

# 50.039 Theory and Practice of Deep Learning

## W3-S1 Introduction to Deep Learning using the PyTorch framework

Matthieu De Mari



# Introduction (Week 3)

1. What is the **PyTorch library** and its **benefits**?
2. What is a **PyTorch tensor object** and its typical **attributes**?
3. How to implement some typical **tensor operations**?
4. What is **broadcasting** on tensors?
5. What are **tensor locations** in terms of computation?
6. How to **transform our original NumPy shallow Neural Network** class so it uses **PyTorch** now instead?
7. How to implement a **forward**, **loss** and **accuracy** metric in PyTorch?
8. What are some measurable **performance benefits** of using **PyTorch** over NumPy and **GPUs** over CPUs?

# Introduction (Week 3)

9. What is the **autograd/backprop** module in PyTorch, and how does it use a **computational graph** to **compute all derivatives**?
10. How to use the **autograd** to implement **derivatives** and a **vanilla gradient descent**?
11. How to implement **backprop** in PyTorch for our **shallow Neural Network** class?
12. How to use **PyTorch** to implement **advanced optimizers**?
13. How to use **PyTorch** to implement **advanced initializers**?
14. How to use **PyTorch** to implement **regularization**?
15. How to finally revise our **trainer** function to obtain a minimal, yet complete Neural Network in PyTorch?

# Introduction (Week 3)

16. What are the **Dataset** and **Dataloader** objects in **PyTorch**?
17. How to implement a custom **Dataloader** and **Dataset** object in PyTorch?
18. How to move from binary classification to **multi-class classification**?
19. How to adjust output probabilities using the **softmax** function?
20. How to change the **cross-entropy loss** so it works in **multi-class classification**?
21. How to implement **building blocks** in PyTorch?
22. How to implement and train our first **Deep Neural Network**?
23. What are **additional good practices** in PyTorch?

# Hyperparameters tuning and searching

## Definition (**hyperparameters tuning and searching**):

Earlier, we have identified that several parameters in our neural networks had to be manually decided by the user and called those **hyperparameters**.

Unfortunately, the **NFL theorem** tells us there is often **no closed-form formula** to tell you what is the **best value to use for those hyperparameters**.

Typical **hyperparameters** include:

- Number of layers and their sizes,
- Initializers to use for parameters,
- Activation functions to use,
- Learning rate, momentum and other parameters related to optimizers,
- Etc.

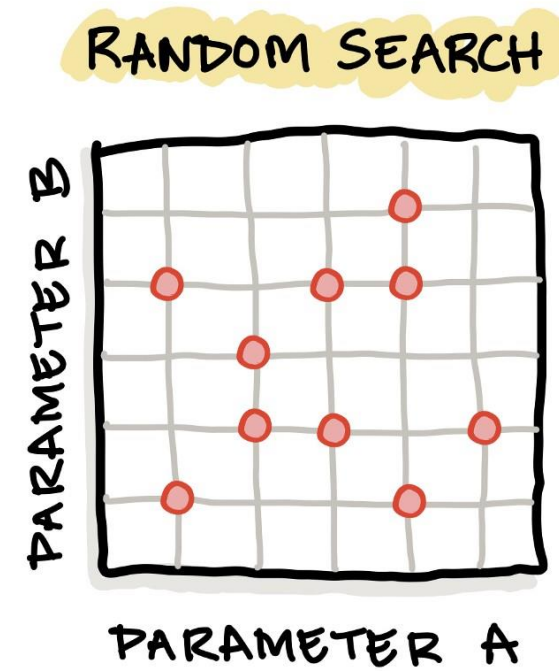
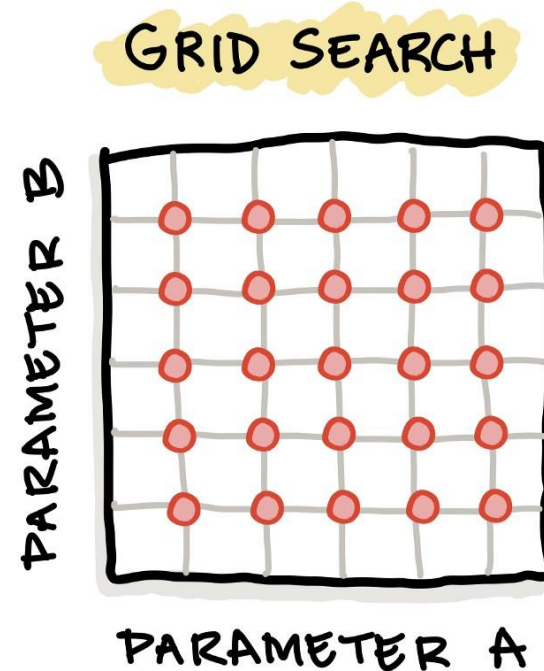
**No choice:** try different values and possibilities.

# Hyperparameters tuning and searching

**Definition (hyperparameter grid searching):**

**Grid search** is the simplest algorithm for hyperparameter tuning.

Basically, we divide the domain of possible values for each of the hyperparameters into a discrete grid. Then, we will try every combination of values of this grid, calculating some performance metrics.

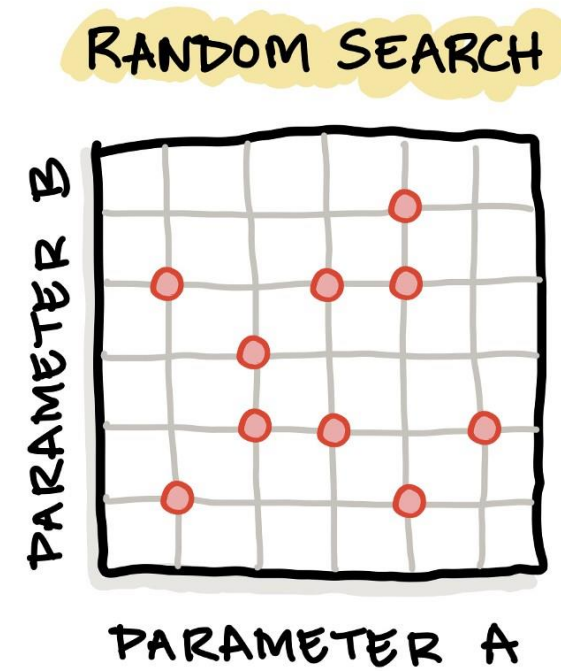
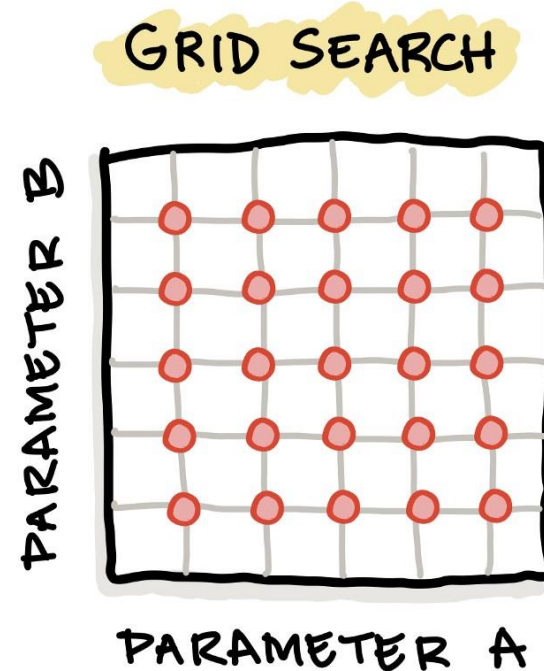


# Hyperparameters tuning and searching

Definition (**hyperparameter random searching**):

**Random search** is the another algorithm for hyperparameter tuning.

**Random search** defines a search space as a bounded domain of hyperparameter values and randomly sample points in that domain. After trying enough values, keep the best hyperparameters configuration.

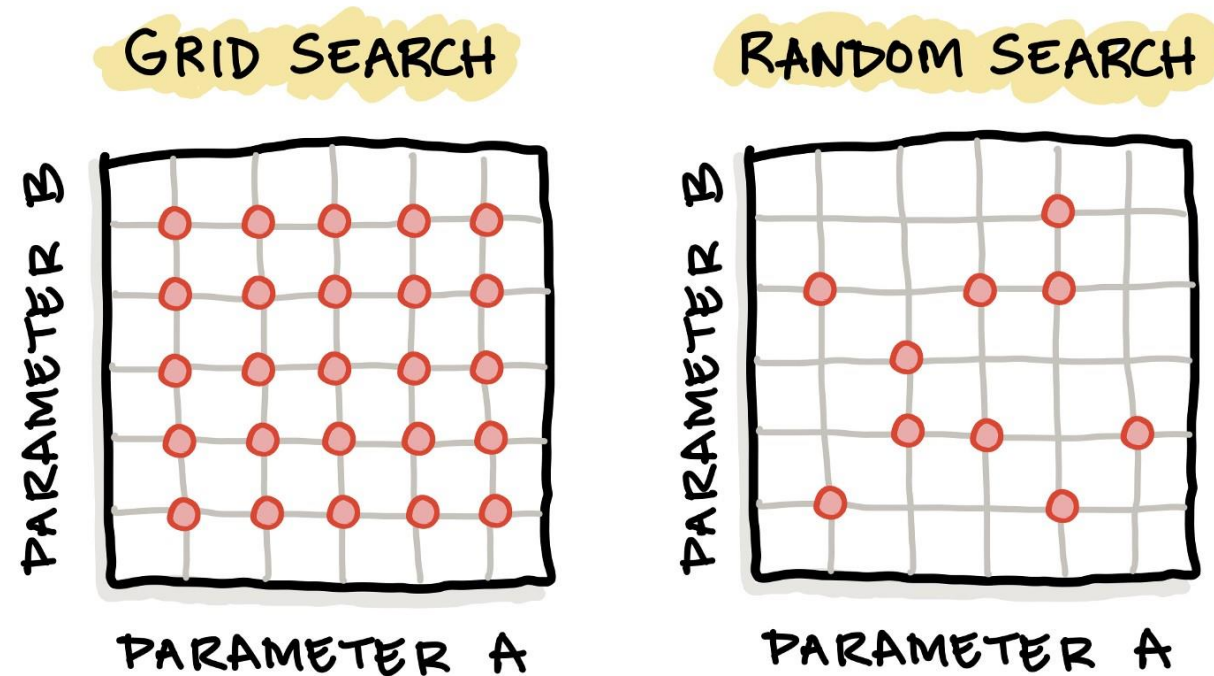


# Hyperparameters tuning and searching

In general ([HypSearch]),

- grid search is often better on small spaces,
- whereas random search is better on large domains, but has no guarantee of returning a good configuration.

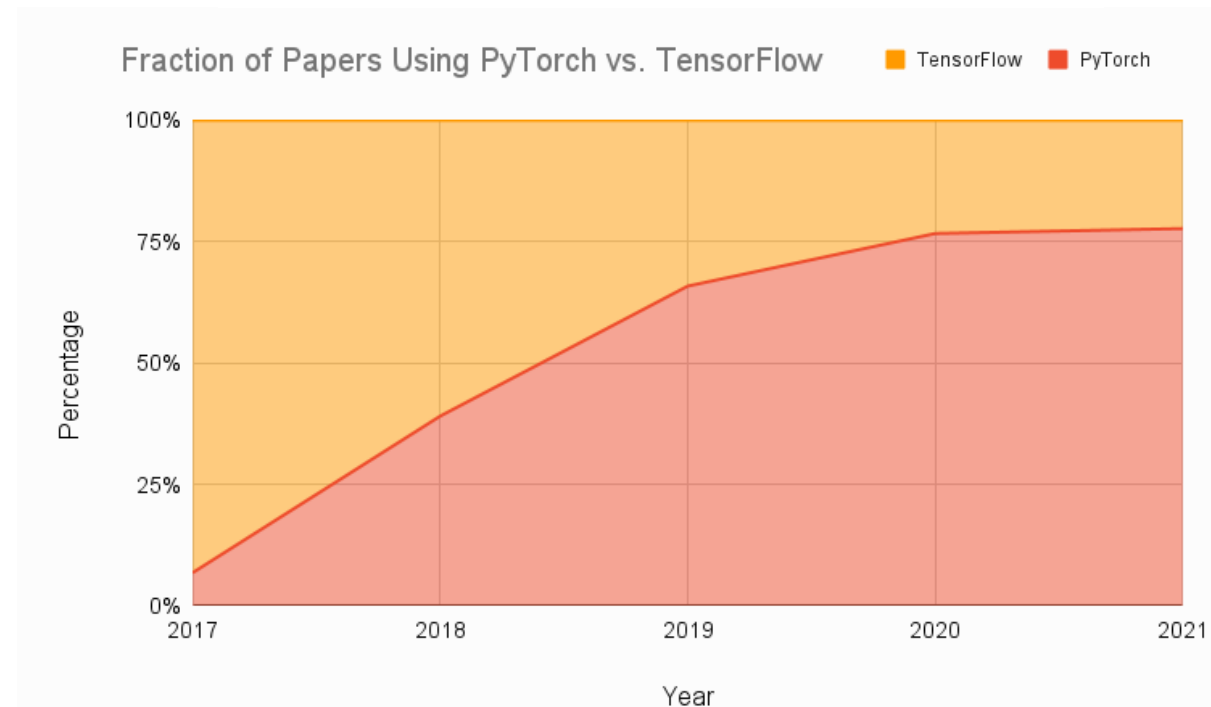
More advanced techniques than these two exist, such as Bayesian Optimization and Evolutionary Optimization.





# Technical pre-requisites

- Framework of choice will be **PyTorch!** (not Tensorflow, not Keras, not MXNet, etc.)
- Increasing popularity and preferred to Google's Tensorflow these days for many reasons.
- Learn more, if curious:  
<https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/>



# On the benefits of using PyTorch

There are several benefits to using PyTorch over NumPy for implementing neural networks:

- **More intuitive interface** for working with tensors and neural networks.
  - NumPy is primarily a numerical computing library.
  - PyTorch is specifically designed with deep learning, and provides a more natural and convenient interface.
- **Large active community** of users.
  - Many tutorials, detailed documentations and implementations on Github.
  - Stack overflow equivalent.
- PyTorch has **better support for GPU acceleration** than NumPy.
  - If a GPU is available, significantly **speed up the training** of our neural network by performing the computations on the GPU using PyTorch.
  - Especially useful for **training large and complex models**.
- **High-level abstractions** for building and training neural networks.
  - Easier to write and debug code,
  - Improve the performance of our model by allowing PyTorch to apply optimization techniques
  - More specifically, **automatic differentiation!**

# At this point, you should have installed PyTorch, and if possible, CUDA

- You can check for CUDA/GPU capabilities, using the line below. If the CUDA has not been properly installed or the GPU is not compatible, you will be using a CPU instead.
- We strongly advise to take a moment to make sure your machine is CUDA enabled, assuming your GPU is compatible. When CUDA is properly installed on a compatible GPU, the line below should display *cuda*, otherwise it will print *cpu*.

```
1 # Use GPU if available, else use CPU
2 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
3 print(device)
```

cuda

# The Tensor object

## Definition (**Tensor**):

The tensor is PyTorch's basic building block and is similar to NumPy arrays.

This is why most of the concepts and methods will look similar

However, these come with additional features, which will be useful later on when building Neural Networks with these tensors.

```
1 # Create a 2D Numpy array and a PyTorch tensor,  
2 # both of size 2 by 5, filled with ones.  
3 ones_array = np.ones((2, 5))  
4 print(ones_array)  
5 ones_tensor = torch.ones(size = (2, 5))  
6 print(ones_tensor)
```

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

# Creating a Tensor

- Tensors can be simply created as in NumPy, by using **zeros()** or **ones()** functions, specifying dimensions of the expected Tensor, with tuples.
- They can also be created from a **list (of lists)** of values, by passing it to the **tensor()** function.
- Or from a **Numpy array**, using the **from\_numpy()** method.

```
1 # Create a 2D Numpy array and a PyTorch tensor,
2 # both of size 2 by 5, filled with ones.
3 ones_array = np.ones((2, 5))
4 print(ones_array)
5 ones_tensor = torch.ones(size = (2, 5))
6 print(ones_tensor)
```

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

```
1 # Create a 1D tensor of size 3, filled zeros.
2 # (Pay attention to the extra comma in the tuple.)
3 zeros_tensor = torch.zeros(size = (3, ))
4 print(zeros_tensor)
```

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

```
1 # Create a Tensor from a list, directly
2 l = [1, 2, 3, 4]
3 list_tensor = torch.tensor(l)
4 print(list_tensor)
```

```
tensor([1, 2, 3, 4])
```

```
1 # From a numPy array
2 numpy_array = np.array([0.1, 0.2, 0.3])
3 numpy_tensor = torch.from_numpy(numpy_array)
4 print(numpy_tensor)
```

```
tensor([0.1000, 0.2000, 0.3000], dtype=torch.float64)
```

# Tensor datatypes (dtypes)

## Definition (**dtypes**):

PyTorch tensors have an attribute called **dtype**, which tracks the **datatype of values stored in the tensor**.

See link below for more details on the possible dtypes.

[https://www.tensorflow.org/api\\_docs/python/tf/dtypes](https://www.tensorflow.org/api_docs/python/tf/dtypes)

In general, **operations between tensors require compatible, and sometimes identical dtypes**.

Here, the dot product failed because one tensor was int, and the other was float/double.

```
1 # Create a Tensor from a list, directly
2 # Forcing dtype to be integers on 32bits.
3 l = [1, 2, 3, 4]
4 list_tensor = torch.tensor(l, dtype = torch.int32)
5 print(list_tensor)
6 # Changing to float 64bits
7 l2 = [2, 4, 6, 8]
8 list_tensor2 = torch.tensor(l2, dtype = torch.double)
9 print(list_tensor2)
10 # Some operations on tensors with different datatypes might be problematic
11 list_tensor3 = torch.dot(list_tensor, list_tensor2)
12 print(list_tensor3)
```

```
tensor([1, 2, 3, 4], dtype=torch.int32)
tensor([2., 4., 6., 8.], dtype=torch.float64)
```

RuntimeError

Traceback (most recent call last)

Cell In[16], line 11

```
9 print(list_tensor2)
10 # Some operations on tensors with different datatypes might be problematic
--> 11 list_tensor3 = torch.dot(list_tensor, list_tensor2)
12 print(list_tensor3)
```

RuntimeError: dot : expected both vectors to have same dtype, but found Int and Double

# Tensor datatypes (dtypes)

## Definition (**dtypes**):

PyTorch tensors have an attribute called **dtype**, which tracks the **datatype of values stored in the tensor**.

See link below for more details on the possible dtypes.

[https://www.tensorflow.org/api\\_docs/python/tf/dtypes](https://www.tensorflow.org/api_docs/python/tf/dtypes)

In general, **operations between tensors require compatible, and sometimes identical dtypes**.

You can change the dtype of a tensor

- by either **specifying it explicitly during its creation**;
- or by using the **type()** method on the tensor, casting a new dtype.

```
1 # Create a Tensor from a list, directly
2 # Forcing dtype to be integers on 32bits.
3 l = [1, 2, 3, 4]
4 list_tensor = torch.tensor(l, dtype = torch.int32)
5 print(list_tensor)
6 # Changing to float 64bits
7 list_tensor2 = list_tensor.type(torch.float64)
8 print(list_tensor2)
```

```
tensor([1, 2, 3, 4], dtype=torch.int32)
tensor([1., 2., 3., 4.], dtype=torch.float64)
```

# Tensor random initializers

Tensors can also be **initialized using random generators**, as in NumPy.

For instance we can:

- use **rand()** for drawing random values in a **[0, 1] uniform distribution**,
- or use **randn()** to draw values from a **normal distribution with zero mean and variance one**.

Seeding is done with **torch.manual\_seed()**.

- Functions and methods exist for calculating mean values of a tensor, its standard deviation/variance, etc.

```
1 # Create a 3D tensor, of size 3 by 2 by 2, filled with random values
2 # drawn from a uniform [0, 1] distribution.
3 rand_unif_tensor = torch.rand(size = (3, 2, 2))
4 print(rand_unif_tensor)
5 # Calculate mean with function (should be close to 0.5)
6 val = torch.mean(rand_unif_tensor)
7 print(val)
```

```
tensor([[[[0.1515, 0.2990],
           [0.0024, 0.4920]],

         [[0.4696, 0.5268],
           [0.0309, 0.7782]],

         [[0.9518, 0.7263],
           [0.8879, 0.9096]]]])
tensor(0.5188)
```



# Tensor random initializers

Tensors can also be **initialized using random generators**, as in NumPy.

For instance we can:

- use **rand()** for drawing random values in a **[0, 1] uniform distribution**,
- or use **randn()** to draw values from a **normal distribution with zero mean and variance one**.

Seeding is done with **torch.manual\_seed()**.

- Functions and methods exist for calculating mean values of a tensor, its standard deviation/variance, etc.

```
1 # Seeding
2 torch.manual_seed(17)
3
4 # Create a 4D tensor, of size 4 by 2 by 3 by 7, filled with random values
5 # drawn from a normal distribution with zero mean and variance one.
6 rand_normal_tensor = torch.randn(size = (4, 2, 3, 7))
7 print(rand_normal_tensor.shape)
8
9 # Calculate mean with method (should be close to 0)
10 # (With see 17, should be 0.0865)
11 val = rand_normal_tensor.mean()
12 print(val)
```

```
torch.Size([4, 2, 3, 7])
tensor(0.0865)
```

# Additional attributes of Tensors

- Tensors come with many attributes and methods, e.g. asking the shape of a tensor, like in NumPy, is done using the **shape** attribute.

```
1 print(rand_normal_tensor.shape)
```

```
torch.Size([4, 2, 3, 2])
```

# Additional attributes of Tensors

- Tensors come with many attributes and methods, e.g. asking the shape of a tensor, like in NumPy, is done using the **shape** attribute.
- Too many of them for me to cover, so **RTFM!**

```
1 print(dir(rand_normal_tensor))
```

```
['H', 'T', '__abs__', '__add__', '__and__', '__array__', '__array_priority__', '__array_wrap__', '__bool__', '__class__', '__complex__', '__contains__', '__deepcopy__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__div__', '__dlpack__', '__dlpack_device__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__iand__', '__idiv__', '__ifloordiv__', '__ilshift__', '__imod__', '__imul__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__ior__', '__ipow__', '__irshift__', '__isub__', '__iter__', '__itruediv__', '__ixor__', '__le__', '__len__', '__long__', '__lshift__', '__lt__', '__matmul__', '__mod__', '__module__', '__mul__', '__ne__', '__neg__', '__new__', '__nonzero__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rfloordiv__', '__rlshift__', '__rmatmul__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__setitem__', '__setstate__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__torch_dispatch__', '__torch_function__', '__truediv__', '__weakref__', '__xor__', '_addmm_activation', '_autocast_to_full_precision', '_autocast_to_reduced_precision', '_backward_hooks', '_base', '_cdata', '_coalesced_', '_conj', '_conj_physical', '_dimI', '_dimV', '_fix_weakref', '_grad', '_grad_fn', '_indices', '_is_view', '_is_zerotensor', '_make_subclass', '_make_wrapper_subclass', '_neg_view', '_nested_tensor_layer_norm', '_nnz', '_python_dispatch', '_reduce_ex_internal', '_storage', '_to_dense', '_update_names', '_values', '_version', 'abs', 'abs_', 'absolute', 'absolute_', 'acos', 'acos_', 'acosh', 'acosh_', 'add', 'add_', 'addbmm', 'addbmm_', 'addcddiv', 'addcddiv_', 'addcmul', 'addcmul_',
```

# Basic operations on Tensors

All the typical element-wise operations work on tensors. For instance, we can:

- **access elements using the square bracket notation**, multiple square bracket notations and multiple indexes in a single square bracket;

```
1  # Create a 3D tensor, of size 3 by 2 by 2, filled with random values
2  # drawn from a uniform [0, 1] distribution.
3  torch.manual_seed(17)
4  rand_unif_tensor = torch.rand(size = (3, 2, 2))
5  print(rand_unif_tensor)
6
7  # Indexing
8  element1 = rand_unif_tensor[2]
9  print("Element1: ", element1)
10 element2 = rand_unif_tensor[2][0]
11 print("Element2: ", element2)
12 element3 = rand_unif_tensor[2, 0, 1]
13 print("Element3: ", element3)
```

```
tensor([[[0.4342, 0.5351],
         [0.8302, 0.1239]],

        [[0.0293, 0.5494],
         [0.3825, 0.5463]],

        [[0.4683, 0.0172],
         [0.0214, 0.3664]]])
Element1: tensor([[0.4683, 0.0172],
                  [0.0214, 0.3664]])
Element2: tensor([0.4683, 0.0172])
Element3: tensor(0.0172)
```

# Basic operations on Tensors

All the typical element-wise operations work on tensors. For instance, we can:

- **access elements using the square bracket notation**, multiple square bracket notations and multiple indexes in a single square bracket;
- **slice a tensor** using the square bracket notation and colon symbol;

```
1 # Create a 3D tensor, of size 3 by 2 by 2, filled with random values
2 # drawn from a uniform [0, 1] distribution.
3 torch.manual_seed(17)
4 rand_unif_tensor = torch.rand(size = (3, 2, 2))
5 print(rand_unif_tensor)
6
7 # Slicing
8 slice1 = rand_unif_tensor[0:2]
9 print("Slice1: ", slice1)
10 slice2 = rand_unif_tensor[:,2]
11 print("Slice2: ", slice2)
12 slice3 = rand_unif_tensor[1:]
13 print("Slice3: ", slice3)
14 slice4 = rand_unif_tensor[0, :, :]
15 print("Slice4: ", slice4)
```

```
tensor([[[[0.4342, 0.5351],
          [0.8302, 0.1239]],

         [[0.0293, 0.5494],
          [0.3825, 0.5463]],

         [[0.4683, 0.0172],
          [0.0214, 0.3664]]]])
Slice1: tensor([[[[0.4342, 0.5351],
          [0.8302, 0.1239]],

         [[0.0293, 0.5494],
          [0.3825, 0.5463]]]])
Slice2: tensor([[[[0.4342, 0.5351],
          [0.8302, 0.1239]],

         [[0.0293, 0.5494],
          [0.3825, 0.5463]]]])
Slice3: tensor([[[[0.0293, 0.5494],
          [0.3825, 0.5463]],

         [[0.4683, 0.0172],
          [0.0214, 0.3664]]]])
Slice4: tensor([[[0.4342, 0.5351],
          [0.8302, 0.1239]]])
```

# Basic operations on Tensors

All the typical element-wise operations work on tensors. For instance, we can:

- **access elements using the square bracket notation**, multiple square bracket notations and multiple indexes in a single square bracket;
- **slice a tensor** using the square bracket notation and colon symbol;
- **update elements** of a tensor using the square bracket notation;

```

1  # Create a 3D tensor, of size 3 by 2 by 2, filled with random values
2  # drawn from a uniform [0, 1] distribution.
3  torch.manual_seed(17)
4  rand_unif_tensor = torch.rand(size = (3, 2, 2))
5  print(rand_unif_tensor)
6
7  # Before
8  element4 = rand_unif_tensor[2, 1, 1]
9  print("Element4: ", element4)
10 # Updating
11 rand_unif_tensor[2, 1, 1] = 0.5
12 # After
13 element4 = rand_unif_tensor[2, 1, 1]
14 print("New Element4: ", element4)

```

```

tensor([[[[0.4342, 0.5351],
          [0.8302, 0.1239]],

         [[0.0293, 0.5494],
          [0.3825, 0.5463]],

         [[0.4683, 0.0172],
          [0.0214, 0.3664]]]])
Element4:  tensor(0.3664)
New Element4:  tensor(0.5000)

```

# Basic operations on Tensors

All the typical element-wise operations work on tensors. For instance, we can:

- **access elements using the square bracket notation**, multiple square bracket notations and multiple indexes in a single square bracket;
- **slice a tensor** using the square bracket notation and colon symbol;
- **update elements** of a tensor using the square bracket notation;
- **browse through elements of a tensor** using a **for loop**.

```

1  # Create a 3D tensor, of size 3 by 2 by 2, filled with random values
2  # drawn from a uniform [0, 1] distribution.
3  torch.manual_seed(17)
4  rand_unif_tensor = torch.rand(size = (3, 2, 2))
5  print(rand_unif_tensor)
6
7  # Browsing
8  for sub_tensor in rand_unif_tensor:
9      print("---")
10     print(sub_tensor)

```

```

tensor([[[[0.4342, 0.5351],
           [0.8302, 0.1239]],

          [[0.0293, 0.5494],
           [0.3825, 0.5463]],

          [[0.4683, 0.0172],
           [0.0214, 0.3664]]]])

---
tensor([[0.4342, 0.5351],
        [0.8302, 0.1239]])

---
tensor([[0.0293, 0.5494],
        [0.3825, 0.5463]])

---
tensor([[0.4683, 0.0172],
        [0.0214, 0.3664]])

```

# Arithmetic operations on Tensors

All NumPy array operations work on Tensors and equivalent methods have been written in torch as well.

- Element-wise Addition/Subtraction,

```
1 # Define two simple 2D tensors
2 a = torch.tensor([[1, 2, 3], [1, 2, 3]])
3 b = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
1 # Element-wise addition
2 c = a + b
3 print(c)
4 c = torch.add(a, b)
5 print(c)
```

```
tensor([[2, 4, 6],
        [5, 7, 9]])
tensor([[2, 4, 6],
        [5, 7, 9]])
```

```
1 # Element-wise subtraction
2 c = a - b
3 print(c)
4 c = torch.sub(a, b)
5 print(c)
```

```
tensor([[ 0,  0,  0],
        [-3, -3, -3]])
tensor([[ 0,  0,  0],
        [-3, -3, -3]])
```



# Arithmetic operations on Tensors

All NumPy array operations work on Tensors and equivalent methods have been written in torch as well.

- Element-wise Addition/Subtraction,
- Element-wise Multiplication/Division,

```
1 # Define two simple 2D tensors
2 a = torch.tensor([[1, 2, 3], [1, 2, 3]])
3 b = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
1 # Element-wise multiplication
2 c = a * b
3 print(c)
4 c = torch.mul(a,b)
5 print(c)
```

```
tensor([[ 1,  4,  9],
        [ 4, 10, 18]])
tensor([[ 1,  4,  9],
        [ 4, 10, 18]])
```

```
1 # Element-wise division
2 c = a / b
3 print(c)
4 c = torch.div(a, b)
5 print(c)
```

```
tensor([[1.0000, 1.0000, 1.0000],
        [0.2500, 0.4000, 0.5000]])
tensor([[1.0000, 1.0000, 1.0000],
        [0.2500, 0.4000, 0.5000]])
```

# Arithmetic operations on Tensors

All NumPy array operations work on Tensors and equivalent methods have been written in torch as well.

- Element-wise Addition/Subtraction,
- Element-wise Multiplication/Division,
- Transposition,

```
1 # Define two simple 2D tensors
2 a = torch.tensor([[1, 2, 3], [1, 2, 3]])
3 b = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
1 # Transposition
2 c = a.T
3 print(a)
4 print(c)
```

```
tensor([[1, 2, 3],
        [1, 2, 3]])
tensor([[1, 1],
        [2, 2],
        [3, 3]])
```

```
1 # Transpose and swap dimensions 0 and 1
2 # (could specify other dimensions if ND tensor)
3 d = b.transpose(0, 1)
4 print(b)
5 print(d)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

# Arithmetic operations on Tensors

All NumPy array operations work on Tensors and equivalent methods have been written in torch as well.

- Element-wise Addition/Subtraction,
- Element-wise Multiplication/Division,
- Transposition,
- Matrix multiplication and dot product,
- Etc.

```
1 # Define two simple 2D tensors
2 a = torch.tensor([[1, 2, 3], [1, 2, 3]])
3 b = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
1 # Transpose and swap dimensions 0 and 1
2 # (could specify other dimensions if ND tensor)
3 d = b.transpose(0, 1)
4 print(b)
5 print(d)
```

```
1 # Matrix multiplication
2 e = torch.matmul(a, d)
3 print(e)
```

```
tensor([[14, 32],
        [14, 32]])
```

```
1 # Define two simple 1D tensors
2 a = torch.tensor([1, 2, 3])
3 b = torch.tensor([4, 5, 6])
4
5 # Dot operation, used for computing
6 # the dot product of two 1D tensors.
7 f = torch.dot(a, b)
8 print(f)
9 g = torch.matmul(a, b.T)
10 print(g)
```

```
tensor(32)
tensor(32)
```

# A quick note on broadcasting

Tensors, just like NumPy arrays, support broadcasting.

Two tensors are “broadcastable” if the following rules hold:

- Each tensor has at least one dimension.
- When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.
- If two tensors  $x$ ,  $y$  are “broadcastable”, the resulting tensor size is calculated as follows:
- If the number of dimensions of  $x$  and  $y$  are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
- Then, for each dimension size, the resulting dimension size is the max of the sizes of  $x$  and  $y$  along that dimension.

# A quick note on broadcasting

```
1 # Same shapes are always broadcastable
2 # (i.e. the above rules always hold)
3 x = torch.ones(5, 7, 3)
4 y = torch.ones(5, 7, 3)
5 z = (x+y)
6 print(z.shape)
```

torch.Size([5, 7, 3])

```
1 # Tensors x and y are not broadcastable,
2 # because x does not have at least 1 dimension
3 x = torch.ones((0,))
4 y = torch.ones(2,2)
5 z = (x+y)
6 print(z.shape)
```

-----  
**RuntimeError**

Traceback (most recent call last)

Cell In[26], line 5

3 x = torch.ones((0,))

4 y = torch.ones(2,2)

----> 5 z = (x+y)

6 print(z.shape)

**RuntimeError:** The size of tensor a (0) must match the size of tensor b (2) at non-singleton dimension 1

# A quick note on broadcasting

```
1 # You can line up trailing dimensions
2 # Tensors x and y are broadcastable.
3 # 1st trailing dimension: both have size 1
4 # 2nd trailing dimension: y has size 1, using size of x and broadcasting
5 # 3rd trailing dimension: x size is same as y size
6 # 4th trailing dimension: y dimension doesn't exist, using x only
7 x = torch.ones(5, 3, 4, 1)
8 y = torch.ones(3, 1, 1)
9 z = (x+y)
10 print(z.shape)
```

torch.Size([5, 3, 4, 1])

```
1 # However, x and y are not broadcastable,
2 # because of third trailing dimension (2 != 3).
3 x = torch.ones(5, 2, 4, 1)
4 y = torch.ones(3, 1, 1)
5 z = (x+y)
6 print(z.shape)
```

-----  
**RuntimeError**

Traceback (most recent call last)

Cell In[29], line 5

```
3 x = torch.ones(5, 2, 4, 1)
4 y = torch.ones(3, 1, 1)
----> 5 z = (x+y)
      6 print(z.shape)
```

**RuntimeError:** The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 1

# Tensor locations

**Recall:** PyTorch has **better support for GPU acceleration** than NumPy.

- If a GPU is available, significantly **speed up the training** of our neural network by performing the computations on the GPU using PyTorch.
- Especially useful for **training large and complex models**.

CUDA  
Graphs + PyTorch

The logo features the word "CUDA" in dark grey, "Graphs" in green, and "PyTorch" in dark grey. A small grey plus sign is positioned between "Graphs" and "PyTorch". Above the "Py" in "PyTorch" is the PyTorch logo, which is a red flame-like shape.

# Tensor locations

By default, all tensors are used by the CPU. If device enabled for GPU/CUDA computation, transfer the tensor to the GPU for faster computation.

This is done in three ways:

- Using **.to(device)** method will transfer to the best device available for computation (device variable defined earlier, when we checked for cuda/cpu).

```
1 # A tensor will by default be hosted on CPU
2 a = torch.ones(2, 3)
3 print(a)
4 print(a.device)
```

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

cpu

```
1 # Best option, use GPU/CUDA if available, else use CPU
2 b = torch.ones(2, 3).to(device)
3 print(b)
4 print(b.device)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

cuda:0



# Tensor locations

By default, all tensors are used by the CPU. If device enabled for GPU/CUDA computation, transfer the tensor to the GPU for faster computation.

This is done in three ways:

- Using **.to(device)** method will transfer to the best device available for computation (device variable defined earlier, when we checked for cuda/cpu).
- Using **.cpu()** or **.cuda()** will force transfer to the cpu or cuda respectively. Note that it might fail if you machine is not CUDA compatible.

```
1 # Force tensor to CPU
2 c = torch.ones(2, 3).cpu()
3 print(c)
4 print(c.device)
```

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

```
1 # Force tensor to GPU/CUDA
2 # (will fail if not CUDA compatible)
3 d = torch.ones(2, 3).cuda()
4 print(d)
5 print(d.device)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
cuda:0
```

# Tensor locations

By default, all tensors are used by the CPU. If device enabled for GPU/CUDA computation, transfer the tensor to the GPU for faster computation.

This is done in three ways:

- Using **.to(device)** method will transfer to the best device available for computation (device variable defined earlier, when we checked for cuda/cpu).
- Using **.cpu()** or **.cuda()** will force transfer to the cpu or cuda respectively. Note that it might fail if you machine is not CUDA compatible.

- In doubt, you can check the **device** attribute of your tensors to find where their computations will occur.

```
1 # Best option, use GPU/CUDA if available, else use CPU
2 b = torch.ones(2, 3).to(device)
3 print(b)
4 print(b.device)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
cuda:0
```

# Tensor locations

- **Keep in mind: two tensors with different devices cannot be used in the same operation! (same logic as dtypes).**

```
1 # Operations require tensors
2 # to be on same device
3 c = torch.ones(2, 3).cpu()
4 print(c)
5 print(c.device)
6 d = torch.ones(2, 3).cuda()
7 print(d)
8 print(d.device)
9 f = c + d
10 print(f)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
cpu
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
cuda:0
```

```
-----
RuntimeError                                Traceback (most recent call last)
Cell In[18], line 9
      7 print(d)
      8 print(d.device)
----> 9 f = c + d
```

**RuntimeError:** Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!

# A bit of practice for you to try at home

In order to practice your PyTorch Tensor skills, you may try to manually implement your own version of typical algorithms we ran on lists/Numpy arrays in previous classes, using the basic operations on PyTorch tensors.

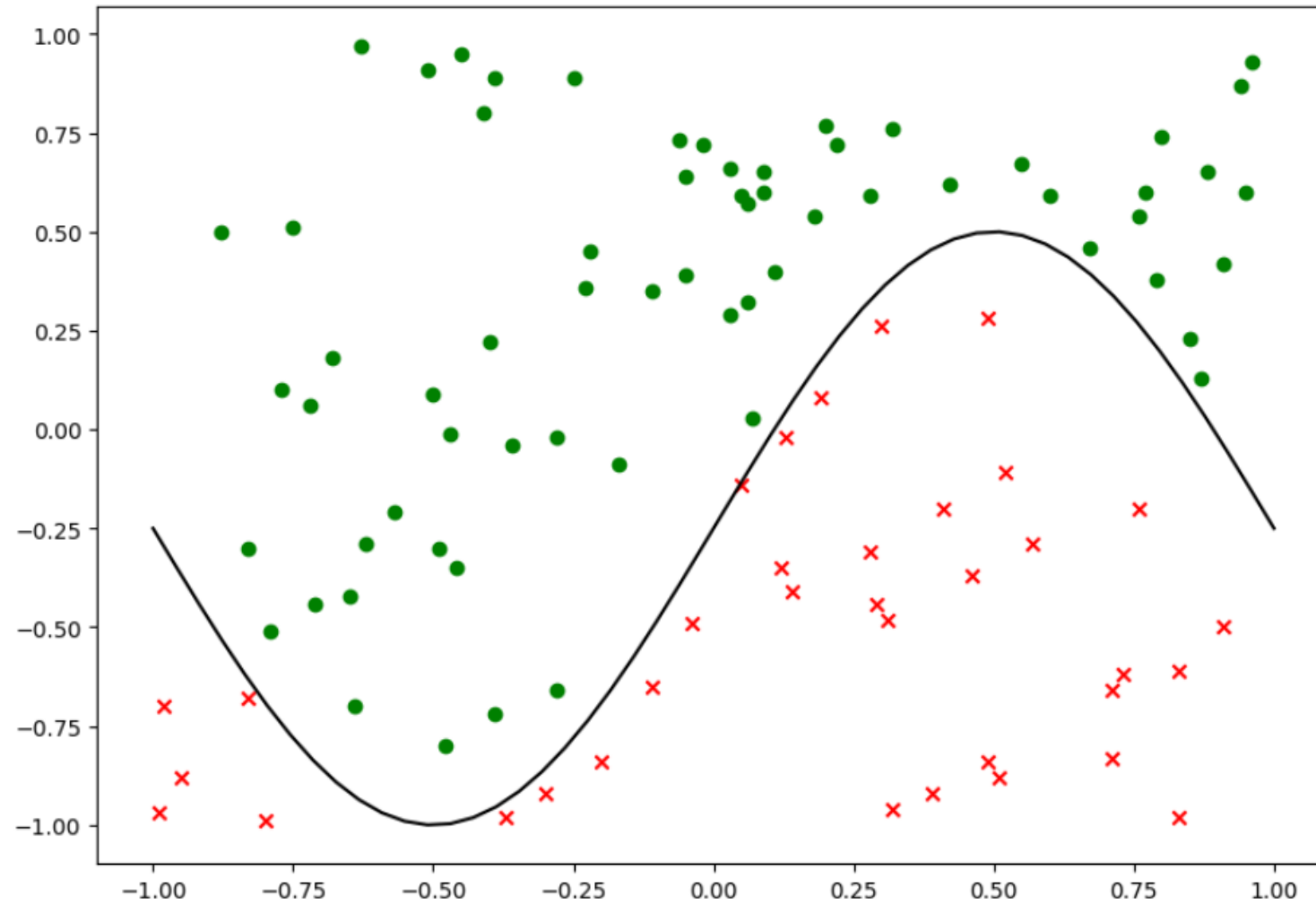
For instance, try writing algorithms:

- Finding the maximum, minimum, average, median values of a given 1D tensor,
- Transposing a given 2D tensor,
- Sorting a given 1D tensor (bubble sort, insertion sort, selection sort, quick sort, merge sort),
- Generating a 1D array containing the first K Fibonacci numbers with K given,
- Etc.

Later, you can check their performance times compared to their numpy/pytorch implementations when running them on both CPU and CUDA (if available).

In which scenarios is it slower to implement said functions and run them on GPU?

# Same dataset as before



# Our shallow neural network class

We will reuse our previous ShallowNeuralNet class from Week2 Notebook 7, which:

- implements a Shallow neural network using two fully connected layers and sigmoid activation functions,
- uses a Stochastic Mini-Batch gradient descent, with Adam as its optimizer,
- uses a random normal initialization,
- comes with a forward() method for predictions,
- comes with a backward() and train() method for backpropagation training,
- comes with a cross-entropy loss function and an accuracy calculating loss function,
- comes with a display function, to show training curves on both the loss and the accuracy,
- comes with save and load functions.

Let us start by replicating the init, forward, loss and accuracy methods in PyTorch.

# Writing a PyTorch neural network

The main differences between the original class and the PyTorch version are:

- The PyTorch version of the class **should inherit from torch.nn.Module** and call its parent's init method using `super()`.
- This is necessary because PyTorch uses classes inherited from `torch.nn.Module` to keep track of the layers and their **parameters** in a neural network.
- Instead of using NumPy arrays for the weights and biases, the PyTorch version uses **torch.nn.Parameter** objects, which is a **subclass of tensors, whose values can be trained and modified later**.

```

1  # Our class will inherit from the torch.nn.Module
2  # used to write all model in PyTorch
3  class ShallowNeuralNet_PT(torch.nn.Module):
4
5      def __init__(self, n_x, n_h, n_y, device):
6          # Super __init__ for inheritance
7          super().__init__()
8
9          # Network dimensions (as before)
10         self.n_x = n_x
11         self.n_h = n_h
12         self.n_y = n_y
13
14         # Device
15         self.device = device
16
17         # Initialize parameters using the torch.nn.Parameter type (a subclass of Tensors).
18         # We immediatly initialize the parameters using a random normal.
19         # The RNG is done using torch.randn instead of the NumPy RNG.
20         # We add a conversion into float64 (the same float type used by Numpy to generate our data)
21         # And send them to our GPU/CPU device
22         self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
23                                                  dtype = torch.float64, device = device)*0.1)
24         self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
25                                                  dtype = torch.float64, device = device)*0.1)
26         self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
27                                                  dtype = torch.float64, device = device)*0.1)
28         self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
29                                                  dtype = torch.float64, device = device)*0.1)
30
31         self.W1.retain_grad()
32         self.b1.retain_grad()
33         self.W2.retain_grad()
34         self.b2.retain_grad()

```



# Writing a PyTorch neural network

In PyTorch, **retain\_grad** is an attribute of tensors that **allows the gradients to be accumulated for a particular tensor, during backpropagation.**

Setting **retain\_grad** to **True** for a tensor will tell PyTorch to keep track of operations happening during the forward pass, later allowing to compute the gradients of intermediate tensors.

This will eventually serve to perform parameters updates via backpropagation.

**To keep it simple: Trainable parameters should allow for gradients to be retained for backpropagation.**

# Rewriting the forward pass

The forward pass can be rewritten. You can replace the NumPy operations with their PyTorch equivalents. Similarly, our activation function sigmoid is replaced with PyTorch's **torch.sigmoid** function.

```
def forward(self, inputs):  
    # Instead of using np.matmul(), we use its equivalent in PyTorch,  
    # which is torch.matmul()!  
    # (Most numpy matrix operations have their equivalent in torch, check it out!)  
    # Wx + b operation for the first layer  
    Z1 = torch.matmul(inputs, self.W1)  
    Z1_b = Z1 + self.b1  
    # Sigmoid is already implemented in PyTorch, feel free to reuse it!  
    A1 = torch.sigmoid(Z1_b)  
  
    # Wx + b operation for the second layer  
    # (Same as first layer)  
    Z2 = torch.matmul(A1, self.W2)  
    Z2_b = Z2 + self.b2  
    y_pred = torch.sigmoid(Z2_b)  
    return y_pred
```

# Rewriting the loss and accuracy functions

- Similarly, we can rewrite the **loss** and **accuracy** functions using torch functions in place of the Numpy ones, also vectorizing operations as much as possible.

```
def CE_loss(self, pred, outputs):  
    # We will use an epsilon to avoid NaNs on the log() values  
    eps = 1e-10  
    # As before with matmul, most operations in NumPy have their equivalent in torch (e.g. log and sum)  
    losses = outputs * torch.log(pred + eps) + (1 - outputs) * torch.log(1 - pred + eps)  
    loss = -torch.sum(losses)/outputs.shape[0]  
    return loss  
  
def accuracy(self, pred, outputs):  
    # Calculate accuracy for given inputs and outputs  
    # We will, again, rely as much as possible on the torch methods and functions.  
    return ((pred >= 0.5).int() == outputs).float().mean()
```

```

1 class ShallowNeuralNet_PT(torch.nn.Module):
2     def __init__(self, n_x, n_h, n_y, device):
3         super().__init__()
4         self.n_x = n_x
5         self.n_h = n_h
6         self.n_y = n_y
7         self.device = device
8         self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
9                                                 dtype = torch.float64, device = device)*0.1)
10        self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
11                                                dtype = torch.float64, device = device)*0.1)
12        self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
13                                                dtype = torch.float64, device = device)*0.1)
14        self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
15                                                dtype = torch.float64, device = device)*0.1)
16
17        self.W1.retain_grad()
18        self.b1.retain_grad()
19        self.W2.retain_grad()
20        self.b2.retain_grad()
21
22    def forward(self, inputs):
23        A1 = torch.sigmoid(torch.matmul(inputs, self.W1) + self.b1)
24        return torch.sigmoid(torch.matmul(A1, self.W2) + self.b2)
25
26    def CE_loss(self, pred, outputs):
27        eps = 1e-10
28        losses = outputs * torch.log(pred + eps) + (1 - outputs) * torch.log(1 - pred + eps)
29        return -torch.sum(losses)/outputs.shape[0]
30
31    def accuracy(self, pred, outputs):
32        return ((pred >= 0.5).int() == outputs).float().mean()

```

# Trying out our new neural network

Before using this Neural Network on our dataset, we need to convert them to PyTorch Tensor objects and send them to GPU (if available).

```
1 train_inputs_pt = torch.from_numpy(train_inputs).to(device)
2 train_outputs_pt = torch.from_numpy(train_outputs).to(device)
```

```
1 # Define a neural network structure
2 n_x = 2
3 n_h = 10
4 n_y = 1
5 np.random.seed(37)
6 shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y, device)
7 train_pred = shallow_neural_net_pt.forward(train_inputs_pt)
8 acc = shallow_neural_net_pt.accuracy(train_pred, train_outputs_pt)
9 loss = shallow_neural_net_pt.CE_loss(train_pred, train_outputs_pt)
10 print(train_pred.shape)
11 print(train_outputs_pt.shape)
12 print(acc, loss)
13 print(acc.item(), loss.item())
```

```
torch.Size([1000, 1])
```

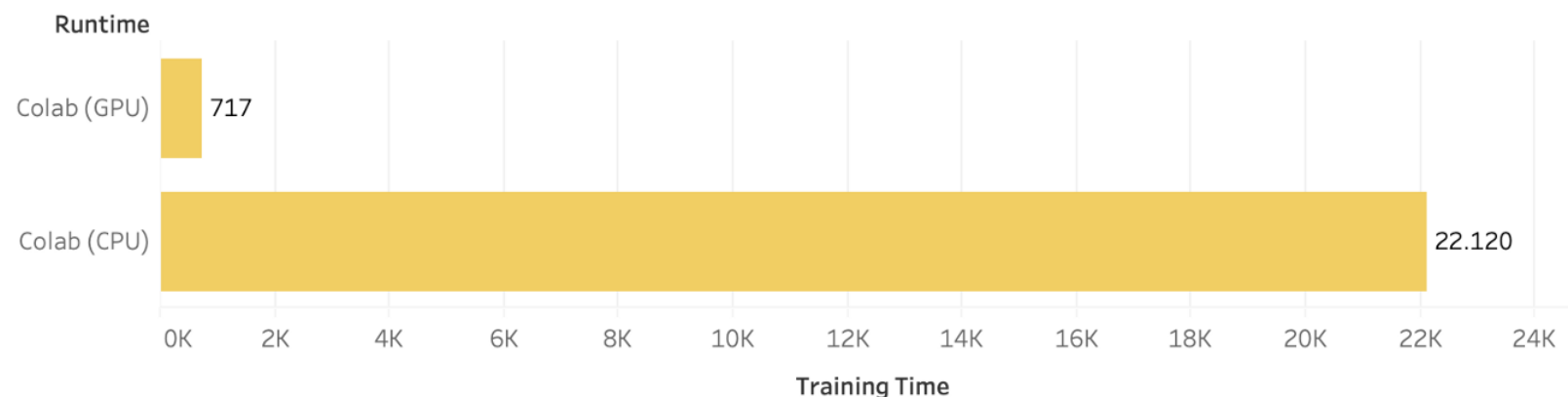
```
torch.Size([1000, 1])
```

```
tensor(0.6260, device='cuda:0') tensor(0.6881, device='cuda:0', dtype=torch.float64, grad_fn=<DivBackward0>)
0.6260000467300415 0.688087761172729
```

# Computation times comparison

- Let us run both models (the NumPy one and the PyTorch one) and ask them to perform 1000 times the accuracy computation.
- On my machine (CUDA, Nvidia GTX 1060), the PyTorch model is **roughly 30 times faster!**
- Obviously depends on your machine (could achieve even more).

CPU vs GPU Training Time Comparison in Seconds



# Computation times comparison

```
1  # Calculate computation times (NumPy NN)
2  start = time()
3  for i in range(1000):
4      train_acc = shallow_neural_net.accuracy(train_inputs, train_outputs)
5  end = time()
6  time_np = end - start
7  print(time_np)
8
9  # Calculate computation times (PyTorch NN)
10 start = time()
11 for i in range(1000):
12     train_acc_pt = shallow_neural_net_pt.accuracy(train_inputs_pt, train_outputs_pt)
13 end = time()
14 time_pt = end - start
15 print(time_pt)
16
17 # Ratio
18 ratio = time_np/time_pt
19 print(ratio)
```

2.0732526779174805

0.07289958000183105

28.43984393141093