

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)

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

```
1 import torch
2
3 # From data
4 x = torch.tensor([1, 2, 3])          # From list
5 y = torch.tensor([[1, 2], [3, 4]])  # 2D tensor
6
7 # Random tensors
8 z = torch.randn(3, 4)               # Normal distribution
9 u = torch.rand(2, 3)                # Uniform [0, 1)
10
11 # Special tensors
12 zeros = torch.zeros(3, 3)
13 ones = torch.ones(2, 4)
14 eye = torch.eye(3)                  # Identity matrix
15
16 # With specific dtype and device
17 x = torch.tensor([1.0], dtype=torch.float64,
18                  device='cuda' if torch.cuda.is_available()
19                  else 'cpu')
```

Tensor Operations

```
1  # Basic arithmetic
2  a = torch.tensor([1., 2., 3.])
3  b = torch.tensor([4., 5., 6.])
4
5  c = a + b          # Element-wise addition
6  d = a * b          # Element-wise multiplication
7  e = a @ b          # Dot product (same as torch.dot)
8
9  # Matrix operations
10 A = torch.randn(3, 4)
11 B = torch.randn(4, 5)
12 C = A @ B          # Matrix multiplication [3, 5]
13
14 # In-place operations (use with caution!)
15 a.add_(1)           # Modifies a in-place
16 a.mul_(2)           # Dangerous with autograd!
```

Device Management

```
1  # Proper device selection (CUDA > MPS > CPU)
2  device = torch.device(
3      'cuda' if torch.cuda.is_available()
4      else 'mps' if hasattr(torch.backends, 'mps')
5                      and torch.backends.mps.is_available()
6      else 'cpu'
7  )
8
9  # Moving tensors between devices
10 x = torch.randn(3, 4)
11 x = x.to(device)          # Move to selected device
12 x = x.cpu()              # Back to CPU
13
14 # Creating directly on device
15 y = torch.randn(3, 4, device=device)
16
17 # Check device
18 print(x.device)           # cuda:0 or cpu
19 print(x.is_cuda)         # True/False
```

Broadcasting Rules

PyTorch automatically expands tensors for element-wise operations:

```
1  # Scalar broadcast
2  a = torch.randn(3, 4)
3  b = 2.0
4  c = a * b           # b broadcasts to [3, 4]
5
6  # Vector broadcast
7  a = torch.randn(3, 4) # [3, 4]
8  b = torch.randn(4)    # [4] -> broadcasts to [3, 4]
9  c = a + b            # OK
10
11 # Matrix broadcast
12 a = torch.randn(3, 1, 4) # [3, 1, 4]
13 b = torch.randn(1, 5, 4) # [1, 5, 4]
14 c = a + b              # [3, 5, 4]
```

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

Shape Manipulation

```
1 x = torch.randn(12)
2
3 # Reshape (creates new tensor if needed)
4 y = x.reshape(3, 4)      # [12] -> [3, 4]
5 y = x.reshape(2, 2, 3)   # [12] -> [2, 2, 3]
6
7 # View (shares memory, must be contiguous)
8 z = x.view(3, 4)         # [12] -> [3, 4]
9
10 # Add/remove dimensions
11 a = torch.randn(3, 4)
12 b = a.unsqueeze(0)       # [3, 4] -> [1, 3, 4]
13 c = b.squeeze(0)         # [1, 3, 4] -> [3, 4]
14
15 # Transpose and permute
16 d = a.T                  # Transpose (2D only)
17 e = a.transpose(0, 1)    # Swap specific dims
18 f = a.permute(1, 0)      # Reorder dimensions
```

Indexing and Slicing

```
1 x = torch.randn(3, 4, 5)
2
3 # Basic indexing
4 a = x[0]           # First batch: [4, 5]
5 b = x[:, 0, :]     # First row: [3, 5]
6 c = x[..., -1]     # Last column: [3, 4]
7
8 # Advanced indexing
9 indices = torch.tensor([0, 2])
10 d = x[indices]      # Select batches 0 and 2
11
12 # Boolean masking
13 mask = x > 0
14 positive = x[mask]  # All positive values
15
16 # Gather and scatter
17 idx = torch.tensor([[0, 1], [2, 0]])
18 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

```
1 import os, random, numpy as np, torch
2
3 def setup_seed(seed=42):
4     random.seed(seed)
5     np.random.seed(seed)
6     torch.manual_seed(seed)
7     if torch.cuda.is_available():
8         torch.cuda.manual_seed_all(seed)
9         # For exact reproducibility (slower)
10        torch.backends.cudnn.deterministic = True
11        torch.backends.cudnn.benchmark = False
12
13 # Call at start of script
14 setup_seed(42)
15
16 # Also set environment variables
17 os.environ['PYTHONHASHSEED'] = str(42)
```

Tensor Attributes and Metadata

```
1 x = torch.randn(3, 4, 5, requires_grad=True)
2
3 # Shape information
4 print(x.shape)          # torch.Size([3, 4, 5])
5 print(x.size())         # torch.Size([3, 4, 5])
6 print(x.ndim)           # 3 (number of dimensions)
7 print(x.numel())        # 60 (total elements)
8
9 # Data type and device
10 print(x.dtype)          # torch.float32
11 print(x.device)         # cpu or cuda:0
12 print(x.layout)         # torch.strided
13
14 # Gradient information
15 print(x.requires_grad)  # True
16 print(x.grad)           # None (before backward)
17 print(x.is_leaf)        # True (user-created)
```

Type Casting and Conversion

```
1 # Create tensor with specific type
2 x = torch.tensor([1, 2, 3], dtype=torch.float64)
3
4 # Type conversion
5 y = x.float()           # Convert to float32
6 z = x.long()            # Convert to int64
7 w = x.half()            # Convert to float16
8
9 # To/from NumPy (shares memory on CPU!)
10 np_array = x.numpy()    # Tensor -> NumPy
11 tensor = torch.from_numpy(np_array) # NumPy -> Tensor
12
13 # To Python scalars
14 scalar_tensor = torch.tensor(3.14)
15 python_float = scalar_tensor.item() # Extract value
16
17 # Detach from computation graph
18 x_detached = x.detach()  # No gradient tracking
```

Tensor Comparison Operations

```
1 a = torch.tensor([1, 2, 3])
2 b = torch.tensor([3, 2, 1])
3
4 # Element-wise comparison
5 print(a == b)          # [False, True, False]
6 print(a > b)           # [False, False, True]
7 print(a <= b)          # [True, True, False]
8
9 # Aggregated comparisons
10 print(torch.all(a == b))    # False
11 print(torch.any(a == b))    # True
12
13 # Close comparison (for floats)
14 c = torch.tensor([1.0, 2.0, 3.0])
15 d = torch.tensor([1.0001, 2.0, 3.0])
16 print(torch.allclose(c, d, atol=1e-3)) # True
17 print(torch.isclose(c, d, atol=1e-4))  # [False, True, True]
```

Statistical Operations

```
1 x = torch.randn(3, 4, 5)
2
3 # Basic statistics
4 mean = x.mean()           # Overall mean
5 std = x.std()             # Standard deviation
6 var = x.var()             # Variance
7
8 # Along dimensions
9 mean_dim0 = x.mean(dim=0)  # [4, 5]
10 sum_dim12 = x.sum(dim=[1, 2]) # [3]
11
12 # Min/max operations
13 min_val = x.min()
14 max_val, max_idx = x.max(dim=1) # Returns values and indices
15
16 # Quantiles
17 median = x.median()
18 q1 = torch.quantile(x, 0.25)
```


Linear Algebra Operations

```
1 A = torch.randn(3, 3)
2 B = torch.randn(3, 4)
3 v = torch.randn(3)
4
5 # Matrix multiplication
6 C = A @ B                # [3, 3] @ [3, 4] -> [3, 4]
7 C = torch.mm(A, B)       # Same as @
8 v2 = A @ v               # Matrix-vector product
9
10 # Batch operations
11 batch_A = torch.randn(10, 3, 3)
12 batch_B = torch.randn(10, 3, 4)
13 batch_C = torch.bmm(batch_A, batch_B) # [10, 3, 4]
14
15 # Decompositions
16 U, S, V = torch.svd(A)    # SVD
17 L, U = torch.lu(A)        # LU decomposition
18 eigenvalues, eigenvectors = torch.eig(A, eigenvectors=True)
```

Common Tensor Patterns in RL

```
1  # States and actions
2  states = torch.randn(32, 4)      # [batch, state_dim]
3  actions = torch.randn(32, 2)     # [batch, action_dim]
4
5  # Q-values
6  q_values = torch.randn(32, 4)    # [batch, num_actions]
7  action_indices = torch.argmax(q_values, dim=1) # [batch]
8
9  # Rewards and returns
10 rewards = torch.randn(32, 1)     # [batch, 1]
11 gamma = 0.99
12 returns = rewards + gamma * next_values
13
14 # Masking for done states
15 done_mask = torch.tensor([0, 1, 0, ...]) # 1 if done
16 next_values = next_values * (1 - done_mask)
17
18 # Gathering Q-values for taken actions
19 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

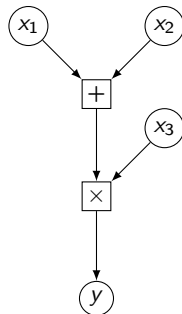
Computational Graphs

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

```
1  # Leaf tensors (user-created with requires_grad=True)
2  x = torch.randn(3, requires_grad=True)
3  w = torch.randn(3, 4, requires_grad=True)
4
5  # Non-leaf tensors (results of operations)
6  y = x @ w.T          # y.requires_grad = True (inherited)
7  z = y.sum()          # z.requires_grad = True
8
9  # Check tensor properties
10 print(x.is_leaf)      # True
11 print(y.is_leaf)      # False
12 print(x.grad_fn)      # None (leaf tensor)
13 print(y.grad_fn)      # <MmBackward>
14
15 # Gradients accumulate only in leaf tensors
16 z.backward()
17 print(x.grad)         # Has gradient
18 print(y.grad)         # None (non-leaf)
```

The Backward Pass

```
1  # Simple example
2  x = torch.tensor([2.0], requires_grad=True)
3  y = x ** 2          # y = x^2
4  z = 2 * y           # z = 2x^2
5
6  # Compute gradients
7  z.backward()        # dz/dx = 4x = 8
8
9  print(x.grad)       # tensor([8.])
10
11 # Gradient accumulation (be careful!)
12 x.grad.zero_()      # Reset gradient
13 y2 = x ** 3
14 y2.backward()
15 print(x.grad)       # tensor([12.]) = 3x^2
16
17 # Multiple backwards require retain_graph
18 loss1 = (x ** 2).sum()
19 loss2 = (x ** 3).sum()
20 loss1.backward(retain_graph=True)
21 loss2.backward()    # Works because of retain_graph
```


Chain Rule in Action

```
1  # Multi-layer computation
2  x = torch.randn(2, 3, requires_grad=True)
3  W1 = torch.randn(4, 3, requires_grad=True)
4  W2 = torch.randn(5, 4, requires_grad=True)
5
6  # Forward pass
7  h = torch.relu(x @ W1.T)      # [2, 4]
8  y = h @ W2.T                  # [2, 5]
9  loss = y.mean()               # scalar
10
11 # Backward pass applies chain rule
12 loss.backward()
13
14 # Gradients computed via chain rule:
15 #  $dL/dW2 = dL/dy * dy/dW2$ 
16 #  $dL/dW1 = dL/dy * dy/dh * dh/dW1$ 
17 #  $dL/dx = dL/dy * dy/dh * dh/dx$ 
18
19 print(W2.grad.shape)           # [5, 4]
20 print(W1.grad.shape)           # [4, 3]
21 print(x.grad.shape)            # [2, 3]
```

Controlling Gradient Flow

```
1  # Stop gradients with detach()
2  x = torch.randn(3, requires_grad=True)
3  y = x ** 2
4  z = y.detach()          # Stop gradient here
5  w = z * 2
6
7  w.sum().backward()      # No gradient flows to x
8  print(x.grad)          # None
9
10 # Stop gradients with torch.no_grad()
11 x = torch.randn(3, requires_grad=True)
12 with torch.no_grad():
13     y = x ** 2          # No graph built
14 print(y.requires_grad)  # False
15
16 # Gradient clipping (prevent exploding gradients)
17 torch.nn.utils.clip_grad_norm_(parameters, max_norm=1.0)
18 torch.nn.utils.clip_grad_value_(parameters, clip_value=1.0)
```

Common Autograd Patterns

```
1 # Pattern 1: Training loop
2 for epoch in range(num_epochs):
3     optimizer.zero_grad()          # Clear gradients
4     output = model(input)          # Forward pass
5     loss = criterion(output, target)
6     loss.backward()                # Compute gradients
7     optimizer.step()               # Update weights
8
9 # Pattern 2: Gradient accumulation
10 accumulation_steps = 4
11 for i, (input, target) in enumerate(dataloader):
12     output = model(input)
13     loss = criterion(output, target) / accumulation_steps
14     loss.backward()
15
16     if (i + 1) % accumulation_steps == 0:
17         optimizer.step()
18         optimizer.zero_grad()
```

Higher-Order Gradients

```
1  # Second-order gradients (Hessian)
2  x = torch.randn(3, requires_grad=True)
3  y = (x ** 3).sum()
4
5  # First derivative
6  grad = torch.autograd.grad(y, x, create_graph=True)[0]
7  print(grad)           # 3x^2
8
9  # Second derivative
10 grad2 = torch.autograd.grad(grad.sum(), x)[0]
11 print(grad2)          # 6x
12
13 # Using backward twice
14 x = torch.randn(3, requires_grad=True)
15 y = (x ** 4).sum()
16 y.backward(create_graph=True)
17 first_grad = x.grad.clone()
18 x.grad.sum().backward()
19 second_grad = x.grad - first_grad
```

Custom Autograd Functions

```
1 class CustomReLU(torch.autograd.Function):
2     @staticmethod
3     def forward(ctx, input):
4         ctx.save_for_backward(input)
5         return input.clamp(min=0)
6
7     @staticmethod
8     def backward(ctx, grad_output):
9         input, = ctx.saved_tensors
10        grad_input = grad_output.clone()
11        grad_input[input < 0] = 0
12        return grad_input
13
14 # Use custom function
15 custom_relu = CustomReLU.apply
16 x = torch.randn(5, requires_grad=True)
17 y = custom_relu(x)
18 y.sum().backward()
19 print(x.grad) # Gradient only where x > 0
```

Gradient Checking

```
1 def gradient_check(f, x, eps=1e-6):
2     """Check gradients using finite differences"""
3     # Analytic gradient via autograd
4     x.requires_grad = True
5     y = f(x)
6     y.backward()
7     analytic_grad = x.grad.clone()
8
9     # Numerical gradient
10    x.requires_grad = False
11    numerical_grad = torch.zeros_like(x)
12
13    for i in range(x.numel()):
14        x_pos = x.clone()
15        x_pos.view(-1)[i] += eps
16        x_neg = x.clone()
17        x_neg.view(-1)[i] -= eps
18        numerical_grad.view(-1)[i] = (f(x_pos) - f(x_neg)) / (2 * eps)
19
20    # Compare
21    error = (analytic_grad - numerical_grad).abs().max()
22    return error < 1e-4
```

Memory Management with Autograd

```
1  # Memory-efficient gradient computation
2  # Problem: Large intermediate tensors
3  x = torch.randn(1000, 1000, requires_grad=True)
4  y = x @ x @ x @ x @ x  # Many intermediates stored
5
6  # Solution 1: Gradient checkpointing
7  from torch.utils.checkpoint import checkpoint
8
9  def expensive_function(x):
10     return x @ x @ x @ x
11
12  # Recompute intermediates during backward
13  y = checkpoint(expensive_function, x)
14
15  # Solution 2: In-place operations (use carefully!)
16  x = torch.randn(1000, 1000)
17  x.requires_grad = True
18  # x.add_(1) # ERROR: Can't use in-place on leaf
19  y = x.clone()
20  y.add_(1)  # OK: In-place on non-leaf
```

Debugging Autograd Issues

```
1 # Common issues and solutions
```

```
2  
3 # 1. None gradients
```

```
4 x = torch.randn(3) # No requires_grad!
```

```
5 y = x ** 2
```

```
6 y.backward() # ERROR: requires_grad needed
```

```
7  
8 # 2. Gradient not flowing
```

```
9 x = torch.randn(3, requires_grad=True)
```

```
10 y = x.detach() ** 2 # Detach breaks flow
```

```
11 y.backward() # ERROR: grad_fn is None
```

```
12  
13 # 3. In-place operation error
```

```
14 x = torch.randn(3, requires_grad=True)
```

```
15 y = x ** 2
```

```
16 x[0] = 0 # ERROR: In-place on needed tensor
```

```
17  
18 # 4. Double backward without retain_graph
```

```
19 loss.backward()
```

```
20 loss.backward() # ERROR: Graph already freed
```


Profiling Autograd Operations

```
1 import torch.autograd.profiler as profiler
2
3 # Profile forward and backward passes
4 with profiler.profile(use_cuda=torch.cuda.is_available()) as prof:
5     x = torch.randn(100, 100, requires_grad=True)
6     y = x @ x
7     z = y.sum()
8     z.backward()
9
10 # Print profiling results
11 print(prof.key_averages().table(sort_by="cpu_time_total"))
12
13 # Export to Chrome tracing
14 prof.export_chrome_trace("trace.json")
15
16 # Record specific operations
17 with profiler.record_function("my_operation"):
18     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

```
1 import torch.nn as nn
2
3 class SimpleNet(nn.Module):
4     def __init__(self, input_dim, hidden_dim, output_dim):
5         super(SimpleNet, self).__init__()
6         self.fc1 = nn.Linear(input_dim, hidden_dim)
7         self.relu = nn.ReLU()
8         self.fc2 = nn.Linear(hidden_dim, output_dim)
9
10    def forward(self, x):
11        x = self.fc1(x)
12        x = self.relu(x)
13        x = self.fc2(x)
14        return x
15
16 # Create and use model
17 model = SimpleNet(10, 20, 5)
18 x = torch.randn(32, 10) # [batch, input_dim]
19 output = model(x)        # [batch, output_dim]
```

Common Layer Types

```
1  # Linear layers
2  linear = nn.Linear(10, 20, bias=True)
3
4  # Convolutional layers
5  conv1d = nn.Conv1d(in_channels=3, out_channels=16, kernel_size=5)
6  conv2d = nn.Conv2d(3, 32, kernel_size=3, padding=1)
7
8  # Recurrent layers
9  lstm = nn.LSTM(input_size=10, hidden_size=20, num_layers=2)
10 gru = nn.GRU(10, 20, batch_first=True)
11
12 # Normalization layers
13 batch_norm = nn.BatchNorm1d(20)
14 layer_norm = nn.LayerNorm([20])
15
16 # Dropout for regularization
17 dropout = nn.Dropout(p=0.5)
```

Activation Functions

```
1  # Common activations
2  relu = nn.ReLU()
3  sigmoid = nn.Sigmoid()
4  tanh = nn.Tanh()
5  softmax = nn.Softmax(dim=1)
6
7  # Advanced activations
8  leaky_relu = nn.LeakyReLU(negative_slope=0.01)
9  elu = nn.ELU(alpha=1.0)
10 gelu = nn.GELU()
11 swish = nn.SiLU()  # Also known as Swish
12
13 # In functional form
14 import torch.nn.functional as F
15 x = torch.randn(10, 20)
16 y = F.relu(x)
17 z = F.softmax(x, dim=1)
```

Loss Functions

```
1  # Regression losses
2  mse_loss = nn.MSELoss()
3  l1_loss = nn.L1Loss()
4  smooth_l1 = nn.SmoothL1Loss()  # Huber loss
5
6  # Classification losses
7  cross_entropy = nn.CrossEntropyLoss()
8  nll_loss = nn.NLLLoss()
9  bce_loss = nn.BCELoss()
10 bce_with_logits = nn.BCEWithLogitsLoss()
11
12 # Example usage
13 predictions = model(inputs)      # [batch, classes]
14 targets = torch.randint(0, 10, (batch_size,))
15 loss = cross_entropy(predictions, targets)
16
17 # Custom weights for imbalanced classes
18 weights = torch.tensor([1.0, 2.0, 1.5])
19 weighted_ce = nn.CrossEntropyLoss(weight=weights)
```

Optimizers

```
1 # Common optimizers
2 sgd = torch.optim.SGD(model.parameters(), lr=0.01,
3                        momentum=0.9, weight_decay=1e-4)
4 adam = torch.optim.Adam(model.parameters(), lr=1e-3,
5                          betas=(0.9, 0.999), eps=1e-8)
6 rmsprop = torch.optim.RMSprop(model.parameters(), lr=0.01)
7 adamw = torch.optim.AdamW(model.parameters(), lr=1e-3,
8                            weight_decay=0.01)
9
10 # Learning rate scheduling
11 scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
12                                             step_size=30, gamma=0.1)
13 cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
14     optimizer, T_max=100)
15
16 # Update learning rate
17 for epoch in range(num_epochs):
18     train_epoch()
19     scheduler.step()
```


Parameter Management

```
1 # Access parameters
2 for name, param in model.named_parameters():
3     print(name, param.shape)
4
5 # Freeze parameters
6 for param in model.fc1.parameters():
7     param.requires_grad = False
8
9 # Count parameters
10 total_params = sum(p.numel() for p in model.parameters())
11 trainable_params = sum(p.numel() for p in model.parameters()
12                          if p.requires_grad)
13
14 # Parameter groups with different learning rates
15 optimizer = torch.optim.Adam([
16     {'params': model.fc1.parameters(), 'lr': 1e-4},
17     {'params': model.fc2.parameters(), 'lr': 1e-3}
18 ])
```

Weight Initialization

```
1  # Xavier/Glorot initialization
2  def init_weights_xavier(m):
3      if isinstance(m, nn.Linear):
4          nn.init.xavier_uniform_(m.weight)
5          if m.bias is not None:
6              nn.init.zeros_(m.bias)
7
8  # He/Kaiming initialization (better for ReLU)
9  def init_weights_he(m):
10     if isinstance(m, nn.Linear):
11         nn.init.kaiming_normal_(m.weight, mode='fan_out',
12                                 nonlinearity='relu')
13         if m.bias is not None:
14             nn.init.zeros_(m.bias)
15
16 # Apply to model
17 model.apply(init_weights_he)
18
19 # Custom initialization
20 model.fc1.weight.data.normal_(0, 0.01)
21 model.fc1.bias.data.fill_(0)
```

Regularization Techniques

```
1 class RegularizedNet(nn.Module):
2     def __init__(self, input_dim, hidden_dim, output_dim,
3         dropout_rate=0.5):
4         super().__init__()
5         self.fc1 = nn.Linear(input_dim, hidden_dim)
6         self.dropout1 = nn.Dropout(dropout_rate)
7         self.bn1 = nn.BatchNorm1d(hidden_dim)
8         self.fc2 = nn.Linear(hidden_dim, output_dim)
9
10    def forward(self, x):
11        x = self.fc1(x)
12        x = self.bn1(x)
13        x = F.relu(x)
14        x = self.dropout1(x)
15        x = self.fc2(x)
16        return x
17
18    # L2 regularization via weight_decay
19    optimizer = torch.optim.Adam(model.parameters(),
20        weight_decay=1e-4)
```

Complete Training Pattern

```
1 def train_epoch(model, dataloader, optimizer, criterion):
2     model.train() # Set to training mode
3     total_loss = 0
4
5     for batch_idx, (data, target) in enumerate(dataloader):
6         data, target = data.to(device), target.to(device)
7
8         # Forward pass
9         optimizer.zero_grad()
10        output = model(data)
11        loss = criterion(output, target)
12
13        # Backward pass
14        loss.backward()
15        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
16        optimizer.step()
17
18        total_loss += loss.item()
19
20    return total_loss / len(dataloader)
```

Evaluation Pattern

```
1 def evaluate(model, dataloader, criterion):
2     model.eval() # Set to evaluation mode
3     total_loss = 0
4     correct = 0
5
6     with torch.no_grad(): # Disable gradient computation
7         for data, target in dataloader:
8             data, target = data.to(device), target.to(device)
9
10            output = model(data)
11            loss = criterion(output, target)
12            total_loss += loss.item()
13
14            # Calculate accuracy
15            pred = output.argmax(dim=1)
16            correct += pred.eq(target).sum().item()
17
18 accuracy = 100. * correct / len(dataloader.dataset)
19 avg_loss = total_loss / len(dataloader)
20 return avg_loss, accuracy
```

Saving and Loading Models

```
1  # Save model state dict (recommended)
2  torch.save(model.state_dict(), 'model.pth')
3
4  # Load model state dict
5  model = SimpleNet(input_dim, hidden_dim, output_dim)
6  model.load_state_dict(torch.load('model.pth'))
7
8  # Save complete checkpoint
9  checkpoint = {
10     'epoch': epoch,
11     'model_state_dict': model.state_dict(),
12     'optimizer_state_dict': optimizer.state_dict(),
13     'loss': loss,
14     'best_accuracy': best_accuracy
15 }
16 torch.save(checkpoint, 'checkpoint.pth')
17
18 # Load checkpoint
19 checkpoint = torch.load('checkpoint.pth')
20 model.load_state_dict(checkpoint['model_state_dict'])
21 optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Mixed Precision Training

```
1 from torch.cuda.amp import autocast, GradScaler
2
3 # Initialize scaler for mixed precision
4 scaler = GradScaler()
5
6 for epoch in range(num_epochs):
7     for batch_idx, (data, target) in enumerate(dataloader):
8         optimizer.zero_grad()
9
10        # Mixed precision forward pass
11        with autocast():
12            output = model(data)
13            loss = criterion(output, target)
14
15        # Scaled backward pass
16        scaler.scale(loss).backward()
17        scaler.step(optimizer)
18        scaler.update()
19
20 # Note: Only beneficial on GPUs with Tensor Cores
21 # (NVIDIA V100, RTX 20xx, 30xx, 40xx, A100, etc.)
```

Model Compilation with torch.compile

```
1  # PyTorch 2.0+ feature for faster execution
2  import torch
3
4  # Compile model for optimization
5  model = SimpleNet(input_dim, hidden_dim, output_dim)
6  compiled_model = torch.compile(model)
7
8  # Different compilation modes
9  model_reduce = torch.compile(model, mode="reduce-overhead")
10 model_max = torch.compile(model, mode="max-autotune")
11
12 # Disable for debugging
13 model_default = torch.compile(model, mode="default",
14                               disable=True)
15
16 # Backend options
17 model_inductor = torch.compile(model, backend="inductor")
18
19 # Note: First run will be slower (compilation overhead)
20 # 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

```
1 def main():
2     print("="*50)
3     print("Experiment 01: Setup and Sanity")
4     print("="*50)
5
6     # Device selection
7     device = torch.device(
8         'cuda' if torch.cuda.is_available()
9         else 'mps' if hasattr(torch.backends, 'mps')
10            and torch.backends.mps.is_available()
11         else 'cpu'
12     )
13     print(f"Using device: {device}")
14
15     # Test operations
16     x = torch.randn(3, 4).to(device)
17     y = x @ x.T
18     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

```
1  # Automatic differentiation example
2  x = torch.tensor([2.0, 3.0], requires_grad=True)
3  y = x ** 2
4  z = y.sum()
5
6  z.backward()
7  print(f"x.grad = {x.grad}") # [4.0, 6.0]
8
9  # Gradient accumulation
10 x.grad.zero_()
11 for i in range(3):
12     y = (x ** 2).sum()
13     y.backward()
14 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

```
1  # Run integrated test
2  python exp09_integrated_test.py
3
4  # Expected output:
5  # =====
6  # Test 1: Device Setup          [PASS]
7  # Test 2: Autograd              [PASS]
8  # Test 3: nn.Module             [PASS]
9  # Test 4: Optimization          [PASS]
10 # Test 5: Training Loop         [PASS]
11 # Test 6: Checkpointing         [PASS]
12 # =====
13 # 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

Common Pitfalls to Avoid

- × Forgetting to zero gradients before backward
- × Not moving model and data to same device
- × Using in-place operations that break autograd
- × Not using `torch.no_grad()` during evaluation
- × Incorrect broadcasting assumptions
- × Memory leaks from retained gradients

Remember

Always verify tensor shapes, device placement, and gradient flow!

Next Week: RL Fundamentals

Topics:

- Agent-environment interaction
- States, actions, and rewards
- Episode structure and returns
- Gymnasium API introduction
- Basic RL algorithms

Preparation:

- Complete all 9 experiments from today
- Install Gymnasium: `pip install gymnasium`
- Review Markov Decision Processes

Strong foundations enable advanced techniques.

See you next week for RL Fundamentals!