

# بینایی کامپیوتر فصل دوم

## LENET IMPLEMENTATION IN PYTORCH

محمدجواد فدائی اسلام

اردیبهشت ۱۴۰۵

# Import and Hyperparameters

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Device configuration
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# Hyperparameters
batch_size = 64
learning_rate = 0.001
epochs = 5

# Transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

# MNIST dataset

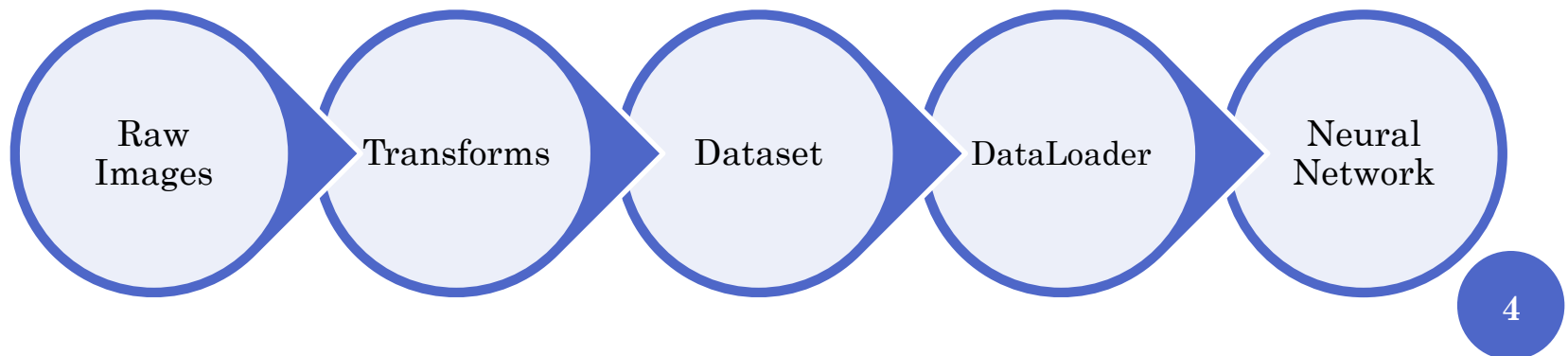
```
# MNIST dataset
train_dataset = datasets.MNIST(
    root="./data",
    train=True,
    transform=transform,
    download=True
)

test_dataset = datasets.MNIST(
    root="./data",
    train=False,
    transform=transform,
    download=True
)

# Data loaders
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

# Data

- In PyTorch, the MNIST pipeline usually has three important parts:
- **Dataset**
- **Transforms**
- **DataLoader**



# Dataset

```
from torchvision import datasets, transforms

# MNIST dataset
train_dataset = datasets.MNIST(
    root="./data",
    train=True,
    transform=transform,
    download=True
)
```

## **MNIST contains**

- 70,000 handwritten digit images

- Size: 28×28 grayscale

- Labels: digits 0–9

## **transform**

- Applies preprocessing to every image before returning it.

- Without transforms:

  - image is a PIL image

- With transforms:

  - image becomes normalized tensor ready for neural network

## **download=True**

- Automatically downloads dataset if missing. Very convenient for first run.

# Dataset-Transform

- Neural networks do not understand images directly.
- Original MNIST image:
  - pixel values: 0–255
  - datatype: image format
- Neural network expects:
  - tensors
  - floating-point values
  - normalized data
- Transforms handle this conversion.

# Dataset-Transform ...

## ○ **transforms.Compose**

- Chains multiple transforms together.

## ○ **transforms.ToTensor()**

- Converts image into PyTorch tensor. (PIL Image -> Tensor)
- Rescales pixel values: (0-255 -> 0.0-1.0)
- **Shape Conversion**
- MNIST image:  $28 \times 28$
- Tensor shape: [1, 28, 28]
  - Where:
  - 1 = grayscale channel
  - 28x28 = image size

## ○ **transforms.Normalize**

- Standardizes data distribution.
- Normalization helps:
  - faster training, more stable gradients, better convergence, improved accuracy
  - Without normalization: training may be slower or unstable

# Dataset-Transform ...

## ○ Common Extra Transforms

### ○ Resize

- `transforms.Resize((32, 32))`
- Changes image size.

### ○ Random Rotation

- `transforms.RandomRotation(10)`
- Useful for Data augmentation and generalization.

### ○ Random Horizontal Flip

- `transforms.RandomHorizontalFlip()`
- Useful for natural images. Not ideal for digits.

# DataLoader

- Dataset gives ONE sample at a time.
- DataLoader creates batches.
- Neural networks train better using batches (faster, GPU efficient, smoother optimization)
- Parameters
  - **batch\_size=64**
    - Each training step uses 64 images.
    - Tensor shape becomes: [64, 1, 28, 28]
  - **shuffle=True**
    - Randomizes image order every epoch.

# CNN Model

```
# CNN Model
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )

        self.fc_layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(128, 10)
        )
```



```

# Initialize model
model = CNN().to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for epoch in range(epochs):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    running_loss += loss.item()

avg_loss = running_loss / len(train_loader)
print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")

```

# Initialize Loss Train Loop

# Evaluation and Save model

```
# Evaluation
model.eval()
correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")

# Save model
torch.save(model.state_dict(), "mnist_cnn.pth")
print("Model saved as mnist_cnn.pth")
```

# Model Architecture

- The CNN contains:
  - Two convolution layers
  - ReLU activations
  - Max pooling
  - Fully connected layers
  - Dropout regularization
- Input image size:
  - $1 \times 28 \times 28$  (grayscale)
- Output:
  - 10 classes (digits 0–9)

```

# Initialize model
model = CNN().to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters())

# Training loop
for epoch in range(epochs):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    running_loss += loss.item()

avg_loss = running_loss / len(train_loader)
print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss}")

```

# Training Steps

- Neural network training is basically:
- 1. Predict
- 2. Measure error
- 3. Compute gradients
- 4. Update weights
- 5. Repeat

# Training Steps

- `Loss.backward()`
  - This computes gradients using backpropagation.
  - for every parameter.  $\partial Loss / \partial Weight$
  - These gradients are stored in: `parameter.grad`
- `optimizer.step()`
  - Uses gradients to update parameters.
  - Without this line: model never learns and weights never change
- `Optimizer.zero_grad()`
  - PyTorch accumulates gradients by default.
  - Without clearing: `new_gradient += old_gradient`

# COMPARING OPTIMIZERS

| Optimizer      | Speed  | Stability             | Common Use         |
|----------------|--------|-----------------------|--------------------|
| SGD            | Medium | Medium                | Classic ML         |
| SGD + Momentum | Faster | Better                | CNNs               |
| Adam           | Fast   | Very Stable           | Most deep learning |
| AdamW          | Fast   | Better regularization | Transformers       |

# Run the Script

```
python mnist_train.py
```