Introduction to PyTorch Programming Patterns

Deep Learning and Modern Applications

Theja Tulabandhula

University of Illinois Chicago

After this lecture, you will be able to:

- Use PyTorch's device abstraction for GPU/CPU computation
- Build data pipelines using DataLoader and transforms
- Define models using the nn.Module class pattern
- Implement the standard training loop correctly
- Control gradient computation and model behavior modes
- Manage model state for saving, loading, and transfer learning

The PyTorch Philosophy

- **Imperative programming**: Write code that executes immediately
- **Dynamic computation graphs**: Build graphs on-the-fly each forward pass
- **Pythonic**: Feels like native Python, uses standard debugging tools
- **Research-friendly**: Easy to experiment and modify

PyTorch vs Other Frameworks

Aspect	PyTorch	TensorFlow 1.x	TensorFlow 2.x
Graph	Dynamic	Static	Eager by default
Debugging	Standard Python	tf.Session complexity	Improved
API Style	Object-oriented	Symbolic	Keras-like
Research Adoption	Very High	Moderate	Growing

PyTorch is now the **dominant framework** in research publications.

The Device Abstraction

PyTorch uses a device abstraction to manage CPU/GPU computation:

```
import torch

# Check GPU availability
device = torch.device("cuda:0" if torch.cuda.is_available()
                      else "cpu")
```

This pattern appears in **every** PyTorch script.

Moving Data to Device

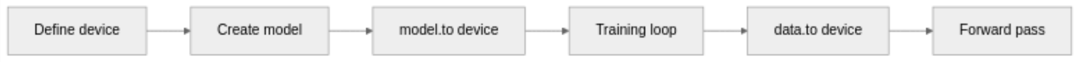
```python
# Create tensor on CPU
x = torch.randn(3, 4)

# Move to device (GPU if available)
x = x.to(device)

# Create directly on device
y = torch.zeros(3, 4, device=device)
```

Key insight: Both data and model must be on the **same device**.

Device Management Pattern



Always move data to device **inside** the training loop.

The Three Components

1. **Dataset**: Holds data, implements `__getitem__` and `__len__`
2. **Transforms**: Preprocessing and augmentation
3. **DataLoader**: Batching, shuffling, parallel loading

Transform Composition

```
from torchvision import transforms

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
```

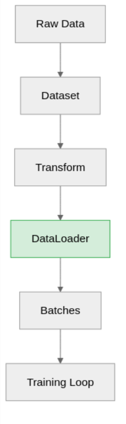
DataLoader Configuration

```
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,        # Randomize order each epoch
    num_workers=4        # Parallel data loading
)
```

Parameter	Training	Validation
shuffle	True	False
batch_size	Tunable	Same or larger
drop_last	Often True	False

Data Loading Flow



DataLoader handles batching and parallel loading automatically.

The nn.Module Contract

Every PyTorch model inherits from `nn.Module`:

```
import torch.nn as nn

class MyNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        # Define layers here

    def forward(self, x):
        # Define computation here
        return x
```

Two required methods: `__init__` and `forward`.

Layer Definition Pattern

```
class SimpleNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_classes):
        super().__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.linear1(x))
        x = self.linear2(x)
        return x
```

Layers defined in `__init__`, used in `forward`.

What nn.Module Gives You

- **Parameter tracking**: All weights automatically registered
- **Device movement**: `.to(device)` moves all parameters
- **State management**: `state_dict()` captures all weights
- **Mode switching**: `train()` and `eval()` methods
- **Gradient control**: Works with autograd system

Common Layer Types

Layer	Purpose	Example
`nn.Linear`	Fully connected	`nn.Linear(784, 256)`
`nn.Conv2d`	2D convolution	`nn.Conv2d(3, 64, 3)`
`nn.ReLU`	Activation	`nn.ReLU()`
`nn.Dropout`	Regularization	`nn.Dropout(0.5)`
`nn.BatchNorm2d`	Normalization	`nn.BatchNorm2d(64)`

The Core Pattern

```
for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()       # 1. Clear gradients
        outputs = model(inputs)     # 2. Forward pass
        loss = criterion(outputs, labels)  # 3. Compute loss
        loss.backward()             # 4. Backward pass
        optimizer.step()            # 5. Update weights
```

This 5-step pattern is **universal** in PyTorch training.

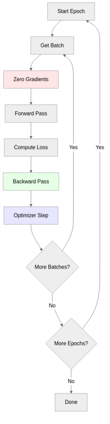
Why Zero Gradients?

- PyTorch **accumulates** gradients by default
- Without zeroing: gradients from previous batch add up
- Must call `optimizer.zero_grad()` before each backward pass

```python
# These are equivalent:
optimizer.zero_grad()

# or manually:
for param in model.parameters():
    if param.grad is not None:
        param.grad.zero_()
```

Training Loop Visualization



Common Loss Functions

```python
import torch.nn as nn

# Classification
criterion = nn.CrossEntropyLoss()  # Multi-class
criterion = nn.BCEWithLogitsLoss() # Binary

# Regression
criterion = nn.MSELoss()           # Mean squared error
criterion = nn.L1Loss()            # Mean absolute error
```

Note: `CrossEntropyLoss` combines LogSoftmax + NLLLoss.

Optimizer Selection

```
import torch.optim as optim

# Stochastic Gradient Descent
optimizer = optim.SGD(model.parameters(),
                      lr=0.01,
                      momentum=0.9)

# Adam (adaptive learning rate)
optimizer = optim.Adam(model.parameters(),
                       lr=0.001)
```

Optimizer Comparison

Optimizer	Pros	Cons	When to Use
SGD	Simple, well-understood	Slow convergence	With LR scheduling
SGD+Momentum	Faster than SGD	Extra hyperparameter	CNNs often
Adam	Fast, adaptive	May generalize worse	Default starting point
AdamW	Better regularization	-	Transformers

Passing Parameters to Optimizer

```python
# All parameters
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Only specific layers
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)

# Different learning rates per layer
optimizer = optim.Adam([
    {'params': model.features.parameters(), 'lr': 1e-4},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
])
```

Two Model Modes

```
model.train()  # Training mode
model.eval()   # Evaluation mode
```

These modes affect behavior of certain layers.

Layers Affected by Mode

Layer	train()	eval()
Dropout	Randomly drops units	No dropout applied
BatchNorm	Uses batch statistics	Uses running statistics
Other layers	No change	No change

Critical: Always set correct mode before forward pass.

Correct Usage Pattern

```python
# Training
model.train()
for inputs, labels in train_loader:
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

# Evaluation
model.eval()
with torch.no_grad():
    for inputs, labels in val_loader:
        outputs = model(inputs)
        # Compute metrics...
```

Common Mistake

```python
# WRONG: Forgot to set eval mode
def predict(model, inputs):
    return model(inputs)  # Dropout still active!

# CORRECT: Set eval mode
def predict(model, inputs):
    model.eval()
    with torch.no_grad():
        return model(inputs)
```

Dropout during inference = **non-deterministic** predictions.

torch.no_grad()

Disables gradient computation:

```python
with torch.no_grad():
    outputs = model(inputs)
    # No gradients computed
    # Cannot call loss.backward()
```

Benefits:

- Reduces memory usage
- Speeds up computation
- Essential for inference

torch.set_grad_enabled()

More flexible gradient control:

```python
# Conditional gradient computation
with torch.set_grad_enabled(phase == 'train'):
    outputs = model(inputs)
    loss = criterion(outputs, labels)

    if phase == 'train':
        loss.backward()
        optimizer.step()
```

Useful for combined train/val loops.

Memory Implications

Context	Gradients Stored	Memory	Use Case
Default	Yes	High	Training
no_grad	No	Low	Inference
inference_mode	No	Lowest	Production

```python
# Lowest memory for inference
with torch.inference_mode():
    outputs = model(inputs)
```

Gradient Context Flow

The state_dict Pattern

```
# Get all model parameters as dictionary
state = model.state_dict()

# Keys are parameter names
print(state.keys())
# ['linear1.weight', 'linear1.bias', 'linear2.weight', ...]
```

This is how PyTorch serializes models.

Saving and Loading Models

```python
# Save model weights
torch.save(model.state_dict(), 'model.pth')

# Load model weights
model = MyNetwork()  # Create architecture first
model.load_state_dict(torch.load('model.pth'))
model.eval()
```

Best practice: Save `state_dict`, not entire model.

Saving Complete Checkpoints

```python
# Save everything for resuming training
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
torch.save(checkpoint, 'checkpoint.pth')

# Resume training
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

Copying Model Weights

```python
import copy

# Deep copy for best model tracking
best_model_wts = copy.deepcopy(model.state_dict())

# Later, restore best weights
model.load_state_dict(best_model_wts)
```

Useful for early stopping and model selection.

The requires_grad Flag

Every tensor has a `requires_grad` attribute:

```
# Check if parameter needs gradients
for name, param in model.named_parameters():
    print(name, param.requires_grad)

# Freeze a parameter
param.requires_grad = False
```

Frozen parameters receive **no gradient updates**.

Freezing for Transfer Learning

```python
# Load pretrained model
model = models.resnet18(pretrained=True)

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

# Replace and train only final layer
model.fc = nn.Linear(512, num_classes)
# New layer has requires_grad=True by default
```

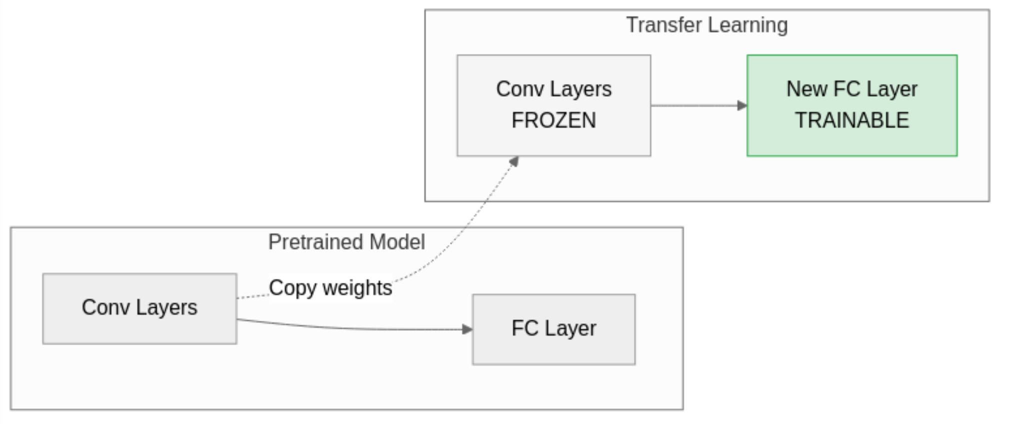
Only the new layer receives gradients.

Selective Freezing

```python
# Freeze by name pattern
for name, param in model.named_parameters():
    if 'layer1' in name or 'layer2' in name:
        param.requires_grad = False

# Only optimize unfrozen parameters
optimizer = optim.Adam(
    filter(lambda p: p.requires_grad, model.parameters()),
    lr=0.001
)
```

Freezing Visualization



Why Schedule Learning Rates?

- **Large LR early**: Fast initial progress
- **Small LR later**: Fine-tune near optimum
- Helps escape local minima
- Improves final convergence

Common Schedulers

```
from torch.optim.lr_scheduler import StepLR, ExponentialLR

# Decay by factor every N epochs
scheduler = StepLR(optimizer, step_size=7, gamma=0.1)

# Decay by factor every epoch
scheduler = ExponentialLR(optimizer, gamma=0.95)

# Cosine annealing
scheduler = CosineAnnealingLR(optimizer, T_max=100)
```

Scheduler in Training Loop

```python
for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    # Step scheduler AFTER epoch
    scheduler.step()

    print(f"LR: {scheduler.get_last_lr()}")
```

Note: Call `scheduler.step()` after each epoch.

Scheduler Comparison

Scheduler	Behavior	Use Case
StepLR	Discrete drops	Most common
ExponentialLR	Smooth decay	Gradual refinement
CosineAnnealingLR	Cosine curve	Vision models
ReduceLROnPlateau	Adaptive	When loss plateaus

Loading Pretrained Models

```
from torchvision import models

# Load with pretrained weights
model = models.resnet18(pretrained=True)

# Or with new API (PyTorch 2.0+)
model = models.resnet18(
    weights=models.ResNet18_Weights.IMAGENET1K_V1
)
```

Models pretrained on ImageNet's 1000 classes.

Modifying for Your Task

```
# Original final layer
print(model.fc)
# Linear(in_features=512, out_features=1000)

# Replace for binary classification
num_classes = 2
model.fc = nn.Linear(model.fc.in_features, num_classes)

# Move to device
model = model.to(device)
```

New layer has random weights, ready to train.

Two Transfer Learning Strategies

Feature Extraction (freeze backbone):

```python
for param in model.parameters():
    param.requires_grad = False
model.fc = nn.Linear(512, num_classes)  # Only this trains
```

Fine-tuning (train everything):

```python
model.fc = nn.Linear(512, num_classes)
# All parameters trainable, but use small LR
optimizer = optim.SGD(model.parameters(), lr=0.001)
```

When to Use Each Strategy

Scenario	Strategy	Learning Rate
Small dataset, similar domain	Feature extraction	Normal
Large dataset	Fine-tuning	Small for pretrained layers
Different domain	Fine-tuning	Very small
Very small dataset	Feature extraction + simple head	Normal

What is Parameterization?

A way to add transformations to existing weights:

```
import torch.nn.utils.parametrize as parametrize

class ScaleWeight(nn.Module):
    def forward(self, W):
        return W * 2.0  # Double all weights

parametrize.register_parametrization(
    layer, "weight", ScaleWeight()
)
```

The weight is now computed through the parameterization.

Use Cases

- **Spectral normalization**: Constrain weight norms
- **Weight orthogonalization**: Maintain orthogonal weights
- **LoRA**: Add low-rank updates (see LoRA lecture)
- **Pruning**: Mask out weights

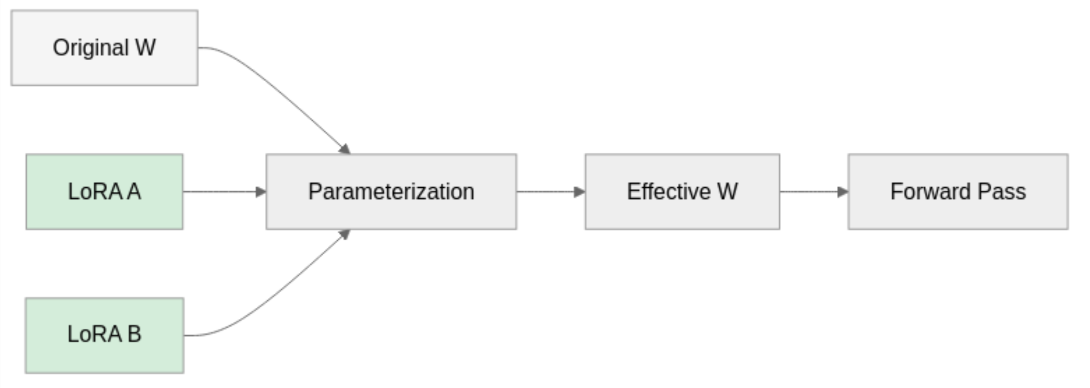
LoRA as Parameterization

```python
class LoRAParametrization(nn.Module):
    def __init__(self, features_in, features_out, rank=8):
        super().__init__()
        self.lora_A = nn.Parameter(torch.randn(rank, features_out))
        self.lora_B = nn.Parameter(torch.zeros(features_in, rank))

    def forward(self, W):
        return W + self.lora_B @ self.lora_A
```

Original weights + low-rank update.

Parameterization Flow



Complete Training Script Template

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader

# 1. Device setup
device = torch.device("cuda" if torch.cuda.is_available()
                      else "cpu")

# 2. Data loading
train_loader = DataLoader(train_dataset, batch_size=32,
                          shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)
```

Template (continued)

```
# 3. Model setup
model = MyNetwork().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer,
                                       step_size=7, gamma=0.1)

# 4. Training loop
for epoch in range(num_epochs):
    model.train()
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
```

Template (continued)

```
    # 5. Validation
    model.eval()
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            # Compute metrics...

    scheduler.step()

# 6. Save model
torch.save(model.state_dict(), 'final_model.pth')
```

M03 Notebooks Overview

Notebook	Patterns Demonstrated
ConvolutionalNet_Classifier	Data transforms, pretrained models, training loop
Overfitting_Regularization	Model comparison, regularization
Transfer_Learning_LoRA	Parameter freezing, parameterization
TSNE_Embedding	Feature extraction from trained models

We will explore these patterns hands-on in the live demos.

Core Patterns

1. **Device Management**: Always define device, move data and model
2. **Data Pipeline**: Dataset $\rightarrow$ Transforms $\rightarrow$ DataLoader
3. **nn.Module**: `__init__` defines layers, `forward` defines computation
4. **Training Loop**: zero_grad $\rightarrow$ forward $\rightarrow$ loss $\rightarrow$ backward $\rightarrow$ step
5. **Loss & Optimizer**: Choose based on task and model

Advanced Patterns

6. **Train/Eval Mode**: Set correctly before inference
7. **Gradient Contexts**: Use `no_grad()` for inference
8. **State Management**: Save/load with `state_dict()`
9. **Parameter Freezing**: Control `requires_grad` for transfer learning
10. **LR Scheduling**: Decay learning rate during training
11. **Pretrained Models**: Load, modify, fine-tune
12. **Parameterization**: Add transformations to weights

What to Remember

- PyTorch patterns are **consistent** across projects
- The training loop structure is **universal**
- Mode switching (`train`/`eval`) is **critical** for correct behavior
- State management enables **reproducibility** and **transfer learning**

PyTorch Documentation

- PyTorch Tutorials: pytorch.org/tutorials
- nn.Module documentation: pytorch.org/docs/stable/nn.html
- Parametrization: pytorch.org/tutorials/intermediate/parametrizations.html

Further Reading

- D2L Book: d2l.ai - Dive into Deep Learning
- CS231n: cs231n.github.io - CNN for Visual Recognition
- Transfer Learning Tutorial: pytorch.org/tutorials/beginner/transfer_learning_tutorial.html