Muhammad Nur Maajid_1103228145

This module introduces the application of transfer learning techniques in deep learning with PyTorch.

The Importance of Transfer Learning:

- Previous models developed had poor performance.
- Transfer learning allows leveraging patterns (weights) learned by other models for our specific problem.
- It enables us to use patterns from computer vision models trained on the ImageNet dataset for our FoodVision Mini project.

What is Transfer Learning:

- Transfer learning lets us use patterns learned by models on other problems for our specific issue.

Benefits of Transfer Learning:

- Utilizes existing models that have proven successful on similar problems.
- Leverages models already trained on data similar to ours, often yielding good results with less custom data.

Applying Transfer Learning to FoodVision Mini:

- Transfer learning will be tested on FoodVision Mini using a computer vision model trained on ImageNet.

0. Initial Setup:

- This step involves steps for importing/downloading necessary modules.
- The code utilizes previously created Python scripts like data_setup.py and engine.py from "Module 05: PyTorch Going Modular."
- Downloads the going_modular directory from the pytorch-deep-learning repository if not present.
- Installs torchinfo if not available for a visual representation of our model.
- Uses the nightly version of torch and torchvision (v0.13+) to run this notebook.
- Regular imports and setups are performed, and the device (CUDA if available, otherwise CPU) is determined.

1. Acquiring Data:

- Before using transfer learning, a dataset is required.
- The pizza_steak_sushi.zip dataset is downloaded from the course GitHub and extracted if not present.

- This dataset contains images of pizza, steak, and sushi.

2. Creating Datasets and DataLoaders:

- With the going_modular directory downloaded, the data_setup.py script is used to set up DataLoaders.
- Special transformations are prepared for images to match trained models from torchvision.models.

2.1. Manual Creation of Transformation Pipeline:

- Manually creates a transformation pipeline using transforms.Compose. These transformations include resizing, pixel value conversion to the [0, 1] range, and normalization with specific means and standard deviations.

2.2. Auto Creation of Transformation Pipeline:

- Utilizes the automatic transformation feature introduced in torchvision v0.13+. This feature allows obtaining transformations suitable for selected pre-trained models without manual creation. For instance, transformations for the EfficientNet_B0 model are derived from its trained weights.
- DataLoaders are created using the create_dataloaders function from data_setup.py, both with manual and automatic transformations.

3. Utilizing Pre-Trained Models:

- A pre-trained model is used to tackle the food image classification task with FoodVision Mini. Previous attempts to build neural networks from scratch in prior notebooks did not yield satisfactory performance.
- Thus, transfer learning is applied. The basic idea is to take a well-performing model in a similar problem space and adjust it for our use case.
- In this context, as we are working on a computer vision problem (image classification with FoodVision Mini), we can find a pre-trained classification model in torchvision.models.

3.1 Choosing a Pre-Trained Model:

- The choice of model depends on the problem and the device used. Typically, higher numbers in model names (e.g., efficientnet_b0 to efficientnet_b7) indicate better performance but larger model size.
- Model choice should consider computational resources, especially for deployment on mobile devices.
- If computational power is unlimited, a larger model can be selected.

3.2 Preparing the Pre-Trained Model:

- For this example, the efficientnet_b0() pre-trained model is used. This model has three main parts: features (a set of convolutional and activation layers to learn basic visual data representations), avgpool (averages the output from the feature layers into a feature vector), and classifier (transforms the feature vector into an output vector matching the required number of output classes).
- The same code used to create transformations is employed to load ImageNet pre-trained weights.

3.3 Summarizing the Model with torchinfo.summary():

- The torchinfo.summary() method is used to understand the model further. It requires the model, input size, desired information columns, column width, and row settings.

3.4 Freezing the Base of the Model and Modifying the Output Layer:

- Transfer learning involves freezing some base layers of the pre-trained model (usually the features part) and then customizing the output (or classification/head) layer for our needs.
- In this example, all base layers are frozen by setting requires_grad=False for each parameter in model.features. Then, the output layer is modified to fit the number of classes we have.

4. Training the Model:

- The now semi-frozen model with a customized classifier is ready for transfer learning.
- For training, a loss function and optimizer are created.
- nn.CrossEntropyLoss() is used for the loss function, as it's a multi-class classification problem.
- torch.optim.Adam() is chosen as the optimizer with lr=0.001.
- The model is trained using the previously defined train() function.
- Training focuses only on the classifier parameters, with other parameters frozen.
- Random seeds are set, and the total training time is calculated.

5. Evaluating the Model with Loss Curve Plotting:

- The model appears to perform well. To further evaluate, loss curves from the training results are plotted.
- The plot_loss_curves() function is used to display these curves.
- Loss curves indicate both training and testing dataset losses and accuracy are moving in the right direction.

6. Predictions on Images from the Test Dataset:

- After quantitatively successful training, the model's qualitative performance is assessed by making predictions on images from the test dataset.
- A pred_and_plot_image() function is created to make predictions on test images.

- The function ensures that images for prediction are formatted similarly to those used to train the model.
- Random images from the test dataset are visualized along with their predictions.
- The prediction results demonstrate a significant improvement compared to the earlier TinyVGG models.

6.1 Predictions on Custom Images:

- Predictions are made on a custom image ("pizza-dad.jpeg").
- The pred_and_plot_image() function is used for making predictions on this image.