PyTorch: From Basics to Advanced

A Hands-on Introduction to Deep Learning with PyTorch

Instructor: mmaleki92

Deep Learning Course

April 9, 2025

Course Overview

PyTorch

Introduction to PyTorch

PyTorch

What is PyTorch?

- An open source machine learning library
- Developed by Facebook's AI Research lab
- Python-first deep learning framework
- Designed for flexibility and speed
- Strong support for GPU acceleration
- Dynamic computational graph (define-by-run)

Why PyTorch?

- **Pythonic**: Integrates well with Python ecosystem
- **Dynamic computation graph**: Easier debugging
- **Imperative programming style**: More intuitive
- **Strong community support**: Rapidly growing ecosystem
- **Research-friendly**: Used in many academic papers
- **Production-ready**: TorchServe, TorchScript, etc.

PyTorch

Installation

Install with pip

```
pip install torch torchvision torchaudio
```

Install with conda

```
conda install pytorch torchvision torchaudio -c pytorch
```

Note

Visit `https://pytorch.org/get-started/locally/` to get installation command specific to your system (CUDA version etc.)

PyTorch

Check Installation

```python
import torch

# Check PyTorch version
print(f"PyTorch version: {torch.__version__}")

# Check if GPU is available
print(f"GPU available: {torch.cuda.is_available()}")

# Print GPU info if available
if torch.cuda.is_available():
    print(f"GPU name: {torch.cuda.get_device_name(0)}")
    print(f"GPU memory: {torch.cuda.get_device_properties(0).total_memory / 1e9} GB")
```

PyTorch

Tensor Basics

PyTorch

What are Tensors?

- Fundamental data structure in PyTorch
- Multi-dimensional arrays (similar to NumPy arrays)
- Can be used on CPU or GPU
- Support automatic differentiation

Dimensions	Math Term	Example
0D	Scalar	Single value (42)
1D	Vector	Array [1, 2, 3]
2D	Matrix	Table [[1,2], [3,4]]
3D	3-Tensor	Cube of numbers
nD	n-Tensor	Higher dimensional data

PyTorch

Creating Tensors

```
import torch

# From Python list
x = torch.tensor([1, 2, 3, 4])
print(x)  # tensor([1, 2, 3, 4])
```

PyTorch

Creating Tensor

```
# With specific data type
x = torch.tensor([1.0, 2.0], dtype=torch.float32)
print(x)  # tensor([1., 2.])
```

PyTorch

Creating Tensor

```
# From NumPy array
import numpy as np
np_array = np.array([1, 2, 3])
x = torch.from_numpy(np_array)
print(x)  # tensor([1, 2, 3])
```

PyTorch

Creating Tensors

```
# Create uninitialized tensor
x = torch.empty(3, 4)  # 3x4 matrix with uninitialized values
```

PyTorch

Common Tensor Creation Functions

```
# Zeros and ones
zeros = torch.zeros(2, 3)  # 2x3 matrix of zeros
ones = torch.ones(2, 3)    # 2x3 matrix of ones
```

PyTorch

Common Tensor Creation Functions

```
# Random tensors
rand = torch.rand(2, 3)     # Random values from uniform distribution [0,1)
randn = torch.randn(2, 3)   # Random values from standard normal distribution
```

PyTorch

Common Tensor Creation Functions

```
# Range and linspace
arange = torch.arange(0, 10, step=2)  # [0, 2, 4, 6, 8]
linspace = torch.linspace(0, 10, steps=5)  # [0, 2.5, 5, 7.5, 10]
```

PyTorch

Common Tensor Creation Functions

```
# Creating tensors like other tensors
like_tensor = torch.zeros_like(rand)  # Same shape as rand, but filled with zeros
```

PyTorch

Tensor Operations

```
# Arithmetic operations
a = torch.tensor([1, 2, 3])
b = torch.tensor([4, 5, 6])

print(a + b)  # Addition: tensor([5, 7, 9])
print(a - b)  # Subtraction: tensor([-3, -3, -3])
print(a * b)  # Element-wise multiplication: tensor([4, 10, 18])
print(a / b)  # Division: tensor([0.2500, 0.4000, 0.5000])

# In-place operations (modifies the tensor)
a.add_(b)     # Equivalent to a = a + b
print(a)      # tensor([5, 7, 9])
```

PyTorch

Tensor Operations

```
# Matrix multiplication
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[5, 6], [7, 8]])
print(torch.matmul(x, y))  # Matrix multiplication
print(x @ y)               # Alternative syntax
```

PyTorch

Tensor Reshaping

```
# Create a tensor
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(x.shape)  # torch.Size([2, 3])
# Reshape
y = x.reshape(3, 2)  # 3 rows, 2 columns
print(y)
# tensor([[1, 2],
#         [3, 4],
#         [5, 6]])
# View (shares memory with original)
z = x.view(6, 1)
print(z)  # tensor([[1], [2], [3], [4], [5], [6]])
# Transpose
print(x.T)  # tensor([[1, 4], [2, 5], [3, 6]])
# Squeeze and unsqueeze (add/remove dimensions)
a = torch.tensor([1, 2, 3])
b = a.unsqueeze(1)  # Add dimension: [3] -> [3, 1]
print(b)  # tensor([[1], [2], [3]])
print(b.squeeze())  # Remove dimension: [3, 1] -> [3]
```

Tensor Indexing and Slicing

```
# Create a tensor
x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing
print(x[0, 0])    # tensor(1) - first element
print(x[1, 2])    # tensor(6) - element at row 1, column 2
# Slicing
print(x[0:2, :])  # First two rows, all columns
# tensor([[1, 2, 3],
#         [4, 5, 6]])
print(x[:, 1:])   # All rows, columns from index 1 onwards
# tensor([[2, 3],
#         [5, 6],
#         [8, 9]])
```

PyTorch

Tensor Indexing and Slicing

```
# Advanced indexing
indices = torch.tensor([0, 2])
print(x[indices])  # Select rows 0 and 2
# tensor([[1, 2, 3],
#         [7, 8, 9]])
# Boolean masking
mask = x > 5
print(x[mask])  # tensor([6, 7, 8, 9])
```

PyTorch

Quiz: Tensor Basics

1. Which of the following creates a tensor with values from 0 to 9?
   - A) `torch.zeros(10)`
   - B) `torch.arange(10)`
   - C) `torch.tensor(10)`
   - D) `torch.ones(10)`
2. What does the operation `torch.randn(3, 4)` create?
   - A) A tensor of 3s and 4s
   - B) A 3×4 tensor with random values from a uniform distribution
   - C) A 3×4 tensor with random values from a normal distribution
   - D) A 3×4 tensor filled with zeros
3. What's the difference between `tensor.reshape()` and `tensor.view()`?

PyTorch

Quiz: Tensor Basics

1. Which of the following creates a tensor with values from 0 to 9?
   - A) `torch.zeros(10)`
   - B) `torch.arange(10)`
   - C) `torch.tensor(10)`
   - D) `torch.ones(10)`
2. What does the operation `torch.randn(3, 4)` create?
   - A) A tensor of 3s and 4s
   - B) A 3×4 tensor with random values from a uniform distribution
   - C) A 3×4 tensor with random values from a normal distribution
   - D) A 3×4 tensor filled with zeros
3. What's the difference between `tensor.reshape()` and `tensor.view()`?

Answers: 1. B) `torch.arange(10)` 2. C) A 3×4 tensor with random values from a normal distribution 3. `view()` shares memory with the original tensor, while `reshape()` may return a copy if necessary

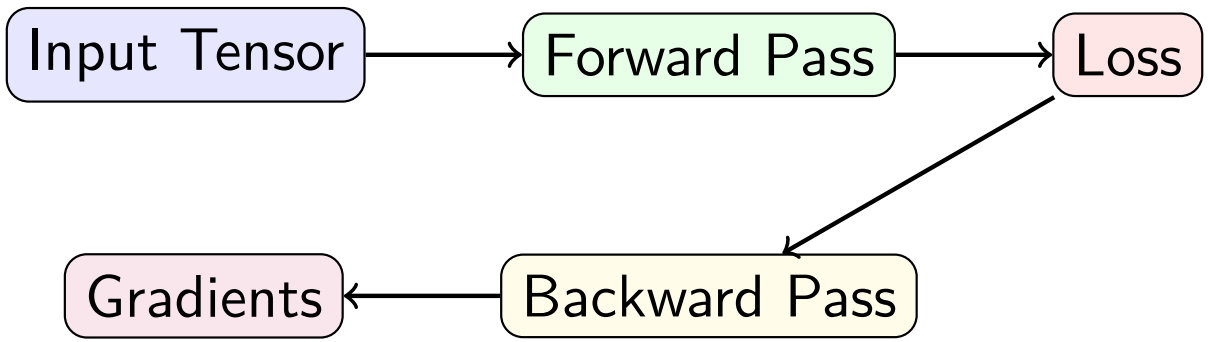
PyTorch

Autograd: Automatic Differentiation

PyTorch

What is Autograd?

- The automatic differentiation engine in PyTorch
- Computes gradients for optimization
- Central to neural network training
- Enables backpropagation with minimal code



PyTorch

Setting up Tensors for Autograd

```python
# Create tensors with gradients enabled
x = torch.tensor([2.0], requires_grad=True)
y = torch.tensor([3.0], requires_grad=True)

# Perform operations
z = x * y + torch.sin(x)

# Compute gradients
z.backward()

# Access gradients
print(f"dz/dx: {x.grad}")  # derivative of z with respect to x
print(f"dz/dy: {y.grad}")  # derivative of z with respect to y
```

PyTorch

Note

- Set `requires_grad=True` to track operations
- Call `backward()` on the final output
- Access gradients through `.grad` attribute

PyTorch

Autograd Example: Linear Regression

```
x = torch.linspace(0, 10, 100)# Generate synthetic data
y_true = 2 * x + 3 + torch.randn(100) * 1.5
# Initialize parameters with gradients
weight = torch.tensor([0.0], requires_grad=True)
bias = torch.tensor([0.0], requires_grad=True)
lr = 0.01 # Learning rate
for epoch in range(100):
    y_pred = weight * x + bias    # Forward pass: compute prediction
    loss = ((y_pred - y_true) ** 2).mean()    # Compute loss (MSE)
    loss.backward()
    # Update parameters (manually)
    with torch.no_grad():  # No need to track gradients during updates
        weight -= lr * weight.grad
        bias -= lr * bias.grad
        # Zero gradients for next iteration
        weight.grad.zero_()
        bias.grad.zero_()
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
        print(f"Weight: {weight.item()}, Bias: {bias.item()}")
```

PyTorch

Autograd with No Grad

- Sometimes we want to prevent gradient tracking:
  - During evaluation phase
  - When updating parameters
  - For efficiency

```
# Method 1: torch.no_grad() context
with torch.no_grad():
    y = x * 2
    # No gradients will be computed

# Method 2: tensor.detach()
y = x.detach() * 2
# y is detached from computation graph

# Method 3: volatile flag (legacy, not recommended)
# Now prefer torch.no_grad() instead

# Checking if gradients are enabled
print(torch.is_grad_enabled())  # True or False
```

PyTorch

Advanced Autograd Features

```python
# Computing jacobian
def compute_jacobian(inputs, outputs):
    jacobian = []
    for i in range(outputs.size(0)):
        # Zero all gradients
        if inputs.grad is not None:
            inputs.grad.zero_()

        # Backward on output i
        outputs[i].backward(retain_graph=True)

        # Store the gradients
        jacobian.append(inputs.grad.clone())

    return torch.stack(jacobian)

# Retain computation graph
y = x * x
z = y * y
z.backward(retain_graph=True)  # Graph is preserved
y.backward()  # Can still call backward again

# Higher order derivatives
x = torch.tensor([1.0], requires_grad=True)
y = x * x * x
grad_1 = torch.autograd.grad(y, x, create_graph=True)[0]  # dy/dx = 3x²
grad_2 = torch.autograd.grad(grad_1, x)[0]  # d²y/dx² = 6x
print(grad_1)  # tensor([3.])
print(grad_2)  # tensor([6.])
```

Quiz: Autograd

1. What happens if you don't set `requires_grad=True` on a tensor?
   - A) The code won't run
   - B) No gradients will be computed for this tensor
   - C) Gradients will still be computed but not stored
   - D) The tensor will be automatically converted to a NumPy array
2. What does the following code do?

```
with torch.no_grad():
    y = model(x)
```

   - A) Computes y without tracking gradients
   - B) Resets all gradients to zero
   - C) Computes higher-order gradients
   - D) Detaches y from the computation graph
3. Why do we need to call `zero_grad()` between iterations?

PyTorch

Quiz: Autograd

1. What happens if you don't set `requires_grad=True` on a tensor?
   - A) The code won't run
   - B) No gradients will be computed for this tensor
   - C) Gradients will still be computed but not stored
   - D) The tensor will be automatically converted to a NumPy array
2. What does the following code do?

```
with torch.no_grad():
    y = model(x)
```

   - A) Computes y without tracking gradients
   - B) Resets all gradients to zero
   - C) Computes higher-order gradients
   - D) Detaches y from the computation graph
3. Why do we need to call `zero_grad()` between iterations?

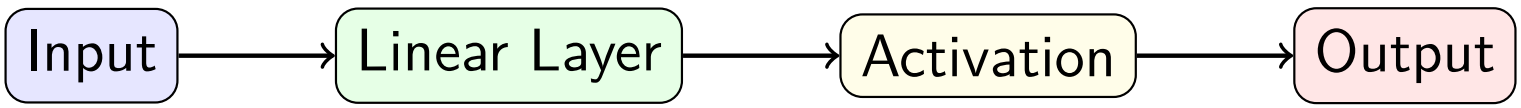
Answers: 1. B) No gradients will be computed for this tensor 2. A) Computes y without tracking gradients 3. PyTorch accumulates gradients between backward passes, so we need to reset them to avoid incorrect updates

PyTorch

Neural Network Basics

PyTorch Neural Network Modules

- **torch.nn**: Module for building neural networks
- **nn.Module**: Base class for all neural network layers
- **nn.functional**: Functional interface to the nn module
- **Key components**:
  - Layers (Linear, Conv2d, etc.)
  - Activation functions (ReLU, Sigmoid, etc.)
  - Loss functions (MSELoss, CrossEntropyLoss, etc.)
  - Optimizers (SGD, Adam, etc.)



PyTorch

Creating a Simple Neural Network

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

# Define a neural network by subclassing nn.Module
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 = self.fc1(x)          # Linear transformation
        x = F.relu(x)            # Apply activation function
        x = self.fc2(x)          # Second linear transformation
        return x

# Create an instance of the network
model = SimpleNetwork(input_size=10, hidden_size=20, output_size=2)

# Print the model architecture
print(model)
```

PyTorch

Common Neural Network Layers

Layer	Function	Common Use Cases
nn.Linear	Fully connected layer	MLP, output layers
nn.Conv1d	1D convolution	Sequence processing
nn.Conv2d	2D convolution	Image processing
nn.MaxPool2d	Max pooling	CNN downsampling
nn.LSTM	Long Short-Term Memory	Sequential data
nn.GRU	Gated Recurrent Unit	Sequential data
nn.Dropout	Random zero-ing	Regularization
nn.BatchNorm2d	Batch normalization	Stabilize training
nn.Embedding	Lookup table	Word embeddings

PyTorch

Common Activation Functions

ReLU (Rectified Linear Unit)

- $f(x) = \max(0, x)$
- Most common activation
- Solves vanishing gradient problem
- Dead neurons problem

Sigmoid

- $f(x) = \frac{1}{1+e^{-x}}$
- Outputs between 0 and 1
- Used for binary classification
- Vanishing gradient problem

Tanh

- $f(x) = \tanh(x)$
- Outputs between -1 and 1
- Zero-centered
- Still has vanishing gradient

Softmax

- $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Outputs sum to 1 (probabilities)
- Used for multi-class classification
- Applied to output layer

```
# Functional interface
x = F.relu(x)
y = F.sigmoid(y)
z = F.softmax(z, dim=1)  # dim specifies dimension to sum over

# Module interface
activation = nn.ReLU()
x = activation(x)
```

Loss Functions

- **MSELoss** (Mean Squared Error)
  - For regression problems
  - MSE $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- **CrossEntropyLoss**
  - For classification problems
  - Combines LogSoftmax and NLLLoss
- **BCELoss** (Binary Cross Entropy)
  - For binary classification
  - Input should be between 0 and 1

```
# Define loss functions
mse_loss = nn.MSELoss()
cross_entropy = nn.CrossEntropyLoss()
bce_loss = nn.BCELoss()

# Calculate loss
regression_loss = mse_loss(predictions, targets)
classification_loss = cross_entropy(predictions, targets)
binary_loss = bce_loss(predictions, targets)
```

PyTorch

Optimizers

- Algorithms for updating model parameters during training
- Different trade-offs between speed, memory, and accuracy

Optimizer	Key Features
SGD	Simple, may need tuning learning rate
Adam	Adaptive learning rates, good default choice
RMSprop	Good for RNNs, adaptive learning rates
Adagrad	Adapts learning rate for each parameter
Adadelta	Improvement over Adagrad

```
import torch.optim as optim

# Define optimizers
sgd = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
adam = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999))
rmsprop = optim.RMSprop(model.parameters(), lr=0.01, alpha=0.99)

# Using an optimizer in training
optimizer = optim.Adam(model.parameters(), lr=0.001)
optimizer.zero_grad()  # Zero gradients
loss.backward()        # Compute gradients
optimizer.step()       # Update parameters
```

Quiz: Neural Network Basics

1. Which of these is required when creating a custom neural network class in PyTorch?
   - A) Implementing the `backward()` method
   - B) Implementing the `forward()` method
   - C) Implementing both `forward()` and `backward()`
   - D) Implementing gradient calculation
2. Which activation function outputs values between 0 and 1?
   - A) ReLU
   - B) Tanh
   - C) Sigmoid
   - D) Leaky ReLU
3. Which optimizer would you typically choose for a new project where you have no prior information about which optimizer works best?

PyTorch

Quiz: Neural Network Basics

1. Which of these is required when creating a custom neural network class in PyTorch?
   - A) Implementing the `backward()` method
   - B) Implementing the `forward()` method
   - C) Implementing both `forward()` and `backward()`
   - D) Implementing gradient calculation
2. Which activation function outputs values between 0 and 1?
   - A) ReLU
   - B) Tanh
   - C) Sigmoid
   - D) Leaky ReLU
3. Which optimizer would you typically choose for a new project where you have no prior information about which optimizer works best?

Answers: 1. B) Implementing the `forward()` method 2. C) Sigmoid 3. Adam is generally a good default choice as it adapts learning rates and works well across many problems

PyTorch

Building and Training a Neural Network

PyTorch

Complete Neural Network Example: MNIST

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

# Define the neural network
class MNISTNetwork(nn.Module):
    def __init__(self):
        super(MNISTNetwork, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Loading Data

```
# Define transforms
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Load training data
train_dataset = datasets.MNIST('../data', train=True, download=True,
                               transform=transform)
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=64, shuffle=True)

# Load test data
test_dataset = datasets.MNIST('../data', train=False,
                              transform=transform)
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=1000)

# Create model, loss function and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MNISTNetwork().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

PyTorch

Training Function

```python
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # Zero gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Calculate loss
        loss = F.nll_loss(output, target)

        # Backward pass
        loss.backward()

        # Update parameters
        optimizer.step()

        # Print progress
        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)} '
                  f'({100. * batch_idx / len(train_loader):.0f}%)]\t'
                  f'Loss: {loss.item():.6f}')
```

PyTorch

Evaluation Function

```python
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():  # Disable gradient computation for evaluation
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)

            # Forward pass
            output = model(data)

            # Sum up batch loss
            test_loss += F.nll_loss(output, target, reduction='sum').item()

            # Get predicted class
            pred = output.argmax(dim=1, keepdim=True)

            # Count correct predictions
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)

    print(f'\nTest set: Average loss: {test_loss:.4f}, '
          f'Accuracy: {correct}/{len(test_loader.dataset)} ({accuracy:.2f}%)\n')

    return accuracy
```

PyTorch

Training Loop

```python
# Number of epochs
num_epochs = 5

# Lists to store metrics
train_losses = []
test_accuracies = []

# Training loop
for epoch in range(1, num_epochs + 1):
    # Train for one epoch
    train(model, device, train_loader, optimizer, epoch)

    # Evaluate model
    accuracy = test(model, device, test_loader)
    test_accuracies.append(accuracy)

    # Save model checkpoint
    torch.save(model.state_dict(), f"mnist_model_epoch_{epoch}.pt")

print("Training complete!")

# Plot results
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 5))
plt.plot(range(1, num_epochs + 1), test_accuracies)
plt.title('Test Accuracy vs. Epoch')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.grid(True)
plt.savefig('training_results.png')
plt.show()
```

Learning Rate Schedulers

- Adjust learning rate during training
- Can improve convergence and final performance
- PyTorch provides various schedulers in `torch.optim.lr_scheduler`

```python
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau

# Step LR: reduces learning rate by gamma every step_size epochs
scheduler1 = StepLR(optimizer, step_size=30, gamma=0.1)

# ReduceLROnPlateau: reduces learning rate when a metric plateaus
scheduler2 = ReduceLROnPlateau(optimizer, 'min', patience=10)

# Using scheduler in training loop
for epoch in range(num_epochs):
    train(...)
    val_loss = validate(...)

    # Option 1: Step LR
    scheduler1.step()

    # Option 2: ReduceLROnPlateau
    scheduler2.step(val_loss)

    current_lr = optimizer.param_groups[0]['lr']
    print(f"Epoch {epoch}, Current LR: {current_lr}")
```

PyTorch

Saving and Loading Models

```python
# Saving a model - Method 1: Save only the model parameters
torch.save(model.state_dict(), 'model_weights.pth')

# Saving a model - Method 2: Save the entire model
torch.save(model, 'entire_model.pth')

# Loading a model - Method 1: Load parameters into an existing model
model = MNISTNetwork()
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()  # Set the model to evaluation mode

# Loading a model - Method 2: Load the entire model
model = torch.load('entire_model.pth')
model.eval()

# Saving checkpoints with additional info
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
}
torch.save(checkpoint, 'checkpoint.pth')

# Loading checkpoints
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

Quiz: Building and Training

1. What is the purpose of `model.eval()` when testing a model?
   - A) It disables dropout and batch normalization layers
   - B) It computes the evaluation metrics automatically
   - C) It saves the model to disk
   - D) It creates a new model for evaluation
2. What is the benefit of using learning rate scheduling?
   - A) It makes training faster
   - B) It can help escape local minima and achieve better results
   - C) It prevents catastrophic forgetting
   - D) It removes the need for validation data
3. When saving a PyTorch model, what's the difference between saving `model.state_dict()` and saving the entire model?

PyTorch

Quiz: Building and Training

1. What is the purpose of `model.eval()` when testing a model?
   - A) It disables dropout and batch normalization layers
   - B) It computes the evaluation metrics automatically
   - C) It saves the model to disk
   - D) It creates a new model for evaluation
2. What is the benefit of using learning rate scheduling?
   - A) It makes training faster
   - B) It can help escape local minima and achieve better results
   - C) It prevents catastrophic forgetting
   - D) It removes the need for validation data
3. When saving a PyTorch model, what's the difference between saving `model.state_dict()` and saving the entire model?

Answers: 1. A) It disables dropout and batch normalization layers 2. B) It can help escape local minima and achieve better results 3. `state_dict()` saves only the model parameters, while saving the entire model includes the architecture definition. `state_dict()` is usually preferred as it's more portable.

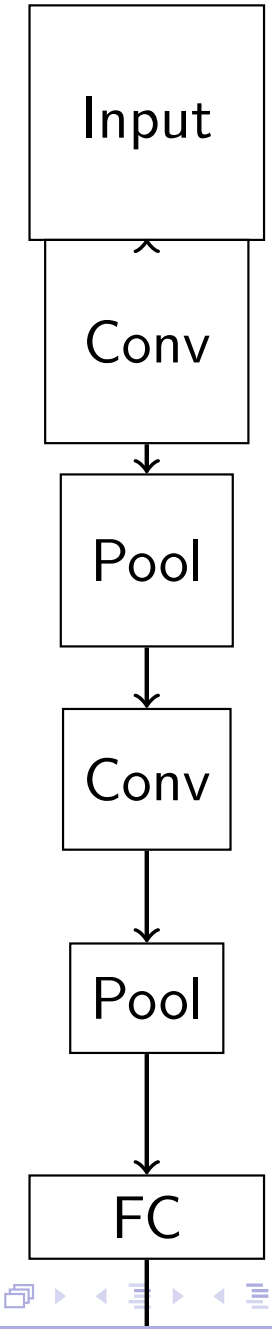
PyTorch

Convolutional Neural Networks (CNNs)

PyTorch

Convolutional Neural Networks (CNNs)

- Specialized for processing grid-like data
- Particularly effective for images
- Key concepts:
  - Local receptive fields
  - Shared weights
  - Pooling
- Common CNN architecture:
  - Convolutional layers
  - Activation functions
  - Pooling layers
  - Fully connected layers



PyTorch

Convolutional Layers

```python
# Basic 2D convolution
conv = nn.Conv2d(in_channels=3,      # Input channels (e.g., RGB)
                 out_channels=16,     # Output feature maps
                 kernel_size=3,       # Filter size (3x3)
                 stride=1,            # Step size
                 padding=1)           # Zero-padding

# Forward pass
x = torch.randn(1, 3, 32, 32)  # (batch_size, channels, height, width)
output = conv(x)
print(output.shape)  # torch.Size([1, 16, 32, 32])

# Parameters
print(f"Number of parameters: {sum(p.numel() for p in conv.parameters())}")
# 3 * 16 * 3 * 3 + 16 = 448 parameters
```

- **Kernel size**: Size of convolutional filter (3x3, 5x5, etc.)
- **Stride**: Step size when sliding the filter
- **Padding**: Zero-padding around the input
- **Dilation**: Spacing between kernel elements

PyTorch

Pooling Layers

Max Pooling

- Takes maximum value within window
- Reduces spatial dimensions
- Provides some translation invariance
- Most commonly used

```
# Max pooling
maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
x = torch.randn(1, 16, 32, 32)
output = maxpool(x)
print(output.shape)  # [1, 16, 16, 16]
```

Average Pooling

- Takes average of values in window
- Smoother downsampling
- Preserves more information
- Used in some architectures

```
# Average pooling
avgpool = nn.AvgPool2d(kernel_size=2, stride=2)
x = torch.randn(1, 16, 32, 32)
output = avgpool(x)
print(output.shape)  # [1, 16, 16, 16]
```

PyTorch

Building a CNN in PyTorch

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # First convolutional block
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Second convolutional block
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Third convolutional block
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Calculate size after convolutions and pooling
        # Input: 3x32x32 -> After 3 pooling layers: 128x4x4
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, 10)  # 10 classes

    def forward(self, x):
        # First block
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)

        # Second block
        x = self.conv2(x)
```

Using Pre-trained Models

```python
import torchvision.models as models

# Load pre-trained models
resnet = models.resnet18(pretrained=True)
vgg = models.vgg16(pretrained=True)
densenet = models.densenet121(pretrained=True)

# Freeze parameters to use as feature extractor
for param in resnet.parameters():
    param.requires_grad = False

# Modify the final layer for your task
num_ftrs = resnet.fc.in_features
resnet.fc = nn.Linear(num_ftrs, 10)  # 10 classes

# Use the model
x = torch.randn(1, 3, 224, 224)  # Input must match expected size
output = resnet(x)
print(output.shape)  # [1, 10]

# Available pre-trained models
print(dir(models))
```

PyTorch

CNN Visualization

```python
import matplotlib.pyplot as plt
from torchvision import transforms

# Function to visualize feature maps
def visualize_feature_maps(model, image, layer_name):
    # Register a hook to get output from a specific layer
    activation = {}
    def get_activation(name):
        def hook(model, input, output):
            activation[name] = output.detach()
        return hook

    # Register hook
    if layer_name == 'conv1':
        model.conv1.register_forward_hook(get_activation(layer_name))

    # Forward pass
    model.eval()
    output = model(image)

    # Get feature maps
    feature_maps = activation[layer_name]

    # Plot feature maps
    num_features = feature_maps.shape[1]
    rows = int(num_features**0.5)
    cols = num_features // rows + (1 if num_features % rows != 0 else 0)

    plt.figure(figsize=(12, 12))
    for i in range(num_features):
        plt.subplot(rows, cols, i+1)
        plt.imshow(feature_maps[0, i].cpu().numpy(), cmap='viridis')
        plt.axis('off')
```

Quiz: Convolutional Neural Networks

1. Consider a convolutional layer with input size $32 \times 32 \times 3$, kernel size 3×3, 16 filters, stride 1, and padding 1. What is the output size?
   - A) $30 \times 30 \times 16$
   - B) $32 \times 32 \times 16$
   - C) $16 \times 16 \times 3$
   - D) $32 \times 32 \times 3$
2. What is the main purpose of pooling layers in CNNs?
   - A) To add non-linearity
   - B) To reduce spatial dimensions and computational load
   - C) To normalize feature maps
   - D) To increase the number of parameters
3. Why are batch normalization layers commonly used in deep CNNs?

PyTorch

Quiz: Convolutional Neural Networks

1. Consider a convolutional layer with input size $32 \times 32 \times 3$, kernel size 3×3, 16 filters, stride 1, and padding 1. What is the output size?
   - A) $30 \times 30 \times 16$
   - B) $32 \times 32 \times 16$
   - C) $16 \times 16 \times 3$
   - D) $32 \times 32 \times 3$
2. What is the main purpose of pooling layers in CNNs?
   - A) To add non-linearity
   - B) To reduce spatial dimensions and computational load
   - C) To normalize feature maps
   - D) To increase the number of parameters
3. Why are batch normalization layers commonly used in deep CNNs?

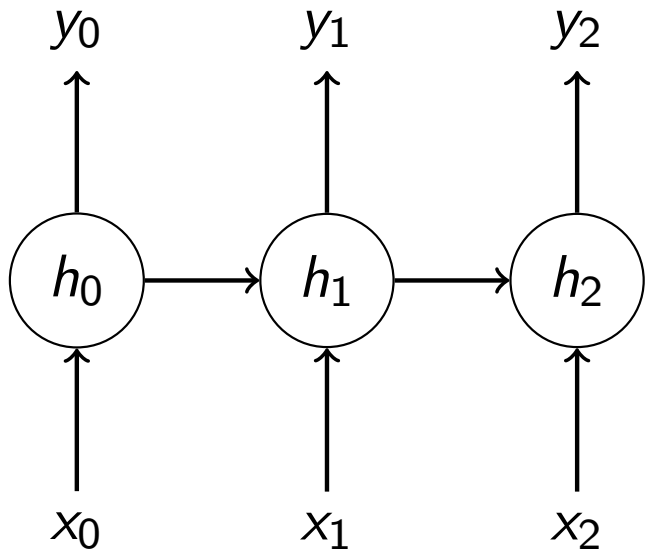
Answers: 1. B) $32 \times 32 \times 16$ (padding preserves spatial dimensions) 2. B) To reduce spatial dimensions and computational load 3. Batch normalization helps stabilize and accelerate training by normalizing layer inputs, mitigating internal covariate shift, and providing some regularization

PyTorch

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs)

- Neural networks designed for sequential data
- Maintain internal state (memory)
- Can process sequences of variable length
- Applications:
  - Natural language processing
  - Time series analysis
  - Speech recognition
  - Music generation
- Types of RNNs in PyTorch:
  - Simple RNN
  - LSTM (Long Short-Term Memory)
  - GRU (Gated Recurrent Unit)



PyTorch

Simple RNN in PyTorch

```python
import torch.nn as nn

# Parameters
input_size = 10    # Size of input features
hidden_size = 20   # Size of hidden state
num_layers = 2     # Number of recurrent layers
batch_size = 3     # Batch size
seq_length = 5     # Sequence length

# Create a simple RNN
rnn = nn.RNN(input_size=input_size,
             hidden_size=hidden_size,
             num_layers=num_layers,
             batch_first=True)  # If True, input shape is (batch, seq, feature)

# Create input tensor
x = torch.randn(batch_size, seq_length, input_size)

# Forward pass
output, hidden = rnn(x)

print(f"Output shape: {output.shape}")  # [batch_size, seq_length, hidden_size]
print(f"Hidden state shape: {hidden.shape}")  # [num_layers, batch_size, hidden_size]
```

PyTorch

LSTM in PyTorch

```python
# Create LSTM
lstm = nn.LSTM(input_size=input_size,
               hidden_size=hidden_size,
               num_layers=num_layers,
               batch_first=True,
               bidirectional=False,
               dropout=0.2)  # Dropout between layers (if num_layers > 1)

# Initial hidden state and cell state
h0 = torch.zeros(num_layers, batch_size, hidden_size)
c0 = torch.zeros(num_layers, batch_size, hidden_size)

# Forward pass
output, (hidden, cell) = lstm(x, (h0, c0))

print(f"Output shape: {output.shape}")  # [batch_size, seq_length, hidden_size]
print(f"Hidden state shape: {hidden.shape}")  # [num_layers, batch_size, hidden_size]
print(f"Cell state shape: {cell.shape}")  # [num_layers, batch_size, hidden_size]

# Bidirectional LSTM
bi_lstm = nn.LSTM(input_size=input_size,
                  hidden_size=hidden_size,
                  num_layers=num_layers,
                  batch_first=True,
                  bidirectional=True)

# Forward pass with bidirectional LSTM
bi_output, (bi_hidden, bi_cell) = bi_lstm(x)

print(f"Bidirectional output shape: {bi_output.shape}")
# [batch_size, seq_length, 2*hidden_size] (2* because bidirectional)
```

h

GRU in PyTorch

```
# Create GRU (Gated Recurrent Unit)
gru = nn.GRU(input_size=input_size,
             hidden_size=hidden_size,
             num_layers=num_layers,
             batch_first=True)

# Initial hidden state
h0 = torch.zeros(num_layers, batch_size, hidden_size)

# Forward pass (GRU doesn't have cell state)
output, hidden = gru(x, h0)

print(f"Output shape: {output.shape}")  # [batch_size, seq_length, hidden_size]
print(f"Hidden state shape: {hidden.shape}")  # [num_layers, batch_size, hidden_size]

# Comparison of RNN types
print("\nComparison of RNN Types:")
print("Simple RNN: + Simple, - Vanishing gradient problem")
print("LSTM: + Solves vanishing gradient, - More parameters")
print("GRU: + Almost as good as LSTM but faster, - Slightly less powerful")
```

PyTorch

Text Classification with LSTM

```python
class TextClassifier(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim,
                 n_layers, bidirectional, dropout, pad_idx):
        super(TextClassifier, self).__init__()

        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_idx)

        # LSTM layer
        self.lstm = nn.LSTM(embedding_dim,
                            hidden_dim,
                            num_layers=n_layers,
                            bidirectional=bidirectional,
                            dropout=dropout if n_layers > 1 else 0,
                            batch_first=True)

        # Output layer
        # If bidirectional, we have 2 * hidden_dim
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

        # Dropout layer
        self.dropout = nn.Dropout(dropout)

    def forward(self, text, text_lengths):
        # text = [batch size, sentence length]

        # Pass text through embedding layer
        embedded = self.embedding(text)
        # embedded = [batch size, sentence length, embedding dim]

        # Pack sequence to handle variable length sequences
        packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths,
                                                            batch_first=True,
```

Training on Padded Sequences

```
import torch.nn.utils.rnn as rnn_utils

def train(model,           device, train_loader, optimizer, criterion):
model.train()
epoch_loss = 0

for batch in train_loader:
# Get text and lengths
text, text_lengths = batch.text

# Move to device
text = text.to(device)
text_lengths = text_lengths.to(device)
labels = batch.label.to(device)

# Zero gradients
optimizer.zero_grad()

# Forward pass
predictions = model(text, text_lengths)

# Calculate loss
loss = criterion(predictions, labels)

# Backward pass
loss.backward()

# Update parameters
optimizer.step()

epoch_loss += loss.item()

return epoch_loss / len(train_loader)
```

Quiz: Recurrent Neural Networks

1. What is the main difference between RNN and LSTM?
   - A) RNNs use convolution operations, LSTMs don't
   - B) RNNs have more parameters than LSTMs
   - C) LSTMs have additional gates to control information flow
   - D) LSTMs can only process fixed-length sequences
2. Why would you use a bidirectional RNN?
   - A) To reduce training time
   - B) To capture context from both past and future elements
   - C) To handle longer sequences
   - D) To reduce overfitting
3. What is the purpose of packing padded sequences in PyTorch?

PyTorch

Quiz: Recurrent Neural Networks

1. What is the main difference between RNN and LSTM?
   - A) RNNs use convolution operations, LSTMs don't
   - B) RNNs have more parameters than LSTMs
   - C) LSTMs have additional gates to control information flow
   - D) LSTMs can only process fixed-length sequences
2. Why would you use a bidirectional RNN?
   - A) To reduce training time
   - B) To capture context from both past and future elements
   - C) To handle longer sequences
   - D) To reduce overfitting
3. What is the purpose of packing padded sequences in PyTorch?

Answers: 1. C) LSTMs have additional gates to control information flow 2. B) To capture context from both past and future elements 3. Packing padded sequences allows efficient computation by ignoring the padded elements, resulting in faster training and saving memory, especially for batches with variable-length sequences.

PyTorch

Transfer Learning

PyTorch

What is Transfer Learning?

- Using knowledge learned from one task for another task
- Leverages pre-trained models to solve new problems
- Benefits:
  - Reduces training time
  - Requires less data
  - Often improves performance
  - Works well for similar domains

PyTorch

Transfer Learning Approaches

Feature Extraction

- Freeze pre-trained layers
- Replace and train only the final layers
- Good when:
  - New dataset is small
  - New data is similar to original

Fine-tuning

- Update some or all layers
- Usually with smaller learning rate
- Good when:
  - New dataset is large
  - New data is different

```
# Feature extraction - freeze all layers
for param in model.parameters():
param.requires_grad = False

# Fine-tuning - train all layers with different learning rates
optimizer = optim.SGD([
{'params': model.base.parameters(), 'lr': 0.001},  # Small LR for pre-trained
{'params': model.fc.parameters(), 'lr': 0.01}      # Larger LR for new layers
], momentum=0.9)
```

PyTorch

Transfer Learning Example: Image Classification

```python
import torchvision.models as models
import torch.nn as nn
import torch.optim as optim

# Load pre-trained ResNet
model = models.resnet50(pretrained=True)

# Option 1: Feature extraction (only train the last layer)
for param in model.parameters():
param.requires_grad = False

# Replace the final layer
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, num_classes)

# Option 2: Fine-tuning (train all layers)
# First train only the modified layers
for param in model.parameters():
param.requires_grad = False
model.fc = nn.Linear(model.fc.in_features, num_classes)

# Train for a few epochs with only the final layer
optimizer = optim.Adam(model.fc.parameters(), lr=0.001)
# ... train for a few epochs ...

# Then unfreeze all layers and train with a smaller learning rate
for param in model.parameters():
param.requires_grad = True

optimizer = optim.Adam(model.parameters(), lr=0.0001)  # Lower learning rate
# ... continue training ...
```

Transfer Learning with Language Models

```
from transformers import BertModel, BertTokenizer

# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bert = BertModel.from_pretrained('bert-base-uncased')

# Create a classifier using BERT embeddings
class BertClassifier(nn.Module):
def __init__(self, num_classes, freeze_bert=True):
super(BertClassifier, self).__init__()
self.bert = BertModel.from_pretrained('bert-base-uncased')
self.dropout = nn.Dropout(0.1)
self.classifier = nn.Linear(768, num_classes)  # BERT hidden size = 768

if freeze_bert:  # Freeze BERT parameters
for param in self.bert.parameters():
param.requires_grad = False

def forward(self, input_ids, attention_mask):
# Forward pass through BERT
outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)

# Use the [CLS] token representation
pooled_output = outputs.pooler_output

# Apply dropout and classification
pooled_output = self.dropout(pooled_output)
logits = self.classifier(pooled_output)

return logits

# Create model
model = BertClassifier(num_classes=3, freeze_bert=True)
```

Quiz: Transfer Learning

1. When would you choose to freeze pre-trained layers rather than fine-tune them?
   - A) When your new dataset is very large
   - B) When your new dataset is small or similar to the original training data
   - C) When your model starts overfitting
   - D) When using CNNs instead of RNNs
2. What is a potential disadvantage of transfer learning?
   - A) It always leads to worse results than training from scratch
   - B) It may introduce biases from the original training data
   - C) It's impossible to modify pre-trained models
   - D) Transfer learning only works for image data
3. Why might you use a smaller learning rate when fine-tuning a pre-trained model?

PyTorch

Quiz: Transfer Learning

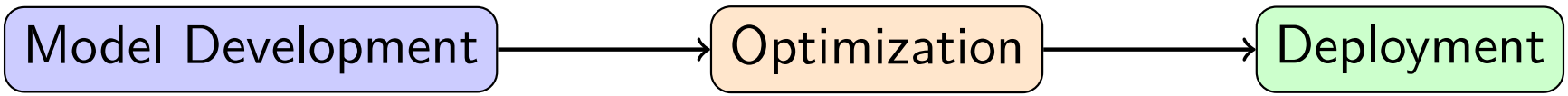
1. When would you choose to freeze pre-trained layers rather than fine-tune them?
   - A) When your new dataset is very large
   - B) When your new dataset is small or similar to the original training data
   - C) When your model starts overfitting
   - D) When using CNNs instead of RNNs
2. What is a potential disadvantage of transfer learning?
   - A) It always leads to worse results than training from scratch
   - B) It may introduce biases from the original training data
   - C) It's impossible to modify pre-trained models
   - D) Transfer learning only works for image data
3. Why might you use a smaller learning rate when fine-tuning a pre-trained model?

Answers: 1. B) When your new dataset is small or similar to the original training data 2. B) It may introduce biases from the original training data 3. Pre-trained weights have already converged to good values, so using a smaller learning rate prevents large updates that might destroy the useful pre-trained features

Model Deployment

Model Deployment Options

- **TorchScript**: Convert PyTorch models to optimized format
- **ONNX**: Open Neural Network Exchange format
- **TorchServe**: Model serving framework for PyTorch
- **Mobile**: Deploy on Android/iOS with PyTorch Mobile
- **Cloud**: Deploy using cloud services (AWS, Azure, GCP)



PyTorch

TorchScript

- Enables saving and loading models without Python code
- Two ways to convert:
  - `torch.jit.trace`: Traces a single execution path
  - `torch.jit.script`: Analyzes the Python code

```
# Converting using trace
example_input = torch.rand(1, 3, 224, 224)
traced_model = torch.jit.trace(model, example_input)
traced_model.save('traced_model.pt')

# Converting using script
scripted_model = torch.jit.script(model)
scripted_model.save('scripted_model.pt')

# Loading and using a TorchScript model
loaded_model = torch.jit.load('traced_model.pt')
prediction = loaded_model(example_input)
```

PyTorch

ONNX Export

- ONNX: Open Neural Network Exchange
- Standardized format for ML models
- Enables interoperability between frameworks
- Supported by many hardware vendors and platforms

```
import torch.onnx

# Define input dimensions
dummy_input = torch.randn(1, 3, 224, 224)

# Export the model to ONNX format
torch.onnx.export(
model,                      # PyTorch model
dummy_input,                # Example input
"model.onnx",               # Output file
export_params=True,         # Export model parameters
opset_version=11,           # ONNX version
do_constant_folding=True,   # Optimize constants
input_names=["input"],      # Input node name
output_names=["output"],    # Output node name
dynamic_axes={
^^I'input': {0: 'batch_size'},  # Variable batch size
^^I'output': {0: 'batch_size'}
}
)

# Now you can run the model with ONNX Runtime
import onnxruntime as ort
ort_session = ort.InferenceSession("model.onnx")
outputs = ort_session.run(
None,
```

Model Quantization

- Reduces model size and inference time
- Less memory and computational requirements
- Types of quantization:
  - Dynamic quantization
  - Static quantization
  - Quantization-aware training

```python
import torch.quantization

# Dynamic Quantization
quantized_model = torch.quantization.quantize_dynamic(
model,  # the original model
{nn.LSTM, nn.Linear},  # specify layers to quantize
dtype=torch.qint8  # the target dtype for quantized weights
)

# Define quantization configuration
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
torch.quantization.prepare(model, inplace=True)

# Calibrate with sample data (feed batches of data to the model)
# ...

# Convert to quantized model
torch.quantization.convert(model, inplace=True)

# Save the quantized model
torch.jit.save(torch.jit.script(model), "quantized_model.pt")
```

Model Serving with TorchServe

```
# 1. Install TorchServe
# pip install torchserve torch-model-archiver

# 2. Create a handler file (handler.py)
from ts.torch_handler.vision_handler import VisionHandler

class ImageClassifierHandler(VisionHandler):
def preprocess(self, data):
# Transform raw input for the model
# ...
return transformed_data

def postprocess(self, data):
# Transform model output
# ...
return transformed_output

# 3. Archive the model (from command line)
# torch-model-archiver --model-name resnet50 \
#                      --version 1.0 \
#                      --model-file model.py \
#                      --serialized-file model.pth \
#                      --handler handler.py \
#                      --extra-files index_to_class.json

# 4. Start TorchServe (from command line)
# torchserve --start --model-store model_store \
#            --models resnet50=resnet50.mar

# 5. Make inference requests
# curl -X POST http://127.0.0.1:8080/predictions/resnet50 \
#     -T input_image.jpg
```

Quiz: Model Deployment

1. What is the main difference between `torch.jit.trace` and `torch.jit.script`?
   - A) Tracing is faster but doesn't capture control flow, scripting handles control flow but may have compatibility issues
   - B) Tracing works for CNNs, scripting works for RNNs
   - C) Tracing creates smaller models, scripting creates more accurate models
   - D) Tracing is deprecated, scripting is the recommended approach
2. What is a benefit of model quantization?
   - A) It makes training faster
   - B) It reduces model size and inference time
   - C) It increases model accuracy
   - D) It enables training on CPUs
3. What is ONNX used for?

PyTorch

Quiz: Model Deployment

1. What is the main difference between `torch.jit.trace` and `torch.jit.script`?
   - A) Tracing is faster but doesn't capture control flow, scripting handles control flow but may have compatibility issues
   - B) Tracing works for CNNs, scripting works for RNNs
   - C) Tracing creates smaller models, scripting creates more accurate models
   - D) Tracing is deprecated, scripting is the recommended approach
2. What is a benefit of model quantization?
   - A) It makes training faster
   - B) It reduces model size and inference time
   - C) It increases model accuracy
   - D) It enables training on CPUs
3. What is ONNX used for?

Answers: 1. A) Tracing is faster but doesn't capture control flow, scripting handles control flow but may have compatibility issues 2. B) It reduces model size and inference time 3. ONNX is a standardized format for machine learning models that enables interoperability between different frameworks and platforms

Best Practices and Advanced Techniques

PyTorch

Debugging Neural Networks

- **Check your data**: Ensure inputs and targets are correct
- **Start simple**: Begin with a simple model and gradually add complexity
- **Visualize**:
  - Input data
  - Gradients (vanishing/exploding)
  - Activations
  - Model predictions
- **Profile memory and performance**:
  - Use `torch.profiler`
  - Monitor GPU memory usage

```
# Check for NaN values
def check_nan(model):
for name, param in model.named_parameters():
if torch.isnan(param).any():
print(f"NaN in {name}")

# Print model gradient norms
def print_grad_norms(model):
for name, param in model.named_parameters():
if param.grad is not None:
grad_norm = param.grad.norm().item()
print(f"{name}: grad_norm = {grad_norm}")
```

Training Large Models

- **Gradient accumulation**: Update less frequently to handle larger batches
- **Mixed precision training**: Use float16 to reduce memory and speed up computation
- **Checkpointing**: Save activations only at selected layers
- **Distributed training**: Use multiple GPUs or machines

```
# Mixed precision training with Automatic Mixed Precision (AMP)
from torch.cuda.amp import autocast, GradScaler

# Initialize scaler
scaler = GradScaler()

# Training loop
for inputs, targets in train_loader:
# Forward pass with autocast (uses float16 where appropriate)
with autocast():
outputs = model(inputs)
loss = criterion(outputs, targets)

# Backward pass with scaled gradients
scaler.scale(loss).backward()

# Update weights, unscaling gradients first
scaler.step(optimizer)

# Update scaler for next iteration
scaler.update()
```

Distributed Training

```
import torch.distributed as dist
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
# Initialize process group
dist.init_process_group("nccl", rank=rank, world_size=world_size)

def cleanup():
dist.destroy_process_group()

def train(rank, world_size):
# Setup process group
setup(rank, world_size)

# Create model and move to GPU with id=rank
model = NeuralNetwork().to(rank)

# Wrap model for distributed training
ddp_model = DDP(model, device_ids=[rank])

# Create data loader for this process
train_sampler = DistributedSampler(
train_dataset,
num_replicas=world_size,
rank=rank
)

train_loader = DataLoader(
train_dataset,
batch_size=batch_size,
sampler=train_sampler
)
```

Hyperparameter Optimization

- **Key hyperparameters**:
  - Learning rate
  - Batch size
  - Network architecture
  - Regularization (dropout, weight decay)
- **Common optimization methods**:
  - Grid search
  - Random search
  - Bayesian optimization
  - Population-based training

```
# Using Ray Tune for hyperparameter optimization
import ray
from ray import tune
from ray.tune.schedulers import ASHAScheduler

def train_model(config):
# Create model with config hyperparameters
model = NeuralNetwork(
hidden_size=config["hidden_size"],
dropout=config["dropout"]
)

optimizer = optim.Adam(model.parameters(), lr=config["lr"])

# Train model (simplified)
for epoch in range(10):
```

Quiz: Best Practices

1. What is gradient accumulation used for?
   - A) To improve model accuracy
   - B) To handle larger effective batch sizes than would fit in memory
   - C) To prevent overfitting
   - D) To speed up convergence
2. Which of these is NOT a benefit of mixed precision training?
   - A) Reduced memory usage
   - B) Faster computation on supported hardware
   - C) Always improves model accuracy
   - D) Enables larger batch sizes
3. What is the advantage of Bayesian optimization over grid search for hyperparameter tuning?

PyTorch

Quiz: Best Practices

1. What is gradient accumulation used for?
   - A) To improve model accuracy
   - B) To handle larger effective batch sizes than would fit in memory
   - C) To prevent overfitting
   - D) To speed up convergence
2. Which of these is NOT a benefit of mixed precision training?
   - A) Reduced memory usage
   - B) Faster computation on supported hardware
   - C) Always improves model accuracy
   - D) Enables larger batch sizes
3. What is the advantage of Bayesian optimization over grid search for hyperparameter tuning?

Answers: 1. B) To handle larger effective batch sizes than would fit in memory 2. C) Always improves model accuracy 3. Bayesian optimization is more efficient because it uses information from previous trials to select promising hyperparameter values, rather than trying all combinations (grid search) or random values

PyTorch

Resources for Further Learning

- **Official PyTorch resources**:
  - PyTorch documentation: `https://pytorch.org/docs/`
  - PyTorch tutorials: `https://pytorch.org/tutorials/`
  - PyTorch examples: `https://github.com/pytorch/examples`
- **Community resources**:
  - PyTorch forum: `https://discuss.pytorch.org/`
  - Papers with code: `https://paperswithcode.com/`
  - Hugging Face: `https://huggingface.co/`
- **Books**:
  - "Deep Learning with PyTorch" by Eli Stevens, Luca Antiga, and Thomas Viehmann
  - "Programming PyTorch for Deep Learning" by Ian Pointer

PyTorch

Thank you!

Any questions?