Name: Subrato Tapaswi	**Class/Roll No.: D16AD/60**	**Grade:**

Title of Experiment: Design and implement a CNN model for image classification

Objective of Experiment: The objective of this experiment is to design and implement a Convolutional Neural Network (CNN) model for image classification. This includes:

1. Building a CNN architecture capable of learning and recognizing features in images.
2. Training the CNN on a labeled dataset to classify images into predefined categories or classes.
3. Evaluating the model's performance using relevant metrics, such as accuracy, precision, and recall.
4. Demonstrating the practical application of deep learning in image recognition tasks and potentially serving as a foundation for more complex computer vision applications.

Description / Theory:

Convolutional Layers: These layers use learnable filters (kernels) to convolve over the input image, detecting various features like edges, textures, and patterns. Convolutional operations enable the network to capture spatial hierarchies in the data.

Activation Functions: After each convolutional layer, activation functions like ReLU (Rectified Linear Unit) introduce non-linearity, helping the network learn complex relationships within the data.

Pooling Layers: Pooling layers (e.g., MaxPooling, AveragePooling) reduce the spatial dimensions of the feature maps, making the model computationally efficient while preserving critical information.

Fully Connected Layers: After feature extraction, fully connected layers are employed to combine learned features and make final classifications. The output layer typically employs softmax activation for multi-class classification.

Loss Function: For image classification, the choice of loss function depends on the task; commonly, categorical cross-entropy measures the difference between predicted class probabilities and true labels.

Training: The model is trained using a labeled dataset through an optimization process. Gradient descent-based optimizers, such as Adam or SGD, adjust the model's weights to minimize the loss function.

Data Augmentation: To improve model generalization, data augmentation techniques like rotation, scaling, and flipping can be applied to generate variations of the training data.

Regularization: Techniques like dropout and batch normalization can be used to prevent overfitting, improving the model's ability to generalize to new, unseen data.

Hyperparameter Tuning: Adjusting hyperparameters such as learning rate, batch size, and the number of filters and layers is essential for optimizing model performance.

Evaluation Metrics: Metrics like accuracy, precision, recall, F1-score, and confusion matrices are used to evaluate the model's performance on a validation or test dataset.

Transfer Learning: Leveraging pre-trained CNN architectures (e.g., VGG, ResNet, Inception) and fine-tuning them for specific tasks can significantly reduce training time and data requirements.

Optimization Techniques: Advanced optimization techniques, such as learning rate scheduling and early stopping, can enhance training efficiency and convergence.

By understanding these theoretical components, one can design and implement an effective CNN model for image classification tasks, tailoring the architecture and techniques to suit the specific problem and dataset.

Algorithm

1. Import the necessary libraries, including TensorFlow/Keras.
2. Load and preprocess your dataset. Replace load_and_preprocess_data() with your data loading and preprocessing code.
3. Create a Sequential model to build the CNN architecture.
4. Add convolutional layers with ReLU activation and max-pooling to extract features.
5. Flatten the feature maps to prepare for fully connected layers.
6. Add fully connected layers for classification.
7. Compile the model with an optimizer, loss function, and evaluation metrics.
8. Train the model on the training data.
9. Evaluate the model's accuracy on the test data.

Program:

In [2]:
```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

In [3]:
```
X_train = X_train.reshape(X_train.shape[0], 32, 32, 3)
X_test = X_test.reshape(X_test.shape[0], 32, 32, 3)
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
```

In [4]:
```
X_train /= 255
X_test /= 255
```

In [5]:
```
n_classes = 10
print("Shape before one-hot encoding: ", y_train.shape)
Y_train = np_utils.to_categorical(y_train, n_classes)
Y_test = np_utils.to_categorical(y_test, n_classes)
print("Shape after one-hot encoding: ", Y_train.shape)
```

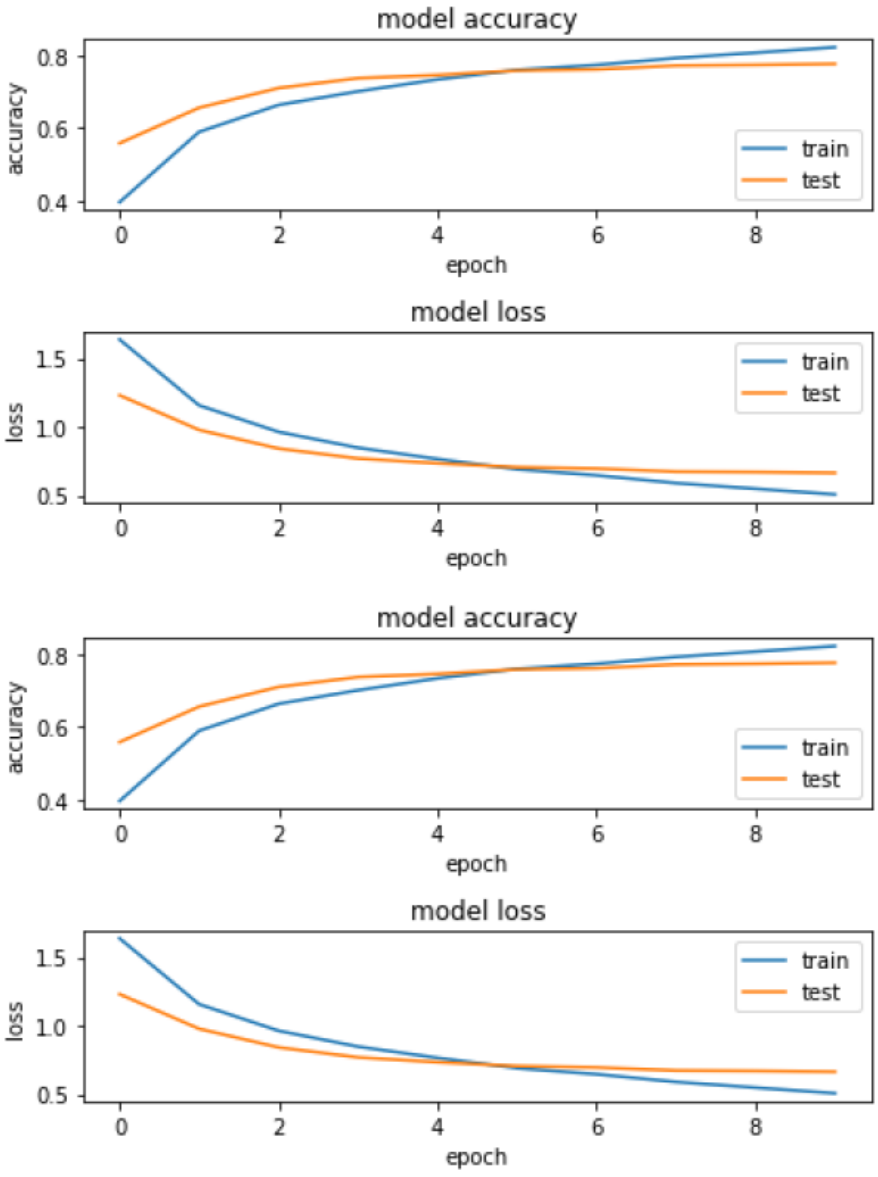
```
Shape before one-hot encoding:  (50000, 1)
Shape after one-hot encoding:  (50000, 10)
```

In [8]:
```
loss, accuracy = model.evaluate(X_test, Y_test, verbose=0)

print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
Test Loss: 0.6650
Test Accuracy: 77.61%
```

Out[9]:



Results and Discussions: The training and validation accuracy plots imply that the model has potential for improvement, possibly through extended training or more sophisticated architectures. Exploring hyperparameter tuning and data augmentation techniques could also enhance the model's capabilities. In essence, this experiment highlights the foundational aspects of CNN-based image classification while highlighting opportunities for optimization and further experimentation.