IMAGE CLASSIFICATION USING CONVOLUTIONAL NEURAL NETWORKS

For the task in hand i.e. to use the training set of images to develop a deep learning model, Convolutional Neural Networks (CNNs) are most suitable. CNNs are ideal for image classification in the project due to their ability to learn hierarchical features from raw pixel data, exploiting spatial locality and sharing weights to reduce overfitting. By employing convolutional layers, CNNs automatically detect local patterns, while pooling layers downsample feature maps, enhancing model robustness. This end-to-end learning approach eliminates manual feature engineering, making CNNs efficient for processing large image datasets and generalizing well to unseen data.

General Framework for building and testing a model using Convolutional Neural Networks (CNNs) are:

1. **Data Loading and Preprocessing:**

   Data loading and preprocessing for this project involve several important steps facilitated by various libraries. Firstly, essential modules like "os" and "zipfile" are utilized to manage file operations, while "numpy" is employed for numerical computations. The "PIL" library aids in image processing, and "sklearn.preprocessing" assists in encoding labels. Additionally, "tensorflow.keras.preprocessing.image" provides the ImageDataGenerator, a vital tool for generating augmented image data.

   The directory paths for the training and validation datasets are stored in variables named train_dir and validation_dir, respectively. These directories are structured to contain subdirectories corresponding to different classes or categories of images.

   ImageDataGenerator is then used to generate batches of image data with real-time data augmentation. This process involves rescaling the pixel values of the images to the range [0, 1]. The flow_from_directory method further aids in generating batches of augmented data from image files in the specified directories. This method automatically labels the images based on the subdirectory structure.

   Moreover, image preprocessing is conducted using generators in Python. The images are resized to a specific size of 224x224 pixels, and a batch size of 32 is set for processing efficiency. Both training and validation data undergo image preprocessing using Python generators. Pixel values are rescaled to a range of 0 to 1 using an ImageDataGenerator object called validation_datagen. The flow_from_directory method is then employed to generate batches of preprocessed validation images, ensuring consistency with the training data. Images are resized, batched, and categorized for multi-class classification tasks, facilitating accurate model evaluation on unseen data.

2. **Model Building:**
   The model building process begins with importing TensorFlow and the necessary Keras components.
   A Sequential model is instantiated, representing a linear stack of layers. Convolutional layers are added to extract features from input images, followed by max-pooling layers to downsample the feature maps and reduce computational complexity. The Flatten layer reshapes the 3D feature maps into a 1D feature vector for input to the dense layers. Dense

layers are then added for classification, with ReLU activation used for hidden layers and softmax activation for the output layer to output class probabilities.
The model is compiled using the Adam optimizer, categorical crossentropy loss function, and accuracy as the evaluation metric. The summary of the model architecture is displayed, providing insights into the layers, output shapes, and total trainable parameters. This step helps in verifying the model structure and ensuring it aligns with the intended design.

Based on the summary of model:

- Model's Complexity: The model consists of three convolutional layers followed by max-pooling layers, flattening layer, and two dense layers. The total number of parameters is 11,169,218, indicating a moderately complex model architecture. The complexity arises from the large number of trainable parameters in the dense layers, especially the first dense layer with 11,075,712 parameters.

- Memory Requirements: The model's total parameters amount to 42.61 MB, which is relatively large. This suggests that the model requires significant memory resources, both during training and inference. The memory requirements may pose challenges, especially when deploying the model on resource-constrained devices or when training multiple models simultaneously.

- Training Efficiency: While the model's architecture is complex, the efficiency of training depends on various factors such as the dataset size, hardware resources, and optimization techniques. With appropriate hardware resources (e.g., GPUs or TPUs) and optimization strategies (e.g., batch normalization, learning rate scheduling), the model can be trained efficiently. However, the large number of parameters may result in longer training times compared to simpler models, necessitating careful monitoring and optimization to ensure training efficiency.

3. **Model Training:**
The model training process involves utilizing the fit() function to train the model. The training data is fed into the model using generators, with the number of epochs set to 10. The model is compiled beforehand using the Adam optimizer, categorical crossentropy loss function, and accuracy as the evaluation metric. Additionally, a CustomPyDataset class is defined to create a custom data generator, facilitating efficient data loading and processing in batches. During training, the model's performance metrics, including accuracy and loss, are recorded for both the training and validation sets after each epoch. The training results illustrate an improvement in accuracy over epochs, indicating the model's learning from the data.

4. **Model Evaluation:**
The model.evaluate() function computes the loss and accuracy metrics of the model on the validation data. The validation loss is approximately 1.44, and the validation accuracy is approximately 0.73.

Here's what these statistics indicate:

Validation Loss (1.44): This represents the average loss incurred by the model on the validation dataset during the evaluation. Loss is a measure of how well the model's predictions match the actual labels. Lower validation loss values indicate better performance, as they suggest that the model's predictions are closer to the true labels.

Validation Accuracy (0.73): This represents the proportion of correctly classified samples in the validation dataset. An accuracy of 0.73 means that the model correctly predicted the class of about 73% of the validation samples. Higher validation accuracy values indicate better performance, as they suggest that the model's predictions are more accurate.

Visualization of model accuracy indicates that as the number of epochs increases, the training accuracy also increases, suggesting that the model is learning from the training data. The validation accuracy, however, varies much more and does not show the same upward trend. Instead, it appears to hover around a lower value with minor fluctuations.
Insight:

High training accuracy with significantly lower validation accuracy might indicate that the model is overfitting to the training data. Overfitting means the model has learned the training data too well, including noise and outliers, and thus may not perform well on unseen data.

Visualisation of model loss indicates that The training loss is decreasing as more epochs are completed, suggesting that the model is learning and improving its performance on the training dataset.
The validation loss initially decreases, indicating that the model's generalization is improving. However, after a certain number of epochs, the validation loss starts to increase, which suggests that the model is beginning to overfit the training data.
Inference:

Overfitting is when a model learns the training data too well, including the noise and details that do not generalize well to new, unseen data. This is typically identified by an increase in validation loss while the training loss continues to decrease.
The epoch at which the validation loss begins to increase is often the point beyond which the model training yields negative returns in terms of generalization performance.
Insight:

The ideal number of epochs for training this model would be where the validation loss is at its minimum before it starts increasing, indicating the best trade-off between underfitting and overfitting.
It may be necessary to implement strategies to combat overfitting, such as early stopping.

Therefore, employing early stopping to fine tune the model:
Early stopping is a technique used during training to prevent overfitting by monitoring a specified metric, in this case, 'val_loss,' on the validation set and stopping the training

process when the metric stops improving for a certain number of epochs defined by the 'patience' parameter. The code initializes an EarlyStopping callback from the TensorFlow library with the criteria of monitoring 'val_loss' and restoring the best weights when training is stopped. This callback is then passed to the model's fit method as part of the callbacks list, enabling early stopping functionality during training. The training process is executed for a maximum of 100 epochs, with the model being trained on the data provided by the 'train_generator' and evaluated on the 'validation_generator.'

Inference:

- The model achieved high training accuracy (around 99%) and low training loss (around 0.007) by the end of training, indicating that it learned the training data well.
- The validation accuracy and loss fluctuated across epochs but did not show significant improvement or degradation. This suggests that the model's performance on unseen data remained relatively stable.

Evaluating the model with early stopping, we see that the model accuracy graph illustrates that the training accuracy remains consistently high, nearing 100%, throughout the training process, indicating the model's strong performance on the training data. In contrast, the validation accuracy starts at a lower point, around 70%.

The model loss graph reveals that the model achieves a very low training loss, indicating its effective performance on the training data. The subsequent evaluation of the model on the validation data confirms these observations, with a validation loss of 1.307 and a validation accuracy of approximately 72.9% which is an improvement from last observation.

5. **Testing the model:**

In the testing phase, the model is evaluated on a specific test image. The image is loaded using OpenCV and displayed using matplotlib, showing its dimensions to be 262x350 pixels with 3 color channels. The image is then resized to 224x224 pixels to match the input size expected by the model. The image is reshaped to have the dimensions (1, 3, 224, 224) to fit the model's input shape requirements. Subsequently, the model predicts the output for the test image, generating a multidimensional array of predictions for different classes. In this case, the model outputs probabilities for the two classes represented in the array. The final step involves converting the pixel-wise predictions to a binary prediction based on majority voting. The binary prediction is determined by checking if the mean of the predictions is

greater than 0.5, resulting in a binary prediction of 0. This process demonstrates how the model processes an input image, generates predictions, and converts them into a binary classification output based on a threshold.
The binary prediction is 0, which means that based on the model's output, the majority voting suggests that the image belongs to class 0 i.e roads which require maintenance and would not be eligible for any payment which is the correct prediction.