Explanation
Image Classification with MNIST

University of Nankai
MSE: Subject AIoT
Somoeurn Virakden

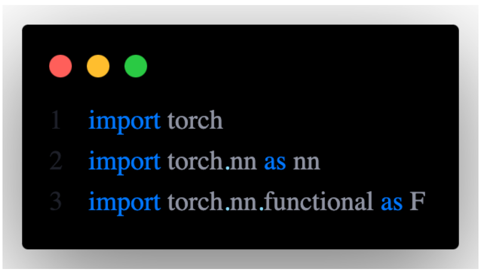
1 Introduction

This notebook reproduces the ResNet18 deep learning model for image classification on the MNIST dataset. The model is based on the ResNet architecture, which utilizes residual connections to improve training performance and mitigate vanishing gradient problems. The model is trained using the Adam optimizer and Cross-Entropy loss to classify images of handwritten digits (0-9) in the MNIST dataset.

2 Model Architecture

2.1 Set-up Library

This block of code is setting up the necessary imports for building and training the neural network using PyTorch.



```python
import torch
import torch.nn as nn
import torch.nn.functional as F
```

- **import torch**: This imports the main PyTorch library. torch is used for creating tensors (multi-dimensional arrays), performing mathematical operations, and building neural networks.
- **import torch.nn as nn**: This imports the torch.nn module, which contains various classes and functions for building neural networks. The alias nn is commonly used as a shorthand for this module. It provides essential layers, loss functions, and utilities to construct deep learning models.
- **import torch.nn.functional as F**: This imports the torch.nn.functional modules as F. This module contains many functions for operations that are used during the forward pass of a neural network.

2.2 ResNet18 Builder

This block defines the core architecture of ResNet, including the residual blocks and the full ResNet network. It handles both feature extraction and classification.

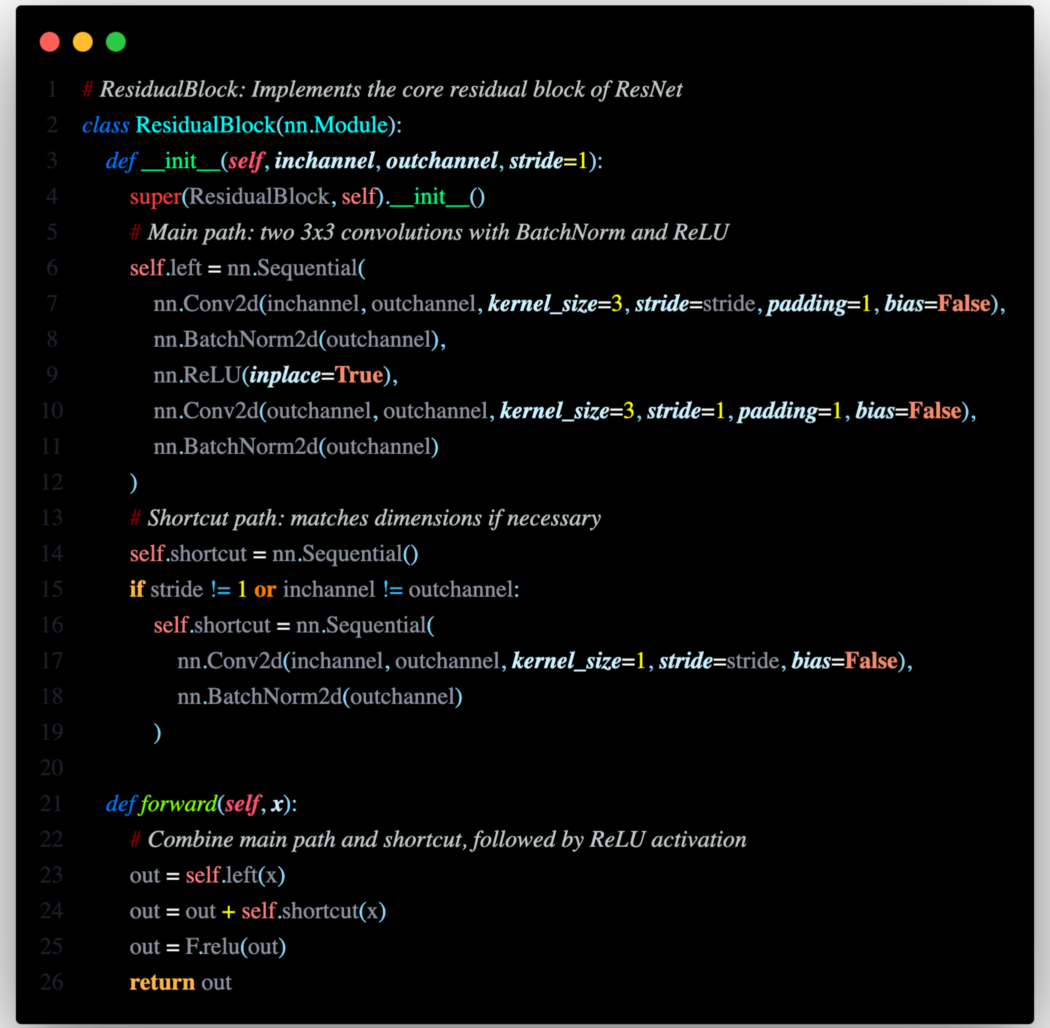
2.2.1 ResidualBlock

The ResidualBlock class defines a single residual block in the ResNet architecture, It is has two function declare such as:

- In function *_init_* has four parameters are self, inchannel, outchannel, stride=1 is constructor method initializes the layers of the residual block:
  - **inchannel**: The number of input channels to the block.
  - **outchannel**: The number of output channels from the block.

  - **stride**: The stride of the convolution; typically, stride is set to 1 for the main path and 2 for downsampling in ResNet architectures.

- **Main Path** (`self.left`) is the path consists of two convolutional layers with batch normalization and ReLU activation.
- **Shortcut Path** (`self.shortcut`) is the path is only created if the input and output channels differ or if the stride is not 1.
- Last function is forward has two parameters are `self, x` is the outputs of both paths are added together before applying ReLU.
  - **Main Path Output** (`out = self.left(x)`) is passes the input through the main path.
  - **Shortcut Addition** (`out = out + self.shortcut(x)`) is adds the input to the processed output.
  - **ReLU Activation** (`F.relu(out)`) is applies the ReLU function after combining the paths.

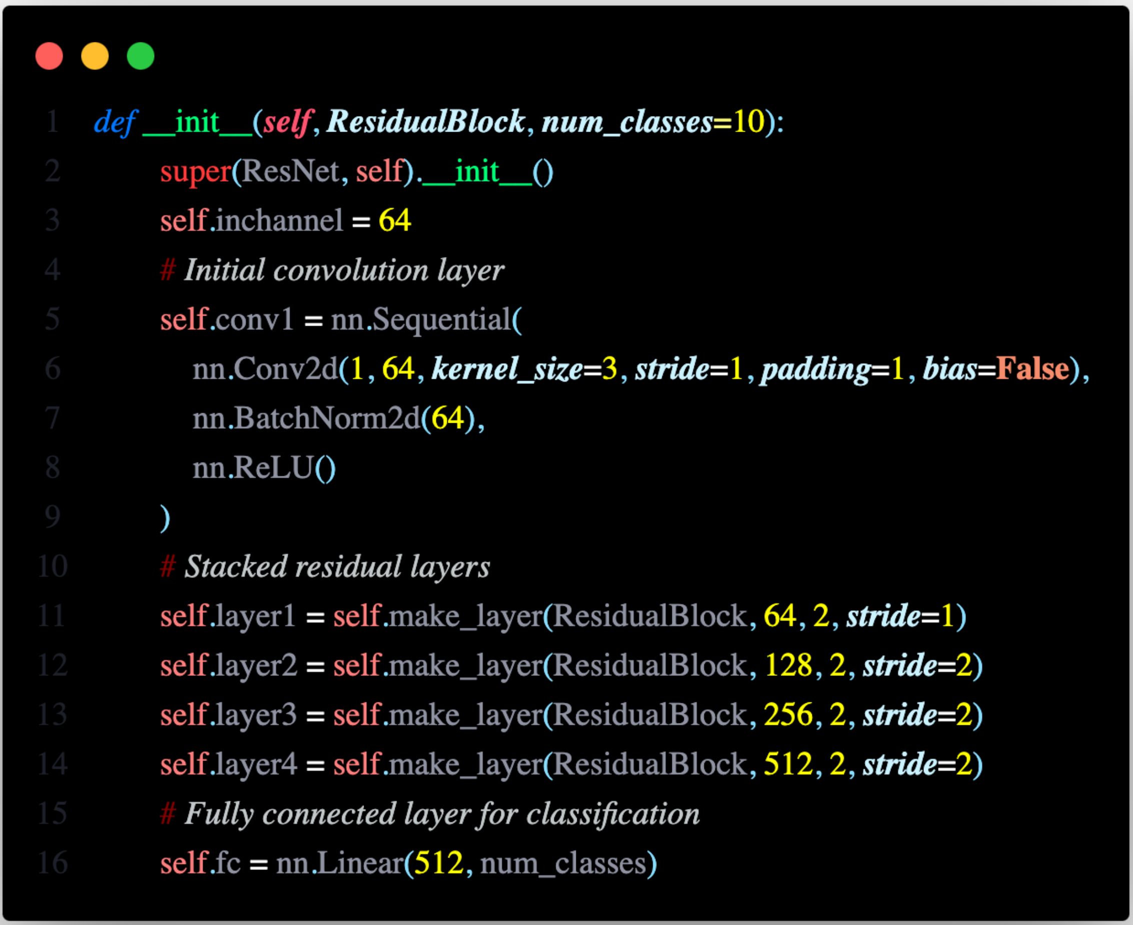


```python
# ResidualBlock: Implements the core residual block of ResNet
class ResidualBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).__init__()
        # Main path: two 3x3 convolutions with BatchNorm and ReLU
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=True),
            nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(outchannel)
        )
        # Shortcut path: matches dimensions if necessary
        self.shortcut = nn.Sequential()
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(outchannel)
            )

    def forward(self, x):
        # Combine main path and shortcut, followed by ReLU activation
        out = self.left(x)
        out = out + self.shortcut(x)
        out = F.relu(out)
        return out
```

2.2.2 ResNetBlock

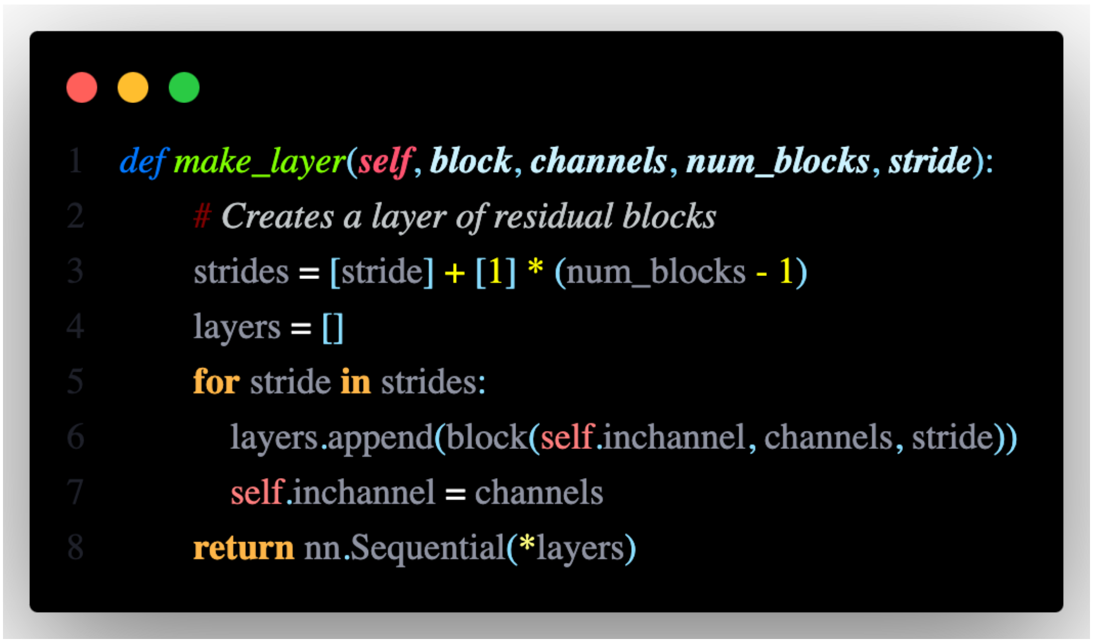
This class defines the overall ResNet architecture by stacking `ResidualBlocks`.

- In function _init_ has three parameters are self, ResidualBlock, num_classes=10 is the constructor method initializes:
  - **Initial Convolution** `self.conv1`
    - Applies a 3×3 convolution followed by BatchNorm and ReLU.
    - Converts 3-channel input to 64 channels.
  - **Residual Layers** `self.layer1` to `self.layer4`:
    - Each layer consists of multiple `ResidualBlocks` created by `make_layer`.
    - As the layers progress, the number of channels doubles $64 \to 128 \to 256 \to 512$, and the spatial resolution is halved.
  - **Fully Connected Layer** `self.fc`:
    - Maps the flattened features to the number of output classes (10 for MNIST)



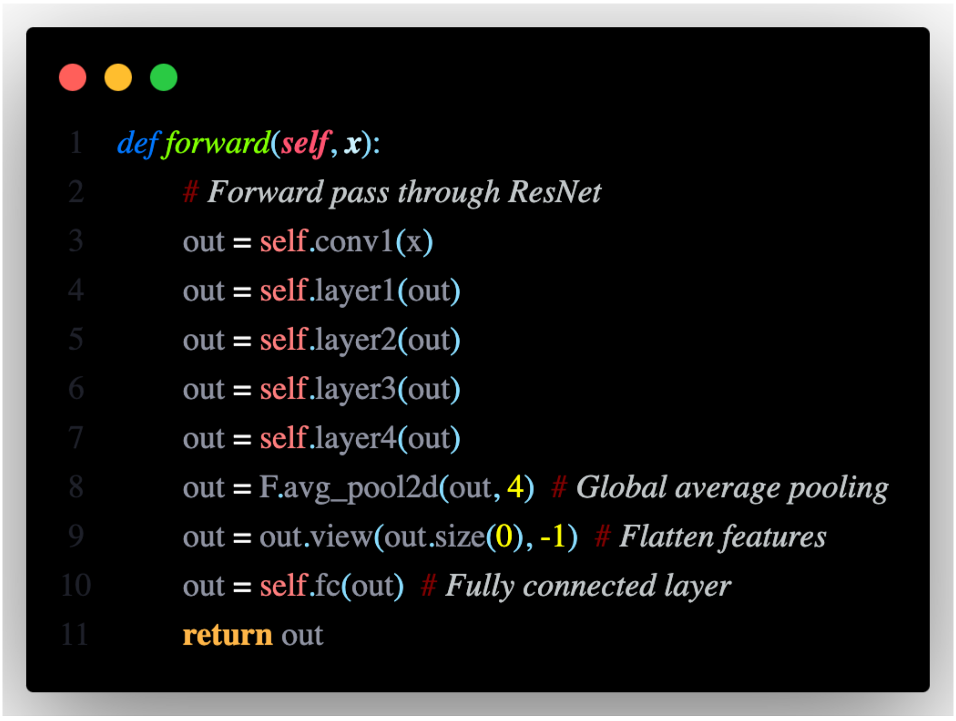
```python
def __init__(self, ResidualBlock, num_classes=10):
    super(ResNet, self).__init__()
    self.inchannel = 64
    # Initial convolution layer
    self.conv1 = nn.Sequential(
        nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1, bias=False),
        nn.BatchNorm2d(64),
        nn.ReLU()
    )
    # Stacked residual layers
    self.layer1 = self.make_layer(ResidualBlock, 64, 2, stride=1)
    self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)
    self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)
    self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)
    # Fully connected layer for classification
    self.fc = nn.Linear(512, num_classes)
```

- In function make_layer has five parameters are self, block, channels, num_blocks, stride this helper function constructs a sequence of residual blocks. The first block in the layer has a stride of `stride` (which controls downsampling), and the remaining blocks have stride 1. The output channels for each layer double as the architecture deepens (64, 128, 256, 512).



```python
def make_layer(self, block, channels, num_blocks, stride):
        # Creates a layer of residual blocks
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)
```

- The last functionforward has two parameters are self, x.
  - Defines the forward pass through the network:
    1. The input passes through the initial convolution.
    2. It sequentially traverses `layer1` to `layer4`.
    3. An average pooling operation (`F.avg_pool2d`) reduces the spatial dimensions to 1×1.
    4. The features are flattened (`view`) and passed to the fully connected layer.

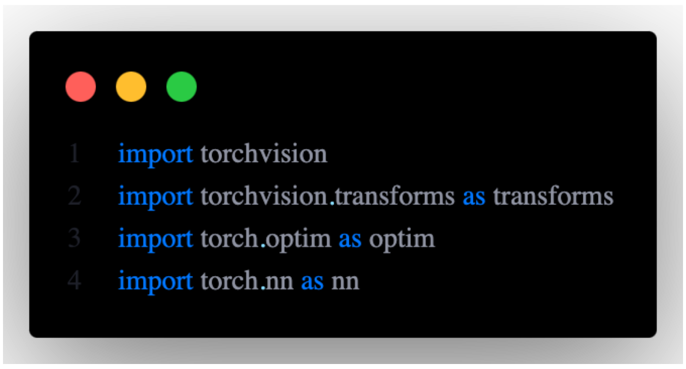


```python
def forward(self, x):
        # Forward pass through ResNet
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)  # Global average pooling
        out = out.view(out.size(0), -1)  # Flatten features
        out = self.fc(out)  # Fully connected layer
        return out
```

3 Prepare and Load Dataset

This block prepares the dataset, sets hyperparameters, and defines the loss function and optimizer for training with the MNIST datasets.

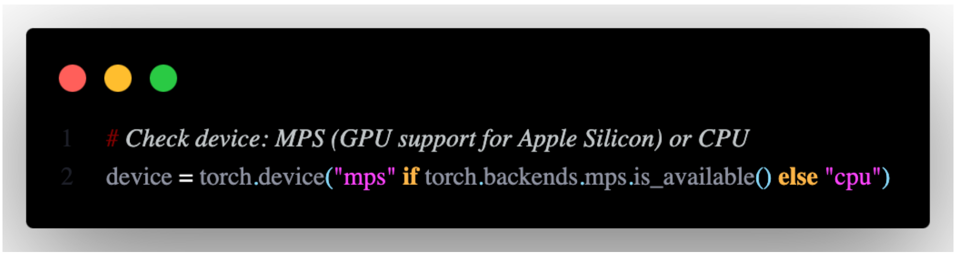
3.1 Importing Required Libraries



```python
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import torch.nn as nn
```

- `import torchvision`: Imports the `torchvision` package, which contains pre-built datasets (such as MNIST) and image transformations commonly used for computer vision tasks.
- `import torchvision.transforms as transforms`: Imports the `transforms` module, which provides functions for applying transformations to images.
- `import torch.optim as optim`: Imports the `optim` module, which contains optimization functions, allowing us to apply various algorithms to optimize the model during training.
- `import torch.nn as nn`: Imports the `torch.nn` module, which provides pre-built layers, loss functions, and utilities to define and train neural networks.

3.2 Device Configuration

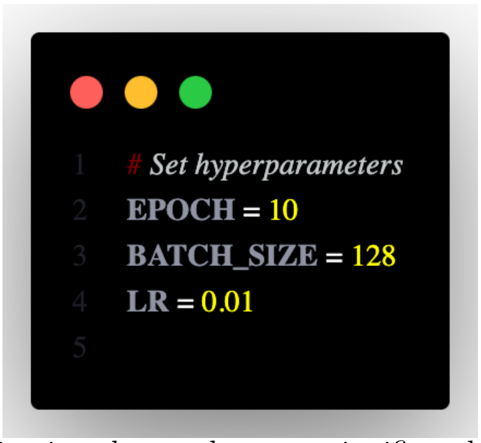


```python
# Check device: MPS (GPU support for Apple Silicon) or CPU
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
```

This line checks whether Metal Performance Shaders (MPS) is available on the system for Apple Silicon devices. If MPS is supported, it uses the GPU for training; otherwise, it defaults to the CPU. This ensures that the model runs on the available hardware for optimal performance.

3.3 Setting Hyperparameters

These define critical hyperparameters for training:

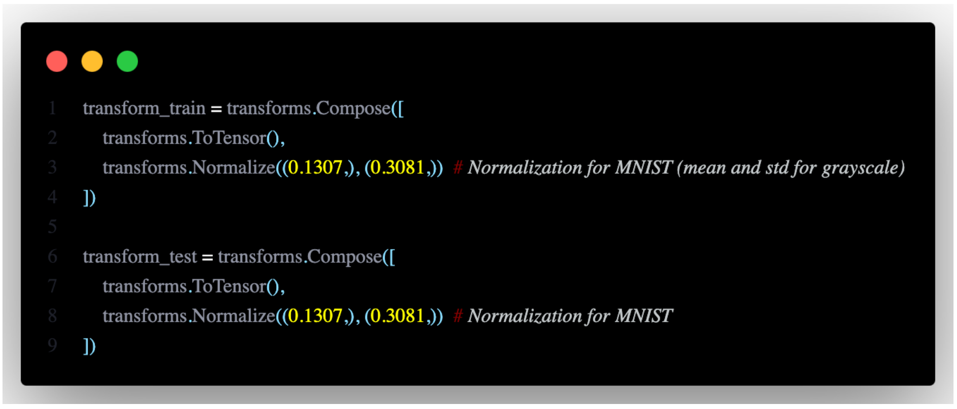


```
# Set hyperparameters
EPOCH = 10
BATCH_SIZE = 128
LR = 0.01
```

- `EPOCH`:Number of iterations through the entire dataset.
- `BATCH_SIZE`:Number of samples processed together before updating model parameters
- `LR`:Learning rate controlling step size at each iteration

Adjusting these values can significantly impact model performance and training speed.

3.4 Data Transformation for Training and Testing



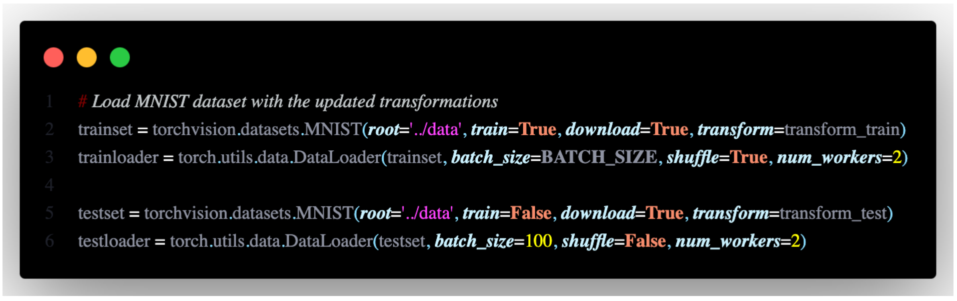
```
transform_train = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # Normalization for MNIST (mean and std for grayscale)
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # Normalization for MNIST
])
```

This section defines data transformations for both training and test datasets:

- `ToTensor()`: Converts PIL images to PyTorch tensors
- `Normalize()`: Standardizes pixel values using mean and standard deviation specific to MNIST dataset

Normalization is crucial for consistent feature scaling and improved model generalization.

3.5 Data Loading



```
# Load MNIST dataset with the updated transformations
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)
```

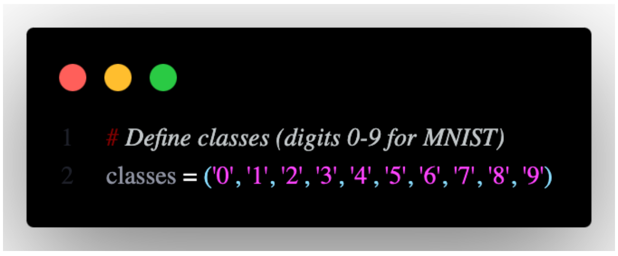
This section loads and prepares the MNIST dataset:

- `MNIST()`: Loads the MNIST dataset.
- `DataLoader()`: Creates efficient data loaders for training and testing.
- `batch_size`: Determines number of samples per batch.
- `shuffle`: Randomizes order of samples in each epoch.
- `num_workers`: Number of subprocesses to load data in parallel.

Proper data loading is essential for efficient training and testing.

3.6 Class Labels for MNIST

- `classes`: A tuple that contains the labels for the MNIST dataset, which correspond to the digits 0 through 9. These are used later during the model evaluation and result interpretation to map the predicted class indices to actual digit labels.



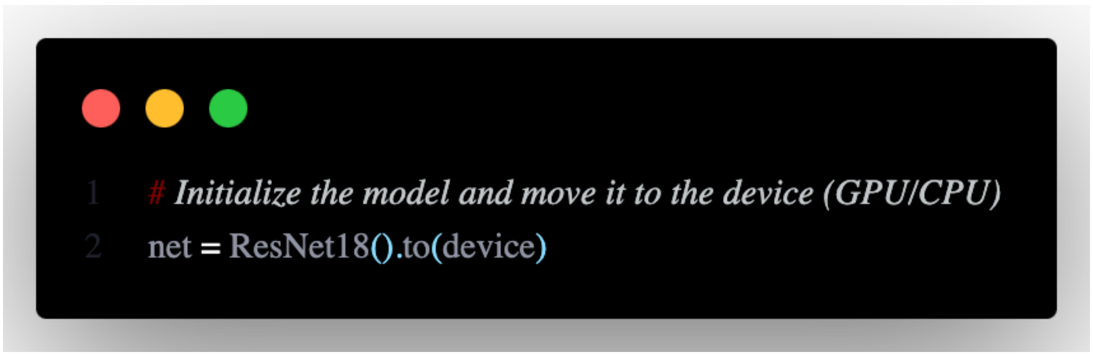
```
# Define classes (digits 0-9 for MNIST)
classes = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9')
```

3.7 Model Initialization

This initializes the ResNet18 model:

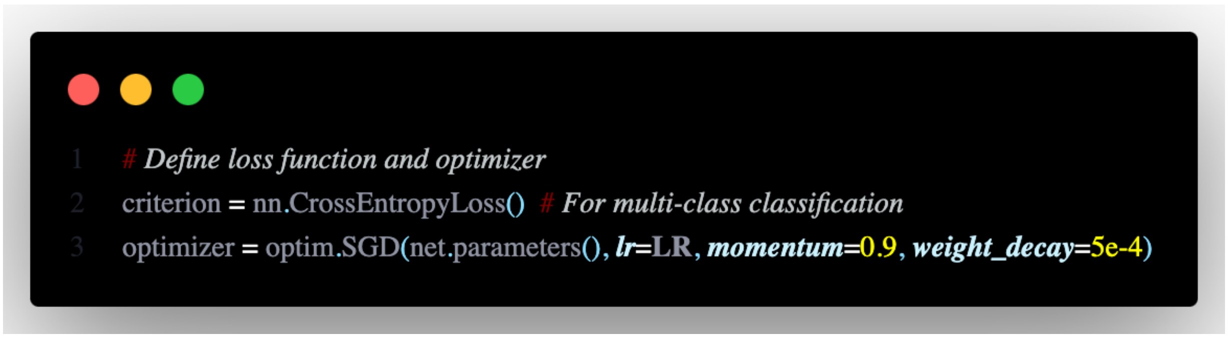
- `ResNet18()`: Instantiates a pre-defined ResNet architecture.
- `.to(device)`: Moves the model to the specified device (CPU or GPU).

Model initialization sets up the neural network architecture for training.



```
# Initialize the model and move it to the device (GPU/CPU)
net = ResNet18().to(device)
```

3.8 Defining Loss Function and Optimize



```python
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()  # For multi-class classification
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9, weight_decay=5e-4)
```

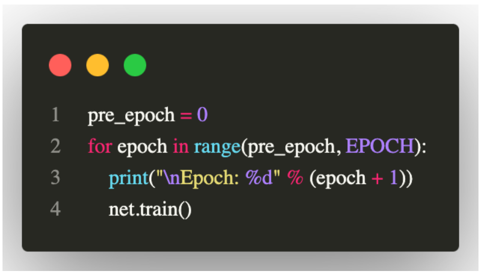
This section defines the loss function and optimizer:

- `CrossEntropyLoss()`: Measures the error between model predictions and true labels.
- `SGD()`: Implements stochastic gradient descent optimization algorithm.
- `lr`: Learning rate controls step size at each iteration.
- `momentum`: Helps escape local minima in optimization.
- `weight_decay`: Regularization term to prevent overfitting.

The choice of loss function and optimizer significantly impacts model training dynamics.

4 Training

4.1 Pre-training Epochs



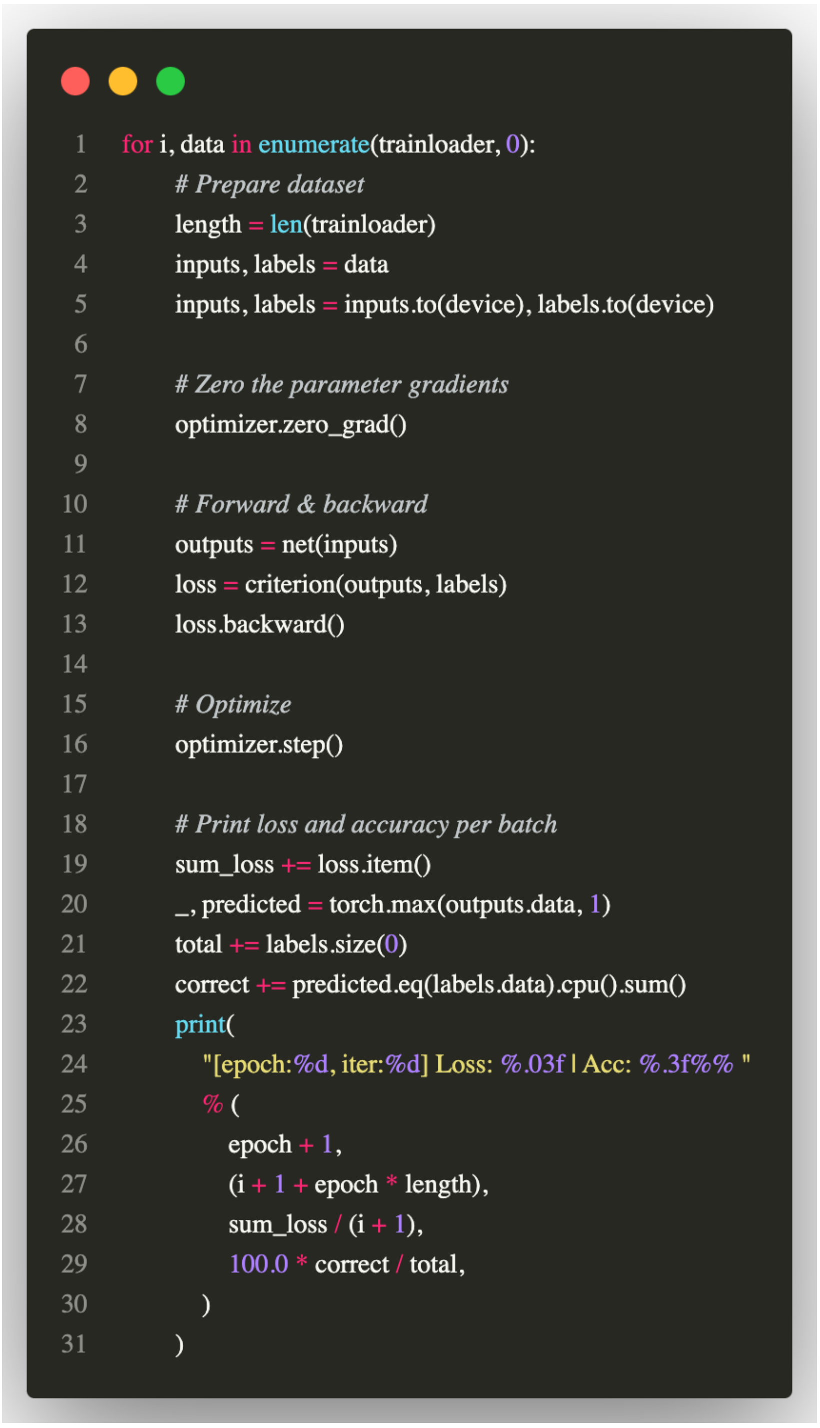
```python
pre_epoch = 0
for epoch in range(pre_epoch, EPOCH):
    print("\nEpoch: %d" % (epoch + 1))
    net.train()
```

- `pre_epoch`: Allows for fine-tuning after an initial training phase.
- Outer loop: Iterates through training epochs.
- `net.train()`: Sets the model to training mode.

4.2 Training Loop

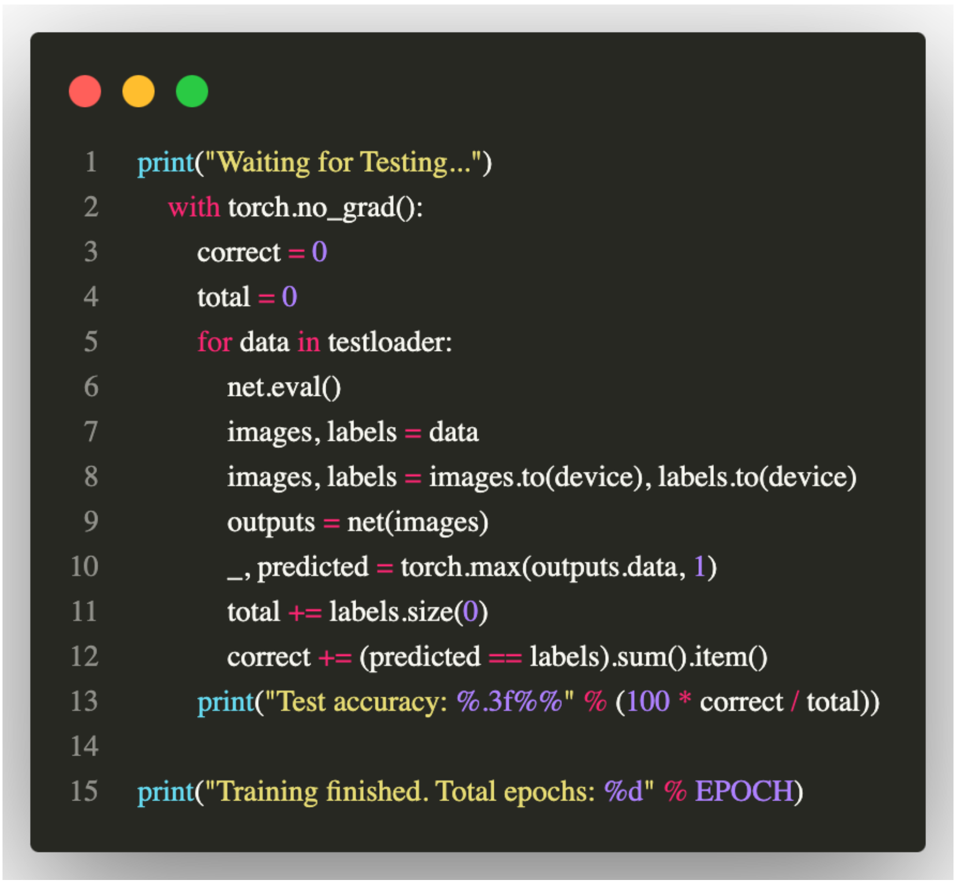
- Inner loop: Iterates through batches in the training dataset.
- Data preparation: Moves inputs and labels to the correct device
- Gradient reset: Clears existing gradients before new calculation
- Forward pass: Runs input through the network
- Loss calculation: Compares model output to ground truth

- Backward pass: Computes gradients
- Parameter update: Adjusts model parameters based on gradients
- Metrics calculation: Tracks total loss, correct predictions, and total samples
- Progress printing: Displays loss and accuracy for each batch



```python
for i, data in enumerate(trainloader, 0):
        # Prepare dataset
        length = len(trainloader)
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward & backward
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()

        # Optimize
        optimizer.step()

        # Print loss and accuracy per batch
        sum_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += predicted.eq(labels.data).cpu().sum()
        print(
            "[epoch:%d, iter:%d] Loss: %.03f | Acc: %.3f%% "
            % (
                epoch + 1,
                (i + 1 + epoch * length),
                sum_loss / (i + 1),
                100.0 * correct / total,
            )
        )
```

5 Testing

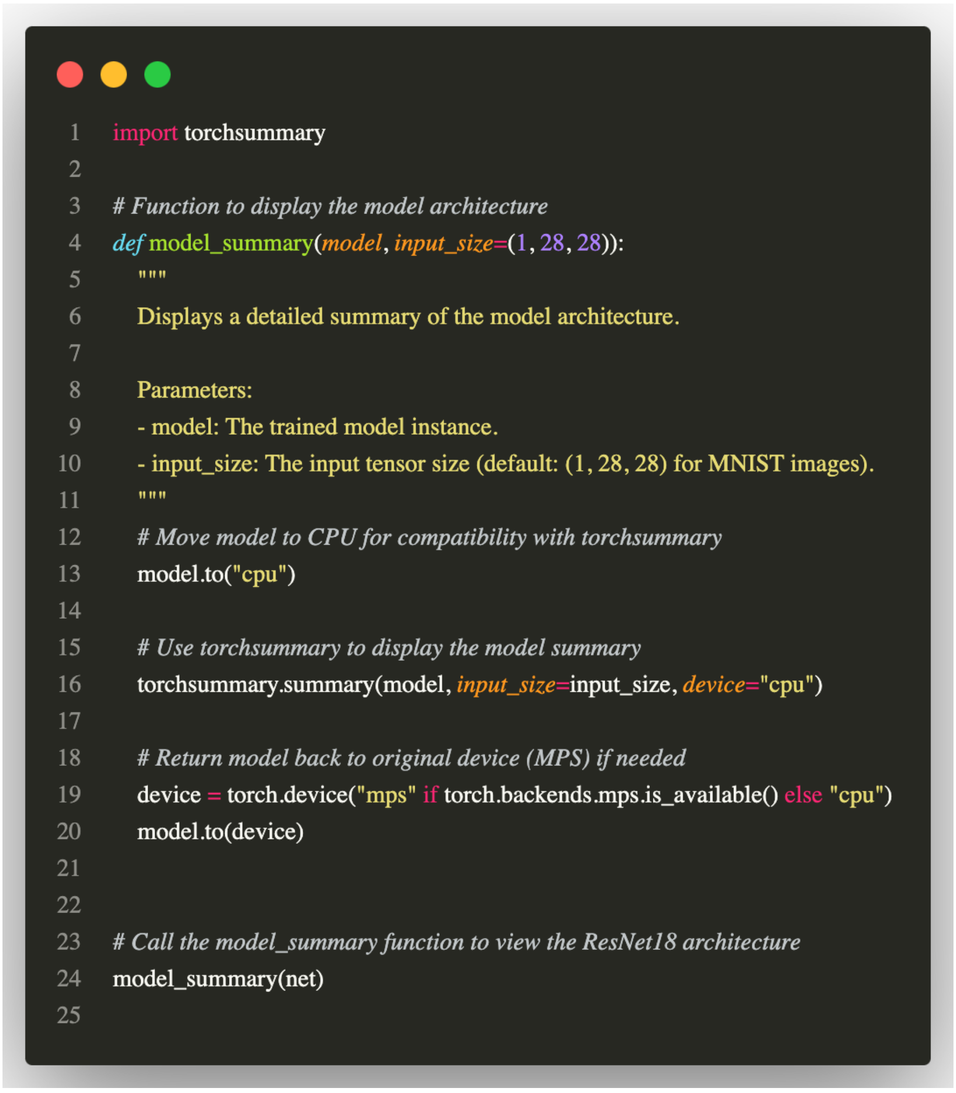


```
print("Waiting for Testing...")
    with torch.no_grad():
        correct = 0
        total = 0
        for data in testloader:
            net.eval()
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        print("Test accuracy: %.3f%%" % (100 * correct / total))

print("Training finished. Total epochs: %d" % EPOCH)
```

- `torch.no_grad()`: Disables gradient calculation during evaluation
- Outer loop: Iterates through test dataset
- `net.eval()`: Sets model to evaluation mode
- Metrics calculation: Counts correct predictions and total samples
- Accuracy calculation: Computes percentage of correct predictions
- Prints completion message with total number of epochs trained

6 Summary Model

This code below defines a function called model_summary that generates a detailed summary of a PyTorch neural network model. The function is designed to provide comprehensive information about the model architecture, including layer types, input/output shapes, and the number of parameters in each layer.



```python
import torchsummary

# Function to display the model architecture
def model_summary(model, input_size=(1, 28, 28)):
    """
    Displays a detailed summary of the model architecture.

    Parameters:
    - model: The trained model instance.
    - input_size: The input tensor size (default: (1, 28, 28) for MNIST images).
    """
    # Move model to CPU for compatibility with torchsummary
    model.to("cpu")

    # Use torchsummary to display the model summary
    torchsummary.summary(model, input_size=input_size, device="cpu")

    # Return model back to original device (MPS) if needed
    device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")
    model.to(device)


# Call the model_summary function to view the ResNet18 architecture
model_summary(net)
```

Key points about this code:

- It uses the `torchsummary` library, which is similar to Keras' `model.summary()` method.
- The function takes two parameters: the model instance and an optional input size (defaulting to MNIST image dimensions).
- It temporarily moves the model to CPU for compatibility with `torchsummary`
- The summary includes information such as layer types, output shapes, number of parameters, and total parameter count.
- After generating the summary, it moves the model back to its original device (CPU or MPS).
- The function is called at the end with the trained model `net`.

Purpose and importance:

- Debugging: Helps identify issues with the model architecture.

- Optimization: Provides insights into the number of parameters and computational complexity.
- Documentation: Serves as a quick reference for the model architecture.
- Understanding: Allows developers to quickly grasp the structure of complex neural networks.

Overall Why i need to add summary of model? is crucial for understanding the model's architecture after training, making it easier to analyze its performance and make informed decisions about further improvements or modifications.