Steps :

1. **Collect and preprocess the data:** The first step is to collect a dataset of images and preprocess them to prepare them for training the model. This involves **resizing the images** to a common size, **converting** them to **grayscale** or **RGB** format, and **normalizing the pixel** values to a range between **0** and **1**.

2. **Build a model:** The next step is to **build a model** that can learn to **classify the images** using convolutional neural network (CNN).

3. **Train the model:** Once the model is built, the next step is to train it on the dataset. This involves **feeding the training images** through the model and **adjusting the model's parameters** to minimize the error between **the predicted** and **true labels**.

4. **Evaluate the model:** After the model is trained, the next step is to evaluate its performance on a separate test dataset. This involves **measuring the accuracy** of the model in predicting the correct labels for the test images.

▪ Dataset that i used:

➢ Number of Images:

- The CIFAR-10 dataset consists of a total of 60,000 color images.

- There are 10 classes, and each class has 6,000 images.

- The 60,000 images are divided into 50,000 training images and 10,000 test images.

➢ Image Dimensions:

- Each image is 32x32 pixels in size.

- The images are in color, so each pixel has three color channels (RGB).

➢ Data Batches:

- The dataset is organized into six batches in total.

- Five batches are used for training, each containing 10,000 images.

- One batch is used for testing, containing 10,000 images.

➢ Class Distribution:

- The test batch includes exactly 1,000 randomly-selected images from each of the 10 classes.

- The training batches, on the other hand, contain the remaining images, and the distribution within each batch may vary. However, across all training batches, there are exactly 5,000 images from each class.

In the model building:

- ✓ Input layer: 32x32x3 images
- ✓ Convolutional layer with 32 filters, 3x3 kernel size, and ReLU activation
- ✓ Convolutional layer with 32 filters, 3x3 kernel size, and ReLU activation
- ✓ Max pooling layer with 2x2 pool size
- ✓ Convolutional layer with 64 filters, 3x3 kernel size, and ReLU activation
- ✓ Convolutional layer with 64 filters, 3x3 kernel size, and ReLU activation
- ✓ Max pooling layer with 2x2 pool size

DETAILS

- **`x_train`:** This variable contains the training images. Each image is represented as a 3D array, typically with dimensions (height, width, channels). In the case of CIFAR-10, the images are 32x32 pixels with three color channels (RGB).

- **`y_train`**: This variable contains the corresponding labels for the training images. Each label is an integer representing the class of the corresponding image. CIFAR-10 has 10 classes, such as 'airplane,' 'automobile,' 'bird,' etc.

- **`x_test`:** This variable contains the test images, and like `x_train`, it's a 3D array.

- **`y_test`:** This variable contains the labels for the test images.

- **Normalizing pixel values:**

The purpose of normalizing the pixel values to a range between 0 and 1 is to ensure numerical stability and convergence during training. Dividing each pixel value by 255.0, it's because the original pixel values in images are typically integers ranging from 0 to 255 for each channel in an RGB image. Dividing by 255.0 scales these values to the range [0, 1].

▪ Converting labels into one-hot encoded vectors :

When you're working with a classification problem, you often have labels like 'cat,' 'dog,' or 'car.' However, for a neural network to work well with these labels, it's helpful to convert them into a format called one-hot encoding.

“to_categorical”: is a function that turns your labels into one-hot encoded vectors.

“num_classes=10”: is used because you have 10 classes (like 'airplane,' 'car,' etc. in CIFAR-10).

▪ Convolutional Layers (`Conv2D`):

- Convolutional layers are the core building blocks of Convolutional Neural Networks (CNNs). They learn local patterns and features from the input images.

- **`(3, 3)`:** is the size of the convolutional kernel or filter. It's a 3x3 grid that slides over the input.
- **`activation='relu'`:** means Rectified Linear Unit is used as the activation function after convolution. It introduces non-linearity.
- **`padding='same'`:** ensures that the spatial dimensions of the output feature maps match the input dimensions. It pads zeros if necessary.
- **`input_shape=(32, 32, 3)`:** specifies the input shape. In this case, it's a 32x32 image with three color channels (RGB).

▪ MaxPooling Layers (`MaxPooling2D`):

-MaxPooling layers downsample the spatial dimensions of the input feature maps. They capture the most important information.

- **`(2, 2)`:** is the size of the pooling window.

- **Fully Connected Layers (`Dense`):**
  - **`Flatten()`:** is used to flatten the 3D output to a 1D vector before the fully connected layers.
  - **`Dense(128, activation='relu')`:** represents a fully connected layer with 128 neurons and a Rectified Linear Unit activation function.
  - **`Dense(10, activation='softmax')`:** is the output layer with 10 neurons (for the 10 classes in CIFAR-10) and a softmax activation function. It converts the network's output into probability scores for each class.

Summary:

- The model starts with convolutional layers to extract features.

- MaxPooling layers reduce spatial dimensions.

- Flattening converts the 3D output to 1D.

- Fully connected layers process the flattened features.

- The output layer predicts the probability distribution over the classes.

OUTPUT: This will create a model with the following layers:

Input layer: 32x32x3 images

Convolutional layer with 32 filters, 3x3 kernel size, and ReLU activation

Convolutional layer with 32 filters, 3x3 kernel size, and ReLU activation

Max pooling layer with 2x2 pool size

Convolutional layer with 64 filters, 3x3 kernel size, and ReLU activation

Convolutional layer with 64 filters, 3x3 kernel size, and ReLU activation

Max pooling layer with 2x2 pool size

Training the Model:

- **Compiling the Model:**

- Before training a model, you need to compile it. Compilation involves configuring the model for training.

- **`optimizer='adam'`:** This specifies the optimization algorithm. Adam is a popular optimization algorithm known for its efficiency and effectiveness in training deep neural networks.
- **`loss='categorical_crossentropy'`:** This is the loss function. For multi-class classification problems like CIFAR-10, categorical crossentropy is commonly used. It measures the difference between the predicted probabilities and the true one-hot encoded labels.
- **`metrics=['accuracy']`:** During training, you want to monitor the accuracy of the model on the training and validation datasets.

- **Training the Model:**

- After compilation, you train the model using the training data. This involves feeding the model with input data (`x_train`) and corresponding labels (`y_train`) and adjusting the model's weights based on the computed loss.

- **`epochs=10`:** The number of times the model will be trained on the entire training dataset. One epoch is a complete pass through the entire dataset.
- **`batch_size=32`:** The number of samples used in each update of the model's weights. Using mini-batches instead of the entire dataset at once helps with computational efficiency.
- **`validation_data=(x_test, y_test)`:** The validation dataset is used to evaluate the model's performance on data it hasn't seen during training. It helps to monitor for overfitting.

- **Training Loss:**
  - **Purpose:** It measures how well the model is performing on the training data.

- **Interpretation:** As training progresses, you want the training loss to decrease. A decreasing training loss indicates that the model is learning to fit the training data better.
- **Caution:** If the training loss continues to decrease but the validation loss starts increasing, it might be a sign of overfitting.

- **Validation Loss:**
  - **Purpose:** It measures how well the model is generalizing to new, unseen data (the validation set).
  - **Interpretation:** A decreasing validation loss indicates that the model is generalizing well. However, an increasing validation loss may suggest overfitting or that the model is not learning useful patterns.
  - **Caution:** Large differences between training and validation losses can be a sign of overfitting.
- **Training Accuracy:**
  - **Purpose:** It measures the accuracy of the model on the training data.
  - **Interpretation:** You want the training accuracy to increase over time as the model learns. It represents the proportion of correctly classified examples in the training set.
  - **Caution:** High training accuracy alone does not guarantee good generalization. Always check the validation accuracy.
- **Validation Accuracy:**
  - **Purpose:** It measures the accuracy of the model on the validation data.
  - **Interpretation:** An increasing validation accuracy indicates good generalization. It represents how well the model is expected to perform on new, unseen data.
  - **Caution:** If the validation accuracy is significantly lower than the training accuracy, it might be a sign of overfitting.
- **What to Monitor:**

- **Overfitting:**

- Keep an eye on the validation loss and accuracy. If the training loss continues to decrease while the validation loss starts increasing, the model may be overfitting the training data.

- **Training Progress:**

- Monitor the training loss and accuracy to ensure they are improving over epochs. However, don't rely solely on the training metrics; validation metrics are crucial for assessing generalization.

- **Plotting Accuracy:**
  - **`history.history['accuracy']`:** This retrieves the training accuracy values stored in the `history` object during the training process.
  - **`history.history['val_accuracy']`:** This retrieves the validation accuracy values stored in the `history` object during the training process.
  - **`plt.plot()`:** This function is used to create line plots. Here, it's used to plot the training accuracy (`'accuracy'`) and validation accuracy (`'val_accuracy'`) over epochs.
  - **`plt.xlabel('Epoch')` and `plt.ylabel('Accuracy')`:** These set the labels for the x-axis (epochs) and y-axis (accuracy).
  - **`plt.legend()`:** This displays a legend to distinguish between the training and validation accuracy curves.
  - **`plt.show()`:** This displays the plot.

- **Plotting Loss:**
  - **`history.history['loss']`:** This retrieves the training loss values stored in the `history` object during the training process.

- **`history.history['val_loss']`:** This retrieves the validation loss values stored in the `history` object during the training process.
- **`plt.plot()`:** Similar to the accuracy plot, this function is used to create line plots for training loss (`'loss'`) and validation loss (`'val_loss'`).
- **`plt.xlabel('Epoch')` and `plt.ylabel('Loss')`:** These set the labels for the x-axis (epochs) and y-axis (loss).
- **`plt.legend()`:** This displays a legend to distinguish between the training and validation loss curves.
- **`plt.show()`:** This displays the plot.

Purpose:

These plots provide a visual representation of how your model is learning and generalizing. Here's what you can infer from these plots:

- **Accuracy Plots:**

- Increasing accuracy over epochs suggests that your model is learning well on the training data.

- A large gap between training and validation accuracy might indicate overfitting.

- **Loss Plots:**

- Decreasing loss over epochs indicates that your model is improving.

- A large gap between training and validation loss might indicate overfitting.

Making Predictions:

- **`model.predict(x_test)`:** This line uses the trained model (`model`) to make predictions on the test dataset (`x_test`).The `predict` function returns

the predicted probabilities for each class for each input image in the test set. Each row in `predictions` corresponds to an image, and each column corresponds to a class.

- **np.argmax(predictions, axis=1)`:** This line uses NumPy's `argmax` function to find the index of the maximum value along axis 1 (across the classes) for each prediction.
- **The result, `predicted_classes`:** is an array containing the predicted class indices for each input image in the test set.Now, `predicted_classes` contains the predicted class indices for each image in the test set.

After obtaining `predicted_classes`, you can compare these predictions to the true labels (`y_test`) to evaluate the model's performance or use them for other downstream tasks.

- We can then compare the predicted classes with the true labels to evaluate the model's performance:

To evaluate the accuracy of the model on the test set:

- **`np.argmax(y_test, axis=1)`:** This line uses NumPy's `argmax` function to find the index of the maximum value along axis 1 (across the classes) for each true label in the test set (`y_test`).The result is an array containing the true class indices for each image in the test set.
- **`accuracy_score(np.argmax(y_test, axis=1), predicted_classes)`:** This line calculates the accuracy of the model by comparing the true class indices (`np.argmax(y_test, axis=1)`) with the predicted class indices (`predicted_classes`).The`accuracy_score` function computes the accuracy as the proportion of correctly predicted instances.
- **`print('Test accuracy:', accuracy)`:** This line prints the computed test accuracy to the console.

- Interpretation:

  - If the accuracy is **0.8**, it means that the model correctly predicted the class for 80% of the images in the test set.

  - **A higher accuracy score** indicates better performance, while a lower accuracy may suggest issues with the model.

  Using accuracy as an evaluation metric is common, especially in classification tasks. However, it's essential to consider other metrics and explore the confusion matrix, precision, recall, and F1 score, especially in imbalanced datasets or when different types of errors have different consequences.

  In summary, the provided code snippet assesses the accuracy of the model on the test set and prints the result.