PyTorch: A Comprehensive Introduction

Sayan Chaki

sayan.chaki@inria.fr

March 20, 2025

Outline

What is PyTorch?

- Open-source machine learning library developed by Facebook's AI Research lab (FAIR)
- Flexible, intuitive, and Pythonic approach to building deep learning models
- Uses dynamic computational graphs (define-by-run approach)
- Particularly suitable for research and experimentation

Key Features of PyTorch

- **Dynamic Computation Graph**: Constructed at runtime, allows intuitive debugging
- **Pythonic Interface**: Integrates seamlessly with Python
- **Strong GPU Acceleration**: Excellent support for GPU computation
- **Rich Ecosystem**: Includes torchvision, torchtext, and more
- **Production Readiness**: TorchScript and ONNX integrations

PyTorch Adoption

- Significant traction in academic research due to flexibility
- Increasingly adopted in industry settings
- Used by companies like Tesla, Uber, and Facebook in production
- Popular for research due to its dynamic nature and ease of debugging

Tensors: The Core Data Structure

- Fundamental building blocks of PyTorch
- Multi-dimensional arrays similar to NumPy's ndarray
- Additional capabilities:
  - GPU acceleration
  - Automatic differentiation
- Primary data structure for all operations in PyTorch

Tensor Dimensions

- **0D tensor (scalar)**: A single value
- **1D tensor (vector)**: A list of values
- **2D tensor (matrix)**: A table of values
- **3D tensor**: A cube of values
- **Higher dimensional tensors**: More complex data structures

Creating Tensors

```python
import torch

# Create a scalar (0D tensor)
scalar = torch.tensor(5)

# Create a vector (1D tensor)
vector = torch.tensor([1, 2, 3])

# Create a matrix (2D tensor)
matrix = torch.tensor([[1, 2], [3, 4]])

# Create a 3D tensor
tensor_3d = torch.tensor([[[1, 2], [3, 4]],
                          [[5, 6], [7, 8]]])
```

Tensor Properties

```
# Check tensor's shape
print(matrix.shape)  # Output: torch.Size([2, 2])

# Check tensor's data type
print(matrix.dtype)  # Output: torch.int64

# Check tensor's device
print(matrix.device)  # Output: device(type='cpu')

# Move tensor to GPU (if available)
if torch.cuda.is_available():
    tensor_gpu = tensor.to('cuda')
```

Tensor Operations

```
# Addition
x = torch.tensor([1, 2, 3])
y = torch.tensor([4, 5, 6])
z = x + y  # Element-wise addition

# Matrix multiplication
a = torch.tensor([[1, 2], [3, 4]])
b = torch.tensor([[5, 6], [7, 8]])
c = torch.matmul(a, b)
# Or using the @ operator
c = a @ b
```

Tensors and Autograd

```python
# Create a tensor with requires_grad=True
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

# Perform operations
y = x * x
z = y.mean()

# Compute gradients
z.backward()

# Access gradients
print(x.grad)  # Output: tensor([0.6667, 1.3333, 2.0000])
```

Tensors in Deep Learning

Tensors are used to represent various types of data:

- Input data (images, text, etc.)
- Model parameters (weights and biases)
- Activations during forward pass
- Gradients during backward pass
- Loss values and prediction outputs

Building Neural Networks in PyTorch

- PyTorch provides the `torch.nn` module for neural network design
- Key steps:
  - Define network architecture
  - Initialize parameters
  - Implement the forward pass
- Based on the foundational `nn.Module` class

nn.Module: The Building Block

Key aspects of the `nn.Module` class:

- Base class for all neural network models in PyTorch
- **Automatic Parameter Registration**: Parameters defined in `__init__()` are tracked
- **forward() Method**: Defines the data flow during the forward pass
- **Parameter Access**: Provides methods like `parameters()` and `named_parameters()`

Basic Neural Network Class

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size)
    :
        super(SimpleNetwork, self).__init__()

        # Define layers
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Define forward pass
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Common Neural Network Layers

PyTorch provides a wide range of pre-implemented layers:

- **Linear Layers**: `nn.Linear`
- **Convolutional Layers**: `nn.Conv1d`, `nn.Conv2d`, `nn.Conv3d`
- **Recurrent Layers**: `nn.RNN`, `nn.LSTM`, `nn.GRU`
- **Normalization Layers**: `nn.BatchNorm1d`, `nn.LayerNorm`
- **Pooling Layers**: `nn.MaxPool2d`, `nn.AvgPool2d`
- **Dropout Layers**: `nn.Dropout`

Activation Functions

PyTorch provides activation functions in both modules and functional forms:

- ReLU: `nn.ReLU()` or `F.relu()`
- Sigmoid: `nn.Sigmoid()` or `F.sigmoid()`
- Tanh: `nn.Tanh()` or `F.tanh()`
- Softmax: `nn.Softmax()` or `F.softmax()`
- LeakyReLU: `nn.LeakyReLU()` or `F.leaky_relu()`
- ELU: `nn.ELU()` or `F.elu()`

Convolutional Neural Network Example

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

        # Convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding
=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3,
padding=1)

        # Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(64 * 8 * 8, 512)
        self.fc2 = nn.Linear(512, 10)

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

Convolutional Neural Network Example (cont.)

```
    def forward(self, x):
        # First conv block
        x = F.relu(self.conv1(x))
        x = self.pool(x)

        # Second conv block
        x = F.relu(self.conv2(x))
        x = self.pool(x)

        # Flatten
        x = x.view(-1, 64 * 8 * 8)

        # Fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x
```

Sequential API

For simpler networks with a linear topology:

```
model = nn.Sequential(
    nn.Linear(784, 256),
    nn.ReLU(),
    nn.Linear(256, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)
```

Model Initialization

PyTorch provides various weight initialization methods:

```
def weight_init(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)

model.apply(weight_init)
```

Backpropagation in PyTorch

- Cornerstone algorithm for training neural networks
- Computes gradients to update model parameters
- PyTorch's **autograd** handles backpropagation automatically
- Enables efficient training of complex neural networks

Autograd: Key Components

- **Dynamic Computational Graph**: Built on-the-fly during forward pass
- **Tensor History**: Tensors with `requires_grad=True` track their history
- **Gradient Computation**: The `.backward()` method computes gradients
- **Gradient Accumulation**: Gradients accumulate in the `.grad` attribute

Training Loop in PyTorch

```python
# Initialize model, loss function, and optimizer
model = SimpleNetwork(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# Training loop
for epoch in range(num_epochs):
    for inputs, targets in data_loader:
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, targets)

        # Backward pass and optimization
        optimizer.zero_grad()  # Clear previous gradients
        loss.backward()        # Compute gradients
        optimizer.step()       # Update parameters
```

Backpropagation Steps

1. **Forward Pass**: Model processes inputs and computes outputs
2. **Loss Computation**: Measures difference between predictions and targets
3. **Gradient Clearing**: `optimizer.zero_grad()` clears previous gradients
4. **Backward Pass**: `loss.backward()` computes gradients for all parameters
5. **Parameter Update**: `optimizer.step()` updates model parameters

Autograd Mechanics

When you call `loss.backward()`, PyTorch:

- Traverses the computational graph backward
- Applies the chain rule of calculus
- For scalar loss L and parameter θ, computes $\frac{\partial L}{\partial \theta}$
- Each operation has a corresponding gradient function
- Gradients flow from the loss to input parameters

Gradient Flow Control

PyTorch provides mechanisms to control gradient flow:

```
# Detach a tensor from the computational graph
detached_tensor = tensor.detach()

# Temporarily disable gradient tracking
with torch.no_grad():
    # Operations here don't track gradients
    result = model(input_data)

# Prevent gradient computation for specific parameters
frozen_layer.weight.requires_grad = False
```

Custom Autograd Functions

For advanced use cases, create custom autograd functions:

```python
class CustomFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Forward computation
        result = input * 2
        ctx.save_for_backward(input)
        return result

    @staticmethod
    def backward(ctx, grad_output):
        # Backward computation
        input, = ctx.saved_tensors
        grad_input = grad_output * 2
        return grad_input
```

Optimizers in PyTorch

PyTorch provides various optimization algorithms:

```
# Different optimizer examples
sgd_optimizer = torch.optim.SGD(
    model.parameters(), lr=0.01, momentum=0.9)

adam_optimizer = torch.optim.Adam(
    model.parameters(), lr=0.001, betas=(0.9, 0.999))

rmsprop_optimizer = torch.optim.RMSprop(
    model.parameters(), lr=0.01, alpha=0.99)
```

Popular Optimizers

- **SGD**: Simple gradient descent with momentum option
- **Adam**: Adaptive moment estimation, per-parameter learning rates
- **RMSprop**: Adapts learning rates based on moving average of squared gradients
- **Adagrad**: Accumulates squared gradients to adjust learning rates
- **LBFGS**: Limited-memory BFGS, a quasi-Newton method

Conclusion

- PyTorch offers a flexible, intuitive framework for deep learning
- Tensors are the fundamental building blocks
- Neural networks are built using the `nn.Module` architecture
- Autograd handles backpropagation automatically
- Dynamic computation graphs make PyTorch particularly suitable for research
- Rich ecosystem of tools and libraries for various applications

Thank You!

Questions?