Reinforcement Learning

Lecture 2: Deep Learning Essentials

Taehoon Kim

Sogang University MIMIC Lab
https://mimic-lab.com

Fall Semester 2025

Learning Objectives

By the end of this lecture, you will:

- Understand tensor operations and automatic differentiation
- Master PyTorch's nn.Module system
- Implement gradient checking for verification
- Build complete training pipelines with proper device handling
- Apply regularization and initialization techniques
- Complete 9 hands-on experiments

Prerequisites

- Python programming experience
- Basic linear algebra (matrix operations)
- Calculus (derivatives and chain rule)

Tensors and PyTorch Fundamentals

The Foundation of Deep Learning

Topics

- What are tensors?
- PyTorch tensor operations
- Device management
- Broadcasting and shape manipulation

What is a Tensor?

Definition: Multidimensional array with hardware acceleration support

- **Scalar:** 0D tensor (single number)
- **Vector:** 1D tensor [n]
- **Matrix:** 2D tensor [m, n]
- **3D Tensor:** [batch, height, width]
- **4D Tensor:** [batch, channels, height, width]

Key Properties

- Device placement (CPU, CUDA, MPS)
- Data type (float32, float64, int64, etc.)
- Gradient tracking (requires_grad)

Creating Tensors

```python
import torch

# From data
x = torch.tensor([1, 2, 3])           # From list
y = torch.tensor([[1, 2], [3, 4]])    # 2D tensor

# Random tensors
z = torch.randn(3, 4)    # Normal distribution
u = torch.rand(2, 3)     # Uniform [0, 1)

# Special tensors
zeros = torch.zeros(3, 3)
ones = torch.ones(2, 4)
eye = torch.eye(3)       # Identity matrix

# With specific dtype and device
x = torch.tensor([1.0], dtype=torch.float64,
                 device='cuda' if torch.cuda.is_available()
                 else 'cpu')
```

Tensor Operations

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

c = a + b          # Element-wise addition
d = a * b          # Element-wise multiplication
e = a @ b          # Dot product (same as torch.dot)

# Matrix operations
A = torch.randn(3, 4)
B = torch.randn(4, 5)
C = A @ B          # Matrix multiplication [3, 5]

# In-place operations (use with caution!)
a.add_(1)          # Modifies a in-place
a.mul_(2)          # Dangerous with autograd!
```

Device Management

```python
# Proper device selection (CUDA > MPS > CPU)
device = torch.device(
    'cuda' if torch.cuda.is_available()
    else 'mps' if hasattr(torch.backends, 'mps')
                  and torch.backends.mps.is_available()
    else 'cpu'
)

# Moving tensors between devices
x = torch.randn(3, 4)
x = x.to(device)           # Move to selected device
x = x.cpu()                # Back to CPU

# Creating directly on device
y = torch.randn(3, 4, device=device)

# Check device
print(x.device)            # cuda:0 or cpu
print(x.is_cuda)          # True/False
```

Broadcasting Rules

PyTorch automatically expands tensors for element-wise operations:

```
# Scalar broadcast
a = torch.randn(3, 4)
b = 2.0
c = a * b              # b broadcasts to [3, 4]

# Vector broadcast
a = torch.randn(3, 4)  # [3, 4]
b = torch.randn(4)     # [4] -> broadcasts to [3, 4]
c = a + b              # OK

# Matrix broadcast
a = torch.randn(3, 1, 4)  # [3, 1, 4]
b = torch.randn(1, 5, 4)  # [1, 5, 4]
c = a + b                  # [3, 5, 4]
```

Rule: Dimensions are compatible if they are equal or one is 1

Shape Manipulation

```
x = torch.randn(12)

# Reshape (creates new tensor if needed)
y = x.reshape(3, 4)      # [12] -> [3, 4]
y = x.reshape(2, 2, 3)   # [12] -> [2, 2, 3]

# View (shares memory, must be contiguous)
z = x.view(3, 4)         # [12] -> [3, 4]

# Add/remove dimensions
a = torch.randn(3, 4)
b = a.unsqueeze(0)       # [3, 4] -> [1, 3, 4]
c = b.squeeze(0)         # [1, 3, 4] -> [3, 4]

# Transpose and permute
d = a.T                  # Transpose (2D only)
e = a.transpose(0, 1)    # Swap specific dims
f = a.permute(1, 0)      # Reorder dimensions
```

Indexing and Slicing

```
x = torch.randn(3, 4, 5)

# Basic indexing
a = x[0]                # First batch: [4, 5]
b = x[:, 0, :]          # First row: [3, 5]
c = x[..., -1]          # Last column: [3, 4]

# Advanced indexing
indices = torch.tensor([0, 2])
d = x[indices]          # Select batches 0 and 2

# Boolean masking
mask = x > 0
positive = x[mask]      # All positive values

# Gather and scatter
idx = torch.tensor([[0, 1], [2, 0]])
gathered = torch.gather(x[0], 0, idx)
```

Memory and Performance Tips

- **Contiguous memory:** Use `.contiguous()` after transpose
- **In-place operations:** Suffix with `_` but avoid with autograd
- **No gradient:** Use `torch.no_grad()` for inference
- **Half precision:** Use `float16` or `bfloat16` for speed
- **Pin memory:** Use `pin_memory=True` in DataLoader

Common Pitfalls

- Forgetting to move model and data to same device
- In-place operations breaking gradient computation
- Memory leaks from retaining gradients
- Not using `torch.no_grad()` during evaluation

Reproducibility Setup

```python
import os, random, numpy as np, torch

def setup_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)
        # For exact reproducibility (slower)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

# Call at start of script
setup_seed(42)

# Also set environment variables
os.environ['PYTHONHASHSEED'] = str(42)
```

Tensor Attributes and Metadata

```python
x = torch.randn(3, 4, 5, requires_grad=True)

# Shape information
print(x.shape)           # torch.Size([3, 4, 5])
print(x.size())          # torch.Size([3, 4, 5])
print(x.ndim)            # 3 (number of dimensions)
print(x.numel())         # 60 (total elements)

# Data type and device
print(x.dtype)           # torch.float32
print(x.device)          # cpu or cuda:0
print(x.layout)          # torch.strided

# Gradient information
print(x.requires_grad)  # True
print(x.grad)            # None (before backward)
print(x.is_leaf)         # True (user-created)
```

Type Casting and Conversion

```python
# Create tensor with specific type
x = torch.tensor([1, 2, 3], dtype=torch.float64)

# Type conversion
y = x.float()          # Convert to float32
z = x.long()           # Convert to int64
w = x.half()           # Convert to float16

# To/from NumPy (shares memory on CPU!)
np_array = x.numpy()          # Tensor -> NumPy
tensor = torch.from_numpy(np_array)  # NumPy -> Tensor

# To Python scalars
scalar_tensor = torch.tensor(3.14)
python_float = scalar_tensor.item()  # Extract value

# Detach from computation graph
x_detached = x.detach()       # No gradient tracking
```

Tensor Comparison Operations

```python
a = torch.tensor([1, 2, 3])
b = torch.tensor([3, 2, 1])

# Element-wise comparison
print(a == b)          # [False, True, False]
print(a > b)           # [False, False, True]
print(a <= b)          # [True, True, False]

# Aggregated comparisons
print(torch.all(a == b))      # False
print(torch.any(a == b))      # True

# Close comparison (for floats)
c = torch.tensor([1.0, 2.0, 3.0])
d = torch.tensor([1.0001, 2.0, 3.0])
print(torch.allclose(c, d, atol=1e-3))  # True
print(torch.isclose(c, d, atol=1e-4))   # [False, True, True]
```

Statistical Operations

```
x = torch.randn(3, 4, 5)

# Basic statistics
mean = x.mean()              # Overall mean
std = x.std()                # Standard deviation
var = x.var()                # Variance

# Along dimensions
mean_dim0 = x.mean(dim=0)   # [4, 5]
sum_dim12 = x.sum(dim=[1, 2])  # [3]

# Min/max operations
min_val = x.min()
max_val, max_idx = x.max(dim=1)  # Returns values and indices

# Quantiles
median = x.median()
q1 = torch.quantile(x, 0.25)
```

Linear Algebra Operations

```
A = torch.randn(3, 3)
B = torch.randn(3, 4)
v = torch.randn(3)

# Matrix multiplication
C = A @ B                    # [3, 3] @ [3, 4] -> [3, 4]
C = torch.mm(A, B)           # Same as @
v2 = A @ v                   # Matrix-vector product

# Batch operations
batch_A = torch.randn(10, 3, 3)
batch_B = torch.randn(10, 3, 4)
batch_C = torch.bmm(batch_A, batch_B)  # [10, 3, 4]

# Decompositions
U, S, V = torch.svd(A)       # SVD
L, U = torch.lu(A)           # LU decomposition
eigenvalues, eigenvectors = torch.eig(A, eigenvectors=True)
```

Common Tensor Patterns in RL

```python
# States and actions
states = torch.randn(32, 4)      # [batch, state_dim]
actions = torch.randn(32, 2)     # [batch, action_dim]

# Q-values
q_values = torch.randn(32, 4)    # [batch, num_actions]
action_indices = torch.argmax(q_values, dim=1)  # [batch]

# Rewards and returns
rewards = torch.randn(32, 1)     # [batch, 1]
gamma = 0.99
returns = rewards + gamma * next_values

# Masking for done states
done_mask = torch.tensor([0, 1, 0, ...])  # 1 if done
next_values = next_values * (1 - done_mask)

# Gathering Q-values for taken actions
gathered_q = q_values.gather(1, actions.long())
```

Tensors Summary

Key Concepts Covered

- Tensor creation and initialization
- Device management (CPU/CUDA/MPS)
- Shape manipulation and broadcasting
- Indexing and slicing
- Type conversion and attributes
- Mathematical operations

Remember

- Always manage device placement consistently
- Use appropriate data types for your task
- Leverage broadcasting for efficient operations
- Set seeds for reproducibility

The Engine Behind Deep Learning

Topics

- Computational graphs
- Forward and backward passes
- Gradient computation
- Gradient checking

Why Automatic Differentiation?

Three approaches to compute gradients:

1. **Symbolic Differentiation**
   - Manipulate expressions to find closed form
   - Exact but often impractical for complex functions
2. **Numerical Differentiation**
   - Finite differences: $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
   - Simple but suffers from truncation and rounding errors
3. **Automatic Differentiation**
   - Compose exact local derivatives at machine precision
   - Efficient for scalar objectives with many parameters

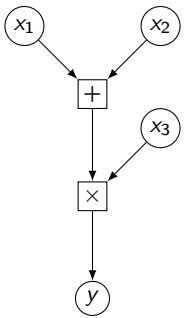
Computational Graphs

Dynamic computational graph:

- Built during forward pass
- Nodes: tensors (data)
- Edges: operations (functions)
- Each operation stores gradient function

Example: $y = (x_1 + x_2) \cdot x_3$

Key point: graph is created on the fly and can change between iterations.



requires_grad and Leaf Tensors

```python
# Leaf tensors (user-created with requires_grad=True)
x = torch.randn(3, requires_grad=True)
w = torch.randn(3, 4, requires_grad=True)

# Non-leaf tensors (results of operations)
y = x @ w.T                # y.requires_grad = True (inherited)
z = y.sum()                # z.requires_grad = True

# Check tensor properties
print(x.is_leaf)           # True
print(y.is_leaf)           # False
print(x.grad_fn)           # None (leaf tensor)
print(y.grad_fn)           # <MmBackward>

# Gradients accumulate only in leaf tensors
z.backward()
print(x.grad)              # Has gradient
print(y.grad)              # None (non-leaf)
```

The Backward Pass

```python
# Simple example
x = torch.tensor([2.0], requires_grad=True)
y = x ** 2                  # y = x^2
z = 2 * y                   # z = 2x^2

# Compute gradients
z.backward()                # dz/dx = 4x = 8

print(x.grad)               # tensor([8.])

# Gradient accumulation (be careful!)
x.grad.zero_()            # Reset gradient
y2 = x ** 3
y2.backward()
print(x.grad)             # tensor([12.]) = 3x^2

# Multiple backwards require retain_graph
loss1 = (x ** 2).sum()
loss2 = (x ** 3).sum()
loss1.backward(retain_graph=True)
loss2.backward()          # Works because of retain_graph
```

Chain Rule in Action

```python
# Multi-layer computation
x = torch.randn(2, 3, requires_grad=True)
W1 = torch.randn(4, 3, requires_grad=True)
W2 = torch.randn(5, 4, requires_grad=True)

# Forward pass
h = torch.relu(x @ W1.T)     # [2, 4]
y = h @ W2.T                  # [2, 5]
loss = y.mean()               # scalar

# Backward pass applies chain rule
loss.backward()

# Gradients computed via chain rule:
# dL/dW2 = dL/dy * dy/dW2
# dL/dW1 = dL/dy * dy/dh * dh/dW1
# dL/dx = dL/dy * dy/dh * dh/dx

print(W2.grad.shape)          # [5, 4]
print(W1.grad.shape)          # [4, 3]
print(x.grad.shape)           # [2, 3]
```

Controlling Gradient Flow

```python
# Stop gradients with detach()
x = torch.randn(3, requires_grad=True)
y = x ** 2
z = y.detach()           # Stop gradient here
w = z * 2

w.sum().backward()       # No gradient flows to x
print(x.grad)            # None

# Stop gradients with torch.no_grad()
x = torch.randn(3, requires_grad=True)
with torch.no_grad():
    y = x ** 2           # No graph built
print(y.requires_grad)   # False

# Gradient clipping (prevent exploding gradients)
torch.nn.utils.clip_grad_norm_(parameters, max_norm=1.0)
torch.nn.utils.clip_grad_value_(parameters, clip_value=1.0)
```

Common Autograd Patterns

```python
# Pattern 1: Training loop
for epoch in range(num_epochs):
    optimizer.zero_grad()        # Clear gradients
    output = model(input)        # Forward pass
    loss = criterion(output, target)
    loss.backward()              # Compute gradients
    optimizer.step()             # Update weights

# Pattern 2: Gradient accumulation
accumulation_steps = 4
for i, (input, target) in enumerate(dataloader):
    output = model(input)
    loss = criterion(output, target) / accumulation_steps
    loss.backward()

    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

Higher-Order Gradients

```python
# Second-order gradients (Hessian)
x = torch.randn(3, requires_grad=True)
y = (x ** 3).sum()

# First derivative
grad = torch.autograd.grad(y, x, create_graph=True)[0]
print(grad)              # 3x^2

# Second derivative
grad2 = torch.autograd.grad(grad.sum(), x)[0]
print(grad2)             # 6x

# Using backward twice
x = torch.randn(3, requires_grad=True)
y = (x ** 4).sum()
y.backward(create_graph=True)
first_grad = x.grad.clone()
x.grad.sum().backward()
second_grad = x.grad - first_grad
```

Custom Autograd Functions

```python
class CustomReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input

# Use custom function
custom_relu = CustomReLU.apply
x = torch.randn(5, requires_grad=True)
y = custom_relu(x)
y.sum().backward()
print(x.grad)  # Gradient only where x > 0
```

Gradient Checking

```python
def gradient_check(f, x, eps=1e-6):
    """Check gradients using finite differences"""
    # Analytic gradient via autograd
    x.requires_grad = True
    y = f(x)
    y.backward()
    analytic_grad = x.grad.clone()

    # Numerical gradient
    x.requires_grad = False
    numerical_grad = torch.zeros_like(x)

    for i in range(x.numel()):
        x_pos = x.clone()
        x_pos.view(-1)[i] += eps
        x_neg = x.clone()
        x_neg.view(-1)[i] -= eps
        numerical_grad.view(-1)[i] = (f(x_pos) - f(x_neg)) / (2 * eps)

    # Compare
    error = (analytic_grad - numerical_grad).abs().max()
    return error < 1e-4
```

Memory Management with Autograd

```python
# Memory-efficient gradient computation
# Problem: Large intermediate tensors
x = torch.randn(1000, 1000, requires_grad=True)
y = x @ x @ x @ x  # Many intermediates stored

# Solution 1: Gradient checkpointing
from torch.utils.checkpoint import checkpoint

def expensive_function(x):
    return x @ x @ x @ x

# Recompute intermediates during backward
y = checkpoint(expensive_function, x)

# Solution 2: In-place operations (use carefully!)
x = torch.randn(1000, 1000)
x.requires_grad = True
# x.add_(1)  # ERROR: Can't use in-place on leaf
y = x.clone()
y.add_(1)    # OK: In-place on non-leaf
```

Debugging Autograd Issues

```
# Common issues and solutions

# 1. None gradients
x = torch.randn(3)  # No requires_grad!
y = x ** 2
y.backward()  # ERROR: requires_grad needed

# 2. Gradient not flowing
x = torch.randn(3, requires_grad=True)
y = x.detach() ** 2  # Detach breaks flow
y.backward()  # ERROR: grad_fn is None

# 3. In-place operation error
x = torch.randn(3, requires_grad=True)
y = x ** 2
x[0] = 0  # ERROR: In-place on needed tensor

# 4. Double backward without retain_graph
loss.backward()
loss.backward()  # ERROR: Graph already freed
```

Profiling Autograd Operations

```python
import torch.autograd.profiler as profiler

# Profile forward and backward passes
with profiler.profile(use_cuda=torch.cuda.is_available()) as prof:
    x = torch.randn(100, 100, requires_grad=True)
    y = x @ x
    z = y.sum()
    z.backward()

# Print profiling results
print(prof.key_averages().table(sort_by="cpu_time_total"))

# Export to Chrome tracing
prof.export_chrome_trace("trace.json")

# Record specific operations
with profiler.record_function("my_operation"):
    result = expensive_computation()
```

Automatic Differentiation Summary

Key Concepts

- Dynamic computational graphs
- Forward pass builds graph, backward computes gradients
- requires_grad enables gradient tracking
- Chain rule automatically applied
- Gradient accumulation and flow control

Best Practices

- Always zero gradients before backward
- Use torch.no_grad() for inference
- Detach tensors to stop gradient flow
- Check gradients with finite differences
- Profile to find bottlenecks

Building Blocks of Deep Learning

Topics

- nn.Module architecture
- Common layers and activation functions
- Loss functions and optimizers
- Training patterns and best practices

nn.Module Basics

```python
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# Create and use model
model = SimpleNet(10, 20, 5)
x = torch.randn(32, 10)  # [batch, input_dim]
output = model(x)         # [batch, output_dim]
```

Common Layer Types

```python
# Linear layers
linear = nn.Linear(10, 20, bias=True)

# Convolutional layers
conv1d = nn.Conv1d(in_channels=3, out_channels=16, kernel_size=5)
conv2d = nn.Conv2d(3, 32, kernel_size=3, padding=1)

# Recurrent layers
lstm = nn.LSTM(input_size=10, hidden_size=20, num_layers=2)
gru = nn.GRU(10, 20, batch_first=True)

# Normalization layers
batch_norm = nn.BatchNorm1d(20)
layer_norm = nn.LayerNorm([20])

# Dropout for regularization
dropout = nn.Dropout(p=0.5)
```

Activation Functions

```python
# Common activations
relu = nn.ReLU()
sigmoid = nn.Sigmoid()
tanh = nn.Tanh()
softmax = nn.Softmax(dim=1)

# Advanced activations
leaky_relu = nn.LeakyReLU(negative_slope=0.01)
elu = nn.ELU(alpha=1.0)
gelu = nn.GELU()
swish = nn.SiLU()  # Also known as Swish

# In functional form
import torch.nn.functional as F
x = torch.randn(10, 20)
y = F.relu(x)
z = F.softmax(x, dim=1)
```

Loss Functions

```python
# Regression losses
mse_loss = nn.MSELoss()
l1_loss = nn.L1Loss()
smooth_l1 = nn.SmoothL1Loss()  # Huber loss

# Classification losses
cross_entropy = nn.CrossEntropyLoss()
nll_loss = nn.NLLLoss()
bce_loss = nn.BCELoss()
bce_with_logits = nn.BCEWithLogitsLoss()

# Example usage
predictions = model(inputs)      # [batch, classes]
targets = torch.randint(0, 10, (batch_size,))
loss = cross_entropy(predictions, targets)

# Custom weights for imbalanced classes
weights = torch.tensor([1.0, 2.0, 1.5])
weighted_ce = nn.CrossEntropyLoss(weight=weights)
```

Optimizers

```python
# Common optimizers
sgd = torch.optim.SGD(model.parameters(), lr=0.01,
                      momentum=0.9, weight_decay=1e-4)
adam = torch.optim.Adam(model.parameters(), lr=1e-3,
                        betas=(0.9, 0.999), eps=1e-8)
rmsprop = torch.optim.RMSprop(model.parameters(), lr=0.01)
adamw = torch.optim.AdamW(model.parameters(), lr=1e-3,
                          weight_decay=0.01)

# Learning rate scheduling
scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                            step_size=30, gamma=0.1)
cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
                                optimizer, T_max=100)

# Update learning rate
for epoch in range(num_epochs):
    train_epoch()
    scheduler.step()
```

Parameter Management

```python
# Access parameters
for name, param in model.named_parameters():
    print(name, param.shape)

# Freeze parameters
for param in model.fc1.parameters():
    param.requires_grad = False

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

# Parameter groups with different learning rates
optimizer = torch.optim.Adam([
    {'params': model.fc1.parameters(), 'lr': 1e-4},
    {'params': model.fc2.parameters(), 'lr': 1e-3}
])
```

Weight Initialization

```python
# Xavier/Glorot initialization
def init_weights_xavier(m):
    if isinstance(m, nn.Linear):
        nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.zeros_(m.bias)

# He/Kaiming initialization (better for ReLU)
def init_weights_he(m):
    if isinstance(m, nn.Linear):
        nn.init.kaiming_normal_(m.weight, mode='fan_out',
                                nonlinearity='relu')
        if m.bias is not None:
            nn.init.zeros_(m.bias)

# Apply to model
model.apply(init_weights_he)

# Custom initialization
model.fc1.weight.data.normal_(0, 0.01)
model.fc1.bias.data.fill_(0)
```

Regularization Techniques

```python
class RegularizedNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim,
                 dropout_rate=0.5):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.dropout1 = nn.Dropout(dropout_rate)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.dropout1(x)
        x = self.fc2(x)
        return x

# L2 regularization via weight_decay
optimizer = torch.optim.Adam(model.parameters(),
                             weight_decay=1e-4)
```

Complete Training Pattern

```python
def train_epoch(model, dataloader, optimizer, criterion):
    model.train()  # Set to training mode
    total_loss = 0

    for batch_idx, (data, target) in enumerate(dataloader):
        data, target = data.to(device), target.to(device)

        # Forward pass
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

        # Backward pass
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

Evaluation Pattern

```python
def evaluate(model, dataloader, criterion):
    model.eval()  # Set to evaluation mode
    total_loss = 0
    correct = 0

    with torch.no_grad():  # Disable gradient computation
        for data, target in dataloader:
            data, target = data.to(device), target.to(device)

            output = model(data)
            loss = criterion(output, target)
            total_loss += loss.item()

            # Calculate accuracy
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()

    accuracy = 100. * correct / len(dataloader.dataset)
    avg_loss = total_loss / len(dataloader)
    return avg_loss, accuracy
```

Saving and Loading Models

```python
# Save model state dict (recommended)
torch.save(model.state_dict(), 'model.pth')

# Load model state dict
model = SimpleNet(input_dim, hidden_dim, output_dim)
model.load_state_dict(torch.load('model.pth'))

# Save complete checkpoint
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    'best_accuracy': best_accuracy
}
torch.save(checkpoint, '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'])
```

Mixed Precision Training

```python
from torch.cuda.amp import autocast, GradScaler

# Initialize scaler for mixed precision
scaler = GradScaler()

for epoch in range(num_epochs):
    for batch_idx, (data, target) in enumerate(dataloader):
        optimizer.zero_grad()

        # Mixed precision forward pass
        with autocast():
            output = model(data)
            loss = criterion(output, target)

        # Scaled backward pass
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

# Note: Only beneficial on GPUs with Tensor Cores
# (NVIDIA V100, RTX 20xx, 30xx, 40xx, A100, etc.)
```

Model Compilation with torch.compile

```python
# PyTorch 2.0+ feature for faster execution
import torch

# Compile model for optimization
model = SimpleNet(input_dim, hidden_dim, output_dim)
compiled_model = torch.compile(model)

# Different compilation modes
model_reduce = torch.compile(model, mode="reduce-overhead")
model_max = torch.compile(model, mode="max-autotune")

# Disable for debugging
model_default = torch.compile(model, mode="default",
                              disable=True)

# Backend options
model_inductor = torch.compile(model, backend="inductor")

# Note: First run will be slower (compilation overhead)
# Subsequent runs will be significantly faster
```

Neural Network Modules Summary

Key Components

- nn.Module as base class for all models
- Layers, activations, loss functions, optimizers
- Weight initialization strategies
- Regularization: dropout, batch norm, weight decay
- Training and evaluation patterns

Best Practices

- Always set model.train() and model.eval()
- Use torch.no_grad() during evaluation
- Initialize weights appropriately
- Save and load checkpoints regularly
- Consider mixed precision for large models

9 Progressive Experiments

1. Environment setup and sanity checks (exp01)
2. Tensors and automatic differentiation (exp02)
3. Computational graph visualization (exp03)
4. Building nn.Module networks (exp04)
5. Loss functions and optimizers (exp05)
6. Regularization and initialization (exp06)
7. Complete training pipeline (exp07)
8. Mixed precision and compilation (exp08)
9. Integration test (exp09)

Experiment 1: Setup and Sanity Checks

Goal: Verify environment and basic operations
Tasks:

- Check PyTorch installation and version
- Test device availability (CPU/CUDA/MPS)
- Verify tensor operations
- Test reproducibility with seeds

Run: `python exp01_setup.py`
Expected: Device detected, operations successful

Exp1: Key Code

```python
def main():
    print("="*50)
    print("Experiment 01: Setup and Sanity")
    print("="*50)

    # Device selection
    device = torch.device(
        'cuda' if torch.cuda.is_available()
        else 'mps' if hasattr(torch.backends, 'mps')
                      and torch.backends.mps.is_available()
        else 'cpu'
    )
    print(f"Using device: {device}")

    # Test operations
    x = torch.randn(3, 4).to(device)
    y = x @ x.T
    print(f"Matrix multiply OK, shape: {y.shape}")
```

Experiment 2: Tensors and Autograd

Goal: Master tensor operations and gradients
Tasks:

- Create tensors with gradient tracking
- Perform operations and compute gradients
- Understand gradient accumulation
- Test gradient flow control

Key Learning: How autograd tracks operations

Exp2: Key Code

```python
# Automatic differentiation example
x = torch.tensor([2.0, 3.0], requires_grad=True)
y = x ** 2
z = y.sum()

z.backward()
print(f"x.grad = {x.grad}")  # [4.0, 6.0]

# Gradient accumulation
x.grad.zero_()
for i in range(3):
    y = (x ** 2).sum()
    y.backward()
print(f"Accumulated grad = {x.grad}")  # [12.0, 18.0]
```

Experiment 3: Computational Graph

Goal: Visualize and understand computation graphs
Tasks:

- Build complex computational graphs
- Inspect grad_fn attributes
- Trace backward pass
- Understand graph retention

Critical: Graphs are dynamic and rebuilt each forward

Experiment 4: nn.Module Networks

Goal: Build custom neural networks
Components tested:

- Custom nn.Module classes
- Parameter management
- Forward pass implementation
- Model composition

Expected outcome: Working multi-layer network

Experiment 5: Losses and Optimizers

Goal: Compare different losses and optimizers
Components:

- MSE, Cross-entropy, Huber losses
- SGD, Adam, AdamW optimizers
- Learning rate scheduling
- Gradient clipping

Compare convergence rates and stability

Experiment 6: Regularization

Goal: Apply regularization techniques
Features:

- Dropout layers
- Batch normalization
- Weight decay (L2 regularization)
- Different initialization strategies

Observe effects on training and validation loss

Experiment 7: Training Pipeline

Goal: Complete training and evaluation loop
What to implement:

- Data loading with DataLoader
- Training loop with metrics
- Validation with early stopping
- Checkpoint saving

Full end-to-end training workflow

Experiment 8: Advanced Features

Goal: Use PyTorch 2.0+ features
Best practices:

- Mixed precision training (AMP)
- Model compilation with torch.compile
- Profiling and optimization
- Memory-efficient training

Experiment 9: Integration Test

Goal: Validate complete implementation
Tests performed:

1. Device management and reproducibility
2. Tensor operations and autograd
3. Model construction and initialization
4. Loss computation and optimization
5. Training and evaluation loops
6. Checkpoint save/load functionality

Must pass all tests before proceeding!

Exp9: Expected Output

```
# Run integrated test
python exp09_integrated_test.py

# Expected output:
# ====================================
# Test 1: Device Setup          [PASS]
# Test 2: Autograd              [PASS]
# Test 3: nn.Module             [PASS]
# Test 4: Optimization          [PASS]
# Test 5: Training Loop         [PASS]
# Test 6: Checkpointing         [PASS]
# ====================================
# All tests passed!
```

Key Takeaways

1. **Tensors:** Foundation of all computations in PyTorch
2. **Autograd:** Automatic differentiation via dynamic graphs
3. **Device Management:** Always check CUDA > MPS > CPU
4. **nn.Module:** Building block for all neural networks
5. **Training Pattern:** zero_grad, forward, backward, step
6. **Best Practices:** Reproducibility, checkpointing, profiling