Muhammad Nur Maajid_1103228145

This module serves as a gateway to understanding the fundamentals of machine learning and deep learning. Its primary aim is to leverage historical data to develop algorithms, such as neural networks, capable of identifying patterns within this data. Once these patterns are established, the algorithms can be employed to forecast future events.

In this module, we begin with a basic concept: the straight line. We then explore how to construct a PyTorch model that can learn from this linear pattern and adapt accordingly. The following key steps will be covered in this module:

1. **Data Preparation:** Initially, we'll create simple linear data.
2. **Model Development:** Here, we construct a model to learn the data's patterns, choose a loss function and optimizer, and establish a training loop.
3. **Model Adjustment with Data (Training):** With the data and model in place, we'll let the model seek patterns in the training data.
4. **Prediction and Model Evaluation (Inference):** Now that our model has identified data patterns, we'll compare its findings with actual test data.
5. **Model Saving and Loading:** You might want to use your model elsewhere or revisit it later. We'll discuss how to save and reload models.
6. **Combining All Components:** We'll bring all the above steps together as a cohesive whole.

Data Preparation and Loading In machine learning, data can be anything from numerical tables, images, and videos to audio, protein structures, and text. The machine learning process involves two main parts: transforming data into a representative numerical set and building or selecting a model to learn this representation as effectively as possible. In this module, we begin with creating linear data as an example. The steps involved include:

1. Establishing known parameters for linear data.
2. Generating data using linear regression via PyTorch.
3. Splitting the data into training and testing sets in an 80:20 ratio.
4. Visualizing the data to comprehend the patterns and relationships between features (X) and labels (y). The prepared and visualized data will form the foundation for model building in the next step.

Model Building In this phase, we construct a linear regression model using PyTorch. This model is created as a Python class implementing a simple linear regression function. Essentially, we utilize the torch.nn module, which provides building blocks for computational graphs, like nn.Module, nn.Parameter, and torch.optim. The linear regression model we create has two parameters: weight and bias, which will be adjusted during the learning process.

We also conduct simple testing on the created model by making predictions on test data. Initially, the predictions are poor as the model's parameters are randomly initialized. Next, we'll see how this model can be fine-tuned to make more accurate predictions.

Summary of Part 3: Model Training In this step, we train the linear regression model using PyTorch. This training process involves several key steps:

1. Model Initialization: The model starts with randomly set parameters (weights and bias).
2. Loss Function and Optimizer Creation: The loss function measures how off the model's predictions are from actual values. The optimizer, in this case, Stochastic Gradient Descent (SGD), updates the model's parameters to better match the data patterns.
3. Training Loop: The model iterates through the training data, makes predictions, calculates loss, performs backpropagation, and updates parameters to minimize loss.
4. Testing Loop: The model is tested on unseen data to evaluate how well it generalizes.
5. Loss Curves Visualization: Observing the loss curves over time to assess how well the model has learned from the data. The result of this training is a better-predicting model, and we observe a decrease in loss value with each iteration. At the end of the training, the model's parameters (weights and bias) approach the actual values, indicating that the model has learned the data patterns.

Understanding the concepts of training and evaluating models, as well as the impact of hyperparameters like the number of epochs and learning rate, is crucial in achieving good results. These steps help us grasp the core workflow in PyTorch for developing simple machine learning models.

Making Predictions with a Trained PyTorch Model (Inference) After training the model, the next step is to make predictions (inference) using it. The main steps for making predictions with a PyTorch model are:

1. Setting the Model to Evaluation Mode: Using the model.eval() function ensures the model is in evaluation mode, where certain functions not needed during training are deactivated.
2. Using Inference Mode Context: Employing with torch.inference_mode(): as a context for making predictions ensures that calculations are done quickly and efficiently.
3. Ensuring Calculations on the Same Device: Ensure that both the model and data are on the same device (CPU or GPU) to prevent cross-device errors. Once these steps are taken, we can make predictions using the trained model on test data. In the example above, the prediction results (y_preds) have been generated, and prediction visualization is performed using a plot_predictions() function.

Next, we will explore how to save and load PyTorch models.

Saving and Loading PyTorch Models After training a PyTorch model, the next step is to save it for future use or deployment elsewhere. PyTorch provides several key methods for saving and loading models, detailed below:

1. torch.save: Saves serialized objects to disk using Python's pickle utility. Models, tensors, and other Python objects like dictionaries can be saved using torch.save.

2. torch.load: Uses pickle's deserialization feature to load serialized Python objects (like models, tensors, or dictionaries) back into memory. You can also specify which device the object should be loaded onto (CPU, GPU, etc.).
3. torch.nn.Module.load_state_dict: Loads a model's parameter dictionary (model.state_dict()) using a previously saved state_dict() object. It is recommended to save and load models using only the state_dict(), as this method is more flexible and not dependent on specific class structures or directories.
4. Saving the Model:
    - Create a directory to store the model (in this example, "models").
    - Create a file path to save the model.
    - Use torch.save(obj, f) where obj is the target model's state_dict() and f is the filename to save the model.
5. Loading the Model:
    - Create a new instance of the model to be loaded.
    - Load the saved model's state_dict() using torch.load(f) and insert it into the new model instance using loaded_model.load_state_dict().
6. Making Predictions with the Loaded Model:
    - Set the loaded model to evaluation mode.
    - Use inference_mode context to make predictions. Once the model is loaded, the predictions made by it can be compared with previous predictions to ensure the model has been correctly saved and loaded.

Combining All Steps

1. Data
    - Create data by defining weight and bias.
    - Generate a range of values from 0 to 1 for X.
    - Use X, weight, and bias to create labels (y) using the linear regression formula (y = weight * X + bias).
    - Split the data into training and testing sets (80/20 split).
2. Building a Linear PyTorch Model
    - Construct a linear model using subclass nn.Module and nn.Linear().
    - The model has one linear layer with one input and one output.
3. Training the Model
    - Utilize the loss function nn.L1Loss() and optimizer torch.optim.SGD().
    - Train the model over a series of epochs by placing data on the available device (GPU if available, otherwise default to CPU).
4. Making Predictions
    - Switch the model to evaluation mode.
    - Perform predictions on test data.
5. Saving and Loading the Model
    - Save the model to a file using torch.save() including the model's state_dict().
    - Reload the model using torch.load() and load_state_dict().
    - Evaluate the loaded model to ensure its predictions match the original model.
6. Overall

- Integrate the above steps into a single code flow, making it device agnostic so it can utilize a GPU if available or switch to CPU otherwise.