Tensors

Disclaimer: large parts of the lab are taken from Deep Learning with PyTorch: A 60 Minute Blitz by Soumith Chintala and lectures material of Sebastian Goldt.

PyTorch *uses* tensors, i.e. specialized data structure that are basically the same as a numpy array. The have nothing to do with the learning procedure: they are generic n-dimensional arrays with data in them (inputs, outputs, parameters of the net).

Why do we use them? They are able to run on GPUs or specialized hardware that lead to more fast results. --> Ndarry just use CPU, but we need more efficient data stracture.

I am not familiar with ndarrays (false, see previous Labs), do I have to worry? No worries, let us go thorugh this quick introduction!

First thing first: import Pytorch and numpy

In [2]:
```
# The idea is to compere torch and numpy
import torch # Data structure
import numpy as np
```

Tensors can be *directly* defined from data

In [3]:
```
data = [[1, 2],[3, 4]] # what's the type? Matrix (a list)
x_data = torch.tensor(data) # In this manner you create a tensor
```

Tensors can be created *from NumPy arrays* (and vice versa).

In [4]:
```
np_array = np.array(data) # what's the type? List of numpy array
x_np = torch.from_numpy(np_array) # Torch data structure build from an narray
```

Tensors can be created from *other tensors* with the same properties (say shape, datatype), unless overridden.

In [5]:
```
# Torch element of only zero element!
x_zero = torch.zeros_like(x_data) # retains the properties of x_data
print(f"Zeros: \n {x_zero} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random: \n {x_rand} \n")
```

```
Zeros:
 tensor([[0, 0],
        [0, 0]])

Random:
 tensor([[0.8595, 0.1323],
        [0.1571, 0.7148]])
```

What about the shape? The shape is a tuple with the dimensions. Let us see how to use it

In [6]:
```
shape = (2,3,) # We can define a shape, and after select the number to put inside of the
              # Structure with the shape that we select
rand_tensor = torch.rand(shape)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print(f"Random: \n {rand_tensor} \n")
print(f"Ones: \n {ones_tensor} \n")
print(f"Zeros: \n {zeros_tensor}")
```

```
Random:
 tensor([[0.7908, 0.5807, 0.7244],
        [0.4107, 0.8857, 0.0778]])

Ones:
 tensor([[1., 1., 1.],
        [1., 1., 1.]])

Zeros:
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

The attributes, not modificable, of a tensor are

- shape,
- datatype,
- the device of storage. --> We're using GPU or CPU?

In [8]:
```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}") # GPU or CPU
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

Operations

There are *hundreds* tensor operations: indexing, slicing, mathematical operations, transposing... Give a look at the torch documentation!

Each operation can be performed on GPU (faster then CPU). On Colab, to allocate a GPU go to Edit > Notebook Settings.

In [9]:
```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
  tensor = tensor.to('cuda') # Way to tell pythoch to use GPU if it accessable
```

In [11]:
```
# On jupyter we cannot use GPU, in Colab we did in ML & DL
print(f"Device tensor is stored on: {tensor.device}") # GPU or CPU
```

```
Device tensor is stored on: cpu
```

Some examples now follow: if you are familiar with numpy, it is a piece of cake

Indexing and slicing

In [12]:
```
tensor = torch.ones(4, 4)
tensor[:,1] = 0
print(tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

Joining tensors

In [13]:
```
# Concatenate tensor
t1 = torch.cat([tensor, tensor, tensor], dim=0) # Arranging from columns
print(t1)
t2 = torch.cat([tensor, tensor, tensor], dim=1) # Arranging from rows
print(t2)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

element-wise tensor multiplication

In [14]:
```
# This computes the element-wise product --> so not matrix multiplication
print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")

# Alternative syntax:
print(f"tensor * tensor \n {tensor * tensor}")
```

```
tensor.mul(tensor)
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor * tensor
 tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

matrix tensor multiplication

In [15]:
```
# Matrix multiplication
    # --> pay attention to the dimension of the array before to multiply them!

print(f"tensor.matmul(tensor.T) \n {tensor.matmul(tensor.T)} \n")
# Alternative syntax:
print(f"tensor @ tensor.T \n {tensor @ tensor.T}")
```

```
tensor.matmul(tensor.T)
 tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])

tensor @ tensor.T
 tensor([[3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.],
        [3., 3., 3., 3.]])
```

tensor addition (in-place)

In [16]:
```
print(tensor, "\n")

tensor.add_(5) # change the same tensor --> inplace operation, I modify the tensor that I have

print(tensor)
```

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

Be carefull!

In-place operations (i.e. operations that changes directly the content of a given Tensor without making a copy) are good for memory, but can be problematic when computing derivatives.

From tensors to numpy (only on CPU)

Tensor to NumPy array

In [18]:
```
t = torch.ones(5)
print(f"t: {t}")
print(f"type of t: {type(t)}")

n = t.numpy() # We have the nupy array of the tensor
print(f"n: {n}")
print(f"type of n: {type(n)}")
```

```
t: tensor([1., 1., 1., 1., 1.])
type of t: <class 'torch.Tensor'>
n: [1. 1. 1. 1. 1.]
type of n: <class 'numpy.ndarray'>
```

if the tensor changes, the array changes (not vice versa).

In [19]:
```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.])
n: [2. 2. 2. 2. 2.]
```

From numpy to tensors

In [20]:
```
n = np.ones(5) # In place operation
t = torch.from_numpy(n)
```

Changes in the NumPy array reflects in the tensor.

In [21]:
```
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
n: [2. 2. 2. 2. 2.]
```