Introduction to PyTorch

Riley Harper

UNC-Chapel Hill

What is PyTorch ?

An open-source deep learning (and ML) library for Python, primarily developed by Facebook's AI Research (FAIR) team.

***with major contributions from Adam Paszke, Soumith Chintala, Sam Gross, Gregory Chanan, and Zeming Lin.**

Why PyTorch?

- Easy to learn and use
- Dynamic computation graph (define-by-run)
- Strong community and industry adoption
- Extensive support for deep learning tasks

PyTorch vs Competitors

	PyTorch	TensorFlow	Keras	JAX
Dynamic computation graph	Yes (define-by-run)	No (Static by default, eager execution available)	No (Runs on TensorFlow)	Yes
Ease of use	High (Pythonic)	Moderate	High	Moderate
Deployment options	Moderate	High	High	Moderate
Community support	Strong	Strong	Strong	Growing

PyTorch vs Competitors (cont.)

	PyTorch	TensorFlow	Keras	JAX
Popular for research	Very popular	Popular	Less popular	Increasing
GPU support	Excellent (via CUDA)	Excellent	Via TensorFlow	Excellent
Model interpretability	Good	Moderate	Moderate	Moderate

Tensors

PyTorch's basic data structure, similar to NumPy arrays but with the following additional features,

- **GPU acceleration**, for faster computations.
- **Autograd support**, enabling automatic differentiation for neural network training.
- **Multi-dimensionality**, ranging from scalars to high-dimensional arrays for handling complex data like images and time series.

Tensors Example

```python
import torch
x = torch.tensor([1.0, 2.0, 3.0])
print(x)
```

```
## tensor([1., 2., 3.])
```

Autograd (Automatic Differentiation)

PyTorch's mechanism for automatic differentiation, useful for:

- **Tracking gradients**: Allows tensors to remember operations for backpropagation.
- **Enabling gradient-based optimization**: Useful for training neural networks by adjusting weights using gradients.
- **Ease of use**: Automatically calculates gradients without needing to manually compute derivatives.

Autograd Example

```python
import torch

# Creating a tensor with requires_grad=True to track gradients
x = torch.tensor(3.0, requires_grad=True)
y = x**2
y.backward()  # Perform backpropagation
print(x.grad)  # Print the gradient of y with respect to x
```

Backpropagation

- Fundamental algorithm in deep learning used to train neural networks.
- Computes the gradient of the loss function with respect to each weight in the network, allowing the model to learn by updating these weights using gradient descent.
- Applies the chain rule to compute gradients layer by layer, enabling efficient gradient calculation for large networks
- PyTorch's **Autograd** simplifies this by automatically tracking operations on tensors and computing gradients, making backpropagation more manageable for complex models.

Backpropagation Example

Given the function $y = x^2$, let $x = 3.0$ and compute the gradient of y with respect to x using PyTorch's **Autograd**.

Backpropagation steps:

1. **Define the function**: $y = x^2$
2. **Compute the gradient**: The derivative of y with respect to x is $\frac{dy}{dx} = 2x$
3. **Evaluate the gradient at $x = 3.0$**: $\frac{dy}{dx} = 2(3.0) = 6.0$
4. **PyTorch computes this gradient automatically** and stores it in $x.grad$.

Result:

- The computed gradient is: $\frac{dy}{dx} = 6.0$

Autograd Example w Result

```python
import torch

# Creating a tensor with requires_grad=True to track gradients
x = torch.tensor(3.0, requires_grad=True)
y = x**2
y.backward()  # Perform backpropagation
print(x.grad)  # Print the gradient of y with respect to x
```

```
## tensor(6.)
```

Artificial Neural Networks

Artificial Neural Networks (ANNs) are computational models inspired by the human brain, consisting of layers of interconnected neurons. Each neuron receives input, processes it using a weighted sum and an activation function, and passes the result to the next layer. ANNs are widely used in deep learning for tasks like image recognition, natural language processing, and more [McCulloch & Pitts, 1943][Rosenblatt, 1958][Rumelhart, Hinton, & Williams, 1986][LeCun et al., 1998].

- **Input Layer**: Receives the input data.
  Example: For fMRI image analysis, the input layer might take voxel intensity values from a 3D fMRI scan, where each voxel represents brain activity at a given point in space.
- **Hidden Layers**: Perform computations and transformations on the data.
  Example: In fMRI data analysis, hidden layers could detect patterns or regions of interest related to brain activity, such as identifying active regions during a specific cognitive task.
- **Output Layer**: Produces the final result based on the previous layers.
  Example: The output layer in an fMRI model might classify brain activity as corresponding to different cognitive states, such as distinguishing between a resting state and task-induced activity.
- **Weights and Biases**: Parameters that the model learns to minimize the error.
  Example: During training, the model adjusts the weights and biases to correctly predict patterns in brain activity, such as recognizing specific neural signatures linked to diseases like Alzheimer's.
- **Activation Function**: Introduces non-linearity to allow learning complex patterns.
  Example: The ReLU (Rectified Linear Unit) activation function could be used in an fMRI model to detect complex non-linear relationships between brain regions, ensuring the model captures intricate patterns of brain activity.

Building a Neural Network

```python
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Linear(10, 1)

    def forward(self, x):
        return self.fc(x)

model = Net()
print(model)
```

```
## Net(
##   (fc): Linear(in_features=10, out_features=1, bias=True)
## )
```

Training Loop

```python
import torch.optim as optim

criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Dummy data
inputs = torch.randn(10)
targets = torch.randn(1)

# Training step
optimizer.zero_grad()
output = model(inputs)
loss = criterion(output, targets)
loss.backward()
optimizer.step()

print('Loss:', loss.item())
```

```
## Loss: 0.7900584936141968
```

Neural Network Example for fMRI Data (with OpenNeuro)

```python
import torch
import torch.nn as nn
import nilearn
from nilearn.datasets import fetch_neurovault_motor_task
from nilearn.input_data import NiftiMasker

# Load fMRI data from the NeuroVault motor task dataset
# NeuroVault is a repository for brain imaging data, and the motor t
dataset = fetch_neurovault_motor_task()

# The dataset contains multiple fMRI images, we are selecting the fi
fmri_img = dataset.images[0]

# Masking: Transform the 3D fMRI image into a 2D array where each vo
# NiftiMasker standardizes the data (voxel intensities) and prepares
# This will flatten the 3D image into a 2D array with shape (number
masker = NiftiMasker(standardize=True)
fmri_data = masker.fit_transform(fmri_img)
```

Neural Network Example for fMRI Data (with OpenNeuro)

```
# Define a simple neural network for fMRI data analysis
class fMRI_Net(nn.Module):
    def __init__(self):
        super(fMRI_Net, self).__init__()
        # The first fully connected layer takes the number of voxels
        # and outputs 128 features. The number of input features is
        self.fc1 = nn.Linear(fmri_data.shape[1], 128)  # Input to hi

        # The second fully connected layer reduces 128 features to 6
        self.fc2 = nn.Linear(128, 64)  # Hidden layer

        # The output layer has 2 neurons for binary classification (
        self.fc3 = nn.Linear(64, 2)  # Output layer

        # ReLU activation function for non-linearity, used after the
        self.relu = nn.ReLU()
```

Neural Network Example for fMRI Data (with OpenNeuro)

```
    # Define the forward pass of the network
    def forward(self, x):
        x = self.relu(self.fc1(x))  # Apply first layer and ReLU act
        x = self.relu(self.fc2(x))  # Apply second layer and ReLU ac
        x = self.fc3(x)  # Output layer (no activation function need

        return x

# Create a PyTorch tensor from the first scan's voxel data (fmri_dat
# This converts the fMRI data into a tensor of floats to be used in
fMRI_input = torch.tensor(fmri_data[0], dtype=torch.float32)

# Initialize the neural network
model = fMRI_Net()

# Perform a forward pass through the network with the fMRI input dat
output = model(fMRI_input)
```

Neural Network Example for fMRI Data (with OpenNeuro)

```
# Print the architecture of the model (layers and their sizes)
print(model)
```

```
## fMRI_Net(
##   (fc1): Linear(in_features=45448, out_features=128, bias=True)
##   (fc2): Linear(in_features=128, out_features=64, bias=True)
##   (fc3): Linear(in_features=64, out_features=2, bias=True)
##   (relu): ReLU()
## )
```

```
# Print the output of the model
# This will be a tensor with 2 values representing the output from t
# These 2 values can be interpreted as the scores for the 2 classes
print(output)
```

```
## tensor([-0.1668,  0.2590], grad_fn=<ViewBackward0>)
```

Questions?