Introduction to Pytorch

A Deep Learning Python library
https://pytorch.org/docs/stable/index.html

Tensor = Fancy way to store numbers

- In Python: `x = 5`
- In Pytorch: `t = torch.Tensor(5)`
- From Python to Pytorch: `t = torch.Tensor(x)`
- From Pytorch to Python: `t.item()`
- Scalar / rank 1 tensor

Tensor = Fancy matrix

- Matrix in Python: `l = [[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]`
- Matrix as a tensor: `t = torch.Tensor([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]`
- Tensors can have arbitrary sizes: vector of matrices, a matrix of matrices and so on
- Rank 3 tensor

Tensor = Fancy list

- Python list: `l = [0, 1, 2, 3]`
- Pytorch tensor: `l = torch.Tensor([0, 1, 2, 3])`
- Python to Pytorch: `torch.Tensor(l)`
- Pytorch to Python: `torch.Tensor([0, 1, 2, 3]).tolist()`
- 1-dimensional tensors = vectors, rank 2 tensors

Tensor operations

Python lists

- `l1 = [0, 1, 2, 3]`
- `l2 = [4, 5, 6, 7]`
- `print(l1 * l2)`

Pytorch tensors

- `t1 = torch.Tensor([0, 1, 2, 3])`
- `t2 = torch.Tensor([4, 5, 6, 7])`
- `print(t1 * t2)`

Tensor operations (contd.)

- most are element-wise operations: multiplying 2 tensors means multiplying their corresponding elements
- we can also: add, subtract, divide, raise to power
- `torch.Tensor([1, 2, 3]) * 4 = ?`

Matrix (2-dimensional tensors) operations

- Matrix multiplication: recall multiplying with * does element-wise multiplication => Hadamard product
- Usual matrix multiplication: `@` operator, or `torch.mm`
- Switch the rows and the columns (known as a transpose): `torch.Tensor([[1, 2, 3], [4, 5, 6]]).T`

Sum, mean, norm and abs

- add up everything in a Tensor: `torch.Tensor([1, 2, 3]).sum() == 6`
- compute the mean of everything in a Tensor: `torch.Tensor([1, 2, 3]).mean() == 2`
- don’t like negative numbers?: `torch.Tensor([-1, -2, 3]).abs() == torch.Tensor([1, 2, 3])`
- how big is a tensor?: `torch.norm(torch.Tensor([-1, -2, 3])) == sqrt(1+4+9)`

Ways to make a Tensor

- `torch.arange(10)` => 1-dimensional tensor (vector / list) that contains all numbers from 0 to 9
- `torch.arange(10).reshape(2, 5)` => matrix with 2 rows and 5 columns containing all numbers from 0 to 9
- `torch.arange(10).reshape(-1, 5)` => same as above
- `torch.zeros((5, 10))` => tensor of shape 5 x 10 filled with 0
- `torch.ones((5, 10))` => tensor of shape 5 x 10 filled with 1
- `torch.full((5, 10), value = 15)` => tensor of shape 5 x 10 filled with a value we choose (15)

Random tensors

- `torch.rand((2, 3, 4))` => tensor filled with random numbers between 0 and 1
- `torch.randn((2, 3, 4))` => normally distributed values with mean 0 and variance 1

Tensor metadata

- `dtype` (data type): float32, float64, int8, int16, int32, int64, bool and many others
  - `torch.arange(10).dtype` => torch.int64 a.k.a. torch.long
  - to set dtype: `torch.arange(10).type(torch.float32)`
- `shape:` tensor size across each axis
  - `torch.arange(10).shape` => torch.size([10])
  - `torch.arange(100).reshape(4, 25)` => torch.size([4, 25])
- `device:` is the tensor on the GPU or on the CPU?

A bit about GPUs

CPUs:

- Great at running sequential code
- Extremely complex
- Instruction pipelines, caches etc.
- Might have a few cores

GPUs:

- Contains thousands of CPU-like cores
- Each core is much simpler than an usual CPU
- We can divide a task between the cores in a GPU

Multiply 2 vectors of 1000 elements on a CPU:

```
for i in range(1000):

    out[i] = a[i] * b[i]
```

1000 time units

Multiply 2 vectors of 1000 elements on a GPU:

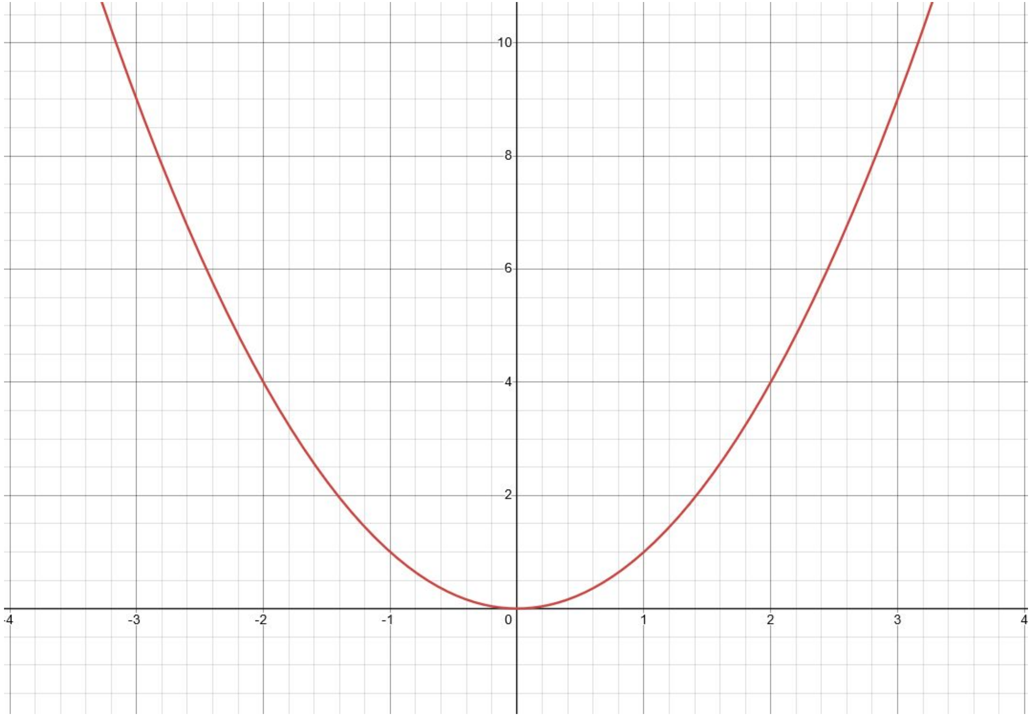
- `core 1 computes a[0] * b[0] and stores it in out[0]`
- …
- `Core 1000 computes a[999] * b[999] and stores results in out[999]`

1 time unit

Pytorch and the GPU

- Tensors can live on our CPU (they actually live in the RAM, meaning we can access them from the CPU) or on the GPU
- Everything we discussed earlier applies to both CPU tensors and GPU tensors
- Move a Tensor to another device: `t.to("cpu"), t.to("cuda")`
- Pytorch may use special, optimized implementations for whatever operations we're doing if the tensors are on the GPU

Derivatives



Why derivatives?

- **We care about the derivative because: it tells us what happens to the function value if we increase the value of its parameters!**
- Positive derivative: we increased the parameter and the function increased as well
- Negative derivative: we increased the parameter and the function decreased
- Derivative is 0?: we increased the parameter and the function didn’t change a bit

Gradient = more derivatives

- Function of more than one variable: `F(x, y) = x^2*y^2`
- Derivative w.r.t. to `x`: what happens to the function if we change `x`?
- Derivative w.r.t. to `y`: what happens to the function if we change `y`?
- Put them together: gradient of the function `F` => what happens to the function if we change `x` and `y`?

No more derivatives!

- Pytorch can automatically compute derivatives / gradients for us!
- `x = torch.Tensor([10, 12, 14, 16]).`**`requires_grad_()`**
- `z = x * x`
- `z = torch.sum(z)`
- `z.`**`backward()`**

- `z = x * x`
- `z = [x1 *  x1, x2 * x2, x3 * x3, x4 * x4]`
- To use **.backward()**, we need to obtain a scalar value: `z` is a vector of 4 elements
- To get a single value: add up all values in **z**
- => `x1^2+x2^2+x3^2+x4^2`
- What is the gradient of this value with respect to `x`?

Broadcasting

- can we multiply a Tensor of shape (1, 5) by a Tensor of shape (2, 5)?

  [[1, 2, 3, 4, 5]] * [[1, 2, 3, 4, 5],

  [4, 5, 6, 7, 8]]

- Broadcasting tries to duplicate the first Tensor until it reaches the size of the second Tensor:

  [[1, 2, 3, 4, 5], * [[1, 2, 3, 4, 5],

  [1, 2, 3, 4, 5]] [4, 5, 6, 7, 8]]

Slicing and dicing

- We have a matrix: `t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`
- `t[0]` => first line: `torch.Tensor([1, 2, 3])`
- `t[-1] = t[2]` => last line: `torch.Tensor([7, 8, 9])`
- `t[0:1]` => first 2 lines: `torch.Tensor([[1, 2, 3], [4, 5, 6]])`
- `t[:, 0:1] =>` first 2 columns: `torch.Tensor([[1, 2], [4, 5], [7, 8]])`
- `t[:, 0:1] = 4` => makes first 2 columns equal to 4

Tensor concatenation

- `t1 = torch.Tensor([[1, 2, 3],`
  `[4, 5, 6]])`
- `t2 = torch.Tensor([[7, 8, 9],`
  `[3, 7, 9]])`
- Make 1 tensor out of the 2: `torch.cat((t1, t2), axis = 0)`
  `torch.cat((t1, t2), axis = 1)`