Course 1

Deep Learning Fundamentals

Comprehensive Course Notes

Module 1: Neural Network Basics

Module 2: Image Classification with MNIST

Module 3: Data Management & Pipelines

Module 4: CNNs & Model Architecture

PyTorch Deep Learning Specialization

January 20, 2026

Contents

1 Module 1: Neural Network Basics

1.1 Introduction to Neural Networks

Neural networks are computational models inspired by biological neurons. At their core, they learn patterns from data to make predictions.

The Machine Learning Pipeline

1. **Data Ingestion:** Gathering raw data from sources
2. **Data Preparation:** Cleaning, normalizing, and formatting
3. **Model Building:** Defining the architecture
4. **Training:** Learning patterns from data
5. **Evaluation:** Testing on unseen data
6. **Deployment:** Using the model in production

1.2 The Single Neuron Model

A single neuron implements a simple linear equation:

Linear Equation

$$\text{Output} = W \times \text{Input} + B$$

Where:

- W = Weight (slope of the line)
- B = Bias (y-intercept)

Listing 1: Creating a Simple Linear Model

```
import torch
import torch.nn as nn

# Single neuron: 1 input -> 1 output
model = nn.Sequential(nn.Linear(1, 1))

# Access learned parameters
layer = model[0]
weight = layer.weight.data  # W
bias = layer.bias.data      # B
```

1.3 Tensors: The Foundation of PyTorch

What are Tensors?

Tensors are multi-dimensional arrays optimized for:

- **GPU acceleration** - Fast parallel computation
- **Automatic differentiation** - Computing gradients automatically
- **Broadcasting** - Operations between different shapes

1.3.1 Creating Tensors

Listing 2: Tensor Creation Methods

```
import torch
import numpy as np

# From Python lists
x = torch.tensor([1, 2, 3])
x = torch.tensor([[1.0], [2.0], [3.0]], dtype=torch.float32)

# From NumPy arrays
numpy_arr = np.array([1, 2, 3])
tensor = torch.from_numpy(numpy_arr)

# Predefined values
zeros = torch.zeros(2, 3)       # 2x3 tensor of zeros
ones = torch.ones(2, 3)         # 2x3 tensor of ones
random = torch.rand(2, 3)       # Random values [0, 1)

# Sequences
range_t = torch.arange(0, 10, step=2)  # [0, 2, 4, 6, 8]
```

1.3.2 Tensor Properties

Listing 3: Essential Tensor Properties

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)

print(x.shape)     # torch.Size([2, 3])
print(x.dtype)     # torch.float32
print(x.device)    # cpu or cuda
```

1.3.3 Reshaping Operations

Operation	Description
`x.shape`	Get tensor dimensions
`x.unsqueeze(dim)`	Add dimension at position dim
`x.squeeze()`	Remove all size-1 dimensions
`x.reshape(shape)`	Change to new shape
`x.transpose(d1, d2)`	Swap two dimensions
`torch.cat([a, b], dim)`	Concatenate tensors

Listing 4: Reshaping Examples

```
x = torch.tensor([[1, 2, 3]])  # Shape: [1, 3]
```

```
# Add batch dimension
x_batch = x.unsqueeze(0)        # Shape: [1, 1, 3]

# Remove size-1 dimensions
x_squeezed = x_batch.squeeze() # Shape: [3]

# Reshape
x = torch.arange(6)             # [0, 1, 2, 3, 4, 5]
x_2d = x.reshape(2, 3)          # [[0,1,2], [3,4,5]]
```

1.3.4 Indexing and Slicing

Listing 5: Tensor Indexing

```
x = torch.tensor([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12]])

# Single element
x[1, 2]           # tensor(7)

# Entire row
x[0]              # tensor([1, 2, 3, 4])

# Entire column
x[:, 1]           # tensor([2, 6, 10])

# Slicing
x[0:2, 1:3]       # First 2 rows, columns 1-2

# Boolean masking
mask = x > 6
x[mask]           # tensor([7, 8, 9, 10, 11, 12])

# Extract scalar value
scalar = x[0, 0].item()  # Python number: 1
```

1.3.5 Mathematical Operations

Listing 6: Tensor Math Operations

```
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

# Element-wise operations
a + b             # tensor([5, 7, 9])
a * b             # tensor([4, 10, 18])
a / b             # Element-wise division

# Dot product
torch.matmul(a, b)  # tensor(32) = 1*4 + 2*5 + 3*6

# Broadcasting (scalar with tensor)
a + 5             # tensor([6, 7, 8])

# Statistics
data = torch.tensor([1.0, 2.0, 3.0, 4.0, 5.0])
```

```
data.mean()        # tensor(3.)
data.std()         # tensor(1.5811)
```

1.4 The Training Loop

Five Essential Steps

Every training iteration follows these steps:

1. `optimizer.zero_grad()` - Clear previous gradients
2. `outputs = model(inputs)` - Forward pass
3. `loss = loss_fn(outputs, targets)` - Calculate error
4. `loss.backward()` - Backpropagation
5. `optimizer.step()` - Update weights

Listing 7: Complete Training Loop

```
import torch.optim as optim

# Setup
model = nn.Sequential(nn.Linear(1, 1))
loss_function = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Training data
distances = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
times = torch.tensor([[7.0], [12.0], [17.0], [22.0]])

# Training loop
for epoch in range(500):
    optimizer.zero_grad()              # Step 1
    outputs = model(distances)         # Step 2
    loss = loss_function(outputs, times)  # Step 3
    loss.backward()                    # Step 4
    optimizer.step()                   # Step 5

    if (epoch + 1) % 100 == 0:
        print(f"Epoch {epoch+1}: Loss = {loss.item():.4f}")
```

1.5 Activation Functions

Why Non-Linear Activation?

Without activation functions, stacking linear layers produces another linear function. Non-linear activations enable learning complex patterns.

ReLU (Rectified Linear Unit)

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Properties:

- Simple and computationally efficient
- Helps avoid vanishing gradient problem
- Most popular activation for hidden layers

Listing 8: Non-Linear Model with ReLU

```
# Model with hidden layer and ReLU activation
model = nn.Sequential(
    nn.Linear(1, 3),     # Input -> 3 hidden neurons
    nn.ReLU(),           # Non-linear activation
    nn.Linear(3, 1)      # Hidden -> Output
)
```

1.6 Data Normalization

Why Normalize?

Normalization (standardization) helps training by:

- Centering data around zero
- Scaling to consistent range
- Improving gradient flow
- Speeding up convergence

Z-Score Normalization

$$x_{\text{normalized}} = \frac{x - \mu}{\sigma}$$

Where μ = mean, σ = standard deviation

Listing 9: Normalizing Data

```
# Calculate statistics
mean = data.mean()
std = data.std()

# Normalize
data_normalized = (data - mean) / std

# De-normalize predictions
prediction_original = (prediction_norm * std) + mean
```

2 Module 2: Image Classification

2.1 The MNIST Dataset

MNIST is a classic benchmark dataset containing:

- 60,000 training images + 10,000 test images
- 28 × 28 grayscale images
- 10 classes (digits 0-9)

2.2 Image Transformations

Listing 10: Standard Image Preprocessing

```
import torchvision.transforms as transforms

# Transformation pipeline
transform = transforms.Compose([
    transforms.ToTensor(),  # PIL -> Tensor, scale to [0,1]
    transforms.Normalize((0.1307,), (0.3081,))  # MNIST stats
])
```

ToTensor() Does Three Things

1. Converts PIL Image to PyTorch Tensor
2. Scales pixel values from [0, 255] to [0, 1]
3. Rearranges dimensions to [C, H, W] (Channels, Height, Width)

2.3 Loading Data with DataLoader

Listing 11: Complete Data Pipeline

```
import torchvision
from torch.utils.data import DataLoader

# Load datasets
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

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

# Create DataLoaders
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
```

```
    shuffle=True     # Shuffle for training
)

test_loader = DataLoader(
    test_dataset,
    batch_size=1000,
    shuffle=False    # No shuffle for testing
)
```

DataLoader Best Practices

- **Training:** shuffle=True (prevents order-dependent learning)
- **Testing:** shuffle=False (reproducible evaluation)
- Larger test batch size is OK (no gradient computation)

2.4 Building a Custom Model with nn.Module

Listing 12: Custom Neural Network Class

```
class SimpleMNISTDNN(nn.Module):
    def __init__(self):
        super(SimpleMNISTDNN, self).__init__()

        # Flatten 28x28 image to 784 vector
        self.flatten = nn.Flatten()

        # Define layers
        self.layers = nn.Sequential(
            nn.Linear(784, 128),  # 28*28 = 784 inputs
            nn.ReLU(),
            nn.Linear(128, 10)    # 10 output classes
        )

    def forward(self, x):
        x = self.flatten(x)  # [batch, 1, 28, 28] -> [batch, 784]
        x = self.layers(x)
        return x
```

2.5 Loss Functions for Classification

Cross-Entropy Loss

For multi-class classification:

$$\mathcal{L}_{CE} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

Where:

- C = number of classes
- y_c = true label (one-hot encoded)
- $\hat{y}_c$ = predicted probability for class c

Listing 13: Classification Setup

```
model = SimpleMNISTDNN()
loss_function = nn.CrossEntropyLoss()  # For classification
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

2.6 Device Management (CPU/GPU)

Listing 14: Device Configuration

```
# Automatic device selection
if torch.cuda.is_available():
    device = torch.device("cuda")
elif torch.backends.mps.is_available():
    device = torch.device("mps")  # Apple Silicon
else:
    device = torch.device("cpu")

# Move model to device
model = model.to(device)

# Move data to device (in training loop)
inputs, targets = inputs.to(device), targets.to(device)
```

2.7 Training and Evaluation Modes

Listing 15: Model Modes

```
# Training mode (enables dropout, batch norm training behavior)
model.train()

# Evaluation mode (disables dropout, uses running stats)
model.eval()

# Disable gradient computation for inference
with torch.no_grad():
    outputs = model(inputs)
```

Always Use These!

- Call `model.train()` before training loops
- Call `model.eval()` before validation/testing
- Use `torch.no_grad()` during inference to save memory

2.8 Complete Training Function

Listing 16: Training Epoch Function

```
def train_epoch(model, loss_fn, optimizer, train_loader, device):
    model = model.to(device)
    model.train()

    total_loss = 0.0
    correct = 0
    total = 0
```

```
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, targets)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = outputs.max(1)
        correct += predicted.eq(targets).sum().item()
        total += targets.size(0)

    avg_loss = total_loss / len(train_loader)
    accuracy = 100. * correct / total
    return avg_loss, accuracy
```

2.9 Evaluation Function

Listing 17: Evaluation Function

```
def evaluate(model, loss_fn, test_loader, device):
    model.eval()

    total_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # No gradients needed
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)

            outputs = model(inputs)
            loss = loss_fn(outputs, targets)

            total_loss += loss.item()
            _, predicted = outputs.max(1)
            correct += predicted.eq(targets).sum().item()
            total += targets.size(0)

    avg_loss = total_loss / len(test_loader)
    accuracy = 100. * correct / total
    return avg_loss, accuracy
```

3 Module 3: Data Management & Pipelines

3.1 Custom Dataset Class

Three Required Methods

Every custom Dataset must implement:

1. `__init__`: Initialize paths, load metadata
2. `__len__`: Return total number of samples
3. `__getitem__`: Return one sample by index

Listing 18: Custom Dataset Template

```
from torch.utils.data import Dataset
from PIL import Image
import os

class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform

        # Load metadata (paths, labels) - NOT images!
        self.image_paths = [...]  # List of paths
        self.labels = [...]       # List of labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        # Lazy loading: load image only when accessed
        img_path = self.image_paths[idx]
        image = Image.open(img_path).convert('RGB')
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label
```

Lazy Loading

Don't load all images in `__init__`! Store only paths and load images in `__getitem__`. This prevents memory overflow with large datasets.

3.2 Data Augmentation

Why Augment?

Data augmentation artificially increases dataset diversity by applying random transformations:

- Prevents overfitting

- Improves model generalization
- Simulates real-world variations

Listing 19: Augmentation Transforms

```
# Training transforms (with augmentation)
train_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomResizedCrop(224),    # Random crop + scale
    transforms.RandomHorizontalFlip(),     # 50% chance flip
    transforms.RandomRotation(15),         # +/- 15 degrees
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])

# Validation transforms (NO augmentation)
val_transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(224),            # Consistent crop
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

Important Rule

Apply augmentation ONLY to training data, NOT to validation or test data. Validation needs consistent data to reliably measure improvements.

3.3 Common Transforms Reference

Transform	Purpose
`ToTensor()`	PIL → Tensor, scale to [0,1]
`Normalize(mean, std)`	Standardize values
`Resize(size)`	Resize to fixed dimensions
`CenterCrop(size)`	Crop from center
`RandomCrop(size)`	Crop from random location
`RandomResizedCrop(size)`	Random crop + resize (scale variation)
`RandomHorizontalFlip()`	50% horizontal flip
`RandomRotation(degrees)`	Random rotation
`ColorJitter(...)`	Random brightness/contrast

3.4 Dataset Splitting

Listing 20: Splitting Dataset

```
from torch.utils.data import random_split

# Split ratios
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size

# Random split
train_dataset, val_dataset = random_split(
```

```
    dataset,
    [train_size, val_size],
    generator=torch.Generator().manual_seed(42)  # Reproducible
)
```

3.5 DataLoader Configuration

Listing 21: DataLoader Options

```
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,            # Shuffle each epoch
    num_workers=4,           # Parallel loading
    pin_memory=True,         # Faster GPU transfer
    drop_last=True           # Drop incomplete batches
)

val_loader = DataLoader(
    val_dataset,
    batch_size=128,          # Larger OK for validation
    shuffle=False,           # No shuffle for validation
    num_workers=4,
    pin_memory=True
)
```

3.6 Robust Error Handling

Listing 22: Handling Corrupted Images

```
def __getitem__(self, idx):
    try:
        image = Image.open(self.image_paths[idx]).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return image, self.labels[idx]
    except Exception as e:
        print(f"Error loading {self.image_paths[idx]}: {e}")
        # Return a different valid sample
        return self.__getitem__((idx + 1) % len(self))
```

3.7 ImageNet Normalization Constants

Standard Values for Pretrained Models

When using pretrained models (ResNet, VGG, etc.), use ImageNet statistics:

```
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
```

4 Module 4: CNNs & Model Architecture

4.1 Why CNNs for Images?

Limitations of Fully Connected Networks

- Treat pixels independently (no spatial awareness)
- Too many parameters for high-resolution images
- Don't leverage local patterns (edges, textures)

CNNs solve this by:

- Learning local patterns through filters
- Sharing weights across spatial locations
- Building hierarchical feature representations

4.2 Convolutional Layer

Convolution Operation

A filter slides across the image, computing element-wise multiplication and sum at each position:

$$\text{output}[i,j] = \sum_m \sum_n \text{input}[i+m, j+n] \times \text{filter}[m,n]$$

Listing 23: Conv2d Parameters

```
nn.Conv2d(
    in_channels=3,      # Input channels (3 for RGB)
    out_channels=32,    # Number of filters to learn
    kernel_size=3,      # Filter size (3x3)
    stride=1,           # Step size
    padding=1           # Border padding (maintains dimensions)
)
```

Padding to Preserve Size

With kernel_size=3 and padding=1, output size = input size.
Formula: $\text{output\_size} = \frac{\text{input\_size} - \text{kernel\_size} + 2 \times \text{padding}}{\text{stride}} + 1$

4.3 Pooling Layer

Max Pooling

Reduces spatial dimensions by keeping the maximum value in each window:

- Reduces computation
- Provides translation invariance
- Prevents overfitting

Listing 24: Max Pooling

```
nn.MaxPool2d(
    kernel_size=2,   # 2x2 window
    stride=2         # Halves dimensions
)

# Example: 28x28 -> MaxPool(2,2) -> 14x14
```

4.4 Complete CNN Architecture

Listing 25: CNN for Image Classification

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super(SimpleCNN, self).__init__()

        # Conv Block 1: 32x32x3 -> 16x16x32
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(2, 2)

        # Conv Block 2: 16x16x32 -> 8x8x64
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(2, 2)

        # Conv Block 3: 8x8x64 -> 4x4x128
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(2, 2)

        # Classifier
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes)

    def forward(self, x):
        # Convolutional blocks
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.pool3(self.relu3(self.conv3(x)))

        # Classifier
        x = self.flatten(x)
        x = self.relu4(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

4.5 nn.Sequential for Cleaner Code

Listing 26: Using nn.Sequential

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
```

```
        super().__init__()

        self.features = nn.Sequential(
            # Block 1
            nn.Conv2d(3, 32, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

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

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 8 * 8, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

4.6 Dropout Regularization

How Dropout Works

During training, randomly sets a fraction of activations to zero:

- Prevents neurons from co-adapting
- Forces redundant representations
- Reduces overfitting

Important: Dropout is automatically disabled during `model.eval()`

4.7 Model Inspection

Listing 27: Inspecting Model Parameters

```
model = SimpleCNN(num_classes=10)

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
trainable = sum(p.numel() for p in model.parameters()
                if p.requires_grad)

print(f"Total parameters: {total_params:,}")
print(f"Trainable: {trainable:,}")

# View architecture
print(model)
```

```python
# Iterate through layers
for name, module in model.named_modules():
    print(name, module)
```

4.8 Feature Map Dimensions

Output Size Calculation

For convolution and pooling:

$$\text{Output} = \left\lfloor \frac{\text{Input} - \text{Kernel} + 2 \times \text{Padding}}{\text{Stride}} \right\rfloor + 1$$

Example with 32×32 input:

- After Conv(k=3, p=1, s=1): $\frac{32-3+2}{1} + 1 = 32$ (same)
- After MaxPool(k=2, s=2): $\frac{32-2}{2} + 1 = 16$ (halved)

4.9 Diagnosing Training Issues

Overfitting Signs

- Training accuracy keeps improving
- Validation accuracy plateaus or decreases
- Large gap between training and validation loss

Solutions:

- Add dropout layers
- Use data augmentation
- Reduce model complexity
- Early stopping
- Add regularization (L2/weight decay)

5 Quiz Review: Key Concepts

5.1 Quiz 1: PyTorch Basics

1. **Loss function** measures error between predictions and actual values
2. **ReLU** enables learning non-linear patterns
3. **Epoch** = one complete pass through training data
4. **loss.backward()** calculates gradients via backpropagation
5. **dtype** specifies the type of numbers in a tensor
6. **Broadcasting** expands tensors to compatible shapes
7. Operations are **element-wise** by default
8. Tensors are **optimized for mathematical operations**
9. `squeeze()` removes size-1 dimensions

5.2 Quiz 2: Training & Evaluation

1. **Normalize** data for more effective training
2. Use `model(x)` not `model.forward(x)` for inference
3. MSE **squares errors** to prevent cancellation and penalize large errors
4. Evaluate on **test set** to measure generalization
5. Standard device setup: `device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')`
6. **Flatten** converts 2D images to 1D vectors
7. **Adam** is generally recommended as default optimizer
8. Shuffle training data, not test data
9. `torch.no_grad()` disables gradient tracking
10. `model.eval()` changes layer behavior for evaluation

5.3 Quiz 3: Data Pipelines

1. Data handling dramatically affects results
2. Batching addresses **efficiency** problems
3. Store paths in `__init__`, load images in `__getitem__`
4. Fix 1-based labels in `__init__`
5. Normalize spreads values for gradient-based learning
6. **Normalize()** is separate from ToTensor()
7. Don't read large files inside `__getitem__`
8. DataLoader returns (batch_images, batch_labels)
9. Augment training only, not validation
10. Handle corrupted images gracefully

5.4 Quiz 4: CNNs

1. Learned filters discover **task-specific patterns**
2. Convolution: element-wise multiply, then sum
3. Two MaxPool(2,2) on 28×28 → 7×7
4. out_channels=32 means 32 different filters
5. nn.Sequential doesn't support conditionals/loops
6. Dynamic graphs built fresh each forward pass
7. `__init__` defines layers, `forward` defines data flow
8. Sequential chains layer calls automatically
9. Count params: `sum(p.numel() for p in model.parameters())`
10. `children()` = direct children, `modules()` = all nested

6 Quick Reference

6.1 Essential Imports

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

6.2 Model Template

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        # Define layers here

    def forward(self, x):
        # Define forward pass here
        return x
```

6.3 Training Template

```
model = MyModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(num_epochs):
    model.train()
    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

    # Evaluate
    model.eval()
    with torch.no_grad():
        # Validation code
```

6.4 Common Layer Sizes

Layer	Input	Operation	Output
Conv2d(3,32,3,p=1)	32×32×3	3×3 conv	32×32×32
MaxPool2d(2,2)	32×32×32	2×2 pool	16×16×32
Conv2d(32,64,3,p=1)	16×16×32	3×3 conv	16×16×64
MaxPool2d(2,2)	16×16×64	2×2 pool	8×8×64
Flatten()	8×8×64	flatten	4096
Linear(4096, 10)	4096	FC	10