PyTorch

Researchers

What is PyTorch ?

- PyTorch is an open-source machine learning framework, primarily used for deep learning
- It's developed by Meta (Facebook) and a large community of contributors
- Think of it as a toolbox filled with specialized tools that make building, training, and deploying machine learning models much easier

Features

- **Tensor Computation**
  - At its core, PyTorch is built around **tensors** which are multi-dimensional arrays like numpy
  - GPU Acceleration: The major advantage of PyTorch (and other deep learning frameworks) is its ability to leverage **GPUs** (Graphics Processing Units). GPUs are designed for parallel processing, making them incredibly fast at the matrix operations that are fundamental to deep learning.
  - PyTorch seamlessly moves tensors between your CPU and GPU, allowing you to take advantage of this speedup.
- **Dynamic Computational Graph**
  - The graph representing your model is built as you execute the code. Each operation (e.g., addition, multiplication, activation function) is added to the graph as it's performed
  - Benefits of Dynamic Graphs:
    - Easier Debugging: Easy to step through code and see exactly what's happening at each stage.
    - Flexibility: Dynamic graphs make it easier to create models with variable-length inputs (e.g., natural language processing tasks like machine translation) and complex control flow.
    - Intuitive: It feels more natural to write code that reflects how you think about the model

Features

- **Autograd (Automatic Differentiation)**
  - Gradients are essential for training neural networks using techniques like backpropagation
  - You define your forward pass and autograd handles the backward pass
- **Modules (nn package)**
  - Provides a collection of pre-built neural network layers, loss functions and optimization algorithms
- **Data Loading & Preprocessing**
  - torchvision: Image processing (datasets, transforms)
  - torchtext: Text data (tokenization, vocabulary building)
  - torchaudio: Audio processing
- **Distributed Training**
  - PyTorch has excellent support for distributed training, allowing you to train models across multiple GPUs or even multiple machines. This is crucial for large-scale projects

Why PyTorch ?

- **Ease of Use & Debugging**: The dynamic graph and Pythonic API make it easier to learn, experiment with, and debug models.
- **Research-Friendly**: PyTorch is widely used in the research community due to its flexibility and ease of modification. Researchers often publish their code in PyTorch, making it accessible to others.
- **Growing Community**: A large and active community provides support, tutorials, and pre-trained models.
- **Production Readiness**: While initially favored for research, PyTorch has matured significantly and is now used in production environments. Tools like TorchScript allow you to optimize and deploy PyTorch models efficiently.
- **Integration with Other Tools**: PyTorch integrates well with other Python libraries and tools, such as NumPy, SciPy, and Jupyter notebooks.

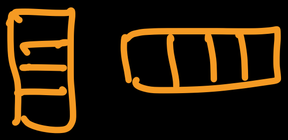
Basic Workflow

- **Define the Model**: Create a neural network using torch.nn modules (layers).
- **Prepare Data**: Load and preprocess your data using torchvision, torchtext, or custom code.
- **Define Loss Function and Optimizer**: Choose a loss function (e.g., torch.nn.CrossEntropyLoss) and an optimization algorithm (e.g., torch.optim.Adam).
- **Training Loop**: → for multiple Epochs
  - Iterate over your data in batches.
  - Perform a forward pass through the model to get predictions.
  - Calculate the loss (difference between predictions and actual values).
  - Perform a backward pass to calculate gradients.
  - Update the model's parameters using the optimizer.
- **Evaluation**: Evaluate your trained model on a separate dataset to assess its performance.

Tensor

What is Tensor?

- **Multi-Dimensional Array**
  - At its heart, a tensor is just a multi-dimensional array of numerical data — similar values
  - Think of it as an extension of NumPy arrays, but with crucial differences related to GPU acceleration and automatic differentiation
- **Generalization of Scalars, Vectors, Matrices**
  - **Scalar (0-D Tensor)** → scalar_tensor = ...[5]   5
    - A single number. Example: torch.tensor(5)
  - **Vector (1-D Tensor)** → 1D array of similar values → [1, 2, 3, 4]



    - A single row or column of numbers. Example: torch.tensor([1, 2, 3])
  - **Matrix (2-D Tensor)** $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \Rightarrow$ [[1, 2], [3, 4]]

    - A table of numbers. Example: torch.tensor([[1, 2], [3, 4]])
  - **Higher-Order Tensors**
    - Tensors with more than two dimensions
    - Think of them as cubes, hypercubes, or even more abstract structures

Creating Tensors

- **torch.tensor():** The most general way to create a tensor from Python data
- **torch.zeros():** Creates a tensor filled with zeros
- **torch.ones():** Creates a tensor filled with ones
- **torch.rand():** Creates a tensor filled with random numbers from a uniform distribution (between 0 and 1)
- **torch.randn():** Creates a tensor filled with random numbers from a standard normal distribution (mean 0, variance 1)
- **torch.empty():** Creates an uninitialized tensor (filled with garbage values). Use this when you're going to immediately fill the tensor with your own data
- **torch.arange():** Creates a 1-D tensor with values in a specified range
- **torch.linspace():** Creates a 1-D tensor with evenly spaced values over a specified interval

Key properties of Tensor

- **shape**: A tuple representing the dimensions of the tensor (rows x cols)
- **dtype**: The data type of the elements in the tensor int32, int16, int8 . .
- **device**: Indicates where the tensor is stored (CPU or GPU)
- **item()**: get data stored in a tensor

size() = No of items present in the tensor

nbytes() = returns total memory required to load tensor

Tensor Operations

- **Broadcasting:** PyTorch automatically handles broadcasting when performing operations on tensors with different shapes, as long as certain conditions are met
- Arithmetic Operators: Perform element-wise arithmetic operations
- **Matrix Multiplication (@ or torch.matmul()):** Performs matrix multiplication
- **Reshaping (reshape() or view()):** Changes the shape of a tensor without changing its data
- **Indexing and Slicing:** Access elements or sub-tensors using indexing and slicing
- **Common Functions:** PyTorch provides a wide range of functions for tensor manipulation
  - torch.max(), torch.min(): Find the maximum and minimum values
  - torch.mean(), torch.sum(): Calculate the mean and sum of elements
  - torch.exp(), torch.log(): Exponential and logarithm functions
  - torch.transpose(): Transpose a tensor (swap rows and columns)

$t = [1, 2, 3, 4, 5]$

<u>broadcasting</u>

* addition $= t + 10 = [1+10, 2+10, 3+10, 4+10] = [11, 12, 13, 14]$
* subtraction $= t - 5 = [-4, -3, -2, -1, 0]$
* power $= t ** 2 = [1, 4, 9, 16, 25]$

Important Considerations

- GPU Acceleration
  - Move tensors to the GPU using .to("cuda") (if a CUDA-enabled GPU is available) for significant performance gains in deep learning computations
- In-Place Operations
  - Some PyTorch functions have an out= argument that allows you to modify the tensor in-place (without creating a copy)
  - Use this with caution, as it can lead to unexpected behavior if not handled correctly
- Memory Management
  - Be mindful of memory usage, especially when working with large tensors
  - Use techniques like deleting unnecessary tensors and using smaller data types to reduce memory footprint

Layers

What is a Layer ?

- In PyTorch, a "layer" fundamentally refers to a modular building block within a neural network
- It's an object that performs a specific transformation on the input data.
- Think of it like a component in a larger system – each layer has its own purpose and contributes to the overall functionality of the network

What a Layer does ?

- **Linear Transformation:** Applies a matrix multiplication and bias addition (like nn.Linear)
- **Convolution:** Performs convolution operations on input data (like nn.Conv2d)
- **Pooling:** Downsamples the input data (like nn.MaxPool2d)
- **Activation:** Applies a non-linear activation function (e.g., ReLU, Sigmoid). While often used as a separate step in code, activation functions are conceptually part of a layer's transformation
- **Normalization:** Normalizes the input data (e.g., Batch Normalization)
- **Embedding:** Maps discrete values to dense vectors (e.g., nn.Embedding)

Characteristics of a Layer

- **Parameters (Learnable):**
  - Most layers have parameters that are learned during training
  - These are typically weights and biases, but can be more complex in some layers
  - These parameters are stored as torch.Tensor objects and automatically tracked by PyTorch's optimization process
- **Forward Pass (forward() method):**
  - Each layer has a forward() method that defines how the input data is transformed into an output
  - This is where the core computation of the layer takes place
- **State (Optional):**
  - Some layers might have internal state that is not learned but influences their behavior
  - For example, RNNs maintain hidden states across time steps
- **Modularity:**
  - Layers are designed to be modular and reusable
  - You can easily combine different layers to create complex network architectures

Types

- **Basic Linear Layers (Fully Connected)**
  - nn.Linear(): The most fundamental layer which performs a linear transformation
- **Convolutional Layers (For Image/Signal Processing)**
  - nn.Conv2d(): 2D Convolution. Used for image processing (e.g., CNNs)
  - nn.Conv1d(): 1D Convolution (for time series data).
  - nn.Conv3d()3D Convolution (for video or volumetric data).
- **Pooling Layers (Downsampling)**
  - nn.MaxPool2d(): Selects the maximum value within a kernel window
  - nn.AvgPool2d(): Calculates the average value within a kernel window
  - nn.AdaptiveAvgPool2d(): Dynamically adjusts the output size to a specified value
  - nn.AdaptiveMaxPool2d(): Dynamically adjusts the output size to a specified value using max pooling

Types

- **Recurrent Layers (For Sequential Data)**
  - **nn.RNN()**: Basic Recurrent Neural Network (RNN) cell
  - **nn.LSTM()**: more powerful than basic RNNs for handling long sequences
  - **nn.GRU()**: a simplified version of LSTM, often faster to train
- **Embedding Layers**
  - **nn.Embedding()**: Maps discrete indices (e.g., word IDs) to dense vectors. Used extensively in natural language processing
- **Normalization Layers**
  - **nn.BatchNorm1d()**: Batch Normalization for 1D data
  - **nn.BatchNorm2d()**: Batch Normalization for 2D data (e.g., image channels)
  - **nn.LayerNorm()**: Layer Normalization – normalizes across features within a layer
  - **nn.GroupNorm()**: Group Normalization – normalizes within groups of channels

Types

- **Activation Functions**
  - **nn.ReLU()**: Rectified Linear Unit – a common activation function
  - **nn.Sigmoid():** Sigmoid function (outputs values between 0 and 1)
  - **nn.Tanh():** Hyperbolic Tangent function (outputs values between -1 and 1)
  - **nn.LeakyReLU()**: Leaky ReLU – addresses the "dying ReLU" problem
  - **nn.Softmax()**: Softmax function (used in multi-class classification)
- **Dropout Layers (Regularization)**
  - **nn.Dropout()**: Randomly sets a fraction p of input units to 0 during training.
- **Sequence Operations**
  - **nn.Flatten():** Flattens the input tensor into a 1D tensor
  - **nn.Unfold():** Unfolds a 2D or 3D tensor into a sequence
  - **nn.Fold()**: Folds a sequence back into a 2D or 3D tensor

Regression

```python
import torch

# Define the model
model = torch.nn.Linear(1, 1)  # Linear layer: input size 1, output size 1

# Prepare data
X = torch.tensor([[1.0], [2.0], [3.0]])  # Input data
y = torch.tensor([[2.0], [4.0], [6.0]])  # Target data

# Define loss function and optimizer
loss_fn = torch.nn.MSELoss()  # Mean Squared Error loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)  # Stochastic Gradient Descent

# Training loop
for epoch in range(100):
    # Forward pass
    y_pred = model(X)

    # Calculate loss
    loss = loss_fn(y_pred, y)

    # Backward pass and optimization
    optimizer.zero_grad()  # Zero the gradients from the previous iteration
    loss.backward()  # Calculate gradients
    optimizer.step()  # Update parameters

print(f"Loss after training: {loss.item()}")
```

Classification

```python
import torch

# Define the model (a simple linear layer)
def model(x, weights, bias):
    """
    Performs a linear transformation followed by a sigmoid activation.

    Args:
        x (torch.Tensor): Input tensor.
        weights (torch.Tensor): Weight matrix.
        bias (torch.Tensor): Bias vector.

    Returns:
        torch.Tensor: Output tensor after transformation and sigmoid activation.
    """
    z = torch.matmul(x, weights) + bias  # Linear transformation
    output = torch.sigmoid(z) # Sigmoid activation for classification (0 or 1)
    return output


# Generate synthetic data
num_samples = 100
input_size = 2  # Number of features per sample
output_size = 1 # Binary classification (0 or 1)

X = torch.randn(num_samples, input_size)  # Input features
y = (torch.rand(num_samples) > 0.5).float() # Generate random labels (0 or 1)


# Initialize weights and bias randomly
weights = torch.randn(input_size, output_size)
bias = torch.randn(output_size)

# Define hyperparameters
learning_rate = 0.1
epochs = 100
```

```python
# Training loop using PyTorch's loss function and optimizer
criterion = torch.nn.functional.binary_cross_entropy  # Binary Cross Entropy Loss
optimizer = torch.optim.SGD(params=[weights, bias], lr=learning_rate) # Stochastic Gradient Descent

for epoch in range(epochs):
    # Forward pass
    predictions = model(X, weights, bias)

    # Calculate loss
    loss = criterion(predictions, y)

    # Backpropagation and optimization
    optimizer.zero_grad()  # Zero the gradients before each backward pass
    loss.backward() # Calculate gradients
    optimizer.step()  # Update weights and bias

    # Print loss every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch+1}, Loss: {loss.item()}")


# Evaluation (after training)
with torch.no_grad():  # Disable gradient calculation during evaluation
    predictions = model(X, weights, bias)
    predicted_labels = (predictions > 0.5).float() # Convert probabilities to labels
    accuracy = torch.sum(predicted_labels == y) / num_samples

print(f"Accuracy: {accuracy.item()}")
```