Muhammad Nur Maajid_1103228145

Computer vision is the field focused on teaching computers how to interpret and understand visual data. This includes developing models to distinguish between categories in images, such as identifying a photo as either a cat or a dog (binary classification) or classifying an image as a cat, dog, or chicken (multi-class classification). Additionally, computer vision encompasses tasks like locating cars in video frames (object detection) or differentiating between various objects in an image (panoptic segmentation).

1. **Acquiring the FashionMNIST Dataset:**
    - FashionMNIST is a computer vision dataset consisting of grayscale images of 10 different types of clothing.
    - It is accessible via PyTorch's torchvision.datasets.FashionMNIST() function, with parameters such as root, train, download, transform, and target_transform. 1.1 **Input and Output Shapes for Computer Vision Models:**
    - Images are represented as tensors of shape [color_channels, height, width].
    - Grayscale images have color_channels=1, while RGB images use color_channels=3.
    - The CHW (Color Channels, Height, Width) or HWC (Color Channels Last) format is employed.
    - NHWC is generally more performance-efficient, but PyTorch defaults to NCHW. 1.2 **Data Visualization:**
    - Examples from the dataset are visualized using matplotlib.
    - The dataset includes 60,000 training samples and 10,000 testing samples.
    - With 10 clothing categories, this dataset presents a multi-class classification challenge.
    - Visualization examples:
        - Displaying the shape and image of a single sample.
        - Showing multiple samples in a 4x4 grid with class labels.
    - Additional considerations:
        - Although aesthetically simple, the principles of model building apply across various computer vision problems.
        - The 60,000 images constitute a relatively small dataset in deep learning terms.
        - PyTorch models offer a faster approach than manually classifying each image.
2. **Preparing the DataLoader:**
    - This section covers data preparation using torch.utils.data.DataLoader. DataLoader facilitates loading data into the model during both training and inference. It converts large Datasets into small, manageable Python iterables called batches or mini-batches, set by the batch_size parameter.
    - Batches are computationally efficient. While ideally, we could process all data at once, large datasets require breaking into batches unless unlimited computational power is available.

- This method also allows the model to improve performance more frequently. With mini-batches, gradient descent occurs more often per epoch (once per mini-batch rather than once per epoch).
- A good starting batch size is 32, but as a tunable hyperparameter, various values, often powers of 2 (e.g., 32, 64, 128, 256, 512), are typically experimented with.
- Code is provided to illustrate creating DataLoaders for both training and testing sets with a predetermined batch size. The DataLoader outputs, including the length for training and testing sets, are displayed.
- The integrity of the data is checked by examining a sample from the training DataLoader. The process of batch retrieval and using DataLoader for training models is successfully implemented in this code.

3. **Model 0: Building a Baseline Model** 3.1 **Setting Up Loss Function, Optimizer, and Evaluation Metrics**
   - This part discusses preparing the loss function, optimizer, and evaluation metrics for classification tasks. CrossEntropyLoss (nn.CrossEntropyLoss()) is used for classification tasks, while stochastic gradient descent (torch.optim.SGD) is selected as the optimizer. Additionally, an accuracy metric is introduced using an accuracy_fn function imported from helper_functions. This accuracy function calculates the percentage of correct predictions. 3.2 **Creating a Function to Measure Experiment Time**
   - A time measurement function (print_train_time) is created to track the duration of model training. This function takes the start and end times of training as inputs and prints the time taken to train the model on the specified device. 3.3 **Constructing the Training Loop and Training the Model on Batch Data**
   - This section implements the training loop to train the baseline model (model_0). The training loop iterates through epochs, processing training batches using the train_dataloader for each epoch. The loop includes forward pass steps, loss calculation, backward pass, and optimizer steps. Additionally, the training loop accumulates training loss per epoch.
   - After training, the loop proceeds to the testing phase, where the model is evaluated on the test set (test_dataloader). The testing loop calculates both loss and accuracy. Overall progress, including training and testing results, is printed during each epoch.
4. **Making Predictions and Obtaining Results from Model 0**
   - In this step, predictions are made using model 0, and its results are obtained. An eval_model function is created to accept a trained model, DataLoader, loss function, and accuracy function. This function calculates the loss and accuracy on the given data set.
   - The outcome of model 0 is as follows:
   - {'model_name': 'FashionMNISTModelV0', 'model_loss': 0.47663894295692444, 'model_acc': 83.42651757188499}
5. **Preparing Device-Agnostic Code:**
   - This step involves creating device-agnostic code to enable GPU usage if available. This code identifies whether a CUDA (GPU) device is available and sets the device accordingly. If a GPU is available, the model will run on the GPU; otherwise, it will use the CPU.

6. **Model 1: Building a Better Model with Non-Linearity**
    - In this step, Model 1 is constructed to be more complex by adding non-linear functions (nn.ReLU()) between each linear layer. This model is designed to explore whether adding non-linearity can improve model performance. 6.1 **Training and Testing Functions**
    - This step involves creating more general training and testing functions to be used with both models. These functions are designed to accept a model, DataLoader, loss function, optimizer, and accuracy function. This helps keep the code clean and modular. 6.2 **Evaluating Model 1**
    - After training, the updated eval_model function is used to evaluate Model 1's results on the test data set.
7. **Model 2: Building a Convolutional Neural Network (CNN)**
    - This step introduces building a Convolutional Neural Network (CNN) or ConvNet to enhance our model's performance in handling visual data. 7.1 **Model Selection**
    - CNNs are known for finding patterns in visual data, so we'll try using a CNN to improve baseline results.
    - The model we'll use is called TinyVGG from the CNN Explainer site.
    - TinyVGG follows a typical CNN structure: Input layer -> [Convolutional layer -> activation layer -> pooling layer] -> Output layer.
    - Model choice depends on the type of data being handled. For structured data, models like Gradient Boosted Models or Random Forests may be used, while for unstructured data like images, audio, or text, CNNs or Transformers are more suitable. 7.2 **TinyVGG Architecture**
    - TinyVGG consists of two main blocks, each with Convolutional, ReLU activation, and MaxPooling layers.
    - General structure: Conv2d -> ReLU -> Conv2d -> ReLU -> MaxPool2d.
    - The FashionMNISTModelV2 class is used to implement TinyVGG using PyTorch's nn.Conv2d and nn.MaxPool2d modules. 7.3 **Building the CNN Model**
    - The FashionMNISTModelV2 class in PyTorch is used to build the CNN model, following the TinyVGG architecture. 7.4 **Examples of Using Conv2d and MaxPool2d**
    - nn.Conv2d is used for feature extraction from images by filtering them with specific kernels.
    - nn.MaxPool2d is used to reduce the spatial dimensions of images.
    - Examples and parameter testing are provided in the code snippets.
8. **Comparing Model Results and Training Time**
    - Three different models have been trained: model_0, model_1, and model_2.
    - model_0 is a basic model with two nn.Linear() layers.
    - model_1 is similar to the basic model but with nn.ReLU() layers between nn.Linear() layers.
    - model_2 is the first CNN model, mimicking the TinyVGG architecture from the CNN Explainer site.
    - These models are built and trained to determine the best performance.
    - The model results are compiled in a DataFrame for comparison.
    - Training time is also added to the result comparison.
9. **Making and Evaluating Random Predictions with the Best Model**

- The best-performing model is model_2.
- A make_predictions() function is created to make predictions with the given model and data.
- Predictions are made on test samples using model_2.
- Prediction probabilities are displayed, and predictions are converted into labels.

10. **Creating a Confusion Matrix for Further Evaluation**
    - A confusion matrix is used as a visual evaluation method.
    - Predictions are made with the trained model (model_2).
    - The confusion matrix is created using torchmetrics.ConfusionMatrix.
    - The confusion matrix is plotted using mlxtend.plotting.plot_confusion_matrix.
11. **Saving and Loading the Best Model**
    - The best model (model_2) is saved by saving its state_dict() using torch.save().
    - The best model is reloaded by creating a new instance of the model class and using load_state_dict().
    - The trained model is retrained to ensure the results are consistent with the model before saving.