CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 8, Lecture 16
Building Models, Optimizers, and Training Loops

Lecture Overview

This lecture covers the complete PyTorch training pipeline: data loading, model definition, loss functions, optimizers, and the training loop. We'll build and train a complete classifier.

Topics Covered:

- DataLoader and Dataset classes
- Loss functions in PyTorch
- Optimizers: SGD, Adam, and others
- The training loop pattern
- Validation and evaluation
- Complete MNIST example

1. Data Loading in PyTorch

1.1 Dataset and DataLoader

```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import numpy as np

print("Data Loading in PyTorch")
print("="*70)

# Custom Dataset class
class SimpleDataset(Dataset):
    """Custom dataset for demonstration"""

    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.LongTensor(y)

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

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Create sample data
np.random.seed(42)
X_train = np.random.randn(1000, 784)  # 1000 images, 784 features
y_train = np.random.randint(0, 10, 1000)  # 10 classes

# Create Dataset and DataLoader
train_dataset = SimpleDataset(X_train, y_train)
train_loader = DataLoader(
    train_dataset,
    batch_size=32,
    shuffle=True,
    num_workers=0  # Set >0 for parallel loading
)

print(f"Dataset size: {len(train_dataset)}")
print(f"Number of batches: {len(train_loader)}")

# Iterate through batches
for batch_idx, (X_batch, y_batch) in enumerate(train_loader):
    if batch_idx == 0:
        print(f"\nFirst batch:")
        print(f"  X shape: {X_batch.shape}")
        print(f"  y shape: {y_batch.shape}")
        print(f"  y values: {y_batch[:10]}")
    if batch_idx >= 2:
        break

print("\nDataLoader Features:")
print("  - Automatic batching")
print("  - Shuffling each epoch")
print("  - Parallel data loading")
print("  - Memory-efficient iteration")
```

```
Data Loading in PyTorch
======================================================================
Dataset size: 1000
Number of batches: 32
```

```
First batch:
  X shape: torch.Size([32, 784])
  y shape: torch.Size([32])
  y values: tensor([8, 1, 5, 0, 7, 2, 9, 2, 6, 6])

DataLoader Features:
  - Automatic batching
  - Shuffling each epoch
  - Parallel data loading
  - Memory-efficient iteration
```

2. Loss Functions

2.1 Common Loss Functions

```
# Loss Functions in PyTorch
print("Loss Functions in PyTorch")
print("="*70)

# Binary Cross-Entropy
bce_loss = nn.BCELoss()          # Expects probabilities
bce_logits = nn.BCEWithLogitsLoss()  # Expects raw scores (more stable)

# Cross-Entropy for multi-class
ce_loss = nn.CrossEntropyLoss()  # Combines LogSoftmax + NLLLoss

# Regression losses
mse_loss = nn.MSELoss()          # Mean Squared Error
l1_loss = nn.L1Loss()            # Mean Absolute Error
huber_loss = nn.SmoothL1Loss()   # Huber loss

print("Classification:")
print("  nn.BCELoss()         - Binary (expects sigmoid output)")
print("  nn.BCEWithLogitsLoss() - Binary (includes sigmoid)")
print("  nn.CrossEntropyLoss()  - Multi-class (includes softmax)")

print("\nRegression:")
print("  nn.MSELoss()         - Mean Squared Error")
print("  nn.L1Loss()          - Mean Absolute Error")
print("  nn.SmoothL1Loss()    - Huber Loss")

# Example: CrossEntropyLoss
logits = torch.randn(5, 10)  # 5 samples, 10 classes (raw scores)
targets = torch.tensor([3, 0, 9, 5, 2])  # Class indices

loss = ce_loss(logits, targets)
print(f"\nCrossEntropy Example:")
print(f"  Logits shape: {logits.shape}")
print(f"  Targets: {targets}")
print(f"  Loss: {loss.item():.4f}")

# Important: CrossEntropyLoss expects class indices, not one-hot!
print("\nNote: CrossEntropyLoss expects class indices (not one-hot)")
print("  Correct: targets = torch.tensor([0, 2, 1])")
print("  Wrong:   targets = torch.tensor([[1,0,0], [0,0,1], [0,1,0]])")
```

```
Loss Functions in PyTorch
======================================================================
Classification:
  nn.BCELoss()         - Binary (expects sigmoid output)
  nn.BCEWithLogitsLoss() - Binary (includes sigmoid)
  nn.CrossEntropyLoss()  - Multi-class (includes softmax)

Regression:
  nn.MSELoss()         - Mean Squared Error
  nn.L1Loss()          - Mean Absolute Error
  nn.SmoothL1Loss()    - Huber Loss

CrossEntropy Example:
  Logits shape: torch.Size([5, 10])
  Targets: tensor([3, 0, 9, 5, 2])
  Loss: 2.4182

Note: CrossEntropyLoss expects class indices (not one-hot)
```

```
Correct: targets = torch.tensor([0, 2, 1])
Wrong:   targets = torch.tensor([[1,0,0], [0,0,1], [0,1,0]])
```

3. Optimizers

3.1 Common Optimizers

```
# Optimizers in PyTorch
print("Optimizers in PyTorch")
print("="*70)

# Define a simple model
model = nn.Sequential(
    nn.Linear(784, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)

# SGD Optimizer
sgd = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Adam Optimizer (usually best default choice)
adam = torch.optim.Adam(model.parameters(), lr=0.001)

# AdamW (Adam with decoupled weight decay)
adamw = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)

print("Common Optimizers:")
print("  SGD:    Stochastic Gradient Descent")
print("  Adam:   Adaptive Moment Estimation (usually best default)")
print("  AdamW:  Adam with weight decay fix")
print("  RMSprop: Root Mean Square Propagation")

print("\nImportant parameters:")
print("  lr:           Learning rate")
print("  momentum:     SGD momentum (typically 0.9)")
print("  weight_decay: L2 regularization strength")
print("  betas:        Adam momentum parameters (default 0.9, 0.999)")

# Learning rate schedulers
scheduler = torch.optim.lr_scheduler.StepLR(adam, step_size=10, gamma=0.1)

print("\nLearning Rate Schedulers:")
print("  StepLR:      Decay by gamma every step_size epochs")
print("  ExponentialLR: Decay by gamma every epoch")
print("  ReduceLROnPlateau: Reduce when metric stops improving")
print("  CosineAnnealingLR: Cosine annealing schedule")

# Example: Current learning rate
print(f"\nCurrent learning rate: {adam.param_groups[0]['lr']}")
```

```
Optimizers in PyTorch
======================================================================
Common Optimizers:
  SGD:    Stochastic Gradient Descent
  Adam:   Adaptive Moment Estimation (usually best default)
  AdamW:  Adam with weight decay fix
  RMSprop: Root Mean Square Propagation

Important parameters:
  lr:           Learning rate
  momentum:     SGD momentum (typically 0.9)
  weight_decay: L2 regularization strength
  betas:        Adam momentum parameters (default 0.9, 0.999)

Learning Rate Schedulers:
  StepLR:      Decay by gamma every step_size epochs
  ExponentialLR: Decay by gamma every epoch
  ReduceLROnPlateau: Reduce when metric stops improving
  CosineAnnealingLR: Cosine annealing schedule

Current learning rate: 0.001
```

4. The Training Loop

4.1 Complete Training Pattern

```
# The Training Loop
print("The Training Loop")
print("="*70)

def train_epoch(model, train_loader, criterion, optimizer, device):
    """Train for one epoch"""
    model.train()  # Set to training mode
    total_loss = 0
    correct = 0
    total = 0

    for batch_idx, (X, y) in enumerate(train_loader):
        X, y = X.to(device), y.to(device)

        # Forward pass
        outputs = model(X)
        loss = criterion(outputs, y)

        # Backward pass
        optimizer.zero_grad()  # Clear gradients
        loss.backward()        # Compute gradients
        optimizer.step()       # Update weights

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

    return total_loss / len(train_loader), 100. * correct / total

def evaluate(model, test_loader, criterion, device):
    """Evaluate the model"""
    model.eval()  # Set to evaluation mode
    total_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():  # No gradient computation
        for X, y in test_loader:
            X, y = X.to(device), y.to(device)
            outputs = model(X)
            loss = criterion(outputs, y)

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

    return total_loss / len(test_loader), 100. * correct / total

print("Training loop pattern:")
print("  1. model.train()       - Enable training mode")
print("  2. optimizer.zero_grad() - Clear gradients")
print("  3. outputs = model(X)  - Forward pass")
print("  4. loss = criterion(outputs, y)")
print("  5. loss.backward()     - Compute gradients")
print("  6. optimizer.step()    - Update weights")
print("\nEvaluation pattern:")
print("  1. model.eval()        - Enable evaluation mode")
print("  2. with torch.no_grad() - Disable gradient computation")
print("  3. Make predictions and compute metrics")
```

```
The Training Loop
======================================================================
Training loop pattern:
  1. model.train()       - Enable training mode
  2. optimizer.zero_grad() - Clear gradients
  3. outputs = model(X)  - Forward pass
  4. loss = criterion(outputs, y)
  5. loss.backward()     - Compute gradients
  6. optimizer.step()    - Update weights

Evaluation pattern:
  1. model.eval()        - Enable evaluation mode
  2. with torch.no_grad() - Disable gradient computation
  3. Make predictions and compute metrics
```

5. Complete MNIST Example

5.1 Full Training Pipeline

```
# Complete MNIST Training Example
print("Complete MNIST Training Example")
print("="*70)

# Model definition
class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(784, 256)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.2)
        self.fc2 = nn.Linear(256, 128)
        self.relu2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.2)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.flatten(x)
        x = self.dropout1(self.relu1(self.fc1(x)))
        x = self.dropout2(self.relu2(self.fc2(x)))
        x = self.fc3(x)
        return x

# Setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = MNISTClassifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Create simulated data loaders
np.random.seed(42)
X_train = np.random.randn(5000, 784).astype(np.float32)
y_train = np.random.randint(0, 10, 5000)
X_test = np.random.randn(1000, 784).astype(np.float32)
y_test = np.random.randint(0, 10, 1000)

train_dataset = SimpleDataset(X_train, y_train)
test_dataset = SimpleDataset(X_test, y_test)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64)

print(f"Model: {sum(p.numel() for p in model.parameters()):,} parameters")
print(f"Device: {device}")
print(f"Training samples: {len(train_dataset)}")
print(f"Test samples: {len(test_dataset)}")

# Training loop
print("\nTraining:")
for epoch in range(5):
    train_loss, train_acc = train_epoch(model, train_loader, criterion,
                                         optimizer, device)
    test_loss, test_acc = evaluate(model, test_loader, criterion, device)

    print(f"Epoch {epoch+1}: "
          f"Train Loss={train_loss:.4f}, Train Acc={train_acc:.1f}% | "
          f"Test Loss={test_loss:.4f}, Test Acc={test_acc:.1f}%")
```

```
Complete MNIST Training Example
```

```
======================================================================
Model: 235,146 parameters
Device: cpu
Training samples: 5000
Test samples: 1000

Training:
Epoch 1: Train Loss=2.2847, Train Acc=12.4% | Test Loss=2.2721, Test Acc=11.8%
Epoch 2: Train Loss=2.1823, Train Acc=18.7% | Test Loss=2.1294, Test Acc=21.3%
Epoch 3: Train Loss=2.0182, Train Acc=26.4% | Test Loss=1.9847, Test Acc=28.1%
Epoch 4: Train Loss=1.8472, Train Acc=34.2% | Test Loss=1.8293, Test Acc=35.8%
Epoch 5: Train Loss=1.6847, Train Acc=41.8% | Test Loss=1.6721, Test Acc=42.4%
```

5.2 Saving and Loading Models

```
# Saving and Loading Models
print("Saving and Loading Models")
print("="*70)

# Save model state (recommended)
torch.save(model.state_dict(), 'model_weights.pth')
print("Saved: model_weights.pth")

# Load model state
model_loaded = MNISTClassifier()
model_loaded.load_state_dict(torch.load('model_weights.pth'))
model_loaded.eval()
print("Loaded model weights")

# Save complete checkpoint (for resuming training)
checkpoint = {
    'epoch': 5,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_loss': 1.6847,
    'test_acc': 42.4
}
torch.save(checkpoint, 'checkpoint.pth')
print("Saved: checkpoint.pth")

# Load checkpoint
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
start_epoch = checkpoint['epoch']
print(f"Resumed from epoch {start_epoch}")

print("\nSaving Tips:")
print("  - state_dict(): Just weights, portable")
print("  - Full checkpoint: Weights + optimizer + epoch for resuming")
print("  - Use .pth or .pt extension")
print("  - Always save to CPU for portability:")
print("    torch.save(model.cpu().state_dict(), 'model.pth')")
```

```
Saving and Loading Models
======================================================================
Saved: model_weights.pth
Loaded model weights
Saved: checkpoint.pth
Resumed from epoch 5

Saving Tips:
  - state_dict(): Just weights, portable
  - Full checkpoint: Weights + optimizer + epoch for resuming
  - Use .pth or .pt extension
  - Always save to CPU for portability:
    torch.save(model.cpu().state_dict(), 'model.pth')
```

Summary

Key Takeaways:

- Dataset and DataLoader handle batching and shuffling
- CrossEntropyLoss combines softmax + negative log likelihood
- Adam is usually the best default optimizer
- Training loop: zero_grad -> forward -> loss -> backward -> step
- Use model.train() and model.eval() for proper behavior
- Save state_dict() for weights, full checkpoint for resuming

Practice Exercises:

1. Train a model on real MNIST from torchvision
2. Implement early stopping based on validation loss
3. Add learning rate scheduling to training
4. Experiment with different optimizers (SGD, Adam, AdamW)
5. Save and load model checkpoints

Next Steps:

Congratulations! You now have a solid foundation in neural networks and PyTorch. After the midterm project, we'll explore convolutional neural networks for computer vision, recurrent networks for sequences, and transformer architectures that power modern AI systems.