**10 Principles of PyTorch**

[[Kasper Junge](https://medium.com/@kasperjuunge?source=post_page-----bbe4bf0c42cd--------------------------------)](https://medium.com/@kasperjuunge?source=post_page-----bbe4bf0c42cd--------------------------------)

[Kasper Junge](https://medium.com/@kasperjuunge?source=post_page-----bbe4bf0c42cd--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F984b37088b50&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40kasperjuunge%2F10-principles-of-pytorch-bbe4bf0c42cd&user=Kasper+Junge&userId=984b37088b50&source=post_page-984b37088b50----bbe4bf0c42cd---------------------post_header-----------)

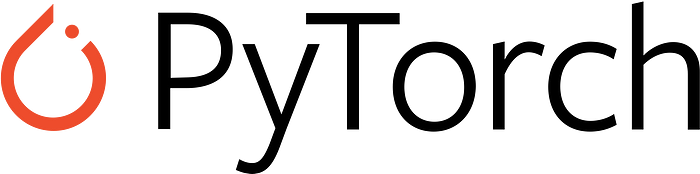
2 min read

·

Oct 20

191

3



Welcome to this concise guide on the principles of PyTorch. Whether you're a beginner or have some experience, understanding these principles can make your journey smoother. Let's gets started!

**1. Tensors: The Building Blocks**

Tensors in PyTorch are multi-dimensional arrays. They are similar to NumPy's ndarrays but can run on GPUs.

import torch  
  
# Create a 2x3 tensor  
tensor = torch.tensor([[1, 2, 3], [4, 5, 6]])  
print(tensor)

**2. Dynamic Computation Graph**

PyTorch uses dynamic computation graphs, meaning the graph is built on-the-fly as operations are executed. This provides flexibility for modifying the graph during runtime.

# Define two tensors  
a = torch.tensor([2.], requires\_grad=True)  
b = torch.tensor([3.], requires\_grad=True)  
  
# Compute result  
c = a \* b  
c.backward()  
  
# Gradients  
print(a.grad) # Gradient w.r.t a

**3. GPU Acceleration**

PyTorch allows easy switching between CPU and GPU. Utilize .to(device) for optimal performance.

device = "cuda" if torch.cuda.is\_available() else "cpu"  
tensor = tensor.to(device)

**4. Autograd: Automatic Differentiation**

PyTorch's autograd provides automatic differentiation for all operations on tensors. Set requires\_grad=True to track computations.

x = torch.tensor([2.], requires\_grad=True)  
y = x\*\*2  
y.backward()  
print(x.grad) # Gradient of y w.r.t x

**5. Modular Neural Networks with nn.Module**

PyTorch provides the nn.Module class to define neural network architectures. Create custom layers by subclassing.

import torch.nn as nn  
  
class SimpleNN(nn.Module):  
  
 def \_\_init\_\_(self):  
 super().\_\_init\_\_()  
 self.fc = nn.Linear(1, 1)  
   
 def forward(self, x):  
 return self.fc(x)

**6. Predefined Layers and Loss Functions**

PyTorch offers various predefined layers, loss functions, and optimization algorithms in the nn module.

loss\_fn = nn.CrossEntropyLoss()  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

**7. Dataset and DataLoader**

For efficient data handling and batching, PyTorch offers the Dataset and DataLoader classes.

from torch.utils.data import Dataset, DataLoader  
  
class CustomDataset(Dataset):  
 # ... (methods to define)  
   
data\_loader = DataLoader(dataset, batch\_size=32, shuffle=True)

**8. Model Training Loop**

Typically, training in PyTorch follows the pattern: forward pass, compute loss, backward pass, and parameter update.

for epoch in range(epochs):  
 for data, target in data\_loader:  
 optimizer.zero\_grad()  
 output = model(data)  
 loss = loss\_fn(output, target)  
 loss.backward()  
 optimizer.step()

**9. Model Serialization**

Save and load your models using torch.save() and torch.load().

# Save  
torch.save(model.state\_dict(), 'model\_weights.pth')  
  
# Load  
model.load\_state\_dict(torch.load('model\_weights.pth'))

**10. Eager Execution and JIT**

While PyTorch operates in eager mode by default, it offers Just-In-Time (JIT) compilation for production-ready models.

scripted\_model = torch.jit.script(model)  
scripted\_model.save("model\_jit.pt")

[Pytorch](https://medium.com/tag/pytorch?source=post_page-----bbe4bf0c42cd---------------pytorch-----------------)