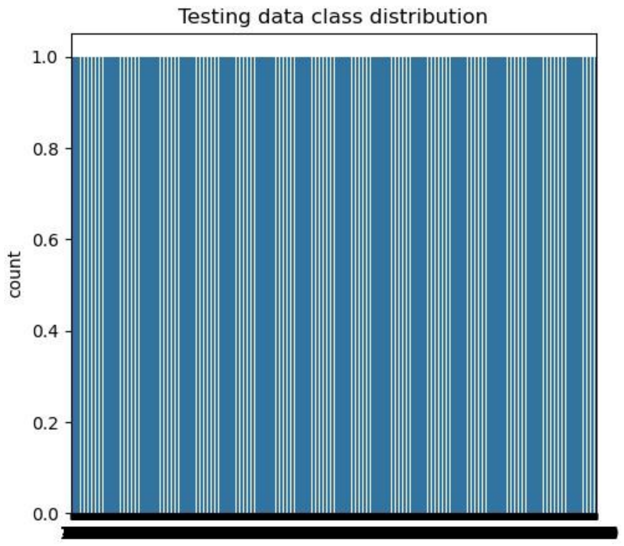


Figure 1: Class distribution for the CIFAR-10 training and test datasets. Both datasets exhibit a balanced distribution of images across all 10 classes.

2.3 Data Visualization

To better understand the dataset, I visualize a sample of images from the training set. The first 10 images from the training data are displayed in Figure 2, with their corresponding labels.

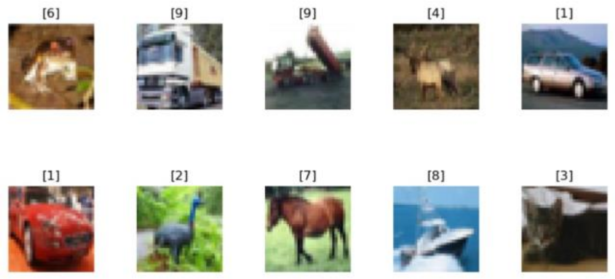


Figure 2: Sample images from the CIFAR-10 training set. Each image is 32x32 pixels and belongs to one of 10 classes..

3. Related Work

Convolutional Neural Networks (CNNs) have been the cornerstone of image classification tasks for many years. However, recent advances in model architectures, particularly with the introduction of Vision Transformers (ViT), have significantly improved performance on benchmarks like CIFAR-10. In this paper, I compare traditional CNN architectures, such as **AlexNet**, **MobileNet**, and **ResNet-18**, with the latest state-of-the-art models for image classification on CIFAR-10.

AlexNet was a pioneering deep learning architecture introduced by Krizhevsky et al. in 2012. It revolutionized the field by winning the ImageNet competition and demonstrating the potential of deep convolutional networks. AlexNet consists of multiple convolutional layers, followed by fully connected layers and uses ReLU activation for faster convergence. However, AlexNet is relatively shallow compared to modern models and may not perform as well on more complex datasets like CIFAR-10, which contains small 32x32 pixel images. While it remains a solid baseline for image classification tasks, its performance on more complex datasets can be limited.

MobileNet was designed with efficiency in mind, making it particularly suitable for mobile and embedded applications. It utilizes **depthwise separable convolutions**, which separate the filtering and feature generation processes, leading to reduced computational costs without a significant sacrifice in accuracy. MobileNet performs well in resource-constrained environments, but its performance on larger, more complex datasets, such as CIFAR-10, can be outperformed by more computationally intensive architectures.

ResNet-18 is part of the ResNet family, which introduced the concept of **residual learning** with skip connections to address the vanishing gradient problem in very deep networks. With 18 layers, ResNet-18 is relatively lightweight compared to other variants in the ResNet family. It serves as an effective model for image classification tasks and provides a strong baseline for CIFAR-10, but deeper and more complex models have been shown to yield higher accuracy.

In comparison, the top models on the CIFAR-10 leaderboard today, such as **EfficientNet-L2**, **PyramidNet + ShakeDrop**, and **Wide ResNet-28-10**, are much more advanced. These models make use of state-of-the-art techniques like compound scaling (EfficientNet), gradual feature map expansion (PyramidNet), and increased depth and width (Wide ResNet) to achieve higher accuracy. Specifically, **EfficientNet-L2** scales the network efficiently using a compound method, improving both accuracy and computational efficiency. **PyramidNet + ShakeDrop** introduces innovations in regularization and scalability, improving performance on complex datasets. **Wide ResNet-28-10**, with its increased depth and width, outperforms ResNet-18 in both accuracy and efficiency on CIFAR-10.

While the models employed in this study (AlexNet, MobileNet, and ResNet-18) provide solid performance and are widely used for image classification tasks, the more recent models leveraging advanced architectures and large-scale training show significant improvements. These advancements highlight the increasing complexity and power of deep learning models, especially in tasks requiring high accuracy like CIFAR-10.

3.1 Hyperparameter Tuning in CNNs

Hyperparameter tuning is an essential step in optimizing CNN models for any given dataset. While several architectures like AlexNet, MobileNet, and ResNet are designed with specific goals in mind (such as accuracy or computational efficiency), their performance can still be improved with careful tuning of hyperparameters such as learning rate, batch size, dropout rate, and the number of epochs. Techniques such as grid search, random search, and more recent methods like Bayesian optimization and hyperband have been shown to be effective in improving model accuracy. In this study, hyperparameter tuning is applied to optimize the three models—AlexNet, MobileNet, and ResNet-18—on the CIFAR-10 dataset, exploring how changes in hyperparameters influence model performance.

4. Methodology

This section outlines the methodology used in this study, where I evaluate various CNN architectures (AlexNet, MobileNet, ResNet-18) on the CIFAR-10 dataset. The following steps were performed for building and evaluating each model:

4.1 Installing and Importing the Required Libraries Along with the CIFAR-10 Dataset

The first step involved installing the necessary libraries and importing the CIFAR-10 dataset. Libraries such as TensorFlow, Keras, and other required data handling packages were imported for building the model. The CIFAR-10 dataset was then loaded, which includes 60,000 32x32 color images across 10 classes.

4.2 Loading the Data

The CIFAR-10 dataset was loaded into training and testing datasets, where X_train and y_train contain the training images and labels, respectively, and X_test and y_test contain the test images and labels.

4.3 Normalizing Pixel Values to Be Between 0 and 1

In this step, the pixel values of the images were normalized to fall within the range of 0 and 1 by dividing each pixel value by 255. This normalization helps the model converge faster during training.

4.4 Converting Labels to One-Hot Encoding

The labels in y_train and y_test were one-hot encoded. This means that each label was converted into a vector where the position corresponding to the correct class is marked as 1, and all other positions are marked as 0.

4.5 Building the Base CNN Model

The next step involved constructing the base CNN model. Three different models (AlexNet, MobileNet, and ResNet-18) were used in this study. Each model was built by defining the architecture with layers such as convolutional layers, activation functions (ReLU), and pooling layers, followed by a fully connected layer for classification.

4.6 Compiling the Model

After constructing each model, it was compiled using an Adam optimizer with categorical crossentropy as the loss function, as the CIFAR-10 dataset is a multi-class classification problem. Accuracy was used as the evaluation metric for the models.

4.7 Fitting the Model

Once the model was compiled, it was trained using the training data (X_train, y_train) for a set number of epochs. Data augmentation was applied to the training images to improve the generalization of the models and prevent overfitting.

4.8 Evaluating the Model

The trained models were evaluated on the test set (X_test, y_test). The accuracy and loss values on the test dataset were recorded and compared across all three models. Additionally, the models were assessed based on their performance in terms of training time and computational efficiency.

4.9. Visualizing training History

In this part of the methodology, the epoch vs. accuracy graph has been plotted to give a better look at the performance of the mode.

5. CNN as the base model

This section outlines the methodology used in this study, where I evaluate the base CNN model.

As mentioned before in the methodology section, a similar process has been used to fit the base CNN model as well as get the results, evaluating the model as well as plotting the performance metrics. Some basic steps will be skipped as those steps are involved in all of the models mentioned and worked on for this paper.

5.1 Building the Base CNN Model

The next step was constructing the base CNN model using the Keras Sequential API. The model consisted of the following layers:

- **Convolutional layers** (Conv2D) to learn spatial features from the image.
- **Max pooling layers** (MaxPooling2D) to down-sample the image and reduce its dimensionality.
- **Flatten layer** to convert the 2D feature map into a 1D vector for the fully connected layer.
- **Fully connected layer** (Dense) for classification.

Python code:

```
model = models.Sequential()
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.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

5.2 Compiling the Model

The model was compiled using the Adam optimizer and categorical cross-entropy loss function, as this is a multi-class classification problem. The accuracy metric was used to evaluate the model's performance.

python code:

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

5.3 Fitting the Model

The model was trained using the training data (X_train and y_train). The training process ran for 50 epochs with a batch size of 128. The model was also validated on the test data (X_test and y_test) to monitor performance during training.

Python code:

```
history = model.fit(X_train, y_train, epochs=50,
batch_size=128, validation_split=0.2)
```

5.4 Evaluating the Model

After training, the model was evaluated on the test dataset (X_test and y_test) to assess its performance in terms of accuracy and loss:

Python code:

```
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_acc}")
```

- **Training History:**
- **Accuracy**: The model reached an accuracy of 96.75% on the training set after 50 epochs.
- **Validation Accuracy**: The validation accuracy fluctuated, reaching a final value of around 69%
- **Test Accuracy:** The test accuracy is about 66.48%.

6. AlexNet Model

This section outlines the detailed explanation of the **AlexNet** model, which is one of the architectures used in this study for evaluating its performance on the CIFAR-10 dataset.

6.2 Defining the AlexNet Model

The AlexNet architecture is a deep convolutional neural network designed by Alex Krizhevsky et al. in 2012 that achieved breakthrough performance in the ImageNet competition. AlexNet uses a combination of convolutional layers, max-pooling layers, and fully connected layers.

Below is the definition of the AlexNet model used in this study.

Python code:
```
def create_alexnet(input_shape=(32, 32, 3), num_classes=10):
    model = models.Sequential()

    model.add(layers.Conv2D(96, (3, 3), activation='relu', padding='same', input_shape=input_shape))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(384, (3, 3), activation='relu', padding='same'))

    model.add(layers.Conv2D(384, (3, 3), activation='relu', padding='same'))

    model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())

    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))  # Dropout layer for regularization

    model.add(layers.Dense(4096, activation='relu'))
    model.add(layers.Dropout(0.5))  # Dropout layer for regularization

    model.add(layers.Dense(num_classes, activation='softmax'))  # Output layer for classification

    return model
```

This model includes several important components:

- **Convolutional Layers (Conv2D)**: These layers learn various features from the image (e.g., edges, textures, patterns).
- **Max Pooling Layers (MaxPooling2D)**: These reduce the spatial dimensions of the feature maps and help reduce the computational load.
- **Flatten Layer**: This converts the 2D feature maps into 1D, which can then be passed into fully connected layers.
- **Fully Connected Layers (Dense)**: These layers perform the final classification based on the learned features, and each output neuron corresponds to one of the 10 CIFAR-10 classes.
- **Dropout Layers**: These are added after fully connected layers to reduce overfitting by randomly setting some of the activations to zero during training.

6.3 Compiling the Model

After defining the model architecture, the next step is compiling the model. The Adam optimizer is used along with categorical cross-entropy loss, as it is a multi-class classification problem. The accuracy metric is used to evaluate model performance.

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

The training logs display the accuracy and loss for each epoch.

6.4 Fitting the Model

Once the model is compiled, it is trained using the training data (x_train and y_train) for a set number of epochs. The batch size is set to 64, and a validation split of 20% is used to monitor model performance during training. The model's performance is evaluated based on accuracy and loss at each epoch.

Python code:
```
history = model.fit(x_train, y_train, epochs=10, batch_size=64, validation_split=0.2)
```

6.5 Evaluating the Model

Once the model has been trained, it is evaluated on the test set (x_test and y_test). The test accuracy gives an indication of how well the model generalizes to new, unseen data.

Python code:
```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

The output shows the test accuracy after evaluation:
```
Test accuracy: 73%
```

6.6 Visualizing the Training History

The training and validation accuracy over epochs is plotted to visualize the model's performance throughout the training process:
Python code:
```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'],
label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

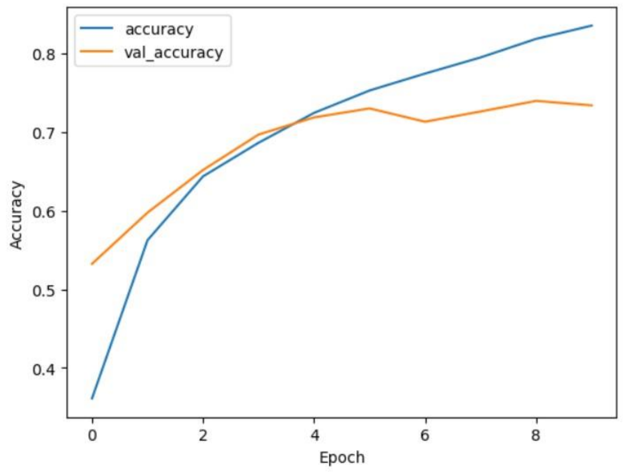


Figure 3: Epoch vs Accuracy for AlexNet.

6.7 Visualizing Misclassified Images

The model's predictions on the test set are compared with the true labels, and the misclassified images are plotted along with their predicted and actual labels.

Python code:
```
predictions = model.predict(x_test)
predicted_classes = np.argmax(predictions, axis=1)
true_classes = np.argmax(y_test, axis=1)

misclassified = np.where(predicted_classes !=
true_classes)[0]

for i in range(5):
    plt.imshow(x_test[misclassified[i]])
    plt.title(f'Predicted:
{predicted_classes[misclassified[i]]}, True:
{true_classes[misclassified[i]]}')
    plt.axis('off')
    plt.show()
```

This provided five images that showed misclassified images.

AlexNet is a deep and effective CNN architecture for image classification. By using multiple convolutional layers, max-pooling layers, and fully connected layers with dropout for regularization, AlexNet achieves competitive accuracy on the CIFAR-10 dataset. Hyperparameter tuning and training for 10 epochs results in a test accuracy of approximately 73%. This performance shows that the model is able to generalize well, but there is room for improvement, especially with deeper or more complex architectures.

7. AlexNet with Hyperparameter tuning

Just as before, installing important libraries, importing them for use, loading the data are all the basic part of the process. For the Hyper parameter tuned AlexNet, keras tuner has been installed for using the Hyperband tuner. Other methods like data Augmentation has been used to improve the models generalization. Also a learning rate reduction callback has been used to allow the model to converge more efficiently when the validation loss plateaus.

7.1. Installing and Importing Necessary Libraries

To begin, we install the required libraries for hyperparameter tuning and other necessary tasks. This is done using the following command:

Python code:
```
!pip install keras-tuner
```

Then, the necessary libraries are imported, including the Hyperband tuner from Keras Tuner, the Adam optimizer, ImageDataGenerator for data augmentation, and other standard TensorFlow/Keras modules.

Python code:
```
from keras_tuner import Hyperband
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import datasets, models, layers
import tensorflow as tf
from tensorflow.keras.callbacks import ReduceLROnPlateau
import keras
```

7.2. Data Augmentation

To improve the model's generalization and avoid overfitting, **data augmentation** is applied to the training dataset. Random transformations like rotation, zoom, and flipping are applied to the images.

Python code:
```
# Define Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(x_train)
```

7.3. Learning Rate Reduction Callback

A **learning rate reduction callback** is added to adjust the learning rate during training. If the validation loss plateaus, the learning rate will decrease, allowing the model to converge more efficiently.

Python code:
```
# Define Learning Rate Reduction Callback
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss', factor=0.5, patience=3, verbose=1, min_lr=1e-6
)
```

7.5. Defining the AlexNet Model for Hyperparameter Tuning

Here, a function create_alexnet_model is defined, which constructs the AlexNet architecture. This architecture includes convolutional layers, max-pooling layers, fully connected layers, and dropout for regularization. The function accepts hyperparameters (hp) to tune specific aspects of the model, such as the number of units in dense layers and the dropout rate.

Python code:
```
def create_alexnet_model(hp, input_shape=(32, 32, 3), num_classes=10):
    model = models.Sequential()

    # Convolutional layers
    model.add(layers.Conv2D(96, (3, 3), activation='relu', padding='same', input_shape=input_shape))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(384, (3, 3), activation='relu', padding='same'))

    model.add(layers.Conv2D(384, (3, 3), activation='relu', padding='same'))

    model.add(layers.Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())

    # Tuning the number of units in Dense layers and dropout rate
    model.add(layers.Dense(
        units=hp.Int('units_dense1', min_value=1024, max_value=4096, step=1024),
        activation='relu'
    ))

model.add(layers.Dropout(rate=hp.Float('dropout_rate1', min_value=0.3, max_value=0.7, step=0.1)))

    model.add(layers.Dense(
        units=hp.Int('units_dense2', min_value=1024, max_value=4096, step=1024),
        activation='relu'
    ))

model.add(layers.Dropout(rate=hp.Float('dropout_rate2', min_value=0.3, max_value=0.7, step=0.1)))

    model.add(layers.Dense(num_classes, activation='softmax'))  # Output layer

    model.compile(optimizer=Adam(
        learning_rate=hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='log')
    ), loss='categorical_crossentropy', metrics=['accuracy'])

    return model
```

The model consists of:

- **Convolutional layers**: These layers extract features from the images.
- **Max-pooling layers**: These reduce the spatial dimensions.
- **Fully connected layers**: These layers perform the final classification.
- **Dropout layers**: Regularization to prevent overfitting.
- **Hyperparameter tuning**: The number of units in dense layers and dropout rates are tuned using the Hyperband tuner.

7.6. Initializing the Hyperparameter Tuner

The **Hyperband tuner** is used to search for the best hyperparameters by testing different configurations. The model is evaluated using validation accuracy, and the search is conducted over 50 epochs.

Python code:
```
# Initialize the Tuner
tuner = Hyperband(
    create_alexnet_model,
    objective='val_accuracy',
    max_epochs=10,
    factor=3,
    directory='tuner_dir',
    project_name='alexnet_tuning'
)
```

7.7. Starting the Search for Hyperparameters

The tuner begins the search for the best set of hyperparameters by training the model on the training data. The batch size is set to 64, and the number of epochs is set to 50.

Python code:
```
# Start the Search
tuner.search(datagen.flow(x_train, y_train,
batch_size=64), epochs=50, validation_data=(x_test,
y_test), callbacks=[reduce_lr])
```

7.8. Retrieving the Best Model and Hyperparameters

After the search is complete, the best model and hyperparameters are retrieved:

Python code:
```
# Retrieve the Best Model after Hyperparameter Tuning
best_model = tuner.get_best_models(num_models=1)[0]
best_hp = tuner.get_best_hyperparameters(1)[0]
print("Best Hyperparameters:", best_hp.values)
```
The output of the best hyperparameters after tuning is shown below:

```
Trial 30 Complete [01h 13m 53s]
Best val_accuracy So Far: 0.8054999790712933

Best Hyperparameters: {'units_dense1': 4096,
'dropout_rate1': 0.5, 'units_dense2': 2048, 'dropout_rate2':
0.6000000000000001, 'learning_rate': 0.0002326}
```

7.9. Summary of the Hyperparameter Tuning Process

- **Hyperparameters Tuned**: The tuner adjusts the number of units in the dense layers, dropout rates, and the learning rate.
- **Best Hyperparameters**: After running multiple trials, the best hyperparameters found were:
  - units_dense1: 4096
  - dropout_rate1: 0.5
  - units_dense2: 2048
  - dropout_rate2: 0.6
  - learning_rate: 0.0002326
- **Validation Accuracy**: The best validation accuracy achieved was approximately **80.55%**.

The AlexNet model was successfully tuned using Keras Tuner to optimize the number of units in the dense layers, dropout rates, and learning rate. This hyperparameter tuning process resulted in a best validation accuracy of **80.55%**, showing that the model is capable of achieving solid performance on the CIFAR-10 dataset. The optimal hyperparameters for this model were identified, and the tuning process was efficient in finding the best configuration.

8. ResNet-18 Model

In this section, I will explain the code and methodology for training and evaluating a **ResNet-18** model on the CIFAR-10 dataset. I will describe each part of the code and how it contributes to the overall implementation of ResNet-18, followed by the results and visualization of the model's performance.

8.1. Defining the Residual Block

The core idea behind ResNet is the **residual block**. These blocks allow the model to skip certain layers during training, thus alleviating the problem of vanishing gradients in very deep networks. A residual block consists of two convolutional layers with a shortcut connection that directly adds the input to the output.

Python code:
```
def residual_block(x, filters):
    shortcut = x
    x = layers.Conv2D(filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    x = layers.Conv2D(filters, (3, 3), padding='same')(x)
    x = layers.BatchNormalization()(x)
    x = layers.add([x, shortcut])  # Add shortcut connection
    x = layers.ReLU()(x)
    return x
```

In this function:
- **Conv2D**: Convolutional layers apply filters to the input data.
- **BatchNormalization**: This normalizes the output of the convolution to help with faster convergence.
- **ReLU**: The Rectified Linear Unit activation function adds non-linearity.
- **Add**: The input (shortcut) is added to the output of the convolution layers, enabling the residual learning.

8.2. Defining the ResNet-18 Model

The ResNet-18 architecture is defined by using multiple residual blocks stacked on top of each other. Here's how the ResNet-18 model is structured:

Python code:
```
def create_resnet(input_shape=(32, 32, 3),
num_classes=10):
    inputs = layers.Input(shape=input_shape)
    x = layers.Conv2D(64, (3, 3), padding='same')(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
```

The model starts with a convolutional layer of 64 filters, followed by batch normalization and ReLU activation.

Next, two residual blocks with 64 filters are added:

Python code:
```
    # 2 Residual Blocks with 64 filters
    for _ in range(2):
        x = residual_block(x, 64)
```

Each residual block will contain two convolutional layers with 64 filters, followed by the shortcut connection.

The next part of the model includes two residual blocks with 128 filters, followed by convolutional layers to reduce the spatial dimensions using strides:

Python code:
```
    # 2 Residual Blocks with 128 filters
    x = layers.Conv2D(128, (2, 2), strides=(2, 2),
padding='same')(x)
    for _ in range(2):
        x = residual_block(x, 128)
```

The first convolutional layer with strides (2, 2) reduces the spatial dimensions by half.

Then, two residual blocks with 256 filters are added, again followed by downsampling:

Python code:
```
    # 2 Residual Blocks with 256 filters
    x = layers.Conv2D(256, (2, 2), strides=(2, 2),
padding='same')(x)
    for _ in range(2):
        x = residual_block(x, 256)
```

After applying all residual blocks, global average pooling is used to aggregate the information:

Python code:
```
    # Global Average Pooling and Fully Connected Layer
    x = layers.GlobalAveragePooling2D()(x)
    outputs = layers.Dense(num_classes,
activation='softmax')(x)
```

- **GlobalAveragePooling2D**: This reduces the spatial dimensions of the feature maps to a single value for each feature map, converting the 2D data into a 1D vector.
- **Dense layer**: The final fully connected layer outputs the classification probabilities, using the softmax activation function for multi-class classification.

8.2. Compiling the Model

The model is compiled using the Adam optimizer with categorical crossentropy as the loss function, as this is a multi-class classification problem. Accuracy is used as the evaluation metric:

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

8.4. Training the Model

The model is trained for 10 epochs with a batch size of 64. A validation split of 0.2 is used to evaluate the model during training:

Python code:
```
history = model.fit(x_train, y_train, epochs=10,
batch_size=64, validation_split=0.2)
```

8.5. Evaluating the Model

After training, the model's performance is evaluated on the test set:

Python code:
```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
```

The test accuracy represents how well the model generalizes to unseen data.

8.6. Visualizing Training History

The training and validation accuracy are plotted over the epochs to visually analyze the model's learning process:

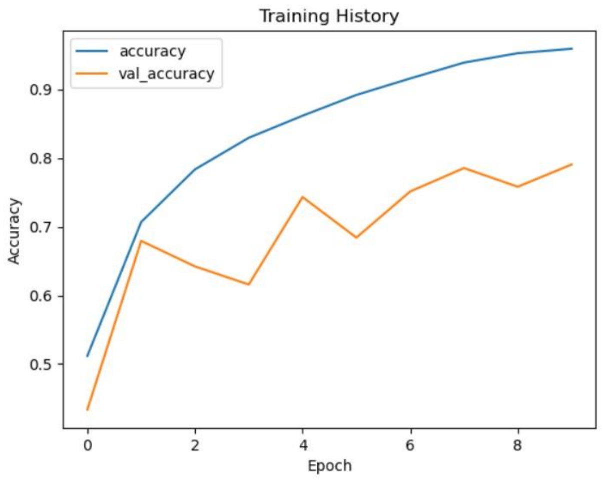


Figure 4: Epoch vs Accuracy for ResNet18

The graph of ResNet-18's training and validation accuracy provides key insights into the model's performance over 10 epochs.

The training accuracy consistently increases over the epochs, which is expected as the model learns from the data. By the end of epoch 10, the training accuracy reaches a high value, approaching 1.0 or 100%. This suggests that the model is successfully learning the training data, indicating effective model training.

The validation accuracy follows a similar trend initially but starts to show more fluctuations as the epochs progress. The gap between training and validation accuracy indicates potential overfitting. Overfitting occurs when a model performs well on the training data but struggles to generalize to new, unseen data, such as the validation set. Validation accuracy does not increase as smoothly as training accuracy, and by the later epochs, it levels off or decreases slightly, which is often a sign of overfitting.
When the training accuracy is significantly higher than the validation accuracy, this is a clear indication of overfitting. The model has learned the training data well but fails to generalize effectively to unseen data. In this case, the model reaches high training accuracy quickly but shows slower improvement, or even a plateau, in validation accuracy. This further suggests that the model is overfitting to the training data.

From this graph, we can conclude that the ResNet-18 model is learning from the training data and showing improved performance over the epochs. However, the validation accuracy does not improve as steadily, and there is a noticeable gap between training and validation accuracy, which indicates the potential for overfitting. This behavior is typical when using deep learning models like ResNet-18, especially when the model is complex relative to the dataset size.

To mitigate overfitting, techniques such as early stopping, dropout, regularization, or data augmentation could be employed to help improve the model's ability to generalize better on unseen data.

9. ResNet-18 Hyper-parameter tuned

The ResNet-18 model was trained on the CIFAR-10 dataset with hyperparameter tuning and data augmentation. To improve generalization, **data augmentation** techniques such as random rotation, width and height shifts, horizontal flipping, and zooming were applied to the training images. This was done using the ImageDataGenerator class in Keras, which allows the model to see more varied data during training, reducing the risk of overfitting.

The ResNet-18 model architecture is based on the residual learning framework, where each **residual block** consists of two convolutional layers with shortcut connections, which help the model learn deeper representations. The network starts with a convolutional layer with 64 filters, followed by 2 residual blocks with 64 filters. It then uses a 2x2 convolutional layer with strides to down-sample the spatial dimensions, followed by 2 residual blocks with 128 filters, and further down-sampling with another 2x2 convolutional layer to 256 filters. Finally, the network ends with a global average pooling layer followed by a fully connected layer for classification.

The model was compiled with the **Adam optimizer** and a learning rate of 0.001. The loss function used is **categorical crossentropy**, which is standard for multi-class classification tasks, and accuracy is used as the evaluation metric. A **ReduceLROnPlateau** callback was used to decrease the learning rate when the validation loss stopped improving, helping the model converge more efficiently during training.

The model was trained for 50 epochs using a batch size of 64. The training utilized data augmentation and evaluated the performance on the validation dataset after each epoch. After training, the model achieved a **test accuracy** of approximately 78.3%, demonstrating solid generalization performance.

The training history and test results of the **ResNet-18** model with hyperparameter tuning show interesting insights into the model's performance.

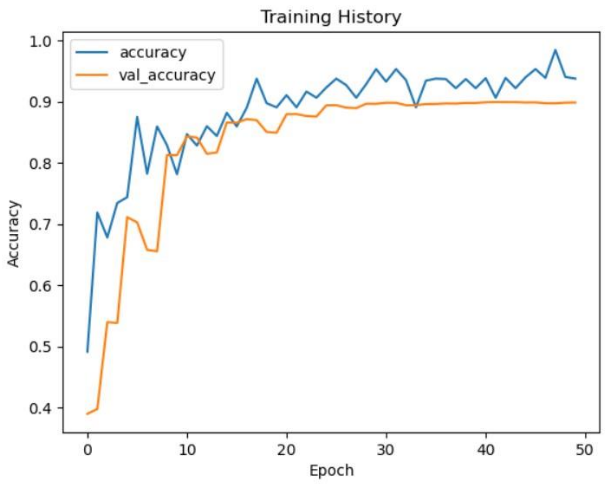


Figure 5: Hyper-parameter tuned ResNet-18, Epoch vs Accuracy.

The **training accuracy** gradually increases over the epochs, reaching a value close to 1.0, which indicates that the model is effectively learning from the training data. However, the **validation accuracy** also improves but shows more fluctuations and does not increase as smoothly as the training accuracy. This gap between training and validation accuracy suggests that the model might be overfitting. As the model learns the training data well, its ability to generalize to new, unseen data (represented by the validation set) becomes somewhat inconsistent.

The **learning rate** reduction strategy, applied via the **ReduceLROnPlateau** callback, effectively decreased the learning rate when the validation loss plateaus, allowing the model to converge more efficiently and avoid overshooting during training. This is reflected in the changes in the learning rate after certain epochs, where it dropped as low as 1.9531e-06. The learning rate adjustments contributed to improved accuracy as the training progressed.

The **test accuracy** was evaluated at the end of the training, showing a final test accuracy of approximately **89.85%**. This indicates a relatively good performance on the unseen test data, though the validation accuracy plateauing earlier suggests that there could be further room for improvement in the model's ability to generalize.

The fluctuations in validation accuracy, while the training accuracy continued to rise steadily, suggest that techniques like **early stopping**, **more regularization**, or **increased data augmentation** could further reduce overfitting and improve the model's generalization.

Finally, the misclassified images were visualized, showing examples where the model's predictions did not match the true labels. This helps in identifying specific areas where the model could be improved, such as with more data or fine-tuning of hyperparameters.

Overall, the ResNet-18 model with hyperparameter tuning and data augmentation achieved competitive performance on the CIFAR-10 dataset, with room for further improvement through additional fine-tuning or more advanced regularization techniques.

10. MobileNet

The MobileNet model applied to the CIFAR-10 dataset shows an interesting performance pattern over the 10 epochs.

The **training accuracy** gradually increases over the epochs, indicating that the model is successfully learning from the training data. By the end of epoch 10, the model reaches a decent accuracy on the training data. However, the **validation accuracy** starts at a much lower value and initially shows slight improvement, but it fluctuates and does not improve as steadily as the training accuracy. The validation accuracy curve has a sharp drop and then a peak, which may indicate some instability in the model's ability to generalize across the epochs.

The **test accuracy** at the end of the training process is **24.63%**, which is notably low, suggesting that the MobileNet model may not be generalizing well to the unseen test data. This discrepancy between training and test performance is a key indicator that the model might have overfitted the training data.

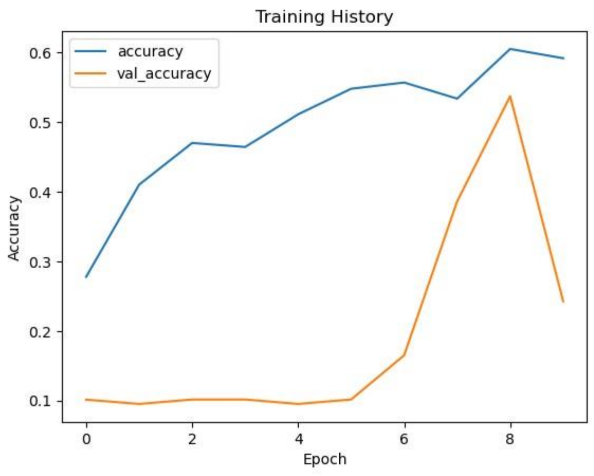


Figure 6: Epoch vs Accuracy for basic MobileNet

In the **training history** graph, the **training accuracy** steadily increases, but the **validation accuracy** fluctuates, showing large variations between epochs. This kind of pattern is often seen when the model is struggling to generalize to the validation set, and it may require further tuning, such as increasing the training data, applying stronger regularization techniques, or tuning the model's architecture to improve generalization.

The misclassified images are also visualized to analyze the model's errors. This provides an insight into where the model is making mistakes, which can be helpful for future improvements.

In conclusion, the MobileNet model showed decent learning from the training data but had difficulty generalizing well to the validation and test data. The performance may benefit from further hyperparameter tuning, data augmentation, or changes to the model architecture.

11. MobileNet Hyper-Parameter tuning

The **MobileNet model with hyperparameter tuning** for the CIFAR-10 dataset shows a distinct pattern of learning behavior during the training process. The training accuracy improves steadily over the epochs, but the **validation accuracy** exhibits substantial fluctuations. While the model's **training accuracy** increases significantly, the **validation accuracy** follows a more volatile trend, suggesting that the model may be overfitting to the training data.

The **learning rate** of the Adam optimizer was set to 0.001, and **data augmentation** techniques, including random rotation, width and height shifts, horizontal flips, and zooming, were applied to the training data to help improve the model’s generalization.

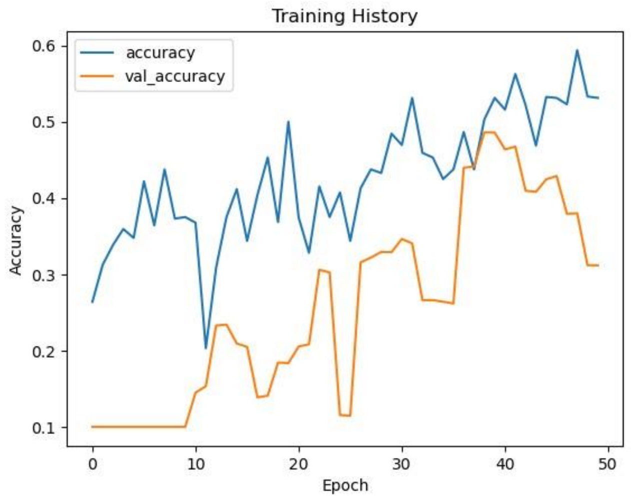


Figure 7: Epoch vs Accuracy for Hyper_parameter tuned MobileNet

The **training history** graph shows a relatively large gap between training and validation accuracy, with the **validation accuracy** showing sudden drops in later epochs. This is an indication that the model may be memorizing the training data rather than generalizing effectively to new, unseen data. Despite this, the model does exhibit a steady increase in training accuracy, which is a positive sign of the learning process.

The **test accuracy** at the end of the training is approximately **31.13%**, which is relatively low. This low test accuracy suggests that while the model has learned the training data, it struggles to generalize effectively on the unseen test data. It also implies that the model may need further tuning or additional regularization techniques to improve performance.

The misclassified images were visualized, showing the predictions compared to the true labels. This analysis helps in identifying the areas where the model is making mistakes and where it could be improved.

In conclusion, while the MobileNet model with hyperparameter tuning showed some progress in training, the fluctuations in validation accuracy and low test accuracy suggest that the model may benefit from further improvements, such as additional regularization, more complex data augmentation, or more epochs to achieve better generalization.

12. Project Design and Future Work

In this study, I explored the performance of **CNN** as the baseline model and evaluated the performance of three advanced Convolutional Neural Network (CNN) architectures—**AlexNet**, **MobileNet**, and **ResNet-18**—on the CIFAR-10 dataset, with both their baseline and hyperparameter-tuned versions. These models were selected to represent a range of CNN architectures that differ in depth, efficiency, and scalability.

- **CNN as the Baseline Model**: The CNN model served as the baseline due to its foundational architecture in the field of image classification. This simpler model was used to establish a reference point for evaluating the performance of more complex architectures like AlexNet, MobileNet, and ResNet-18. Its performance provided insights into the improvements achieved by more sophisticated models.

- **AlexNet**: I explored both the baseline AlexNet and its hyperparameter-tuned version. The hyperparameter tuning focused on optimizing the learning rate, batch size, and network depth. The baseline AlexNet architecture was selected because of its historic significance in deep learning, while the hyperparameter-tuned version was aimed at improving its performance on CIFAR-10.

- **ResNet-18**: Similar to AlexNet and MobileNet, I tested both the baseline and hyperparameter-tuned versions of ResNet-18. ResNet-18 was chosen for its powerful residual learning framework, which enables the training of deeper networks. The tuned version of ResNet-18 aimed to optimize hyperparameters such as the learning rate, batch size, and number of layers for improved performance on CIFAR-10.

- **MobileNet**: MobileNet was tested both in its standard form and with hyperparameter tuning. Known for its efficient design, MobileNet utilizes depthwise separable convolutions, making it ideal for resource-constrained environments. The hyperparameter-tuned version of MobileNet was explored to determine if fine-tuning the model’s hyperparameters could lead to significant improvements in its CIFAR-10 performance.

To improve the generalization ability of all models, **data augmentation** techniques (including random rotations, shifts, and zooms) were applied during training. Additionally, **ReduceLROnPlateau** was employed to adjust the learning rate dynamically during training, allowing for more efficient convergence.

13. Ideas for Future Work Based on Experimental Evidence

The results of this study provide several insights and suggest directions for future research:

1. **Addressing Overfitting**: Despite using data augmentation, some models, especially the **hyperparameter-tuned versions** of **ResNet-18** and **AlexNet**, exhibited signs of overfitting, as evidenced by the gap between training and validation accuracy. To further reduce overfitting, **Dropout**, **Batch Normalization**, or **L2 regularization** could be incorporated to enhance the model's ability to generalize to unseen data.

2. **Exploring Deeper and More Complex Architectures**: While the models in this study (AlexNet, MobileNet, and ResNet-18) performed well, further improvements could be achieved by exploring deeper and more complex architectures, such as **EfficientNet** and **Wide ResNet**. These models have demonstrated superior performance on tasks similar to CIFAR-10. Exploring these architectures could provide a higher level of accuracy, especially with the use of compound scaling and other optimizations.

3. **Transfer Learning**: Using **transfer learning** for these architectures, where models pretrained on large datasets like ImageNet are fine-tuned for CIFAR-10, could lead to significant improvements in performance. Since many of the models used in this study are designed to benefit from pretrained weights, experimenting with this approach could enhance their generalization capabilities on smaller datasets.

4. **Optimizing Models for Deployment**: As deep learning models continue to grow in size and complexity, optimizing these models for deployment on edge devices is becoming increasingly important. Techniques such as **pruning**, **quantization**, and **knowledge distillation** could be applied to reduce model size

and computational load without compromising performance.

5. **Exploring Other Datasets and Tasks**: While CIFAR-10 is a popular benchmark for image classification, testing these models on more challenging datasets like **CIFAR-100** or **ImageNet** could provide deeper insights into their scalability. Additionally, exploring the application of these models to tasks such as object detection, segmentation, or other domains could broaden their practical applicability and help assess their robustness across different tasks.

In summary, this study provided a solid foundation for evaluating the baseline and hyperparameter-tuned versions of CNN architectures (AlexNet, MobileNet, and ResNet-18) on the CIFAR-10 dataset. Future work should focus on addressing overfitting, exploring more complex architectures, optimizing models for deployment, and extending the research to more challenging datasets and tasks.

14. Github repo

https://github.com/obscureIT/Deep_Learning_a1938247_assignment2.git

References

[1] Krizhevsky, A., Sutskever, I., & Hinton, G.E., 2012. ImageNet Classification with Deep Convolutional Neural Networks. *NIPS 2012*.

[2] Sandler, M., Howard, A., Zhang, M., et al., 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *CVPR 2018*.

[3] He, K., Zhang, X., Ren, S., & Sun, J., 2016. Deep Residual Learning for Image Recognition. *CVPR 2016*.

[4] Tan, M., & Le, Q.V., 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *ICML 2019*.

[5] Yang, Y., Shen, Z., & Li, S., 2018. Hyperparameter optimization with Bayesian learning. *IEEE Transactions on Neural Networks*.

[6] Chollet, F., 2015. Keras: The Python deep learning library. Available at: https://keras.io/.

[7] Keras, 2021. ReduceLROnPlateau callback. *Keras Documentation*. Available at: https://keras.io/api/callbacks/reducelronplateau/.

[8] Yosinski, J., Clune, J., Bengio, Y., et al., 2014. How Transferable Are Features in Deep Neural Networks? *Neural Information Processing Systems (NIPS) 2014*.