

جامعة الأخوين

جامعة الأخوين

AL AKHAWAYN

UNIVERSITY

Project 4: Application of Computer Vision

Work of: Meriem Lmoubariki

Grade: 100/ 100

Supervised by: Dr. Tajjeeddine Rachidi

Fall 2024

| | |
|-----------------------------------|----|
| Introduction..... | 3 |
| Methodology..... | 4 |
| Model Architecture | 4 |
| Data Acquisition. | 5 |
| Training Process and Results..... | 9 |
| Hyperparameter Tuning..... | 28 |
| Performance Evaluation | 33 |
| Video Demo..... | 44 |
| Conclusion..... | 45 |
| References | 46 |

INTRODUCTION

This research looks at the field of image classification with Convolutional Neural Networks (CNNs), using the CIFAR-10 dataset and the Keras library. Specialized deep neural networks called CNNs are very good at processing pixel data, which makes them perfect for tasks involving visual recognition. Using a variety of performance metrics, this project attempts to demonstrate the CNN's accuracy in image classification by utilizing Keras, a high-level neural networks API that makes the creation of deep learning models on platforms such as TensorFlow easier.

In domains like image and video recognition, recommender systems, and natural language processing, CNNs play a crucial role in the analysis of visual imagery. Making use of the CIFAR-10 dataset, which is well-known for its vastness and variety, offers a reliable testbed for investigating the robustness and effectiveness of the model in object recognition.

Building and training a CNN architecture, optimizing hyperparameters, and closely assessing performance metrics—particularly accuracy and detection rates—are among the main goals. This project aims to investigate the effects of various network architectures and hyperparameters on model performance in addition to achieving high classification accuracy.

We describe our approach to CNN architecture design, dataset preparation, and thorough analysis of experimental outcomes in the sections that follow. A thorough report detailing the CNN model's adaptation to the

CIFAR-10 dataset, performance optimization techniques, and experimentally derived insights is the project's final product.

Convolutional Neural Networks (CNNs)

A particular type of deep neural networks called convolutional neural networks (CNNs) is specifically made for processing and interpreting visual data. As the foundation of many cutting-edge image recognition systems, they have transformed the field of computer vision. Here are some crucial elements and characteristics of CNNs:

Methodology: Architecture

CNNs have several layers that gradually extract higher-level features from raw pixel data, emulating the animal visual cortex's structure.

Convolutional Layers: CNNs' fundamental building components are these layers. To create feature maps, they are composed of a collection of learnable filters, also known as kernels, that are convolved over the input data. Every filter finds particular features or patterns in the input, like textures, shapes, or edges.

ReLU Activation: Typically, a non-linear activation function such as Rectified Linear Unit (ReLU) is applied following each convolutional operation. By setting all negative values in the feature maps to zero, ReLU introduces non-linearity, enabling the network to discover intricate relationships and enhancing the generalization ability of the model.

Pooling Layers: Using pooling layers (like max pooling), each feature map's spatial dimensions can be decreased without sacrificing the most

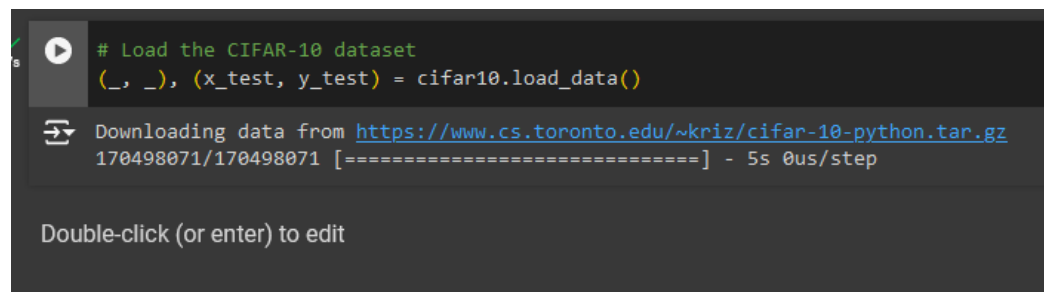
crucial information. By using pooling, one can lessen computational complexity and increase the learned features' resistance to slight distortions and translations in the input data.

Fully Connected Layers: These layers, which come at the end of the CNN architecture, are frequently used to combine the features that the convolutional layers have extracted. The final output layer, which predicts the class probabilities or regression values, is reached through these layers, which link every neuron in one layer to every other neuron in the following layer.

Training: Gradient-based optimization algorithms like Stochastic Gradient Descent (SGD) or its variations are commonly used to train CNNs. Backpropagation aids in network parameter optimization and enhances generalization when paired with strategies like batch normalization and dropout regularization.

Applications: CNNs are extensively utilized in a wide range of computer vision tasks, such as autonomous driving, object detection, facial recognition, medical image analysis, and image classification. They perform exceptionally well in tasks where traditional machine learning techniques falter due to their capacity to automatically extract features from unprocessed data.

Cifar 10



```
# Load the CIFAR-10 dataset
(_, _), (x_test, y_test) = cifar10.load_data()

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [=====] - 5s 0us/step
```

Double-click (or enter) to edit

The CIFAR-10 dataset, which comprises 60,000 32x32 color images divided into 10 classes with 6,000 images per class, is a popular benchmark in computer vision and machine learning. 50,000 training photos and 10,000 test images make up the dataset. Aircraft, cars, trucks, frogs, dogs, cats, deer, horses, and planes are examples of common objects that are represented by the classes. A common use for this dataset is to train and test image classification algorithms because of its diversity and simplicity, where each image is assigned a class.

Keras and TensorFlow

Keras: is an essential high-level neural network application programming interface that was created especially to speed up deep neural network experimentation. Keras, which is written in Python, can easily work with backends like TensorFlow, CNTK, or Theano. Because of its design, which puts an emphasis on extensibility, modularity, and user-friendliness, it is a great option for both researchers and engineers. Keras removes the complexity involved in creating neural networks, allowing programmers to quickly and effectively prototype models.

TensorFlow: An end-to-end open-source platform designed specifically for machine learning, TensorFlow was developed by Google. It offers extensive libraries and tools that enable researchers to create, train, and implement machine learning models in a variety of settings with ease. TensorFlow is an excellent tool for managing workflows in both research and production settings. It can handle large datasets and facilitate the seamless transfer of experimental models into reliable deployments.

The advantages of TensorFlow and Keras are combined in this project. With Keras, building and training models is made easier, enabling rapid testing of various neural network topologies and hyperparameters. TensorFlow offers the stable framework required to effectively scale models and implement them in real-world environments in the interim. This project aims to develop and optimize complex deep learning models efficiently by combining the powerful features of TensorFlow with the simplicity and ease of use of Keras, as especially for tasks involving large datasets such as CIFAR-10.

implementation

we import the cifar10 dataset by using this line of code

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
```

`datasets.load_data().cifar10`: TensorFlow/Keras comes with the CIFAR-10 dataset, which is loaded via this function call.

2 tuples are returned by the function:

(train_images, train_labels): Holds the labels that go with the training images.

(test_images, test_labels): This array holds the labels that go with the testing images.

Next, the variables `train_images`, `train_labels`, `test_images`, and `test_labels` are assigned the images and labels.

Normalization

Code snippet

```
train_images, test_images = train_images / 255.0, test_images / 255.0
```

This code normalizes the image pixel values in the CIFAR-10 dataset from their original range of 0 to 255 to a new range of 0 to 1, normalizing the pixel values. In order to obtain normalized values that are better suited for training neural network models, this is accomplished by dividing each pixel value by 255.0. Through the standardization of the input data, normalization ensures that all pixel values fall within a more manageable, smaller range, which enhances training performance and speed.

Sampling the data

Code snippet

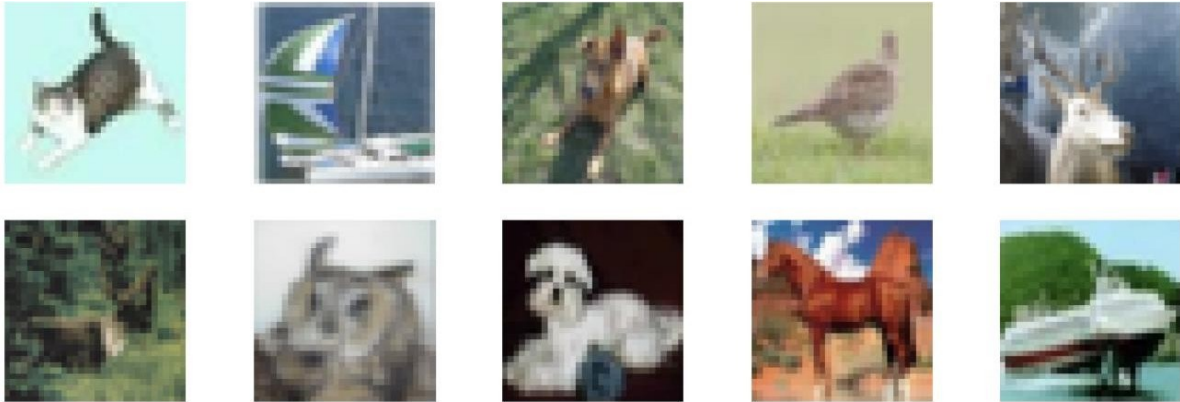
```
label_mapping = {
    0: 'airplane',
    1: 'automobile',
    2: 'bird',
    3: 'cat',
    4: 'deer',
    5: 'dog',
    6: 'frog',
    7: 'horse',
    8: 'ship',
    9: 'truck'
}

plt.figure(figsize=(15, 5))
for i, idx in enumerate(random_indices, 1):
    plt.subplot(2, 5, i)
    plt.imshow(x_test[idx], cmap='gray') # assuming grayscale images
    plt.axis('off')

plt.show()
```

Using the CIFAR-10 test dataset, this code snippet creates a figure that shows a grid of 10 photos (2 rows and 5 columns). Based on the indices in `random_indices`, a random selection is made for each image. Each image is rendered via the `plt.imshow()` function, and `plt.axis('off')` eliminates axis labels and ticks to concentrate only on the visual content. Examining the actual photos used in the predictions or conducting any other study involving the CIFAR-10 dataset is made easier with the help of this visualization. The size of the entire plot can be altered with changes to `plt.figure(figsize=())`, giving the presentation of the images more flexibility.

Result:



Base models

Code snippet

```
model = Sequential([
    Conv2D(32, (3, 3), input_shape=(32, 32, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3)),
    BatchNormalization(),
    Activation('relu'),

    Flatten(),

    Dense(64),
    BatchNormalization(),
    Activation('relu'),

    Dense(10, activation='softmax')
])
```

This code uses the Sequential API provided by Keras to create a convolutional neural network (CNN). To stabilize and quicken the training process, three convolutional layers (Conv2D) are used at the beginning, each followed by batch normalizing (BatchNormalization). To add non-linearity, ReLU activation (Activation('relu')) is applied

following batch normalization. The feature maps' spatial dimensions are decreased through the application of max pooling (MaxPooling2D). The feature maps are prepared for dense layers (Dense) by flattening them (Flatten()) after the convolutional layers. In multi-class classification, two dense layers are utilized, the last of which has a softmax activation (10 classes in this case).

```
def lr_schedule(epoch):  
    lr = 1e-3  
    if epoch > 10:  
        lr *= 0.1  
    return lr
```

The learning rate scheduler is defined by this function (lr_schedule). It modifies the learning rate (lr) by entering the current epoch number. The learning rate is initially set at 1e-3. The learning rate is lowered by a factor of 10 (lr *= 0.1) following epoch 10. Through the dynamic adjustment of the learning rate as training advances, this strategy can aid in stabilizing and improving training.

```
initial_lr = lr_schedule(0)  
model.compile(optimizer=Adam(learning_rate=initial_lr),  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])  
  
lr_scheduler = LearningRateScheduler(lr_schedule)
```

In this case, the model is constructed using the Adam optimizer (Adam(learning_rate=initial_lr)), where lr_schedule(0) (the learning rate for epoch 0) sets the initial learning rate, denoted by initial_lr. For multi-class classification problems, the loss function supplied is categorical cross-entropy (loss='categorical_crossentropy'). During training, accuracy (metrics=['accuracy'])—which represents the proportion of accurately categorized images—is the metric to keep an eye on. Description: Based on the previously defined lr_schedule function, this line initializes a callback (LearningRateScheduler) that will dynamically modify the learning rate during training. By perhaps lowering the learning rate as training goes on, the learning rate scheduler callback can help the model converge more successfully and optimize the training process.

```
history = model.fit(datagen.flow(x_train, y_train, batch_size=32),
                    epochs=5, validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler])
```

Ultimately, the fit approach is employed to train the model. Here, datagen, which was previously set up for data augmentation, is used to create batches of augmented data (x_train and y_train) on the fly using datagen.flow(x_train, y_train, batch_size=32). The model is trained for five epochs (epochs=5), and at the end of each epoch, its performance is assessed using the validation data (x_test, y_test). In order to modify the learning rate during training in accordance with the schedule specified by lr_schedule, the lr_scheduler callback is supplied as a parameter.

```
Epoch 1/5
1563/1563 [=====] - 69s 43ms/step - loss: 1.4552 - accuracy: 0.4761 - val_loss: 1.6949 - val_accuracy: 0.4650 - lr: 0.0010
Epoch 2/5
1563/1563 [=====] - 67s 43ms/step - loss: 1.1681 - accuracy: 0.5851 - val_loss: 1.1732 - val_accuracy: 0.5854 - lr: 0.0010
Epoch 3/5
1563/1563 [=====] - 67s 43ms/step - loss: 1.0558 - accuracy: 0.6300 - val_loss: 1.0801 - val_accuracy: 0.6234 - lr: 0.0010
Epoch 4/5
1563/1563 [=====] - 67s 43ms/step - loss: 0.9899 - accuracy: 0.6517 - val_loss: 1.1283 - val_accuracy: 0.6140 - lr: 0.0010
Epoch 5/5
1563/1563 [=====] - 67s 43ms/step - loss: 0.9399 - accuracy: 0.6702 - val_loss: 1.2234 - val_accuracy: 0.6004 - lr: 0.0010
```

Training epoch progress is shown in the output, which includes measures like training loss and accuracy (loss and accuracy) and validation loss and accuracy (val_loss and val_accuracy). With a starting learning rate of 0.001, the Adam optimizer defines the optimization process that is used to update the model's weights throughout each epoch of iterating through the dataset. A learning rate scheduler (lr_schedule function) controls the learning rate reduction by a factor of 10 following the tenth epoch, ensuring that the model increasingly fine-tunes its parameters as training goes on.

comprehension the model's performance requires a comprehension of the metrics displayed:

Loss and Accuracy: These measures show how well the model fits the data, both on the training and validation sets. Better alignment between expected and actual results is shown by a lower loss, and more accurate classifications are indicated by a greater accuracy.

For example, the training loss (1.4552) and accuracy (0.4761) in the first epoch (Epoch 1) indicate a reasonable performance, indicating that the model's predictions are reasonably in line with the training data. The model performs marginally worse on unseen validation data, according to the validation metrics (val_loss: 1.6949, val_accuracy: 0.4650), which may be the result of overfitting or early learning adjustments. Training accuracy gains (0.6702 by Epoch 5) indicate that the model is learning and becomes more accurate on the training data as training moves forward (Epoch 2 to Epoch 5). But variations in validation loss (val_loss) and accuracy (val_accuracy) over various epochs show that more tweaks are required to improve generalization. The consistent learning rate that was applied during these epochs is indicated by the lr: 0.0010 line. Since changes to learning rate schedules can affect training efficiency and ultimate accuracy, it is important to keep an eye on how this rate influences convergence and model performance.

Performance of this model

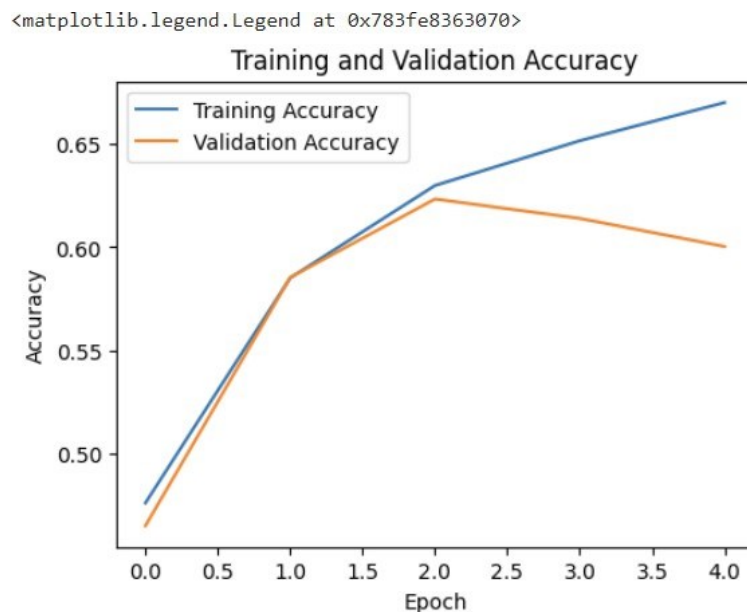
Training and validation accuracy

```
# Plot training and validation accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

The code uses Matplotlib to plot the training and validation accuracy after training. The training accuracy graph (Training Accuracy) shows how well the model fits the training data throughout epochs, with the goal of continuous improvement. This visualization is essential for evaluating the model's performance. In addition, the model's ability to generalize to

previously unseen data is shown by the validation accuracy graph (Validation Accuracy), which aids in the identification of overfitting or underfitting problems. The output graphs help with decision-making regarding model modifications or additional training iterations by offering insights into the model's learning dynamics and generalization outside the training set.

Figure for training and validation accuracy

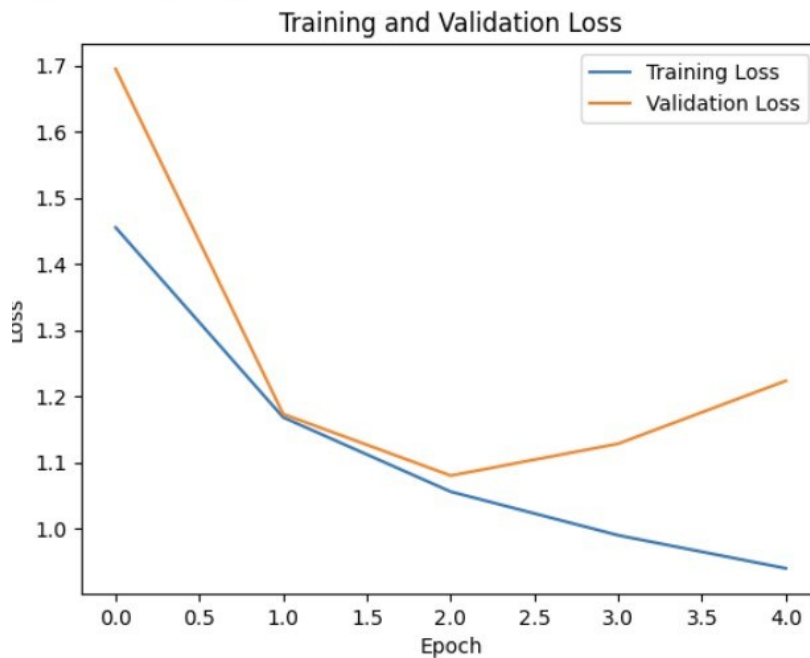


Training and validation loss

```
# Plot training and validation loss
plt.subplot(1, 1, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

Figure for training and validation accuracy

```
matplotlib.legend.Legend at 0x7837e842aec0>
```



Measuring Model Performance with F1 score

The evaluation with `model.evaluate()` is a standard practice in machine learning to assess model performance on unseen test data. It computes metrics such as loss and accuracy, which give a quick snapshot of how well the model predicts the test dataset. Loss indicates the difference between predicted and actual values, while accuracy measures the proportion of correct predictions.

In addition to these standard metrics, calculating the F1 score provides a deeper analysis of the model's performance. Unlike accuracy, which considers only correct predictions, the F1 score considers both precision (the ratio of true positives to all positive predictions) and recall (the ratio of true positives to all actual positives). By averaging these metrics across different classes (weighted by the number of samples), the F1 score offers insights into how well the model performs across various categories, accounting for imbalances in class distribution.

These evaluation metrics are crucial for assessing the model's generalization capability beyond the training set. They help detect potential issues like overfitting (where the model performs well on training data but poorly on unseen data) or underfitting (where the model fails to capture relationships in the data). Insights from these metrics guide improvements in model architecture, hyperparameter tuning, or data preprocessing techniques, aiming to enhance overall performance and reliability in real-world applications.

Code snippet

```
import numpy as np
from sklearn.metrics import f1_score

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=2)
```

In machine learning workflows, this evaluation stage is essential since it gives information about how well the model functions on data that it hasn't encountered during training. While test_accuracy reveals the overall accuracy of the model's predictions, test_loss displays how closely the predictions match the true labels. These metrics aid in evaluating the generalization capacity of the model and offer direction for prospective enhancements, including altering the training procedure, fine-tuning the dataset, or modifying the model's parameters.

Result

```
313/313 - 3s - loss: 1.2234 - accuracy: 0.6004 - 3s/epoch - 8ms/step
```

Code snippet

```
# Calculate F1 score
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)
f1 = f1_score(y_true_classes, y_pred_classes, average='weighted')
```

Combining precision and recall into a single numerical metric, the F1 score evaluates a model's performance in classification tasks. The best possible score is 1, and the range is 0 to 1. Better model performance is indicated by higher F1 scores, which also reflect improved recall (capacity to locate all positive examples) and precision (accuracy of positive predictions).

Result

```
313/313 [=====] - 5s 15ms/step
```

Code snippet

```
print(f'Test accuracy: {test_accuracy:.4f}')
print(f'Test F1 Score: {f1:.4f}')
```

When taken as a whole, these lines offer a succinct overview of the model's performance on the test set, demonstrating both its efficacy and accuracy in terms of the F1 score, which takes precision and recall into account. These metrics are essential for determining how effectively the model balances its predictions across various classes and how well it generalizes to new data.

Result

```
Test accuracy: 0.6004
Test F1 Score: 0.5783
```

Adding additional layer and Introducing Learning schedulerCallback


```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormalization, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import LearningRateScheduler
import matplotlib.pyplot as plt
```

All of these imports together create the framework needed to work with deep learning models, particularly for CIFAR-10 image classification tasks, and to visualize different elements of model evaluation and training. Please don't hesitate to ask questions or ask specific tasks related to creating models or showing findings!

```
] # Load dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

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

```
] # Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)

y_test = to_categorical(y_test, 10)
```

These code fragments help with important CIFAR-10 dataset preparing processes. To create the groundwork for the construction and assessment of the model, the dataset is first loaded and divided into training and testing sets. After that, pixel values are standardized from 0-255 to 0-1, which improves model performance by guaranteeing consistent data scaling. Furthermore, class labels are one-hot encoded, which converts them into a binary matrix format with distinct representations for each class, which is essential for efficient categorical classification tasks. Together, these preparation stages get the CIFAR-10 data ready for reliable model testing and training.

```
datagen = tf.keras.preprocessing.image.ImageDataGenerator(  
    rotation_range=15,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
)  
datagen.fit(x_train)
```

This code applies methods for data augmentation to the training set. With data augmentation, random picture changes (rotations, shifts, flips, etc.) are applied to the training dataset to artificially increase its size and diversity. By exposing the model to more variances of the input data during training, this enhances the resilience and generalization of the model and calculates any internal statistics required in connection with the data augmentation methods listed in datagen. In order to provide enhanced batches of data during model training, this step sets up the datagen object.

Model a

```
model = Sequential([
    Conv2D(32, (3, 3), input_shape=(32, 32, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3)),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3)),
    BatchNormalization(),
    Activation('relu'),

    Flatten(),

    Dense(64),
    BatchNormalization(),
    Activation('relu'),

    Dense(10, activation='softmax')
])
def lr_schedule(epoch):
    lr = 1e-3
    if epoch > 10:
        lr *= 0.1
    return lr

initial_lr = lr_schedule(0)
model.compile(optimizer=Adam(learning_rate=initial_lr),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

lr_scheduler = LearningRateScheduler(lr_schedule)
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=5, validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler])
```

Using TensorFlow's Keras API, this code snippet builds and trains a convolutional neural network (CNN) model for image classification on the CIFAR-10 dataset. Max-pooling layers are used to minimize spatial dimensions after convolutional layers with batch normalization and ReLU activation are included in the model design. Using TensorFlow's ImageDataGenerator, it applies random changes to enrich data and improve model resilience and generalization. Throughout training, the learning rate scheduler modifies the learning rate in an attempt to maximize model performance across epochs. Using the Adam optimizer and categorical crossentropy loss, this code builds a CNN that

aims to achieve high accuracy in classifying CIFAR-10 images while minimizing overfitting through regularization techniques and dynamic learning rate adjustments. It then trains the CNN on augmented data batches and tracks validation performance.

```
Epoch 1/5
782/782 [=====] - 66s 82ms/step - loss: 1.4616 - accuracy: 0.4728 - val_loss: 1.3366 - val_accuracy: 0.5268 - lr: 0.0010
Epoch 2/5
782/782 [=====] - 64s 82ms/step - loss: 1.1655 - accuracy: 0.5858 - val_loss: 1.0653 - val_accuracy: 0.6216 - lr: 0.0010
Epoch 3/5
782/782 [=====] - 67s 86ms/step - loss: 1.0432 - accuracy: 0.6312 - val_loss: 1.3720 - val_accuracy: 0.5384 - lr: 0.0010
Epoch 4/5
782/782 [=====] - 66s 84ms/step - loss: 0.9709 - accuracy: 0.6566 - val_loss: 1.4362 - val_accuracy: 0.5483 - lr: 0.0010
Epoch 5/5
782/782 [=====] - 65s 83ms/step - loss: 0.9154 - accuracy: 0.6793 - val_loss: 0.8569 - val_accuracy: 0.7068 - lr: 0.0010
```

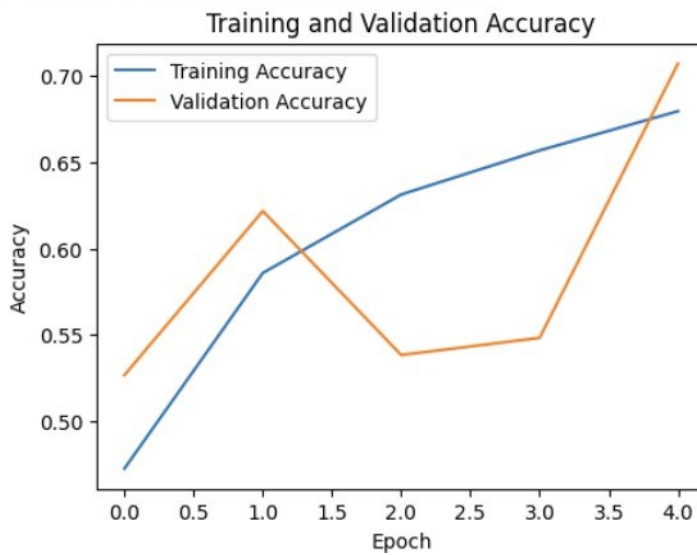
This output helps track training progress and identify possible problems like overfitting or underfitting by revealing how the model is changing (or not) over epochs. In particular on the validation set, the objective is to witness reduced loss and increasing accuracy over epochs, showing that the model is learning to generalize effectively to new data

Performance of this model

```
# Plot training and validation accuracy
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

The training and validation accuracies over several epochs are visually compared in the graph. Both lines should ideally rise and stay near to one another, showing that the model is generalizing and learning well. Significant differences in the two lines could be a sign of underfitting (if both accuracies are low and not improving) or overfitting (if the training accuracy is significantly greater than the validation accuracy).

```
plt.plot(history.history['accuracy'], label='Training Accuracy')
```



```
# Plot training and validation loss
plt.subplot(1, 1, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

The training and validation losses are graphically compared in the plot. When the two curves are almost aligned and both decrease across epochs, the model is probably not overfitting and is training efficiently. Significant divergence or unpredictable behavior between the training and validation losses could point to problems like overfitting or insufficient training.



Model b

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormalization, Activation
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import LearningRateScheduler
import matplotlib.pyplot as plt
```

Dropout: Allows you to randomly drop units during training to prevent overfitting.

EarlyStopping: Monitors a specified metric (like validation loss or accuracy) and stops training when it stops improving, thus preventing overfitting and saving time

```

model = Sequential([
    ... Conv2D(32, (3, 3), input_shape=(32, 32, 3), kernel_regularizer=l2(1e-5)),
    ... BatchNormalization(), Activation('relu'), MaxPooling2D((2, 2)),
    ... Dropout(0.1), ... # Lowered dropout rate

    ... Conv2D(64, (3, 3), kernel_regularizer=l2(1e-5)),
    ... BatchNormalization(), Activation('relu'), MaxPooling2D((2, 2)),
    ... Dropout(0.2), ... # Lowered dropout rate

    ... Conv2D(64, (3, 3), kernel_regularizer=l2(1e-5)),
    ... BatchNormalization(), Activation('relu'),
    ... Dropout(0.3), ... # Lowered dropout rate

    ... Flatten(),
    ... Dense(64, kernel_regularizer=l2(1e-5)),
    ... BatchNormalization(), Activation('relu'),
    ... Dropout(0.4), ... # Lowered dropout rate

    ... Dense(10, activation='softmax')
])

```

Training the model

```

history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=5,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping])

```

```

Epoch 1/5
782/782 [=====] - 71s 87ms/step - loss: 1.7505 - accuracy: 0.3654 - val_loss: 1.4838 - val_accuracy: 0.4614 - lr: 0.0010
Epoch 2/5
782/782 [=====] - 69s 88ms/step - loss: 1.4672 - accuracy: 0.4744 - val_loss: 1.4934 - val_accuracy: 0.4485 - lr: 0.0010
Epoch 3/5
782/782 [=====] - 71s 91ms/step - loss: 1.3525 - accuracy: 0.5200 - val_loss: 1.3101 - val_accuracy: 0.5388 - lr: 0.0010
Epoch 4/5
782/782 [=====] - 70s 90ms/step - loss: 1.2733 - accuracy: 0.5525 - val_loss: 1.0759 - val_accuracy: 0.6186 - lr: 0.0010
Epoch 5/5
782/782 [=====] - 69s 88ms/step - loss: 1.2233 - accuracy: 0.5724 - val_loss: 1.2812 - val_accuracy: 0.5598 - lr: 0.0010

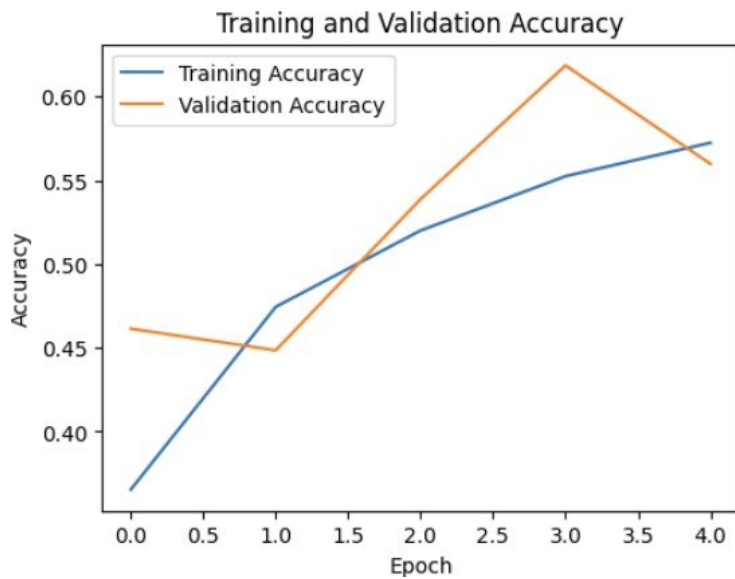
```

Performance of this model

```

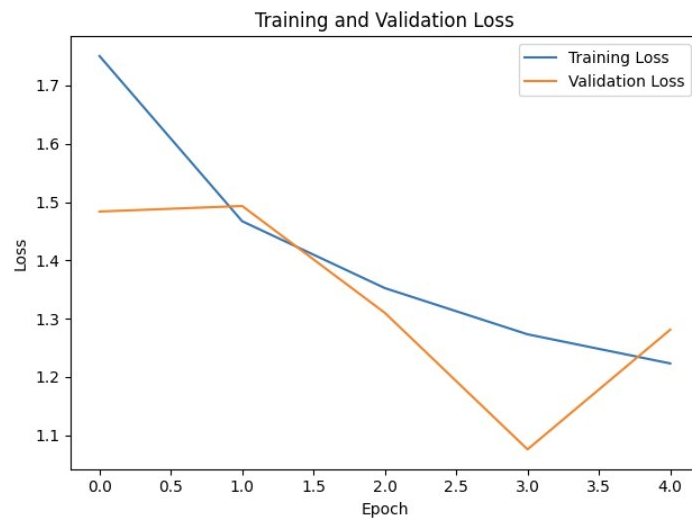
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

```

This figure shows the plot of accuracy

```
plt.subplot(1, 1, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
```



Using the model to predict on Different Number of Batches and Computing Overall Accuracy

```
from tensorflow.keras.models import load_model
import random
import numpy as np
import matplotlib.pyplot as plt
```

The required libraries are imported in these lines. Pre-trained models may be loaded using `load_model`; `random` can be used to generate random numbers; `numpy` can be used to do numerical operations; and `matplotlib.pyplot` can be used to plot images.

```
# Assuming you have the cifar10 dataset loaded
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

The CIFAR-10 dataset, comprising 60,000 32x32 color images divided into 10 classes, with 6,000 images in each class, is loaded by this line. Ten thousand test photos and fifty thousand training images make up the dataset.

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

These lines set the image's pixel values to a normal range of 0 to 1. This is a typical picture data preparation phase.

```
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

These lines use one-hot encoding to transform the class labels—which are integers—into binary class matrices. This is required for the model's training categorical cross-entropy loss function.

```
loaded_model = load_model('my_model.h5')
```

This line loads a pre-trained model from a file named `my_model.h5`.

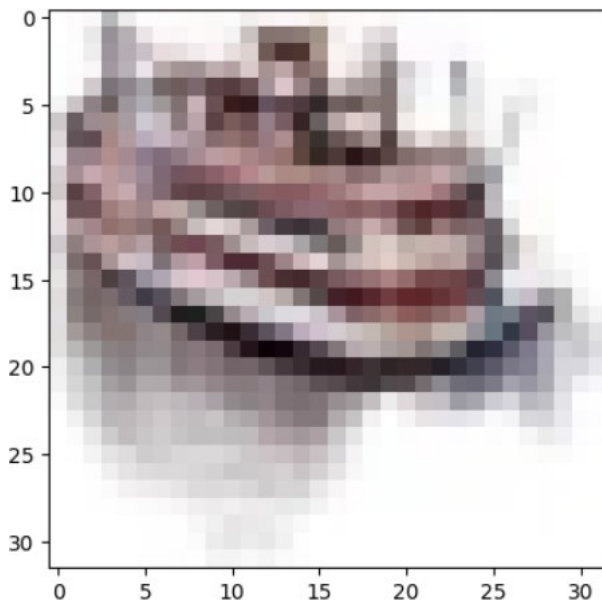
```

# Define label_mapping
label_mapping = {
    0: 'airplane',
    1: 'automobile',
    2: 'bird',
    3: 'cat',
    4: 'deer',
    5: 'dog',
    6: 'frog',
    7: 'horse',
    8: 'ship',
    9: 'truck'
}

```

This dictionary associates the class names in the CIFAR-10 dataset with the numerical class labels (0-9) that correspond to them.

Result



```

print("Predictions for the batch:")
for i, (true_label, pred_label, accuracy) in enumerate(zip(batch_labels, predicted_labels_batch, accuracies)):
    print(f"Image {i + 1}: True label: {label_mapping[np.argmax(true_label)]}, "
          f"Predicted label: {pred_label}, Accuracy: {accuracy}")

```

For every image in the batch, this code snippet offers a comprehensive output of the model's predictions. To evaluate the model's performance on a portion of the CIFAR-10 dataset, it publishes the true label, predicted label, and accuracy for every image. Understanding the model's classification performance and seeing any differences between expected and actual labels is made easier with the help of this output.

Result

```
Predictions for the batch:
Image 1: True label: dog, Predicted label: deer, Accuracy: 0
Image 2: True label: dog, Predicted label: horse, Accuracy: 0
Image 3: True label: deer, Predicted label: truck, Accuracy: 0
Image 4: True label: automobile, Predicted label: automobile, Accuracy: 1
Image 5: True label: dog, Predicted label: deer, Accuracy: 0
```

```
overall_accuracy = sum(accuracies) / len(accuracies)
print(f"\nOverall accuracy for the batch: {overall_accuracy * 100:.2f}%")
```

This code snippet is typically useful in evaluating model performance on a batch of data during model development and testing phases. It provides a quick overview of how accurate the model predictions are on average across multiple samples, helping to assess the model's overall effectiveness in making correct predictions.

Result

```
Overall accuracy for the batch: 20.00%
```

Hyperparameter Tuning:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormalization, Activation
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping
import kerastuner as kt

# Load dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

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

# Convert class vectors to binary class matrices
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Apply data augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
)

# Prefetch data batches for improved data loading efficiency
train_data = datagen.flow(x_train, y_train, batch_size=256)

# Function to build the model with hyperparameters
def build_hypermodel(hp):
    model = Sequential([
        Conv2D(
            filters=hp.Int('conv_1_filters', min_value=32, max_value=64, step=16),
            kernel_size=(3, 3),
            input_shape=(32, 32, 3)
        ),
        BatchNormalization(),
        Activation('relu'),
        MaxPooling2D((2, 2)),

        Conv2D(
            filters=hp.Int('conv_2_filters', min_value=64, max_value=128, step=32),
            kernel_size=(3, 3)
```

```

        input_shape=(32, 32, 3)
    ),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Conv2D(
        filters=hp.Int('conv_2_filters', min_value=64, max_value=128, step=32),
        kernel_size=(3, 3)
    ),
    BatchNormalization(),
    Activation('relu'),
    MaxPooling2D((2, 2)),

    Flatten(),

    Dense(units=hp.Int('dense_units', min_value=64, max_value=128, step=32)),
    BatchNormalization(),
    Activation('relu'),

    Dense(10, activation='softmax')
])

hp_learning_rate = hp.Float('learning_rate', min_value=1e-4, max_value=1e-3, sampling='LOG')
hp_optimizer = hp.Choice('optimizer', values=['adam', 'sgd'])

if hp_optimizer == 'adam':
    optimizer = Adam(learning_rate=hp_learning_rate)
else:
    optimizer = SGD(learning_rate=hp_learning_rate, momentum=0.9)

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

# Initialize the tuner
tuner = kt.RandomSearch(
    build_hypermodel,
    objective='val_accuracy',
    max_trials=10, # Set a fixed number of trials
    directory='random_search',
    project_name='cifar10_tuning'
)

early_stopping = EarlyStopping(monitor='val_accuracy', patience=3)

# Perform hyperparameter tuning
tuner.search(
    x=x_train,

```

```

    y=y_train,
    epochs=5, # Reduced epochs for tuning
    validation_data=(x_test, y_test),
    callbacks=[early_stopping]
)

# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)

# Train the best model with the best hyperparameters
history = best_model.fit(
    train_data,
    epochs=5, # Reduced epochs for training
    validation_data=(x_test, y_test),
    callbacks=[early_stopping]
)

```

These figures show the process to train the best model using the best hyperparameters

This code essentially automates the process of determining a CNN model's optimal hyperparameter configuration on CIFAR-10 with the goal of increasing classification accuracy through methodical model exploration and optimization. In order to maximize model performance and generalize to previously unobserved data, hyperparameter adjustment is essential.

```
Trial 10 Complete [00h 02m 06s]
val_accuracy: 0.6150000095367432

Best val_accuracy So Far: 0.7239999771118164
Total elapsed time: 00h 20m 52s
Epoch 1/5
196/196 [=====] - 29s 142ms/step - loss: 1.4215 - accuracy: 0.4939 - val_loss: 3.5240 - val_accuracy: 0.1056
Epoch 2/5
196/196 [=====] - 28s 142ms/step - loss: 1.1412 - accuracy: 0.5983 - val_loss: 2.1973 - val_accuracy: 0.3362
Epoch 3/5
196/196 [=====] - 27s 140ms/step - loss: 1.0400 - accuracy: 0.6347 - val_loss: 0.9924 - val_accuracy: 0.6489
Epoch 4/5
196/196 [=====] - 28s 141ms/step - loss: 0.9721 - accuracy: 0.6596 - val_loss: 1.2179 - val_accuracy: 0.6052
Epoch 5/5
196/196 [=====] - 27s 137ms/step - loss: 0.9258 - accuracy: 0.6783 - val_loss: 0.9451 - val_accuracy: 0.6729
```

Hyperparameter Tuning Summary: In a total of about 20 minutes and 52 seconds, the Keras Tuner (kt.RandomSearch) finished 10 trials (Trial 10 Complete). In the process, it discovered that the model had produced the best validation accuracy (Best val_accuracy So Far), which was 0.724 (72.4%).

chronologically Training Details: The best model found throughout the hyperparameter tuning process is shown in the following lines as it progresses through training from Epoch 1/5 to Epoch 5/5. Training (loss and accuracy) and validation (val_loss and val_accuracy) metrics are included in each epoch.

Epoch 1: The model's initial values for accuracy and loss on the training set are 1.4215 and 49.39%, respectively. Higher loss and lower accuracy on the validation set (val_loss and val_accuracy) suggest some overfitting or early model change.

Epoch 2: There is a little improvement in the model, indicating learning progress, with less loss and greater accuracy on both training and validation sets.

Epoch 3: The model is learning to generalize better, as seen by further gains in accuracy and loss across both datasets.

Epoch 4: The model demonstrates stability in training by maintaining its performance with constant measurements.

Epoch 5: The last epoch keeps up steady performance measures, indicating that the model can continue to maintain minimal loss and accuracy.

Validation Accuracy: After tuning and training, the model's performance on unseen test data is indicated by the validation accuracy (`val_accuracy`), which reaches 67.29% by the end of the training.

Together, these results highlight how a convolutional neural network may be optimized iteratively for image classification using the CIFAR-10 dataset. They also highlight how model performance changes over different epochs and how fine-tuning hyperparameters can improve accuracy.

```

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
import kerastuner as kt
from tensorflow.keras import layers
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, BatchNormalization, Activation, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import LearningRateScheduler, EarlyStopping
from tensorflow.keras.regularizers import l2
import matplotlib.pyplot as plt
import kerastuner as kt
from tensorflow.keras.optimizers import Adam, SGD, Nadam

```

```

# Load dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

```

```

# Data augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
)
datagen.fit(x_train)

```

```

# Hyperparameters from the image
conv_1_filters = 64
conv_2_filters = 128
conv_3_filters = 96
dense_units = 128
learning_rate = 0.0023502
optimizer = 'nadam'

```

Extra Imports: TensorFlow's Keras optimizer Nadam is one among the new imports included in the code.

Data Augmentation: Using `tf.keras.preprocessing.image` for data augmentation. The training data (`x_train`) is subjected to the `ImageDataGenerator`. By artificially enlarging the training dataset by random transformations such as rotation, shifting, shearing, zooming, and flipping, this strategy improves the generalization of the model.

Hyperparameters: The code defines explicit hyperparameters directly.

Convolutional layer filter count is indicated by the terms `conv_1_filters`, `conv_2_filters`, and `conv_3_filters`.

dense_units: The quantity of units within the completely connected, dense layer.

learning_rate: The optimizer's learning rate.

optimizer: Select the optimizer (in this case, "nadam").

Objective of the Code: This code's main goals are to set up the environment for convolutional neural network (CNN) construction and training using TensorFlow and Keras, prepare the CIFAR-10 dataset for model training, apply data augmentation to increase model robustness, and define specific hyperparameters for the model architecture and optimizer. This setting is necessary to optimize the performance of various model configurations on the CIFAR-10 dataset, which comprises 60,000 32x32 color images divided into 10 classes.

Building models And Performance Evaluation

```
# Build the sequential model
model = Sequential([
    Conv2D(conv_1_filters, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(conv_2_filters, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(conv_3_filters, (3, 3), activation='relu'),
    Flatten(),
    Dense(dense_units, activation='relu'),
    Dense(10, activation='softmax') # Assuming 10 classes as in CIFAR-10
])
```

This code defines a sequential CNN model using Keras' Sequential API. It consists of three convolutional layers followed by max-pooling layers to reduce spatial dimensions, a flattening layer to convert 2D features into a 1D vector, and two dense layers for classification. The ReLU activation function is used for convolutional and dense layers, while softmax is used for multi-class classification

```
# Compile the model with the Nadam optimizer and the given learning rate
model.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

This code specifies the loss function (categorical_crossentropy for multi-class classification), optimizer (Nadam with a defined learning rate), and evaluation metrics (accuracy) to configure the model for training.

```
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

Early stopping is configured to track validation loss (val_loss). If, after five epochs (patiently), the validation loss does not improve, the training will end. When training concludes, the optimal model weights will be reinstated (restore_best_weights=True).

```
# Train the model
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=10,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping])

|
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    epochs=40,
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping])
```

Trains the model with a batch size of 64 using data augmentation (datagen.flow) on the training data (x_train, y_train). Up to 40 epochs of training are allowed, and validation data (x_test, y_test) are supplied. Training can be stopped early if necessary, and the learning rate can be dynamically adjusted via callbacks (lr_scheduler and early_stopping).

```
model.save('my_model.h5')
```

Saves the trained model to a file (my_model.h5) for future use or deployment.

Overview

With dynamic learning rate modification and early stopping, the code seeks to improve the CNN model architecture to categorize images from the CIFAR-10 dataset. The best-performing model is then saved for possible use in real-world applications. This arrangement makes it easier to experiment with various model settings and hyperparameters to get the best results on picture classification jobs.

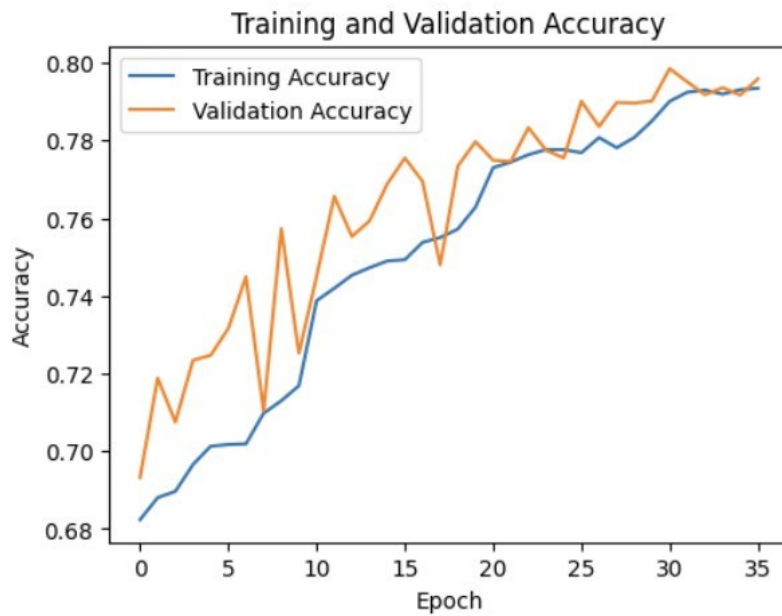
```

epoch 28/40
782/782 [=====] - 31s 40ms/step - loss: 0.6323 - accuracy: 0.7781 - val_loss: 0.6330 - val_accuracy: 0.7898 - lr: 5.8755e-04
Epoch 29/40
782/782 [=====] - 32s 40ms/step - loss: 0.6257 - accuracy: 0.7808 - val_loss: 0.6402 - val_accuracy: 0.7896 - lr: 5.8755e-04
Epoch 30/40
782/782 [=====] - 31s 40ms/step - loss: 0.6206 - accuracy: 0.7851 - val_loss: 0.6330 - val_accuracy: 0.7902 - lr: 5.8755e-04
Epoch 31/40
782/782 [=====] - 31s 40ms/step - loss: 0.5993 - accuracy: 0.7901 - val_loss: 0.5984 - val_accuracy: 0.7985 - lr: 2.9377e-04
Epoch 32/40
782/782 [=====] - 32s 40ms/step - loss: 0.5964 - accuracy: 0.7924 - val_loss: 0.6140 - val_accuracy: 0.7951 - lr: 2.9377e-04
Epoch 33/40
782/782 [=====] - 31s 40ms/step - loss: 0.5926 - accuracy: 0.7930 - val_loss: 0.6357 - val_accuracy: 0.7918 - lr: 2.9377e-04
Epoch 34/40
782/782 [=====] - 32s 41ms/step - loss: 0.5912 - accuracy: 0.7919 - val_loss: 0.6245 - val_accuracy: 0.7936 - lr: 2.9377e-04
Epoch 35/40
782/782 [=====] - 32s 40ms/step - loss: 0.5912 - accuracy: 0.7931 - val_loss: 0.6299 - val_accuracy: 0.7917 - lr: 2.9377e-04
Epoch 36/40
782/782 [=====] - 32s 40ms/step - loss: 0.5838 - accuracy: 0.7935 - val_loss: 0.6161 - val_accuracy: 0.7959 - lr: 2.9377e-04

```

Convolutional neural network (CNN) model training on CIFAR-10 dataset shows a systematic strategy to enhancing generalization and performance. Since the model is still learning from the training set, its accuracy and loss numbers are initially small. The model gradually increases training and validation accuracy as it iteratively improves its weights and biases over the course of training epochs. A scheduler is used to dynamically alter learning rates, which improves convergence and stability by optimizing the model's reaction to gradients during training. Furthermore, the early stopping strategy prevents overfitting and guarantees stable performance on untested data by ensuring that the model stops training when validation accuracy stops increasing. Lastly, the model's optimal performance is captured in the saved model file (my_model.h5), which makes it suitable for implementation in practical applications or additional assessment and improvement in later studies.

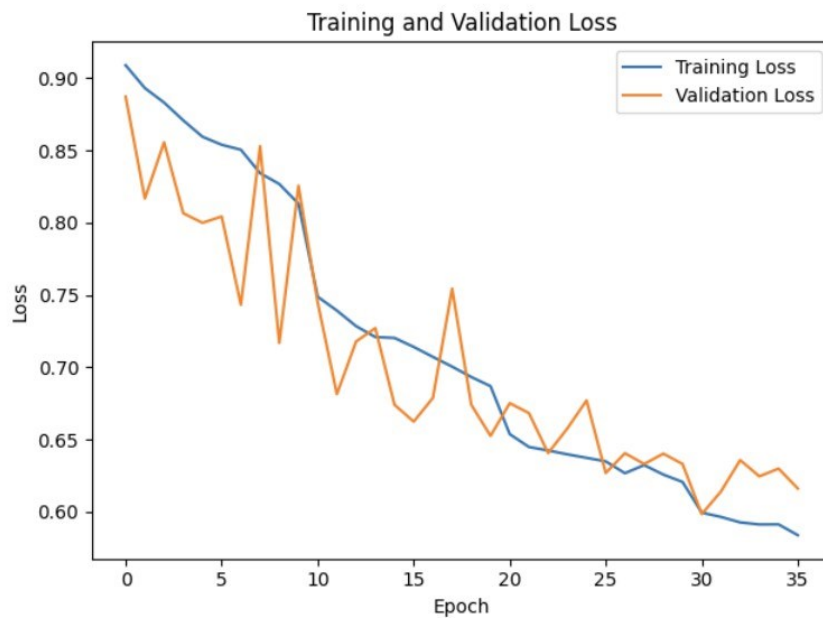
Performance of this model



```
[ ] plt.subplot(1, 1, 1)
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.tight_layout()
    plt.show()
```

This figure shows plotting the Accuracy.

```
plt.subplot(1, 1, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
```



This figure shows plotting the loss.

Training model after hyperparameter tuning

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
from keras.optimizers import Nadam
from keras.utils import to_categorical
from keras.callbacks import LearningRateScheduler, EarlyStopping
import matplotlib.pyplot as plt
from keras.datasets import cifar10

# Load dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Data augmentation
datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
)
datagen.fit(x_train)

# Hyperparameters from the image
conv_1_filters = 64
conv_2_filters = 128
conv_3_filters = 96
dense_units = 128
learning_rate = 0.0023502
optimizer = Nadam()
```

This snippet of code uses the CIFAR-10 dataset in Keras to construct, train, and assess a convolutional neural network (CNN) model for image classification. First, it loads the CIFAR-10 dataset, which has 10,000 test images spread across 10 classes and 50,000 32x32 color training images. It also imports the necessary libraries. In order to prepare the dataset for multi-class classification, the pixel values are scaled to fall between 0 and 1. The labels are then converted to a categorical format.

The application of data augmentation makes use of `tf.keras.preprocessing.image.ImageDataGenerator` adds augmented images produced by rotations, shifts, shears, zooms, and flips to the training set. By subjecting the model to a range of visual variances during training, this strategy improves the model's ability to generalize.

Specific hyperparameters include learning rate (`learning_rate`), units in dense layers (`dense_units`), optimizer (initialized as `Nadam()`), filter sizes for convolutional layers (`conv_1_filters`, `conv_2_filters`, and `conv_3_filters`), and learning rate (`learning_rate`). The CNN model's architecture and optimization settings are specified by these parameters.

Generally, a `Sequential` model would be instantiated and constructed with layers (`Conv2D`, `MaxPooling2D`, `Flatten`, `Dense`) in accordance with the given hyperparameters, albeit the model construction code isn't completely displayed. After that, the model would be constructed using measures like accuracy, the `Nadam` optimizer, and the categorical crossentropy loss function.

`Datagen.flow` is used to feed the augmented data into the model for training, and validation data (`x_test`, `y_test`) is used to track performance. To increase model training efficiency and avoid overfitting, callbacks like `LearningRateScheduler` and `EarlyStopping` dynamically modify learning rates and stop training when validation loss stagnates, respectively.

Results like loss and accuracy measures are usually shown using `matplotlib.pyplot` after training. Ultimately, the trained model (`my_model.h5`) is stored, encapsulating its optimal validation performance and prepared for implementation or additional assessment.

```
# Define different combinations
combinations = [
    {'epochs': 5, 'batch_size': 32},
    {'epochs': 5, 'batch_size': 64},
    {'epochs': 5, 'batch_size': 128},
    {'epochs': 5, 'batch_size': 64},
    {'epochs': 5, 'batch_size': 64}
]
```

This code section defines a list called combinations that describes several epoch and batch size configurations for neural network model training. A distinct set of parameters is specified by each dictionary in the list:

`{'batch_size': 32}, {'epochs': 5}`: This setup instructs the model to train over a period of 5 epochs.

`{'batch_size': 64}, {'epochs': 5}`: In this case, the model is trained for 5 epochs.

`{'batch_size': 128}, {'epochs': 5}`: In this configuration, the model is trained over 5 epochs with a greater batch size of 128.

`{'epochs': 5, 'batch_size': 64}`: This setup replicates training for five epochs with a batch size of 64, much like the second one.

Training runs for 5 epochs with a batch size of 64 in this configuration, which is identical to the second and fourth ones.

These combinations are commonly employed in experimental configurations where the model's performance is assessed across a range of batch sizes and epochs. Through experimentation, one can find the ideal hyperparameter choices that produce the best model performance metrics, such accuracy and loss, enabling the neural network to be trained and generalized to new data with efficacy.


```
plt.figure(figsize=(12, 6))

for i, combo in enumerate(combinations):
    # Build the model
    model = Sequential([
        Conv2D(conv_1_filters, (3, 3), activation='relu', input_shape=(32, 32, 3)),
        MaxPooling2D((2, 2)),
        Conv2D(conv_2_filters, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Conv2D(conv_3_filters, (3, 3), activation='relu'),
        Flatten(),
        Dense(dense_units, activation='relu'),
        Dense(10, activation='softmax')
    ])

```

 <Figure size 1200x600 with 0 Axes>

```
[ ] # Compile the model
model.compile(optimizer=Nadam(learning_rate=learning_rate),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

```

```
[ ] # Set up callbacks
lr_scheduler = LearningRateScheduler(lambda epoch: learning_rate * (0.5 ** (epoch // 10)))
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

```

```
[ ] # Train the model
history = model.fit(datagen.flow(x_train, y_train, batch_size=combo['batch_size']),
                    epochs=combo['epochs'],
                    validation_data=(x_test, y_test),
                    callbacks=[lr_scheduler, early_stopping],
                    verbose=0)

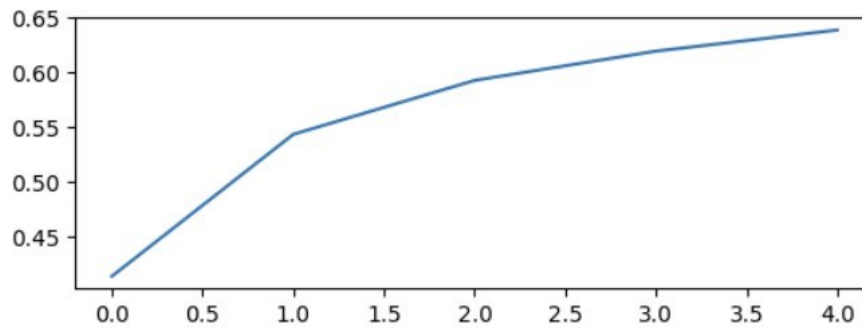
```

```
▶ # Plot accuracy
plt.subplot(2, 1, 1)
plt.plot(history.history['accuracy'], label=f'Combo {i+1}') # Add label here

```

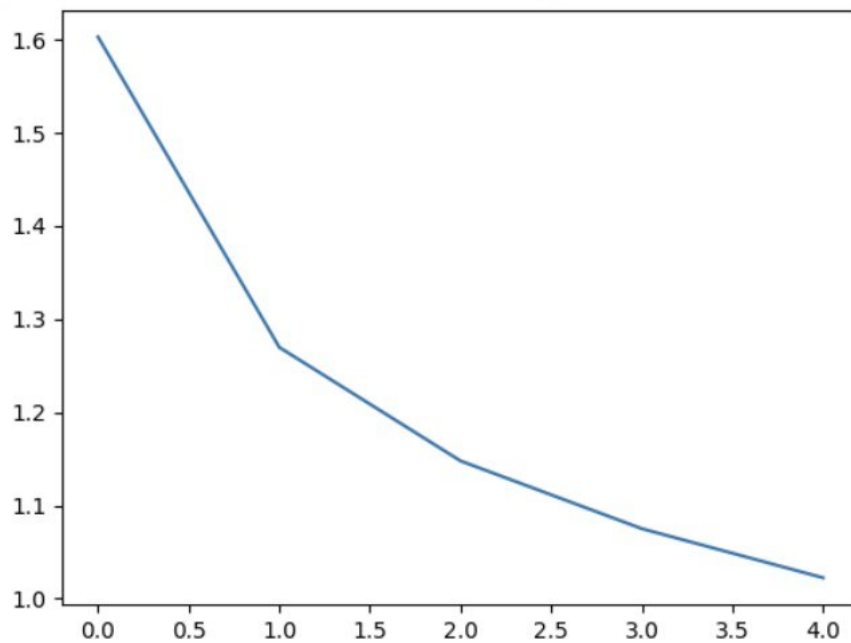
This code is intended to methodically assess the effects of various batch sizes and training epoch combinations on CNN model training accuracy using the CIFAR-10 dataset. It seeks to determine the ideal configuration that produces the greatest model performance for this particular picture classification task by adjusting these parameters and applying the proper callbacks for regularization and optimization.

```
[<matplotlib.lines.Line2D at 0x78bd3cf0040>]
```



```
# Plot loss
plt.subplot(1, 1, 1)
plt.plot(history.history['loss'], label=f'Combo {i+1}')
```

```
[<matplotlib.lines.Line2D at 0x78bd3cf004c0>]
```



Confusion matrix

Code snippet

```
import pandas as pd
import seaborn as sns
from sklearn.metrics import confusion_matrix
```

```
# Define a function to generate and display the confusion matrix as a table
def display_confusion_matrix(true_labels, predicted_labels):
    cm = confusion_matrix(true_labels, predicted_labels)
    cm_df = pd.DataFrame(cm, index=range(10), columns=range(10))
    plt.figure(figsize=(10, 7))
    sns.heatmap(cm_df, annot=True, cmap='Blues', fmt='g')
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.show()
```

```
# Display confusion matrix for the test set
display_confusion_matrix(np.argmax(y_test, axis=1), predicted_classes_test)
```

First, the relevant libraries are imported by the code:

Seaborn is used to create statistical visuals, pandas is used for data manipulation, and confusion_matrix from sklearn.metrics is used to construct the confusion matrix.

The display_confusion_matrix function contains the main body of the code:

Function Parameters: actual and predicted labels for the classification task are represented, respectively, by arrays or lists called true_labels and predicted_labels.

Workflow by Function:

By utilizing confusion_matrix(true_labels, predicted_labels), it calculates the confusion matrix.

To enable visualization, this matrix is converted into a pandas DataFrame (cm_df).

The confusion matrix is then plotted using the heatmap function of Seaborn, with annotations displayed (annot=True) and the matrix color-coded (cmap='Blues').

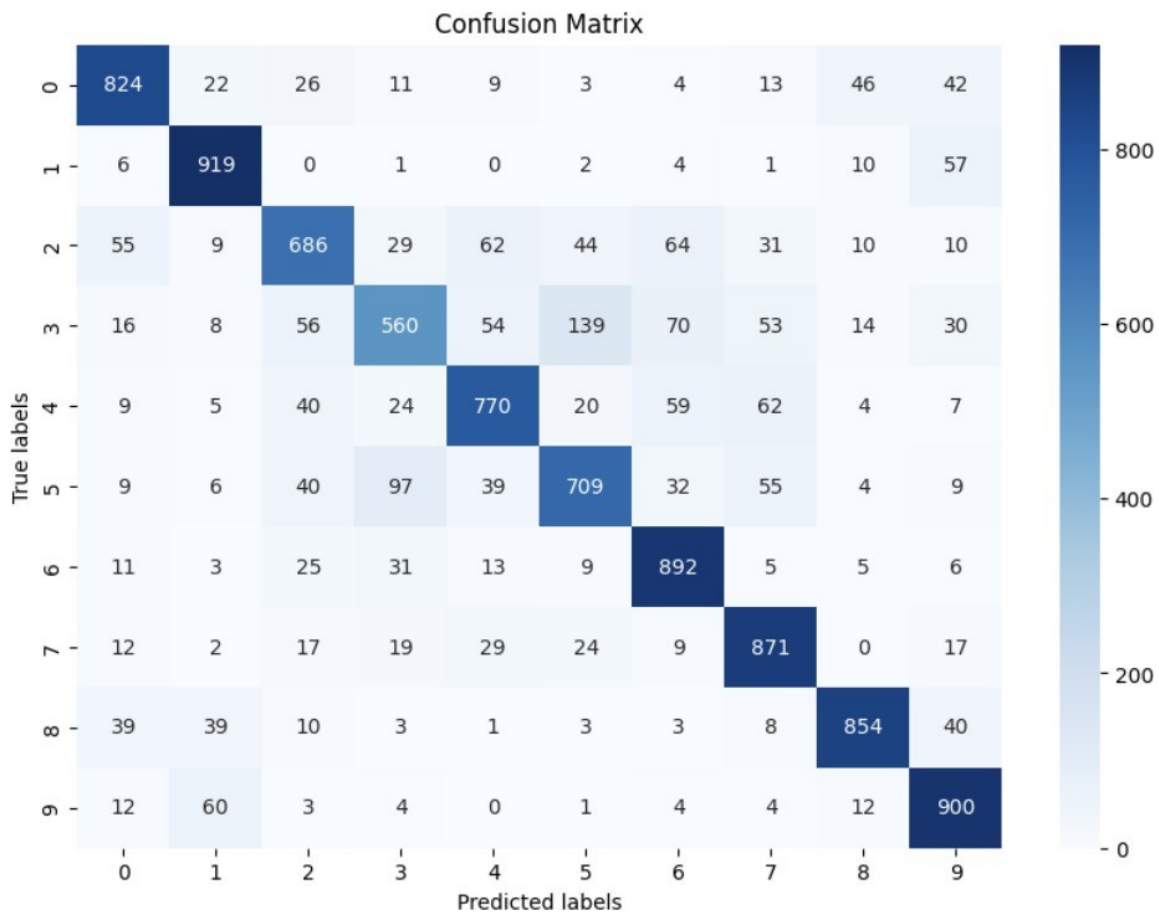
adds labels to the x and y axes (plt.xlabel and plt.ylabel) and a title (plt.title) to personalize the plot.

The function is invoked using:

np.argmax(y_test, axis=1): This returns y_test to category labels for comparison, assuming that it is one-hot encoded.

predicted class labels produced by the model on the test dataset are contained in the predicted classes test variable.

This call creates and presents the confusion matrix, which shows right and wrong predictions made across many classes to give a comprehensive overview of the model's performance.



This figure shows the confusion matrix of the different predictions

Video Demo:

<https://youtu.be/wXUQDArZj-Q>

Conclusion

utilizing the CIFAR-10 dataset, this experiment highlights the delicate balance that needs to be struck between model complexity, training accuracy, and generalization when utilizing CNNs for image classification. Even though our original model had a high training accuracy, it was always plagued by overfitting issues, which emphasizes the vital necessity for careful model training and assessment techniques. By incorporating strategic techniques like dropout layers, data augmentation, and hyperparameter optimization, we greatly enhanced the generalization capacity of our model. These improvements strengthened the model's resilience and reduced overfitting, making it more capable of handling fresh and unforeseen input. Our results highlight the need for a thorough approach to model construction that gives equal weight to both performance indicators and generalization skills. By investigating different regularization techniques, experimenting with sophisticated model architectures, and verifying their effectiveness across larger and more varied datasets, future research could expand upon this foundation. These actions are essential for improving and expanding our models' practical applicability.

References

<https://www.cs.toronto.edu/~kriz/cifar.html>

<https://www.sciencedirect.com/topics/engineering/confusion-matrix#:~:text=A%20confusion%20matrix%20represents%20the,by%20model%20as%20other%20class.>

<https://keras.io/api/callbacks/>

<https://matplotlib.org/stable/index.html>

<https://medium.com/analytics-vidhya/deep-learning-tutorial-with-keras-7a34a1a322cd>

https://www.researchgate.net/figure/Confusion-matrix-for-ten-classes-of-CIFAR-10-dataset_fig4_330552603

<https://www.geeksforgeeks.org/hyperparameter-tuning/>

<https://www.analyticsvidhya.com/blog/2022/02/a-comprehensive-guide-on-hyperparameter-tuning-and-its-techniques/>