In the Name of God

**PyTorch UserManual**

Produced by:

Razieh Dorehgard

Bita Nasiri

Pooria Tajmehrabi

Table of Contents:

Chapter 1: Core Concepts, Datasets, and Transforms

Chapter 2: Model Architecture and Fine-Tuning

* Why to use PyTorch for learning
* Tensors in PyTorch
* AutoGrad in PyTorch

Chapter 3: Training and Evaluating Models

Chapter 3

**Training and Evaluating Models**

**Why to Use PyTorch for Learning**

To embark on explaining why you should definitely consider PyTorch in your machine learning and deep learning projects, we should state: “Because PyTorch is extremely easy to use!”. Owing to having an appealing interface and complete documents, it is quite easy to find the solution to your potential problems. Moreover, Just like [NumPy](https://numpy.org/) provides multidimensional arrays, PyTorch offers tensors. These are generalizations of matrices to N-dimensional space and serve as the basic building blocks of many Deep learning algorithms. However, unlike NumPy arrays, PyTorch tensors can be used on GPUs for accelerated computing. Additionally, in deep learning, we often need to figure out how much to adjust things (called gradients). PyTorch has a tool called [Autograd](https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html) that does this for us automatically. Plus, with PyTorch’s dynamic approach, you can make changes as you work, which is great for some models and research situations.

Convinced? Great, let’s get started then.

**Tensors in PyTorch**

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model’s parameters.

Tensors are similar to NumPy’s ndarrays, except that tensors can run on GPUs or other specialized hardware to accelerate computing. If you’re familiar with ndarrays, you’ll be right at home with the Tensor API. If not, follow along in this quick API walkthrough.

**Tensor Initialization Ways:**

1. Directly from data

data = [[1, 2], [3, 4]]

x\_data = torch.tensor(data)

1. From a NumPy array

data = [[1, 2], [3, 4]]

np\_array = np.array(data)

x\_np = torch.from\_numpy(np\_array)

1. From another tensor

x\_ones = torch.ones\_like(x\_data)

print(f"Ones Tensor: \n {x\_ones} \n")

x\_rand = torch.rand\_like(x\_data, dtype=torch.float)

print(f"Random Tensor: \n {x\_rand} \n")

After the initialization of tensors, you can access their attributes.

**Tensor Attributes:**

Tensor attributes describe their shape, datatype, and the device on which they are stored. Some popular attributes of tensors are:

1. Shape
2. Device
3. Data type

Sample code:

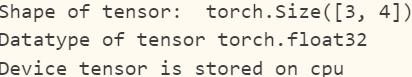
tensor = torch.rand(3, 4)

print("Shape of tensor: ", tensor.shape)

print("Datatype of tensor", tensor.dtype)

print("Device tensor is stored on", tensor.device)

Output should be:



**Tensor Operations**

Over 100 tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random sampling, and more are comprehensively described [here](https://pytorch.org/docs/stable/torch.html). Each of them can be run on the GPU, at typically higher speeds than on a CPU.

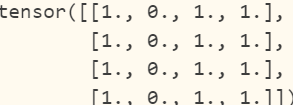
Popular tensor operations are:

1. **Indexing & Slicing**

tensor = torch.ones(4, 4)

tensor[:,1] = 0

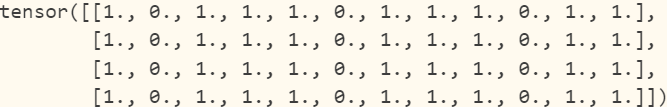
print(tensor)

****

1. **Joining**

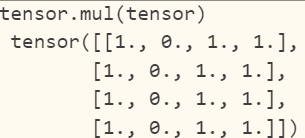
t1 = torch.cat([tensor, tensor, tensor], dim=1)

print(t1)

****

1. **Multiplying**

print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")

****

**AutoGrad in PyTorch**

is PyTorch’s automatic differentiation engine that powers neural network training. In this section, you will get a conceptual understanding of how autograd helps a neural network train.