Car Brand Classification Using Transfer Learning with ResNet50 and TensorFlow Optimization

This code snippet demonstrates how to perform image classification using the ResNet50 model with TensorFlow and Keras. The ResNet50 model is a pre-trained deep learning model that has been trained on a large dataset of images. We will fine-tune this model on our custom dataset to classify images into multiple classes.

Importing Libraries

We begin by importing the necessary libraries: TensorFlow, Keras, and the ResNet50 model. TensorFlow is a popular deep learning library, while Keras provides a high-level API for building deep learning models. The ResNet50 model is a pre-trained image classification model that we will utilize as the base model.

Constants

Next, we define some constants that will be used throughout the code. `IMG_SIZE` represents the desired size of input images, `BATCH_SIZE` defines the number of images processed together during training, and `EPOCHS` specifies the number of times the model will be trained on the entire dataset.

Mixed Precision

We enable mixed precision, a technique that utilizes both floating-point and half-precision numbers for training deep learning models. By using mixed precision, we can significantly speed up the training process.

Data Paths

We define the paths to the training and validation datasets. The training dataset is the collection of images used to train the model, while the validation dataset is used to evaluate the model's performance after each epoch.

Number of Output Classes

We determine the number of output classes by counting the number of subdirectories in the training dataset. Each subdirectory represents a different class of images that the model will classify.

Data Augmentation

To increase the diversity of the training dataset and improve the model's generalization, we apply data augmentation techniques to the training data. These techniques include random flips, rotations, and zooms. Data augmentation helps the model learn robust features by exposing it to a wider range of variations in the training images.

ResNet50 Base Model

We load the ResNet50 base model without its top layer. By excluding the top layer, we can add our own classification layers on top of the base model to adapt it to our specific classification task. We also freeze the weights of the base model to prevent them from being updated during training.

Model Creation

We create the model by sequentially adding the data augmentation layer, the ResNet50 base model, a flatten layer, and a dense layer with a softmax activation function. The data augmentation layer preprocesses the images by rescaling their values. The base model extracts useful features from the images. The flatten layer converts the output feature maps into a vector, and the dense layer with a softmax activation function classifies the images into the different output classes.

Model Compilation

We compile the model by specifying the Adam optimizer, categorical crossentropy loss function, and accuracy metric. The Adam optimizer is known for its fast convergence and is widely used in deep learning. Categorical crossentropy is a suitable loss function for multi-class classification problems. The accuracy metric measures the performance of the model in terms of correctly classified samples.

Data Loading

We load the training and validation datasets using the Keras image dataset API. This API provides a convenient way to load image datasets into Keras models. The datasets are split into training and validation subsets, and the images are resized to the desired `IMG_SIZE`. We also set the `label_mode` to 'categorical' to represent the labels as one-hot encoded vectors.

Callbacks

Two callbacks are defined: early stopping and model checkpoint. Early stopping monitors the validation loss and stops training if it doesn't improve for a certain number of epochs,preventing overfitting. Model checkpoint saves the best model weights based on the validation loss, allowing us to retrieve the best-performing model.

Model Training

We train the model using the `fit()` method, which trains the model on the training dataset and evaluates it on the validation dataset after each epoch. The training process is controlled by the specified number of epochs and the defined callbacks.

Plotting

Finally, we plot the training and validation loss, as well as the training and validation accuracy, using Matplotlib. These plots help visualize the model's training progress and performance.