Progress Report: Image Classification Using CNN on the Oasis Dataset

The aim of this paper is to build a convolutional neural network (CNN) that can classify images from the "Oasis" dataset into appropriate categories. This includes handling data ingestion, exploratory data analysis (EDA), preprocessing, model building, and performance evaluation.

1. Dataset Handling and Setup

- The `kaggle.json` file was uploaded, and the Kaggle API was configured successfully within the Google Colab environment.
- The dataset `imagesoasis` was downloaded using the Kaggle CLI and extracted to the working directory.
- Files were structured in class-wise directories under `/content/oasis_dataset/Data`.

2. Exploratory Data Analysis (EDA)

- Counted the number of images in each class and displayed them using a bar chart.
- Captured basic statistics about image dimensions:
    - Mean height and width
    - Minimum and maximum sizes
    - Verified that all images have 3 channels (RGB).
- Sampled one image from each class for visualization.
- Analyzed grayscale pixel intensity distribution by sampling 50 images from each class.
- Checked and reported:
    - Number of duplicate images using MD5 hashing.
    - Corrupted images that failed to load.

3. Preprocessing and Augmentation

- Images were resized to `128x128`.
- Applied normalization (`rescale=1./255`) to scale pixel values between 0 and 1.
- Employed data augmentation using `ImageDataGenerator`, including:
    - Rotation
    - Width/Height shift
    - Shear
    - Zoom
    - Horizontal flipping
- Split dataset: 80% training and 20% validation.

Visualization of augmented samples helped verify that transformations were being correctly applied.

4. CNN Model Architecture

A custom CNN model was built using the Keras Sequential API with the following layers:

- **Conv2D (32 filters)** → ReLU → MaxPooling → Dropout (0.3)
- **Conv2D (64 filters)** → ReLU → MaxPooling → Dropout (0.3)
- **Flatten** → **Dense (128 units)** → Dropout (0.5)
- **Output Layer:** Dense with `softmax` activation for multi-class classification.

Model compiled with:

- Optimizer: Adam
- Loss: Categorical Crossentropy
- Metric: Accuracy

5. Training and Evaluation

- Used EarlyStopping with a patience of 3 epochs to avoid overfitting.
- Model trained for 10 epochs.

- After training, predictions were generated on the validation set.
- Evaluation included:
  - Accuracy from training history
  - Weighted **F1 Score** for multi-class classification

The final F1 Score was printed to assess classification performance beyond accuracy.

6. Model Performance Summary

The CNN model was trained with **7,392,836 trainable parameters** over **10 epochs**, and the training showed steady but incremental improvements in both accuracy and loss.

Consistent Training Accuracy: The training accuracy steadily improved from 77.6% to 78.35%, indicating that the model is learning, albeit slowly.

Validation Accuracy Plateau: Validation accuracy hovered around 77% for most epochs, with a slight drop in epoch 7, possibly due to overfitting or noisy data.

Validation Loss Fluctuation: While training loss consistently decreased, the validation loss showed fluctuations. This might suggest slight overfitting starting around epoch 4–5.

Best Epoch: Although there was no dramatic jump, the best balance between train and validation performance was around **epoch 4–6**, where validation accuracy and loss were most stable.

7. Challenges Encountered

One of the main challenges during model training was **overfitting**. Initially, I trained the model without any data augmentation and achieved a high training accuracy of around **90%**, but the **validation accuracy was only 70%**, which clearly indicated overfitting.

To address this, I implemented **data augmentation**, which helped improve generalization but also introduced a new issue: **longer training times**. Since the dataset contains over **80,000 images**, augmenting them significantly slowed down the process. To mitigate this, I switched from CPU to **GPU runtime**, which helped speed things up considerably, though training still remained time-consuming, especially for longer runs.

To further combat overfitting, I also added **Dropout layers** and implemented **EarlyStopping**. These techniques helped prevent the model from over-training and ensured that training would stop once validation performance plateaued.

As a result of these improvements, the final model achieved a **training accuracy of 78.35%** and a **validation accuracy of 76.84%**, which shows much better balance and generalization compared to the initial overfitted model. Overall, the model became more robust and reliable, even if the training process was computationally intensive.