**Full Analysis: Improving MLP Accuracy on CIFAR-10**

To improve the performance of a basic Multilayer Perceptron (MLP) model on the CIFAR-10 dataset, we implemented several enhancements in data preprocessing, model architecture, and training configuration. Although MLPs are generally less suited for image classification compared to Convolutional Neural Networks (CNNs), we maintained the MLP structure and achieved a meaningful increase in test accuracy. Initially, the training accuracy was approximately **0.5012**, and test (evaluation) accuracy was around **0.5070**. After implementing the enhancements described below, the **training accuracy slightly improved to 0.5017**, while the **evaluation accuracy increased to 0.5253**, which indicates improved generalization. Here's a breakdown of all the changes made and how they contributed to the overall performance:

* **Input Normalization**: We normalized the image pixel values from the original range of [0, 255] to [0.0, 1.0]. This step helps stabilize and accelerate the training process by ensuring that input values are in a consistent range, which makes gradient descent more effective.
* **Improved MLP Architecture**: We upgraded the architecture by adding multiple dense layers with varying units (512 → 256 → 128), each followed by **Batch Normalization** and **Dropout layers (rate = 0.3)**. Batch normalization stabilizes learning by normalizing the inputs to each layer, and dropout helps prevent overfitting by randomly dropping units during training.
* **Activation Functions**: We continued using **ReLU activation** for hidden layers, which remains a standard due to its simplicity and effectiveness in deep learning models.
* **Loss Function - Label Smoothing**: We used Categorical Crossentropy with label smoothing (0.1). Label smoothing helps the model to avoid being overconfident about its predictions by softening the target distribution. This encourages better generalization and can lead to improved test accuracy.
* **Optimizer**: We used the **Adam optimizer** with a learning rate of 0.001. Adam is known for its adaptive learning rates, which combine the advantages of both RMSProp and momentum, making it ideal for models that train on noisy data.
* **Callbacks for Better Training**: We implemented three key callbacks:
  + EarlyStopping: Monitors the validation loss and stops training when it stops improving for several epochs. This prevents overfitting.
  + ModelCheckpoint: Saves the best-performing model during training.
  + ReduceLROnPlateau: Reduces the learning rate when the validation loss plateaus, allowing the model to fine-tune its weights more delicately in later stages.
* **Increased Epochs**: We trained for up to **50 epochs**, compared to the typical default of 10–20, allowing the model more time to converge. Since we were using dropout and early stopping, the model was unlikely to overfit excessively despite the long training period. This increased the model's chance to explore and learn better feature representations.

In summary, by introducing these enhancements, the model learned more stable and generalizable features, especially evidenced by the improved evaluation accuracy. While the training accuracy did not change significantly (from 0.5012 to 0.5017), the improved test accuracy (from 0.5070 to 0.5253) is a direct indicator of a **better-trained model that performs more reliably on unseen data**.

**Final Updated Code Cells**

Below are all the code cells that were updated.

**🔹 Cell: Data Normalization**

x\_train = x\_train.astype("float32") / 255.0

x\_test = x\_test.astype("float32") / 255.0

**🔹 Cell: Model Architecture**

input\_layer = layers.Input((32, 32, 3))

x = layers.Flatten()(input\_layer)

x = layers.Dense(512, activation="relu")(x)

x = layers.BatchNormalization()(x)

x = layers.Dropout(0.3)(x)

x = layers.Dense(256, activation="relu")(x)

x = layers.BatchNormalization()(x)

x = layers.Dropout(0.3)(x)

x = layers.Dense(128, activation="relu")(x)

x = layers.BatchNormalization()(x)

x = layers.Dropout(0.3)(x)

output\_layer = layers.Dense(NUM\_CLASSES, activation="softmax")(x)

model = models.Model(input\_layer, output\_layer)

model.summary()

**🔹 Cell: Compile Model**

loss\_fn = tf.keras.losses.CategoricalCrossentropy(label\_smoothing=0.1)

opt = optimizers.Adam(learning\_rate=0.001)

model.compile(loss=loss\_fn, optimizer=opt, metrics=["accuracy"])

**🔹 Cell: Training the Model**

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

callbacks = [

EarlyStopping(patience=7, restore\_best\_weights=True),

ModelCheckpoint("best\_mlp\_model.h5", save\_best\_only=True),

ReduceLROnPlateau(monitor="val\_loss", factor=0.3, patience=3, verbose=1)

]

model.fit(

x\_train, y\_train,

batch\_size=128,

epochs=50,

shuffle=True,

validation\_split=0.1,

callbacks=callbacks

)