Neural Network with PyTorch — Classifying Iris Flowers

Dataset: Iris Dataset (download the iris.csv)

Objective: Build and train a neural network using PyTorch to classify iris flowers into one of three species.

1. **Load & Preprocess the Data**
   - Load the dataset from the iris.csv file into a Pandas DataFrame.
   - Encode the target column (species) as integer labels (0, 1, 2).
   - Split the dataset into 80% training and 20% testing.
   - Normalize the feature values (e.g., using StandardScaler).
2. **Visualize the Data**
   - Create box plots for each feature (sepal length, sepal width, petal length, petal width), separated by species to show the distribution and spread per class.
   - Optional: Create a pairplot (scatterplot matrix) of all features, colored by species.
3. **Build the Neural Network**
   - Use PyTorch to define the following architecture:
     - Input layer: 4 neurons (one for each feature)
     - Hidden layer: 16 neurons with ReLU activation
     - Output layer: 3 neurons
   - Use CrossEntropyLoss as the loss function.
   - Use an optimizer such as SGD or Adam.
4. **Train the Model**
   - Train the model for 100 epochs.
   - After every 10 epochs:
     - Calculate and print training and test loss.
     - Calculate and print training and test accuracy.
5. **Create the Following Plots**
   - Plot 1: Training and test loss vs. epochs
   - Plot 2: Training and test accuracy vs. epochs
   - Plot 3: Final confusion matrix (visualized as a heatmap with class labels)

Optional Extension: Understanding and Implementing Backpropagation from Scratch

Objective: Gain a deeper understanding of how neural networks learn by manually implementing the backpropagation algorithm. This will help you understand how neural networks optimize weights and biases using gradients.

1. **Understand the Theory of Backpropagation**
   - Read about how the backpropagation algorithm works:
     - It computes the gradient of the loss function with respect to each weight using the chain rule.
     - These gradients are then used to update the weights in a direction that reduces the loss (typically by moving opposite to the gradient).
2. **Implement Backpropagation from Scratch**
   - Instead of using PyTorch's built-in autograd and optimizer, implement backpropagation manually to compute gradients and update weights using gradient descent.
   - Manually calculate:
     - Forward Pass: Calculate the activations of each layer.
     - Backward Pass: Compute the gradient of the loss with respect to each weight and bias, using the chain rule.
     - Weight Update: Manually update the weights using the computed gradients and a learning rate.
3. **Compare with PyTorch's Built-in Backpropagation**
   - Once you've implemented backpropagation, compare the results (accuracy, loss) with the PyTorch model that uses the built-in autograd system and an optimizer.
   - Verify that both implementations are consistent and produce similar results.
4. **Experiment with Different Learning Rates**
   - After implementing backpropagation manually, experiment with different learning rates to observe how the network's training dynamics change.