Muhammad Nur Maajid_1103228145

This module explores the process of converting a Jupyter or Google Colab notebook into a series of Python scripts. It specifically involves transforming the most valuable code cells from the "PyTorch Custom Datasets" notebook into a collection of Python scripts stored in a directory named "going_modular."

What Does "Going Modular" Mean? "Going modular" refers to the process of transitioning code from a notebook (such as Jupyter Notebook or Google Colab) into a series of Python scripts that offer similar functionality. For example, a notebook's code can be divided into multiple Python files, such as data_setup.py, engine.py, model_builder.py, train.py, and utils.py, tailored to the project's needs.

Why Go Modular? While notebooks are excellent for exploration and experimentation, Python scripts are recommended for larger-scale projects due to their ease of reproduction and execution. Compared to notebooks, Python scripts have advantages in version control, usage of specific sections, and better code packaging.

Pros and Cons of Notebooks vs. Python Scripts:

- **Notebooks:**
  - Pros: Ideal for experimentation, easy to start and share. They are visually and graphically engaging.
  - Cons: Challenging version control, difficulty in using specific parts, and sometimes excessive visuals.
- **Python Scripts:**
  - Pros: Better for packaging code, utilizing Git for version control, and supporting large-scale projects.
  - Cons: Less visual for experimentation, requires running the entire script, and less support from cloud vendors for notebooks.

General Workflow: Machine learning projects often start with notebooks for rapid experimentation, and then the most useful code parts are transferred into Python scripts.

PyTorch in the Wild: Many PyTorch-based machine learning project repositories include instructions for running PyTorch code as Python scripts.

0. Cell Mode vs. Script Mode:

- **Cell Mode:** Notebooks like "05. Going Modular Part 1 (cell mode)" run like typical notebooks, where each cell in the notebook contains either code or markup.
- **Script Mode:** Notebooks like "05. Going Modular Part 2 (script mode)" are similar to cell mode but many code cells can be converted into Python scripts. While it's not necessary to create Python scripts through a notebook, script mode demonstrates one way to transform a notebook into Python scripts.

1. Acquiring Data:

- Data is downloaded from GitHub using Python's requests module to fetch and extract a .zip file.
- The data consists of pizza, steak, and sushi images in a standard image classification format.
- The process results in a directory structure containing "train" and "test" folders for each category.

2. Creating Datasets and DataLoaders (data_setup.py):

- The code for creating PyTorch Datasets and DataLoaders is transformed into a function named create_dataloaders() and written into a file called data_setup.py using %%writefile.
- This function takes directory paths for training and testing data, transformations, batch size, and number of workers to create DataLoaders.
- It returns a tuple (train_dataloader, test_dataloader, class_names) containing DataLoaders for training and testing, and a list of class names.

3. Building a Model (model_builder.py):

- The previously created TinyVGG model from an earlier notebook is placed into a file named model_builder.py using %%writefile.
- This file contains the TinyVGG class definition, creating the TinyVGG architecture in PyTorch.
- The TinyVGG model can now be imported using from going_modular import model_builder.

4. Creating train_step() and test_step() Functions and a train() Function to Combine Them:

- **train_step():** Trains a PyTorch model for one epoch by performing training steps (forward pass, loss calculation, optimization) on a DataLoader.
- **test_step():** Evaluates a PyTorch model for one epoch by performing a forward pass on the testing dataset.
- **train():** Combines train_step() and test_step() for a specified number of epochs and returns the results in a dictionary format.

5. Creating a Function to Save the Model (utils.py):

- **save_model():** Saves a PyTorch model to a target directory using utility functions. This function is included in the utils.py file to avoid repetitive code.

6. Training, Evaluating, and Saving the Model (train.py):

- **train.py:** Brings together all previous functionalities (data_setup.py, engine.py, model_builder.py, utils.py) in a single Python script. This script is used to train a PyTorch model with a single command line input. Hyperparameters and other configurations can be set via command line arguments or directly in the script.