*) Basics of Pytorch (Deep learning framework)

↓

based on Torch

↓

provides two high level features :

i) Tensor computation (like Numpy) with strong GPU acceleration (CUDA)

ii) DNN built on a tape - based autodiff system (or autograd)

→ What are tensors ? Generalisation of matrices & vectors.

(not entirely correct)

↓

matrices → data structures, & tensors provide a relation on how these no.s relate to each other & transform eachother

↓

Deep learning also has these data structures that have a relation on the basis of how they are transformed on product, summation, activation, etc

↓

matrix like properties along with structural properties. In programs, they can be thought of as large dimensional matrices.

(1)

```
import torch
import numpy as np
import matplotlib.pyplot as plt
```

→ pytorch

} library inclusions

(2) `x = torch.ones(3, 2)` → allots a tensor of 3 * 2 dimension with value = 1

`x = torch.zeros(3,2)`

`x = torch.rand(3, 2)` → random value b/w 0 & 1

`x = torch.empty(3,2)` → doesn't initialize values, works with garbage values, and requires to be have values associated.

`y = torch.zeros_like(x)`

→ takes x, and gives us a vector of the shape of x, but with 0 values.

`x = torch.linspace(0, 1, steps = 5)`

↓

initial & final values for a tensor, with total 5 steps

tensor([0.0, 0.2500, 0.5000, 0.7500, 1.0000])

```
x = torch.tensor([[1, 2],
                  [3, 4],
                  [5, 6]])
```

→ explicitly defining tensors

→ Slice Tensors : how we can access parts of a tensor

↓

similar to the way, it is handled in numpy

>> print(x.size()) → torch.Size([3, 2])

>> print(x[:, 1]) → torch.Tensor([2, 4, 6])

↓

pick up all rows, and for each row pick the 1st index.

>> print(x[0, :]) → torch.Tensor([1, 2])

>> y = x[1, 1] → accessing a particular element

↓

tensor(4)

>> y.item() >> (4) → to fetch the item out of a single element tensor

→ Reshaping tensors

$$x = \begin{bmatrix} [1 & 2] \\ [3 & 4] \\ [5 & 6] \end{bmatrix}$$

x → (3, 2)

>> y = x.(view)(2, 3) → reshape

row major order

$$y = \begin{bmatrix} [1 & 2 & 3] \\ [4 & 5 & 6] \end{bmatrix}$$

>> y = x.view(6, -1)

↓

we define all dims except the last, and using -1, it figures the required dimension on its own.

$$y = \begin{bmatrix} [1] \\ [2] \\ [3] \\ [4] \\ [5] \\ [6] \end{bmatrix}$$

*) dimensions of tensors play a very important role in deep learning and leads to many errors.

Imp. to keep track of dimensions

and axes

→ Simple Tensor Operations.

```
>> x = torch.ones(3, 2)
>> y = torch.ones(3, 2)
>> x + y
>> x - y
>> x * y
```

} point wise operations.

```
>> z = y.add(x)
>> z = y.add_(x)
```

↓

addition inplace → modifies y as well here

→ Numpy <> Pytorch (Interfacing Numpy & Pytorch)

```
>> x_np = x.numpy()
```

→ convert to numpy by calling underlying member fn. of Tensor class.

```
>> type(x_np)
<class 'numpy.ndarray'>
>> a = np.random.random(5)
>> a_pt = torch.from_numpy(a)
```

↓

from numpy array to torch.Tensor

*) The conversion from numpy to Tensor is not a copy but rather a bridge or in terms a shallow copy meaning changes made to either are reflected in the other

•) The CPU performance of PyTorch (based on Torch written in Lua) is better than Numpy. Numpy is already an optimization over python lists.

```
[29] %%time
     for i in range(100):
       a = np.random.randn(100,100)
       b = np.random.randn(100,100)
       c = np.matmul(a, b)
```

```
CPU times: user 187 ms, sys: 97.6 ms, total: 285 ms
Wall time: 152 ms
```

```
[30] %%time
     for i in range(100):
       a = torch.randn([100, 100])
       b = torch.randn([100, 100])
       c = torch.matmul(a, b)
```

```
CPU times: user 31.2 ms, sys: 435 µs, total: 31.6 ms
Wall time: 37 ms
```

```
[31] %%time
     for i in range(10):
       a = np.random.randn(10000,10000)
       b = np.random.randn(10000,10000)
       c = a + b
```

```
CPU times: user 1min 40s, sys: 2.5 s, total: 1min 42s
Wall time: 1min 42s
```

```
[32] %%time
     for i in range(10):
       a = torch.randn([10000, 10000])
       b = torch.randn([10000, 10000])
       c = a + b
```

```
CPU times: user 21.3 s, sys: 124 ms, total: 21.4 s
Wall time: 21.4 s
```

→ CUDA support — language extension by NVIDIA to support programming GPUs directly available in C & fortran

>> torch.cuda.device_count() → "checks for the presence of any GPUs on the system"

device : GPU
host : CPU

```
>> print (torch.cuda.device(0))
```

↓

returns a reference to an object within torch referring to the cuda device.

```
>> print (torch.cuda.get_device_name(0))
```

```
>> cuda0 = torch.device('cuda:0')
```

↓

Variable used to refer the CUDA device, where 0 refers to the index based location

```
>> a = torch.ones (3, 2, device=cuda0)
```

↓

this originally used to get made on CPU, but once we specify the cuda device, it gets built on the GPU.

↓

Ops performed on these tensors also run on the GPU

↓

They give a speedup of almost 1000x as compared to CPU, which was itself opt.

```
[10] %%time
     for i in range(10):
       a_cpu = torch.randn([10000, 10000])
       b_cpu = torch.randn([10000, 10000])
       b_cpu.add_(a_cpu)

     CPU times: user 18.6 s, sys: 272 ms, total: 18.9 s
     Wall time: 18.9 s
```

```
%%time
for i in range(10):
  a = torch.randn([10000, 10000], device=cuda0)
  b = torch.randn([10000, 10000], device=cuda0)
  b.add_(a)

CPU times: user 887 µs, sys: 18 ms, total: 18.9 ms
Wall time: 22.7 ms
```

•) for matmul, on CPU we take around 4 mins 11 secs, but for the same dim tensors, we take around 33.3 ms

•) The speedup as expected is much more for parallelizable operations like matrix multiplication.

→ Autograd

```
>> x = torch.ones([3,2], requires_grad=True)
```

↓ list highlighting the shape of the req. tensors

↓ informs pyTorch, that this is something that could be differentiated wrt.

↓ i.e. I can create other tensors, that are relations on x, and hence can be derived w.r.t. x

↓ Tensor on DS level is about storing these multi-dimensional matrices but at a structural level, it relates different tensors with each other

↓ The ability to model relations between different high order matrix like DS is the basic premise behind a tensor.

```
>> y = x + 5
```

```
[3] y = x + 5
    print(y)

tensor([[6., 6.],
        [6., 6.],
        [6., 6.]], grad_fn=<AddBackward0>)
```

```
>> z = y*y + 1

>> t = torch.sum(z)
```

*) to perform the backward pass for a tensor wrt the independent tensor, we use tensor.backward().

↓

```
>> t.backward()
```

*) Now, to get the value of the derivative of the dependent tensor on which backward was called wrt the independent tensor, we use ind_tensor.grad

↓

```
>> print(x.grad)
```

$$y = x + 5$$
$$z = y^2 + 1$$
$$t = \sum_{i=1}^{n} z_i$$

*) Now if the value we call the backward fn is a single value, then it runs great, but if we run it wrt multiple values (a vector), we need to provide an argument which happens to be the gradient of the fn scalar value wrt the current vector.

$$\frac{dt}{dx_i} = \frac{dt}{dz_i}\frac{dz_i}{dy_i}\frac{dy_i}{dx_i}$$

$$\begin{bmatrix} \frac{dz}{dx} \end{bmatrix} = \begin{bmatrix} 2y \frac{dy}{dx} \end{bmatrix}$$

$$\frac{dz}{dx} = 2y$$

$$= (12)$$

↓

```
>> x = torch.ones([3,2], requires_grad=True)
>> y = x+5
>> z = 1/(1+torch.exp(-y))   :: [3,2]
>> s = torch.sum(z)
>> s.backward()
>> x.grad
```

point wise autograd

*) Now to call r.backward() in the same effect as s.backward(), we would need to provide the gradient of s wrt r as an argument in backward, with the same size as r.

```
>> a = torch.ones([3,2])
>> r.backward(a)
>> x.grad()
```

→ computes derivative of r wrt x, but pointwise multiplies with a i.e. the derivative of s wrt r.

$$\frac{\partial s}{\partial x} = \frac{\partial s}{\partial r} \cdot \frac{\partial r}{\partial x}$$ → given by r.backward()

→ Loss fn with autodiff

Training set ↑

```
>> x = torch.randn([20,1], requires_grad = True)
>> y = 3*x - 2
```

↓ ground truth value

↓ independent tensor

```
>> w = torch.tensor([1.], requires_grad = True)
>> b = torch.tensor([1.], requires_grad = True)
>> y_hat = w*x + b
>> loss = torch.sum((y_hat - y)**2)
>> loss.backward()
>> w.grad, b.grad
```

This is the basic workflow around autograd

→ Do it in a loop.

```
learning_rate = 0.01
w = torch.tensor([1.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
for i in range(10):
    x = torch.randn([20,1])
    y = 3*x - 2
    y_hat = w*x + b
    loss = torch.sum((y_hat - y)**2)
    loss.backward()
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad
        w.grad.zero_()
        b.grad.zero_()
print(w.item, b.item())
```

→ to avoid building on the computation graph

↓

There is no book keeping done for ops performed in this block of code

} → sets the grad = 0 for next iteration

*) A cell can be annotated with `%%time` to get the time taken on execution of the cell.

→ the execution loop on GPU

```
%%time

learning_rate = 0.001
N = 1000000
epochs = 200

w = torch.rand([N], requires_grad=True, device='cuda:0')
b = torch.ones([1], requires_grad=True, device='cuda:0')
for i in range(epochs):
    x = torch.randn([N], device='cuda:0')
    y = torch.dot(3*torch.ones([N], device='cuda:0'),
                  x)**2
    y_hat = torch.dot(w,x) + b
    loss = torch.sum((y_hat-y)**2)
    loss.backward()
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad
        w.grad.zero_()
        b.grad.zero_()
```

*) every tensor part of the computation needs to be constructed on the GPU to utilize the speed up. otherwise

it is no use