



Advanced  
Institute for  
Artificial  
Intelligence

# Introduction to PyTorch

---

<https://advancedinstitute.ai>

## References and Resources

- PyTorch Official Documentation
- PyTorch Tutorials
- Deep Learning Book (Goodfellow et al.)
- Dive into Deep Learning (Interactive Book)
- PyTorch Examples Repository
- Fast.ai - Practical Deep Learning



# Part 1

---

What is PyTorch?

# What is PyTorch?

## Introduction

- Open-source **deep learning framework** developed by Meta AI (Facebook)
- Released in 2016, now one of the **most popular** frameworks
- Key features:
  - **Dynamic computational graphs** (define-by-run)
  - **GPU acceleration** for fast computation
  - **Pythonic** - feels natural for Python developers
  - **Automatic differentiation** (autograd)
  - Strong **research community** and industry adoption
- Used by: Meta, Tesla, Uber, Microsoft, OpenAI, and many others

# Why PyTorch?

## Advantages

### □ Ease of use:

- Pythonic API - natural for Python developers
- Intuitive and flexible
- Easy debugging (standard Python debugging tools)

### □ Dynamic graphs:

- Build graphs on-the-fly
- Different computation for each input
- Ideal for research and experimentation

### □ Performance:

- GPU acceleration
- Optimized operations
- Production-ready with TorchScript

### □ Ecosystem:

- Rich libraries: TorchVision, TorchText, TorchAudio

# PyTorch vs TensorFlow

## Main Competitors

### □ PyTorch:

- Dynamic computational graphs (eager execution)
- More Pythonic, easier to debug
- Preferred in research community
- Gaining in production deployments

### □ TensorFlow:

- Static graphs (historically), now also eager with TF 2.x
- Stronger production/deployment tools
- TensorFlow Lite for mobile
- Preferred in industry (historically)

□ **Trend:** Both are converging - PyTorch adding production tools, TensorFlow becoming more Pythonic

□ **For today:** We focus on PyTorch!

# PyTorch Ecosystem

## Related Libraries

- **Core PyTorch:** torch
- **Computer Vision:** torchvision
  - Pre-trained models, datasets, transforms
- **Natural Language Processing:** torchtext
  - Text processing, datasets, vocabulary
- **Audio Processing:** torchaudio
  - Audio transforms, datasets
- **Higher-level APIs:**
  - **PyTorch Lightning:** high-level wrapper
  - **Fast.ai:** even higher-level, for practitioners
- **Deployment:**
  - **TorchServe:** model serving
  - **TorchScript:** production optimization



## Part 2

---

Tensors - The Foundation

# What are Tensors?

## Fundamental Data Structure

□ **Tensor:** Multi-dimensional array (generalization of matrices)

□ **Dimensions:**

- 0D tensor: scalar (single number)
- 1D tensor: vector (array)
- 2D tensor: matrix
- 3D tensor: cube of numbers
- 4D+ tensor: higher dimensions

□ **Similar to NumPy arrays, but:**

- Can run on GPU
- Track gradients for automatic differentiation
- Optimized for deep learning

□ **Everything in PyTorch is a tensor!**

# Creating Tensors

## Basic Operations

```
1 import torch
2
3 # From Python list
4 x = torch.tensor([1, 2, 3])    # 1D tensor
5
6 # From NumPy array
7 import numpy as np
8 arr = np.array([[1, 2], [3, 4]])
9 x = torch.from_numpy(arr)    # 2D tensor
10
11 # Special tensors
12 zeros = torch.zeros(3, 4)      # 3x4 tensor of zeros
13 ones = torch.ones(2, 3)        # 2x3 tensor of ones
14 random = torch.randn(2, 3)     # Random normal distribution
15 identity = torch.eye(3)       # 3x3 identity matrix
```

# Tensor Attributes

## Understanding Tensor Properties

```
1 x = torch.randn(3, 4, 5)
2
3 # Check shape
4 print(x.shape)      # torch.Size([3, 4, 5])
5 print(x.size())     # torch.Size([3, 4, 5]) - same
6
7 # Check data type
8 print(x.dtype)      # torch.float32 (default)
9
10 # Check device (CPU or GPU)
11 print(x.device)    # cpu or cuda:0
12
13 # Number of elements
14 print(x.numel())   # 60 (3 * 4 * 5)
```

# Tensor Operations

## Basic Arithmetic

### □ Element-wise operations:

- Addition:  $a + b$  or `torch.add(a, b)`
- Subtraction:  $a - b$
- Multiplication:  $a * b$
- Division:  $a / b$

### □ Matrix operations:

- Matrix multiplication:  $a @ b$  or `torch.matmul(a, b)`
- Transpose:  $a.T$  or `a.transpose(0, 1)`

### □ In-place operations: (modify tensor directly)

- `a.add_(b)`, `a.mul_(2)`
- Note the underscore suffix!

# Tensor Operations Example

## Common Operations

```
1 # Create tensors
2 a = torch.tensor([[1, 2], [3, 4]])
3 b = torch.tensor([[5, 6], [7, 8]])
4 # Element-wise operations
5 c = a + b          # [[6, 8], [10, 12]]
6 d = a * 2          # [[2, 4], [6, 8]]
7
8 # Matrix multiplication
9 e = a @ b          # [[19, 22], [43, 50]]
10
11 # Reshaping
12 f = a.view(4)      # [1, 2, 3, 4] - flatten
13 g = a.reshape(1, 4) # [[1, 2, 3, 4]]
14
15 # Indexing (like NumPy)
16 print(a[0, 1])    # 2
17 print(a[:, 1])    # [2, 4] - second column
```

## Moving Tensors to GPU

### □ Why GPU?

- Massive parallelization
- 10-100x faster for large operations
- Essential for training deep networks

### □ Check GPU availability:

- `torch.cuda.is_available()`

### □ Move tensors to GPU:

- `x = x.to('cuda')`
- `x = x.cuda()`

### □ Move back to CPU:

- `x = x.to('cpu')`
- `x = x.cpu()`

□ **Important:** All tensors in an operation must be on same device!

# GPU Example

## Using CUDA

```
1 # Check if GPU is available
2 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
3
4 # Create tensor on GPU directly
5 x = torch.randn(1000, 1000, device=device)
6
7 # Or move existing tensor to GPU
8 y = torch.randn(1000, 1000)
9 y = y.to(device)
10
11 # Perform operations (much faster on GPU!)
12 z = x @ y # Matrix multiplication on GPU
13
14 # Move back to CPU for NumPy conversion
15 z_cpu = z.cpu()
16 z_numpy = z_cpu.numpy()
```



## Part 3

---

Automatic Differentiation

# What is Autograd?

## Automatic Differentiation

□ **Autograd:** PyTorch's automatic differentiation engine

□ **Why do we need it?**

- Deep learning = optimizing functions
- Optimization requires **gradients** (derivatives)
- Manual gradient calculation is error-prone and tedious

□ **How it works:**

- PyTorch tracks all operations on tensors
- Builds a **computational graph**
- Automatically computes gradients via **backpropagation**

□ **Enable gradient tracking:**

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

□ **Compute gradients:**

- `loss.backward()`

# Autograd Example

## Computing Gradients

```
1 # Create tensor with gradient tracking
2 x = torch.tensor([2.0], requires_grad=True)
3 # Forward pass: compute function
4 y = x ** 2 + 3 * x + 1 # y = x2 + 3x + 1
5 # Backward pass: compute gradient dy/dx
6 y.backward()
7 # Access gradient
8 print(x.grad) # dy/dx = 2x + 3 = 2(2) + 3 = 7
9
10 # For multiple steps, remember to zero gradients
11 x.grad.zero_() # Important!
12
13 # Another computation
14 z = x ** 3
15 z.backward()
16 print(x.grad) # dz/dx = 3x2 = 3(4) = 12
```

# Computational Graph

## How Autograd Tracks Operations

### □ Dynamic computational graph:

- Built on-the-fly during forward pass
- Each operation creates a node
- Tracks dependencies between tensors

### □ Example: $z = (x + y) * w$

- Node 1:  $a = x + y$
- Node 2:  $z = a * w$
- Graph:  $x, y, w \rightarrow a \rightarrow z$

### □ Backward pass:

- Traverse graph in reverse
- Apply chain rule automatically
- Accumulate gradients

### □ After `backward()`: Graph is destroyed (for memory efficiency)

# Gradient Accumulation

## Important Concept

```
1 x = torch.tensor([2.0], requires_grad=True)
2
3 # First computation
4 y = x ** 2
5 y.backward()
6 print(x.grad) # 4.0
7
8 # Second computation (without zeroing!)
9 z = x ** 3
10 z.backward()
11 print(x.grad) # 16.0 = 4.0 + 12.0 (accumulated!)
12 # Always zero gradients before new backward pass
13 x.grad.zero_()
14 w = x ** 4
15 w.backward()
16 print(x.grad) # 32.0 (fresh gradient)
```

## Important Concept

- **Key point:** Gradients accumulate by default!
- **Always** `zero_grad()` in training loops



## Part 4

---

Training with PyTorch: Linear Regression

# Why Start with Linear Regression?

## Simple but Complete Example

- **Linear regression:** Fitting a line to data
  - Simple:  $y = wx + b$
  - But uses ALL PyTorch training concepts!
- **Advantages for learning:**
  - Easy to visualize
  - Fast to train
  - Shows complete workflow
  - Same patterns apply to complex models
- **You'll learn:** Model definition, Loss functions, Optimizers, Training loop
- **Then:** Apply same concepts to neural networks!

# Problem Setup

## Linear Regression Task

- **Goal:** Predict house prices based on size
- **Model:**  $\text{price} = w \times \text{size} + b$ 
  - $w$  = weight (slope)
  - $b$  = bias (intercept)
- **Training data:**
  - Multiple (size, price) pairs
  - Example:  $(50\text{m}^2, \text{R\$}200\text{k}), (80\text{m}^2, \text{R\$}320\text{k}), \dots$
- **Task:** Find best  $w$  and  $b$ 
  - Minimize error between predictions and actual prices
- **Loss function:** Mean Squared Error (MSE)
  - $\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$

# Generate Synthetic Data

## Creating Training Data

```
1 import torch
2 import matplotlib.pyplot as plt
3 # Set random seed for reproducibility
4 torch.manual_seed(42)
5
6 # Generate synthetic data: y = 2x + 3 + noise
7 n_samples = 100
8 X = torch.randn(n_samples, 1) * 10 # House sizes (random)
9 y = 2 * X + 3 + torch.randn(n_samples, 1) * 2 # Prices with noise
10
11 # Visualize data
12 plt.scatter(X.numpy(), y.numpy(), alpha=0.5)
13 plt.xlabel('House Size (m2)')
14 plt.ylabel('Price (R\$/ thousands)')
15 plt.title('Training Data')
16 plt.show()
17 print(f"Data shape: X={X.shape}, y={y.shape}")
```

# Method 1: Manual Implementation

## Understanding the Basics

### Define model manually:

- Create weight and bias as tensors
- Enable gradient tracking

### Training components:

- **1. Forward pass:** compute predictions
- **2. Compute loss:** how far off are we?
- **3. Backward pass:** compute gradients
- **4. Update weights:** gradient descent

### Learning rate:

- Controls step size:  $w_{new} = w_{old} - \text{lr} \times \frac{\partial L}{\partial w}$
- Too large: unstable, too small: slow

# Manual Training Loop

## Step-by-Step Implementation

```
1 # Initialize parameters randomly
2 w = torch.randn(1, 1, requires_grad=True)
3 b = torch.randn(1, requires_grad=True)
4
5 # Hyperparameters
6 learning_rate = 0.01
7 num_epochs = 100
8
9 # Training loop
10 for epoch in range(num_epochs):
11     # 1. Forward pass: compute predictions
12     y_pred = X @ w + b # Matrix multiplication + bias
13
14     # 2. Compute loss (Mean Squared Error)
15     loss = torch.mean((y_pred - y) ** 2)
16     ...
```

# Manual Training Loop

## Step-by-Step Implementation

```
1  ...
2  # Training loop
3  for epoch in range(num_epochs):
4      ...
5          # 3. Backward pass: compute gradients
6          loss.backward()
7
8          # 4. Update weights (gradient descent)
9          with torch.no_grad():
10              w -= learning_rate * w.grad
11              b -= learning_rate * b.grad
12          # 5. Zero gradients for next iteration
13          w.grad.zero_()
14          b.grad.zero_()
15          if (epoch + 1) % 10 == 0: print(f'Epoch {epoch+1}, Loss: {loss.item():.4f}')
```

# Understanding the Training Loop

## Key Steps Explained

### □ Step 1: Forward Pass

- $y_{\text{pred}} = X @ w + b$
- Compute predictions using current weights

### □ Step 2: Compute Loss

- `loss = mean((y_pred - y) ** 2)`
- Measure error between predictions and true values

### □ Step 3: Backward Pass

- `loss.backward()`
- Automatically compute gradients:  $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}$

# Understanding the Training Loop

## Key Steps Explained

### □ Step 4: Update Weights

- $w -= lr * w.grad$
- Move in direction that reduces loss

### □ Step 5: Zero Gradients

- `w.grad.zero_()`
- Clear for next iteration (gradients accumulate!)

## Method 2: Using nn.Module

### PyTorch's Standard Way

- **nn.Module:** Base class for all models

- **Advantages:**

- Cleaner, more organized code
- Automatic parameter management
- Easy to extend to complex models
- Same pattern for all models

- **Two methods to implement:**

- `__init__`: define layers/parameters
- `forward`: define computation

- **This is the standard approach!**

- Manual method was for understanding
- Always use nn.Module in practice

# Define Model with nn.Module

## Creating a Model Class

```
1 import torch.nn as nn
2
3 class LinearRegressionModel(nn.Module):
4     def __init__(self, input_dim, output_dim):
5         super(LinearRegressionModel, self).__init__()
6         # Define a linear layer: y = Wx + b
7         self.linear = nn.Linear(input_dim, output_dim)
8
9     def forward(self, x):
10        # Define forward pass
11        return self.linear(x)
12
13 # Create model instance
14 model = LinearRegressionModel(input_dim=1, output_dim=1)
15
16 # Check model parameters
17 print(model)
```

# Loss Functions in PyTorch

## Measuring Error

### □ For regression:

- `nn.MSELoss()` - Mean Squared Error (most common)
  - $MSE = \frac{1}{n} \sum (y - \hat{y})^2$
- `nn.L1Loss()` - Mean Absolute Error
  - $MAE = \frac{1}{n} \sum |y - \hat{y}|$
- `nn.SmoothL1Loss()` - Huber loss

### □ Usage:

- `criterion = nn.MSELoss()`
- `loss = criterion(predictions, targets)`

### □ Why MSE for linear regression?

- Penalizes large errors more
- Differentiable everywhere
- Has nice mathematical properties

# Optimizers in PyTorch

## Weight Update Algorithms

□ **Optimizer:** Updates model parameters to minimize loss

□ **Common optimizers:**

- **SGD:** `torch.optim.SGD(params, lr=0.01)`
  - Classic gradient descent
- **Adam:** `torch.optim.Adam(params, lr=0.001)`
  - Adaptive learning rates, most popular
- **RMSprop, AdamW, Adagrad...**

□ **For linear regression:** SGD is fine

□ **For neural networks:** Adam often works better

□ **Usage:**

- `optimizer.zero_grad()` - clear gradients
- `loss.backward()` - compute gradients
- `optimizer.step()` - update weights

# Complete Training with nn.Module

## The Standard Pattern

```
1 # 1. Create model
2 model = LinearRegressionModel(input_dim=1, output_dim=1)
3
4 # 2. Define loss function
5 criterion = nn.MSELoss()
6
7 # 3. Define optimizer
8 optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
9
10 # 4. Training loop
11 num_epochs = 100
12 losses = []
13
14 for epoch in range(num_epochs):
15     # Forward pass
16     y_pred = model(X)
17     loss = criterion(y_pred, y)
```

# The Training Loop Pattern

## Remember This!

- This pattern is universal in PyTorch:

Step	Code
1. Forward pass	<code>outputs = model(inputs)</code>
2. Compute loss	<code>loss = criterion(outputs, targets)</code>
3. Zero gradients	<code>optimizer.zero_grad()</code>
4. Backward pass	<code>loss.backward()</code>
5. Update weights	<code>optimizer.step()</code>

- Works for:

- Linear regression (what we just did)
- Neural networks

# Visualizing Training Progress

## Monitoring the Loss

```
1 import matplotlib.pyplot as plt
2
3 # Plot training loss
4 plt.figure(figsize=(10, 5))
5 plt.plot(losses)
6 plt.xlabel('Epoch')
7 plt.ylabel('Loss')
8 plt.title('Training Loss Over Time')
9 plt.grid(True, alpha=0.3)
10 plt.show()
11
12 # Plot predictions vs actual
13 model.eval() # Set to evaluation mode
14 with torch.no_grad():
15     predictions = model(X)
16
17 plt.figure(figsize=(10, 5))
```

# Evaluating the Model

## Making Predictions

### □ After training, use the model:

- Set to evaluation mode: `model.eval()`
- Disable gradients: with `torch.no_grad()`:

### □ Why evaluation mode?

- For linear regression: no difference
- For neural networks: disables dropout, changes batchnorm
- **Good habit to always use it!**

### □ Why no\_grad?

- Saves memory (don't need gradients for inference)
- Speeds up computation

### □ Get model parameters:

- `w = model.linear.weight.item()`
- `b = model.linear.bias.item()`

# Inference Example

## Using the Trained Model

```
1 # Set to evaluation mode
2 model.eval()
3
4 # Create new data points
5 new_sizes = torch.tensor([[30.0], [50.0], [100.0]])
6
7 # Make predictions (no gradients needed)
8 with torch.no_grad():
9     predicted_prices = model(new_sizes)
10
11 # Print results
12 print("House Size → Predicted Price")
13 for size, price in zip(new_sizes, predicted_prices):
14     print(f"{size.item():.0f} m² → R${price.item():.1f}k")
15
16 # Extract learned parameters
17 w = model.linear.weight.item()
```

# Saving and Loading Models

## Preserving Your Work

### □ Save model:

- `torch.save(model.state_dict(), 'linear_model.pth')`
- Saves only parameters (recommended)

### □ Load model:

- Create model instance first
- `model = LinearRegressionModel(1, 1)`
- `model.load_state_dict(torch.load('linear_model.pth'))`
- `model.eval()`

### □ Why state\_dict?

- More flexible
- Portable across code changes
- Smaller file size

### □ File extension: Usually .pth or .pt

# Save and Load Example

## Complete Workflow

```
1 # Save the trained model
2 torch.save(model.state_dict(), 'linear_regression_model.pth')
3 print("Model saved!")
4
5 # Later... load the model
6 # 1. Create the same model architecture
7 loaded_model = LinearRegressionModel(input_dim=1, output_dim=1)
8
9 # 2. Load the saved parameters
10 loaded_model.load_state_dict(torch.load('linear_regression_model.pth'))
11
12 # 3. Set to evaluation mode
13 loaded_model.eval()
14 ...
```

# Save and Load Example

## Complete Workflow

```
1  ...
2  # 4. Use it for predictions
3  with torch.no_grad():
4      test_input = torch.tensor([[75.0]])
5      prediction = loaded_model(test_input)
6      print(f"Prediction for 75 m2: R\\$ {prediction.item():.1f}k")
```

# From Linear Regression to Neural Networks

## The Same Patterns Apply!

### What we learned:

- Define model with nn.Module
- Choose loss function
- Choose optimizer
- Training loop: forward → loss → backward → update
- Evaluation mode and inference
- Saving and loading

### For neural networks:

- **Same training loop!**
- Just change: model architecture (add layers, activations)
- Maybe: different loss function
- Maybe: different optimizer (Adam instead of SGD)

### Next: See a complete neural network example (MNIST)

## Troubleshooting Training

### **Loss not decreasing:**

- Learning rate too small → increase it
- Learning rate too large → decrease it
- Check if gradients are being computed

### **Loss is NaN:**

- Learning rate too large
- Numerical instability (gradient explosion)

### **Slow convergence:**

- Try different optimizer (Adam vs SGD)
- Normalize your input data

### **Forgot to zero gradients:**

- Gradients accumulate → wrong updates
- Always call `optimizer.zero_grad()`

# Key Takeaways - Training

## Summary

### □ **Linear regression = simplest ML model**

- But shows ALL PyTorch training concepts!

### □ **Training components:**

- Model (nn.Module)
- Loss function (nn.MSELoss)
- Optimizer (torch.optim.SGD or Adam)
- Training loop (the 5-step pattern)

### □ **Critical steps:**

- Always `zero_grad()` before backward
- Use `model.eval()` for inference
- Use `torch.no_grad()` to save memory

### □ **This same pattern works for ANY model!**

- Neural networks, CNNs, RNNs, Transformers...



## Part 5

---

Building Neural Networks

# Neural Networks in PyTorch

## torch.nn Module

- **torch.nn:** Building blocks for neural networks
- **Key components:**
  - **nn.Module:** Base class for all models
  - **Layers:** Linear, Conv2d, LSTM, etc.
  - **Activation functions:** ReLU, Sigmoid, Tanh
  - **Loss functions:** MSELoss, CrossEntropyLoss
- **Two ways to build networks:**
  - **Sequential:** simple linear stack of layers
  - **Module class:** more flexibility, custom forward pass
- We'll focus on the **Module class** approach

# Simple Neural Network

## Defining a Model

```
1 import torch.nn as nn
2
3 class SimpleNet(nn.Module):
4     def __init__(self, input_size, hidden_size, output_size):
5         super(SimpleNet, self).__init__()
6         # Define layers
7         self.fc1 = nn.Linear(input_size, hidden_size)
8         self.relu = nn.ReLU()
9         self.fc2 = nn.Linear(hidden_size, output_size)
10
11     def forward(self, x):
12         # Define forward pass
13         x = self.fc1(x)          # First linear layer
14         x = self.relu(x)         # Activation
15         x = self.fc2(x)          # Second linear layer
16
17         return x
```

# Common Layers

## Building Blocks

### □ Fully Connected (Linear):

- `nn.Linear(in_features, out_features)`
- Standard dense layer:  $y = Wx + b$

### □ Convolutional:

- `nn.Conv2d(in_channels, out_channels, kernel_size)`
- For images and spatial data

### □ Recurrent:

- `nn.LSTM(input_size, hidden_size)`
- `nn.GRU(input_size, hidden_size)`
- For sequences (text, time series)

### □ Normalization:

- `nn.BatchNorm1d/2d, nn.LayerNorm`

### □ Dropout:

- `nn.Dropout(p=0.5)` - regularization

# Activation Functions

## Non-linearity

### □ Why activations?

- Without them, network is just linear regression
- Activations add **non-linearity**

### □ Common activations:

- **ReLU:** `nn.ReLU()` - most common
  - $f(x) = \max(0, x)$
- **Sigmoid:** `nn.Sigmoid()` - outputs in [0,1]
  - $f(x) = 1/(1 + e^{-x})$
- **Tanh:** `nn.Tanh()` - outputs in [-1,1]
- **Softmax:** `nn.Softmax()` - for classification
  - Converts logits to probabilities

### □ Modern variants: LeakyReLU, ELU, GELU, Swish



## Part 7

---

Best Practices & Tips

# Model Saving and Loading

## Preserving Your Work

### □ Save model:

- `torch.save(model.state_dict(), 'model.pth')`
- Saves only weights (recommended)

### □ Load model:

- `model = MNISTNet()`
- `model.load_state_dict(torch.load('model.pth'))`
- `model.eval()`

### □ Save entire model: (not recommended)

- `torch.save(model, 'model.pth')`
- Can break if code changes

### □ Save checkpoint (for resuming training):

- Save: model state, optimizer state, epoch, loss

# Checkpoint Example

## Saving Everything

```
1 # Save checkpoint
2 checkpoint = {
3     'epoch': epoch,
4     'model_state_dict': model.state_dict(),
5     'optimizer_state_dict': optimizer.state_dict(),
6     'loss': loss,
7 }
8 torch.save(checkpoint, 'checkpoint.pth')
9
10 # Load checkpoint
11 checkpoint = torch.load('checkpoint.pth')
12 model.load_state_dict(checkpoint['model_state_dict'])
13 optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
14 epoch = checkpoint['epoch']
15 loss = checkpoint['loss']
16 model.train() # or model.eval()
```

# Common Pitfalls

## Avoid These Mistakes

- ❑ **Forgetting to zero gradients:**
  - Always call `optimizer.zero_grad()`
- ❑ **Not setting eval mode:**
  - Use `model.eval()` for inference
- ❑ **Mixing CPU and GPU tensors:**
  - All tensors must be on same device
- ❑ **Not using `torch.no_grad()`:**
  - Wastes memory during inference
- ❑ **Wrong tensor shapes:**
  - Check shapes with `.shape`
  - Use `.view()` or `.reshape()` carefully
- ❑ **Learning rate too high:**
  - Start with 0.001 for Adam, tune if needed

# Debugging Tips

## Finding Problems

### □ Check tensor shapes:

- Print `x.shape` at each step
- Use `torchinfo` or `torchsummary`

### □ Verify gradients:

- Check if `requires_grad=True`
- Print gradients: `param.grad`

### □ Start simple:

- Overfit on small batch first
- If model can't overfit, architecture issue

### □ Monitor training:

- Plot loss curves
- Use TensorBoard for visualization

### □ Use debugger:

- PyTorch works with standard Python debuggers (`pdb`, `ipdb`)

# Performance Tips

## Making Training Faster

### Use GPU:

- Essential for large models/datasets
- Move model and data to CUDA

### Larger batch sizes:

- Better GPU utilization
- But: need more memory, may need lower LR

### Mixed precision training:

- `torch.cuda.amp` for automatic mixed precision
- Faster and uses less memory

### Data loading:

- Use multiple workers: `num_workers > 0`
- Pin memory: `pin_memory=True`

### Compile your model: (PyTorch 2.0+)

- `model = torch.compile(model)`

# Going Further

## Advanced Topics

### □ Transfer Learning:

- Use pre-trained models from `torchvision.models`
- Fine-tune on your data

### □ Learning Rate Scheduling:

- `torch.optim.lr_scheduler`
- Adjust learning rate during training

### □ Regularization:

- Dropout, BatchNorm, weight decay
- Data augmentation

### □ Custom Datasets:

- Extend `torch.utils.data.Dataset`

### □ Distributed Training:

- `torch.nn.DataParallel`
- `torch.distributed`

# Learning Resources

## Where to Go Next

### □ Official Resources:

- PyTorch Tutorials: [pytorch.org/tutorials](https://pytorch.org/tutorials)
- Documentation: [pytorch.org/docs](https://pytorch.org/docs)
- Examples: [github.com/pytorch/examples](https://github.com/pytorch/examples)

### □ Courses:

- Fast.ai: Practical Deep Learning
- Deep Learning Specialization (Coursera)
- Stanford CS231n (Computer Vision)

### □ Books:

- Deep Learning with PyTorch (Stevens et al.)
- Programming PyTorch for Deep Learning (Subramanian)

### □ Community:

- PyTorch Forums, Stack Overflow, Reddit r/MachineLearning

# Key Takeaways

## Summary

### □ PyTorch is:

- Flexible, Pythonic deep learning framework
- Dynamic graphs, easy debugging
- Strong ecosystem and community

### □ Core concepts:

- Tensors - multi-dimensional arrays
- Autograd - automatic differentiation
- nn.Module - building blocks for models

### □ Training workflow:

- Define model → Loss + Optimizer → Training loop
- Forward → Loss → Backward → Update

### □ Remember:

- Zero gradients before backward
- Set train/eval mode appropriately

## Questions?