🧠 Image Classification Project Using PyTorch (Detailed Summary)

This project involves building an image classification model using the **MNIST dataset** and the **PyTorch** deep learning framework. The goal is to classify grayscale images of handwritten digits (0–9). It follows an end-to-end machine learning pipeline: data preprocessing, model design, training, evaluation, and saving the trained model for reuse.

📊 Dataset: MNIST

The **MNIST dataset** is a benchmark in computer vision, containing:

- **60,000 training images**
- **10,000 test images**
- Each image is **28x28 pixels**, grayscale, and represents a digit from 0 to 9.

The dataset is loaded using torchvision.datasets, and a transformation pipeline is applied to:

- Convert images into tensors
- Normalize pixel values (scaling from [0, 255] to [0, 1] range)

⚙️ Data Preparation

Data is loaded in mini-batches using PyTorch's DataLoader, which:

- Handles efficient memory usage
- Batches and shuffles the training data
- Allows parallel data loading

This setup ensures that the model sees a diverse mix of examples in each epoch, promoting generalization.

🧮 Model Architecture

A **simple feedforward neural network** is implemented by subclassing nn.Module, which includes:

- An **input layer** for 784 features (28×28 flattened pixels)
- One or more **hidden layers** (e.g., 128 and 64 neurons), activated with **ReLU**
- An **output layer** with 10 neurons corresponding to digit classes

No convolutional layers are used, keeping the model architecture intentionally simple for foundational learning.

🧠 Deep Learning Concepts Applied

- **Forward Propagation**: Input is passed through layers to generate predictions

- **Loss Function**: CrossEntropyLoss measures the difference between predicted and true labels
- **Backpropagation**: Computes gradients of the loss with respect to model weights
- **Optimization**: The **SGD** or **Adam** optimizer updates model weights based on gradients

---

🚀 Training

The training loop runs for multiple **epochs**. Each epoch:

- Loads a batch of data
- Passes it through the model to compute predictions
- Calculates loss
- Performs backpropagation
- Updates the model's weights

This iterative process helps the model gradually learn digit patterns.

🧪 Evaluation

After training, the model is tested on unseen data:

- It is switched to **evaluation mode** (model.eval())
- Inference is done without computing gradients (torch.no_grad())
- **Accuracy** is calculated as:

Accuracy=Correct PredictionsTotal Test Samples×100\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{Total Test Samples}} \times 100Accuracy=Total Test SamplesCorrect Predictions×100

📈 **Achieved Accuracy**: 85%

This indicates that the model has learned meaningful features from the data, though there's room for improvement using techniques like convolutional layers or regularization.

💾 Saving and Loading

After training, the model's parameters are saved using torch.save(model.state_dict()). The model can later be reloaded using torch.load() and load_state_dict() for inference or fine-tuning—supporting reproducibility and deployment.

✅ Key Takeaways

- Hands-on experience with PyTorch's full deep learning pipeline
- Understanding of core concepts: activation functions, loss, backpropagation, gradient descent

- Exposure to model evaluation metrics and real-world dataset handling
- Built a working model that achieves **85% accuracy** on test data with a basic architecture

---

📌 Conclusion

This project is a foundational deep learning exercise that applies theoretical knowledge to a practical problem. With an accuracy of 85%, the model performs well given its simplicity, and it provides a strong starting point for further experimentation with CNNs, dropout, data augmentation, or learning rate scheduling.