**DOG BREED CLASSIFICATION USING TRANSFER LEARNING**

Build a dog breed classifier using transfer learning. This method allows us to use a pre-trained deep learning model and fine-tune it to classify images of different dog breeds.

**Benefits of Transfer Learning in Dog Breed Classification**

* **Faster Training**: Pre-trained models have already learned to identify key image features, saving time on training.
* **Improved Accuracy**: Using a pre-trained model on a similar task helps in achieving better performance than training from scratch.
* **Less Data Requirement**: Transfer learning requires fewer data for fine-tuning, making it ideal for tasks like dog breed classification where labeled data might be limited.

**IMPLEMENTATION:**

**Step 1: Import Necessary Libraries**

* **Pandas:**For data manipulation and preprocessing.
* **Numpy:** For numerical computations and array manipulations.
* **Matplotlib and Seaborn:**For visualizing the dataset and model performance.
* **Scikit-learn (Sklearn):**For data preprocessing, splitting datasets, and model evaluation.
* **OpenCV:** For image processing tasks such as resizing, cropping, and converting images.
* **TensorFlow and Keras:** For building and training the deep learning model using transfer learning.

**Step 2: Loading Dataset for Dog Breed Classification**

The dataset contains **10,000 images** of **120 different dog breeds**. The dataset includes:

* **Training images**: Contains labeled images of dog breeds.
* **Test images**: Unlabeled images used for testing the model.
* **CSV file**: Contains metadata about the images and their corresponding dog breed labels.

**Step 3: Exploratory Data Analysis**

Now that we have the dataset, let’s perform some basic [Exploratory Data Analysis (EDA)](https://www.geeksforgeeks.org/what-is-exploratory-data-analysis/).

**Step 4: Data Preparation**

When working with large datasets in deep learning, memory limitations often prevent loading the entire dataset at once. To efficiently handle data loading and augmentation, tools like TensorFlow’s tf.data.Dataset and Albumentations are used to create optimized input pipelines and apply real-time image augmentations.

First, the dataset is split into training and validation sets, enabling model training on one subset and evaluation on another.

**Step 5: Applying Image Augmentation**

**Step 6: Building the Input Pipeline**

Now, let’s define utility functions to handle image loading, augmentation, and normalization. We will create functions to read images from disk, resize them, normalize the pixel values, and apply augmentations.

Below we have implemented some utility functions which will be used while building the input pipeline.

* **decode\_image:**This function will read the image from the path and resize them to be of the same size along with it will normalize as well. Finally, we will convert the labels into **one\_hot** vectors as well.
* **process\_data:**This is the function that will be used to introduce image augmentation to the image.

Now by using the above function we will be implementing our training data input pipeline and the validation data pipeline.

From here we can confirm that the images have been converted into (128, 128) shapes and batches of 64 images have been formed.

**Step 7: Model Building Using Transfer Learning**

**1. Load Pre-trained InceptionV3 Model**

We first load the **InceptionV3 model** from TensorFlow’s Keras API with the weights pre-trained on ImageNet. The **include\_top=False** argument excludes the fully connected layers at the top of the network, allowing us to customize the final layers for our task.

**2. Inspect the Model’s Depth**

InceptionV3 is a deep network with many layers, which makes it effective in learning complex features from images. Let’s check the number of layers in this pre-trained model.

This deep architecture, consisting of 311 layers, makes it highly efficient at extracting detailed features from images.

**3. Freeze Pre-Trained Layers**

Since the convolutional layers of the InceptionV3 model have already been trained on millions of images, we freeze these layers so that their weights are not updated during our fine-tuning process.

This tells us that the last convolutional layer outputs a 6×6 grid of feature maps with 768 channels.

**5. Define the Custom Model Architecture**

Using the **Keras Functional API**, we can build a custom classification head on top of the pre-trained model. This includes flattening the output, adding fully connected layers, [BatchNormalization](https://www.geeksforgeeks.org/applying-batch-normalization-in-keras-using-batchnormalization-class/" \t "_blank)for stable training, Dropout for regularization, and finally, an output layer with softmax activation for multi-class classification.

**6. Implement Callbacks**

[Callbacks](https://www.geeksforgeeks.org/tf-keras-callbacks-callback-tensorflow-callbacks/) are used to monitor the model’s performance during training. We use the following callbacks:

* **EarlyStopping:**Stops training if validation AUC doesn’t improve for 3 consecutive epochs, preventing overfitting.
* **ReduceLROnPlateau:**Reduces the learning rate when the validation loss plateaus, helping the model converge better.
* **Custom Callback:** Stops training if the validation AUC exceeds 0.99.

**from** **keras.callbacks** **import** EarlyStopping, ReduceLROnPlateau

**Step 8: Train the Model**

We train the model using the**fit()** method with training and validation datasets, a maximum of 50 epochs, and the callbacks defined above.

The output will display the training and validation loss, as well as the AUC score after each epoch. If the validation AUC exceeds 0.99, the training will stop early.

**Step 9: Evaluate the Model**

Once the model is trained, we evaluate its performance on the test dataset. We visualize the training history to observe the model’s learning curve and make sure it has converged effectively.

The training and validation AUC curves are plotted, showing how the model’s performance evolved over time. The test loss and test AUC are displayed, providing insight into how well the model generalizes to unseen data.