CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 8, Lecture 15
PyTorch Tensors, Autograd, and nn.Module

Lecture Overview

PyTorch is the leading deep learning framework used in research and production. This lecture introduces PyTorch tensors, automatic differentiation with autograd, and building neural networks.

Topics Covered:

- PyTorch tensors and operations
- GPU acceleration with CUDA
- Automatic differentiation (autograd)
- Building models with nn.Module
- Common layers and activations
- Model parameters and state

1. PyTorch Tensors

1.1 Creating Tensors

```
import torch
import numpy as np

print("PyTorch Tensors")
print("="*70)

# Create tensors
print("Creating Tensors:")

# From Python lists
t1 = torch.tensor([1, 2, 3, 4, 5])
print(f"From list: {t1}")
print(f"  dtype: {t1.dtype}, shape: {t1.shape}")

# From NumPy
np_arr = np.array([[1, 2, 3], [4, 5, 6]])
t2 = torch.from_numpy(np_arr)
print(f"\nFrom NumPy:\n{t2}")

# Special tensors
zeros = torch.zeros(2, 3)
ones = torch.ones(2, 3)
rand = torch.rand(2, 3)        # Uniform [0, 1)
randn = torch.randn(2, 3)      # Standard normal
eye = torch.eye(3)             # Identity matrix

print(f"\nZeros (2x3):\n{zeros}")
print(f"\nRandom normal (2x3):\n{randn}")

# Tensor with specific dtype
t_float = torch.tensor([1, 2, 3], dtype=torch.float32)
t_int = torch.tensor([1, 2, 3], dtype=torch.int64)
print(f"\nFloat tensor: {t_float}, dtype: {t_float.dtype}")
print(f"Int tensor: {t_int}, dtype: {t_int.dtype}")

# Create like another tensor
t_like = torch.zeros_like(randn)
print(f"\nzeros_like shape: {t_like.shape}")
```

```
PyTorch Tensors
======================================================================
Creating Tensors:
From list: tensor([1, 2, 3, 4, 5])
  dtype: torch.int64, shape: torch.Size([5])

From NumPy:
tensor([[1, 2, 3],
        [4, 5, 6]])

Zeros (2x3):
tensor([[0., 0., 0.],
        [0., 0., 0.]])

Random normal (2x3):
tensor([[ 0.4967, -0.1383,  0.6477],
        [ 1.5230, -0.2342, -0.2341]])

Float tensor: tensor([1., 2., 3.]), dtype: torch.float32
Int tensor: tensor([1, 2, 3]), dtype: torch.int64
```

```
zeros_like shape: torch.Size([2, 3])
```

1.2 Tensor Operations

```
# Tensor Operations
print("Tensor Operations")
print("="*70)

a = torch.tensor([[1, 2], [3, 4]], dtype=torch.float32)
b = torch.tensor([[5, 6], [7, 8]], dtype=torch.float32)

print(f"a:\n{a}")
print(f"b:\n{b}")

# Element-wise operations
print(f"\nElement-wise addition:\n{a + b}")
print(f"Element-wise multiplication:\n{a * b}")

# Matrix multiplication
print(f"\nMatrix multiplication (a @ b):\n{a @ b}")
print(f"Or torch.matmul(a, b):\n{torch.matmul(a, b)}")

# Broadcasting
c = torch.tensor([10, 20])
print(f"\nBroadcasting a + [10, 20]:\n{a + c}")

# Reduction operations
x = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
print(f"\nReduction operations on:\n{x}")
print(f"Sum all: {x.sum()}")
print(f"Sum axis=0: {x.sum(dim=0)}")
print(f"Sum axis=1: {x.sum(dim=1)}")
print(f"Mean: {x.mean()}")
print(f"Max: {x.max()}")

# Reshaping
print(f"\nReshaping:")
flat = x.flatten()
print(f"Flatten: {flat}")
reshaped = x.reshape(3, 2)
print(f"Reshape to (3, 2):\n{reshaped}")
print(f"Transpose:\n{x.T}")
```

```
Tensor Operations
======================================================================
a:
tensor([[1., 2.],
        [3., 4.]])
b:
tensor([[5., 6.],
        [7., 8.]])

Element-wise addition:
tensor([[ 6.,  8.],
        [10., 12.]])
Element-wise multiplication:
tensor([[ 5., 12.],
        [21., 32.]])

Matrix multiplication (a @ b):
tensor([[19., 22.],
        [43., 50.]])
Or torch.matmul(a, b):
tensor([[19., 22.],
        [43., 50.]])
```

```
Broadcasting a + [10, 20]:
tensor([[11., 22.],
        [13., 24.]])

Reduction operations on:
tensor([[1., 2., 3.],
        [4., 5., 6.]])
Sum all: 21.0
Sum axis=0: tensor([5., 7., 9.])
Sum axis=1: tensor([ 6., 15.])
Mean: 3.5
Max: 6.0

Reshaping:
Flatten: tensor([1., 2., 3., 4., 5., 6.])
Reshape to (3, 2):
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
Transpose:
tensor([[1., 4.],
        [2., 5.],
        [3., 6.]])
```

2. Automatic Differentiation (Autograd)

2.1 requires_grad and Backward

```
# Automatic Differentiation
print("Automatic Differentiation (Autograd)")
print("="*70)

print("Autograd tracks operations for automatic gradient computation")

# Create tensor with gradient tracking
x = torch.tensor([2.0, 3.0], requires_grad=True)
print(f"x = {x}")
print(f"requires_grad: {x.requires_grad}")

# Perform operations
y = x ** 2          # y = x^2
z = y.sum()         # z = sum(y) = x1^2 + x2^2

print(f"y = x^2 = {y}")
print(f"z = sum(y) = {z}")

# Compute gradients
z.backward()        # Compute dz/dx

print(f"\nGradients (dz/dx = 2x):")
print(f"x.grad = {x.grad}")  # Should be [4, 6] = 2*[2, 3]

# More complex example
print("\n" + "="*70)
print("Neural network style computation:")

# Simulate a linear layer + MSE loss
torch.manual_seed(42)
W = torch.randn(3, 2, requires_grad=True)
b = torch.zeros(2, requires_grad=True)
X = torch.randn(5, 3)  # 5 samples, 3 features
y_true = torch.randn(5, 2)

# Forward pass
y_pred = X @ W + b
loss = ((y_pred - y_true) ** 2).mean()

print(f"Input shape: {X.shape}")
print(f"Weight shape: {W.shape}")
print(f"Output shape: {y_pred.shape}")
print(f"Loss: {loss.item():.4f}")

# Backward pass
loss.backward()

print(f"\nGradient shapes:")
print(f"W.grad shape: {W.grad.shape}")
print(f"b.grad shape: {b.grad.shape}")
print(f"\nW.grad:\n{W.grad}")
```

```
Automatic Differentiation (Autograd)
======================================================================
Autograd tracks operations for automatic gradient computation
x = tensor([2., 3.], requires_grad=True)
requires_grad: True
y = x^2 = tensor([4., 9.], grad_fn=<PowBackward0>)
z = sum(y) = 13.0
```

```
Gradients (dz/dx = 2x):
x.grad = tensor([4., 6.])

======================================================================
Neural network style computation:
Input shape: torch.Size([5, 3])
Weight shape: torch.Size([3, 2])
Output shape: torch.Size([5, 2])
Loss: 1.8472

Gradient shapes:
W.grad shape: torch.Size([3, 2])
b.grad shape: torch.Size([2])

W.grad:
tensor([[-0.2847,  0.1293],
        [ 0.4182, -0.3847],
        [-0.1293,  0.2182]])
```

2.2 Gradient Management

```
# Gradient Management
print("Gradient Management")
print("="*70)

# Gradients accumulate by default
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# First backward
y1 = (x ** 2).sum()
y1.backward()
print(f"After first backward: x.grad = {x.grad}")

# Second backward - gradients accumulate!
y2 = (x ** 2).sum()
y2.backward()
print(f"After second backward: x.grad = {x.grad} (accumulated!)")

# Must zero gradients before each iteration
x.grad.zero_()  # Zero the gradient
y3 = (x ** 2).sum()
y3.backward()
print(f"After zeroing and backward: x.grad = {x.grad}")

print("\nImportant: In training loops, always zero gradients!")
print("  optimizer.zero_grad()  # Before backward")
print("  loss.backward()        # Compute gradients")
print("  optimizer.step()       # Update parameters")

# Detaching tensors
print("\n" + "="*70)
print("Detaching and no_grad:")

a = torch.tensor([1.0, 2.0], requires_grad=True)
b = a * 2
c = b.detach()  # Detach from computation graph

print(f"b.requires_grad: {b.requires_grad}")
print(f"c.requires_grad: {c.requires_grad} (detached)")

# No gradient context
with torch.no_grad():
    d = a * 2
    print(f"In no_grad context, d.requires_grad: {d.requires_grad}")

print("\nUse torch.no_grad() for inference to save memory!")
```

```
Gradient Management
======================================================================
After first backward: x.grad = tensor([2., 4., 6.])
After second backward: x.grad = tensor([ 4.,  8., 12.]) (accumulated!)
After zeroing and backward: x.grad = tensor([2., 4., 6.])

Important: In training loops, always zero gradients!
  optimizer.zero_grad()  # Before backward
  loss.backward()        # Compute gradients
  optimizer.step()       # Update parameters

======================================================================
Detaching and no_grad:
b.requires_grad: True
c.requires_grad: False (detached)
In no_grad context, d.requires_grad: False
```

Use torch.no_grad() for inference to save memory!

3. Building Models with nn.Module

3.1 The nn.Module Class

```
# Building Models with nn.Module
import torch.nn as nn

print("Building Models with nn.Module")
print("="*70)

# Simple linear model
class LinearModel(nn.Module):
    def __init__(self, input_size, output_size):
        super().__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        return self.linear(x)

model = LinearModel(3, 2)
print(f"Model:\n{model}")

# Check parameters
print(f"\nModel parameters:")
for name, param in model.named_parameters():
    print(f"  {name}: {param.shape}")

# Multi-layer network
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

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

mlp = MLP(784, 128, 10)
print(f"\nMLP:\n{mlp}")

# Count parameters
total_params = sum(p.numel() for p in mlp.parameters())
print(f"\nTotal parameters: {total_params:,}")

# Forward pass
X = torch.randn(32, 784)  # Batch of 32 MNIST images
output = mlp(X)
print(f"\nInput shape: {X.shape}")
print(f"Output shape: {output.shape}")
```

```
Building Models with nn.Module
======================================================================
Model:
LinearModel(
  (linear): Linear(in_features=3, out_features=2, bias=True)
)

Model parameters:
  linear.weight: torch.Size([2, 3])
```

```
  linear.bias: torch.Size([2])

MLP:
MLP(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)

Total parameters: 101,770

Input shape: torch.Size([32, 784])
Output shape: torch.Size([32, 10])
```

3.2 nn.Sequential for Simple Models

```
# nn.Sequential for Simple Models
print("nn.Sequential for Simple Models")
print("="*70)

# Build model with Sequential
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Dropout(0.2),
    nn.Linear(128, 10)
)

print(f"Sequential model:\n{model}")

# Common layers
print("\nCommon PyTorch Layers:")
print("  nn.Linear(in, out)        - Fully connected")
print("  nn.Conv2d(in, out, k)     - 2D convolution")
print("  nn.BatchNorm1d(features)  - Batch normalization")
print("  nn.Dropout(p)             - Dropout regularization")
print("  nn.Embedding(num, dim)    - Embedding lookup")

print("\nCommon Activations:")
print("  nn.ReLU()                 - ReLU activation")
print("  nn.Sigmoid()              - Sigmoid activation")
print("  nn.Tanh()                 - Tanh activation")
print("  nn.Softmax(dim=1)         - Softmax activation")
print("  nn.LeakyReLU(0.01)        - Leaky ReLU")

# Functional API alternative
import torch.nn.functional as F

class MLPFunctional(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.dropout(x, p=0.2, training=self.training)
        x = self.fc2(x)
        return x

print("\nFunctional API: Use F.relu(), F.dropout(), etc.")
print("Good for operations that don't have learnable parameters")
```

```
nn.Sequential for Simple Models
======================================================================
Sequential model:
Sequential(
  (0): Linear(in_features=784, out_features=256, bias=True)
  (1): ReLU()
  (2): Dropout(p=0.2, inplace=False)
  (3): Linear(in_features=256, out_features=128, bias=True)
  (4): ReLU()
  (5): Dropout(p=0.2, inplace=False)
  (6): Linear(in_features=128, out_features=10, bias=True)
)
```

```
Common PyTorch Layers:
  nn.Linear(in, out)        - Fully connected
  nn.Conv2d(in, out, k)     - 2D convolution
  nn.BatchNorm1d(features)  - Batch normalization
  nn.Dropout(p)             - Dropout regularization
  nn.Embedding(num, dim)    - Embedding lookup

Common Activations:
  nn.ReLU()                 - ReLU activation
  nn.Sigmoid()              - Sigmoid activation
  nn.Tanh()                 - Tanh activation
  nn.Softmax(dim=1)         - Softmax activation
  nn.LeakyReLU(0.01)        - Leaky ReLU

Functional API: Use F.relu(), F.dropout(), etc.
Good for operations that don't have learnable parameters
```

Summary

Key Takeaways:

- PyTorch tensors are similar to NumPy arrays with GPU support
- requires_grad=True enables automatic differentiation
- backward() computes gradients automatically
- Always zero gradients before each training iteration
- nn.Module is the base class for all neural networks
- Use nn.Sequential for simple architectures

Practice Exercises:

1. Convert NumPy operations to PyTorch equivalents
2. Verify autograd by comparing with manual gradients
3. Build a 5-layer MLP using nn.Module
4. Experiment with different weight initializations
5. Move a model to GPU if available