# Practice of Deep Learning

## Day 2, Part 2/4

Matthieu De Mari

# About this lecture

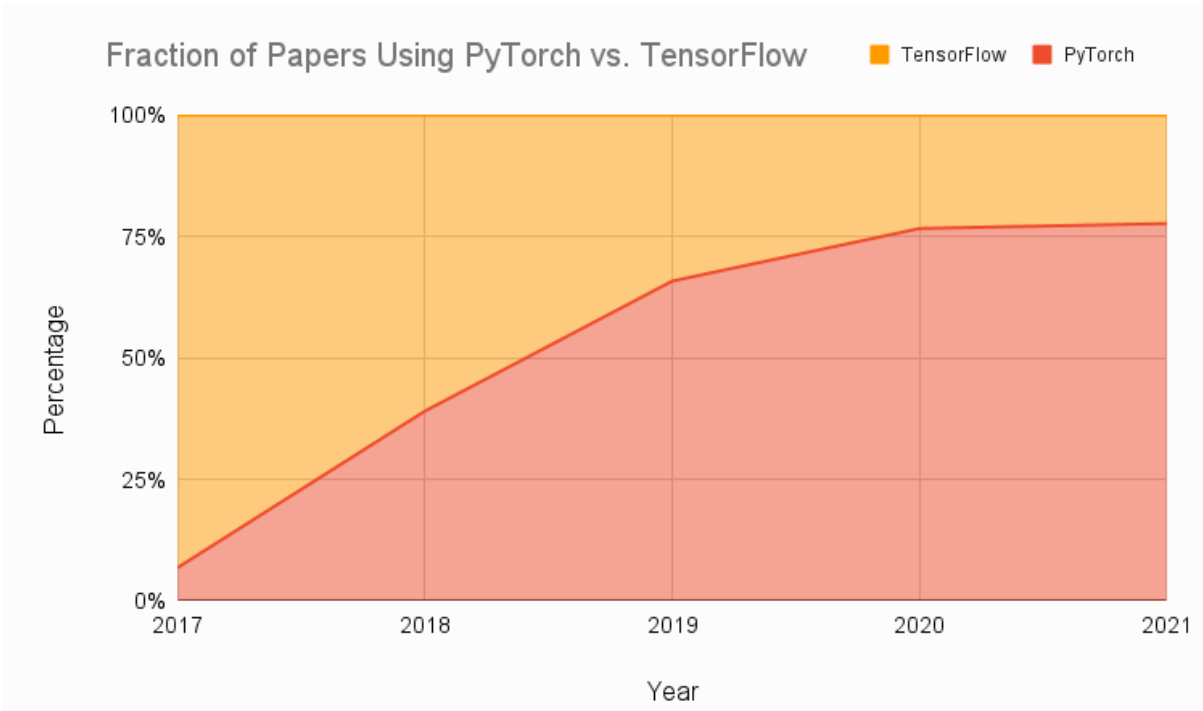
1. What is the **PyTorch library** and its **benefits**?
2. What is a **PyTorch tensor object** and its typical **attributes**?
3. How to implement some typical **tensor operations**?
4. What are **tensor locations** in terms of computation?
5. How to **transform our original NumPy shallow Neural Network**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

# About this lecture

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

# 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.

# On the benefits of using PyTorch

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

- 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**.
  - (More on this on Week 13!)
- **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 code 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 (see W1S1 bonus).
- 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 very 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.

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

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

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

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

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

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

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

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.

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

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 $[\mathbf{0}, \mathbf{1}]$ **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 $[\mathbf{0}, \mathbf{1}]$ **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])
```

- Not planning to cover all of these attributes and methods, because…

# 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__', '__co
mplex__', '__contains__', '__deepcopy__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__div__', '__dlpack__', '__dlpa
ck_device__', '__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__g
t__', '__hash__', '__iadd__', '__iand__', '__idiv__', '__ifloordiv__', '__ilshift__', '__imod__', '__imul__', '__index__', '__i
nit__', '__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_vie
w', '_is_zerotensor', '_make_subclass', '_make_wrapper_subclass', '_neg_view', '_nested_tensor_layer_norm', '_nnz', '_python_di
spatch', '_reduce_ex_internal', '_storage', '_to_dense', '_update_names', '_values', '_version', 'abs', 'abs_', 'absolute', 'ab
solute_', 'acos', 'acos_', 'acosh', 'acosh_', 'add', 'add_', 'addbmm', 'addbmm_', 'addcdiv', 'addcdiv_', 'addcmul', 'addcmul_',
```

# Basic operations on Tensors

All the typical element-wise operations on arrays 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;

```
# Create a 3D tensor, of size 3 by 2 by 2, filled with random values
# drawn from a uniform [0, 1] distribution.
torch.manual_seed(17)
rand_unif_tensor = torch.rand(size = (3, 2, 2))
print(rand_unif_tensor)

# Indexing
element1 = rand_unif_tensor[2]
print("Element1: ", element1)
element2 = rand_unif_tensor[2][0]
print("Element2: ", element2)
element3 = rand_unif_tensor[2, 0, 1]
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 on arrays 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;
- **update elements** of a tensor using the square bracket notation;

```
# Create a 3D tensor, of size 3 by 2 by 2, filled with random values
# drawn from a uniform [0, 1] distribution.
torch.manual_seed(17)
rand_unif_tensor = torch.rand(size = (3, 2, 2))
print(rand_unif_tensor)

# Before
element4 = rand_unif_tensor[2, 1, 1]
print("Element4: ", element4)
# Updating
rand_unif_tensor[2, 1, 1] = 0.5
# After
element4 = rand_unif_tensor[2, 1, 1]
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 on arrays 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;
- **update elements** of a tensor using the square bracket notation;
- **browse through elements of a tensor** using a **for loop**.

```python
# Create a 3D tensor, of size 3 by 2 by 2, filled with random values
# drawn from a uniform [0, 1] distribution.
torch.manual_seed(17)
rand_unif_tensor = torch.rand(size = (3, 2, 2))
print(rand_unif_tensor)

# Browsing
for sub_tensor in rand_unif_tensor:
    print("---")
    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 substraction
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,

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

```
# Element-wise multiplication
c = a * b
print(c)
c = torch.mul(a,b)
print(c)
```

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

```
# Element-wise division
c = a / b
print(c)
c = torch.div(a, b)
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,
- Along with more Linear Algebra stuff (now is the time to revise!).

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

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

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

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

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

# Dot operation, used for computing
# the dot product of two 1D tensors.
f = torch.dot(a, b)
print(f)
g = torch.matmul(a, b.T)
print(g)
```

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

# Tensor locations

**<u>Recall:</u>** 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**.

# 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 was 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 was 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 your 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 was 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 your 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

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

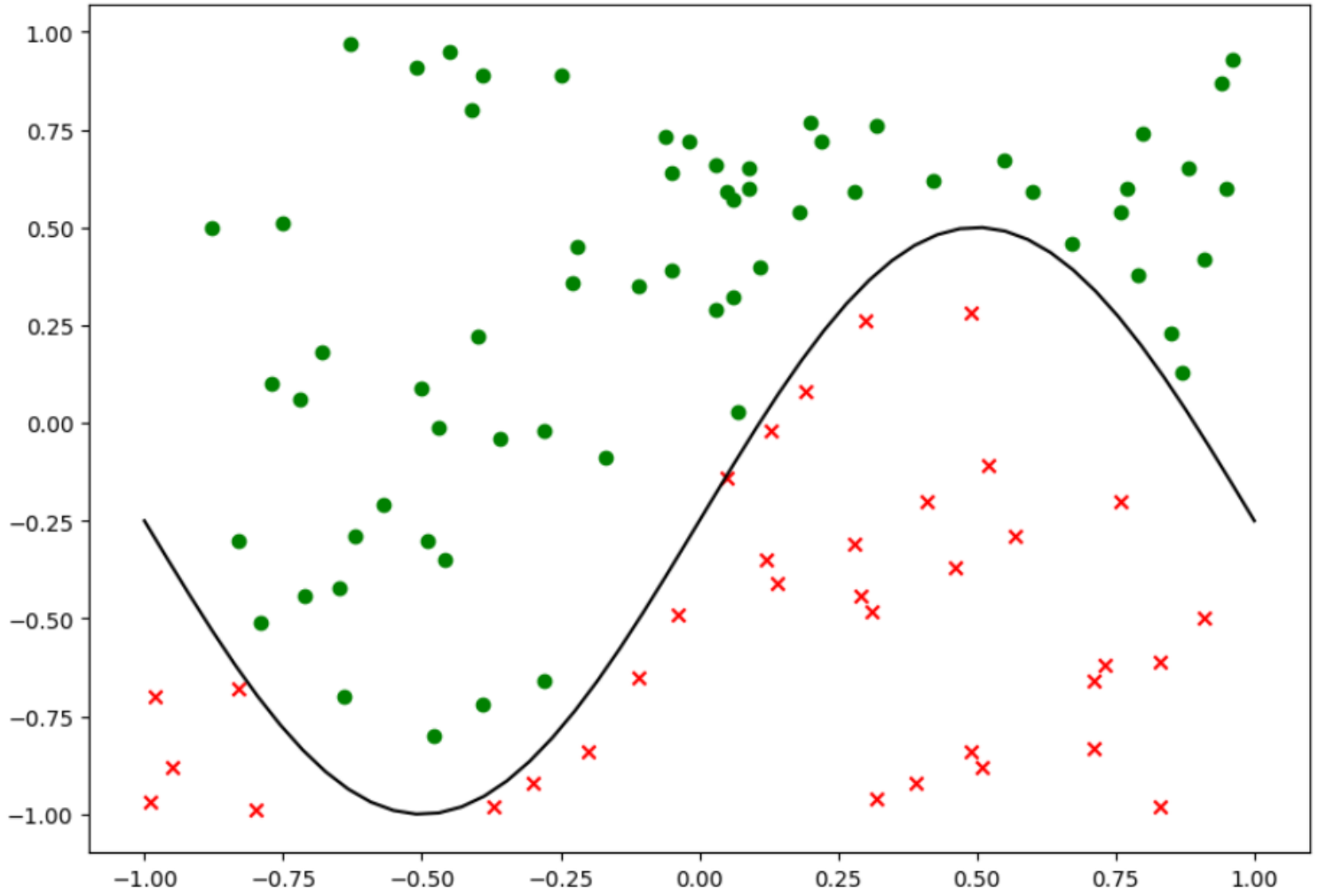
```
1  # Operations require tensors
2  # to be on same device
3  c = torch.ones(2, 3).cpu()
4  print(c)
5  print(c.device)
6  d = torch.ones(2, 3).cuda()
7  print(d)
8  print(d.device)
9  f = c + d
10 print(f)
```

```
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!
```

# Same dataset as before



# Our shallow neural network class

We will reuse our previous **ShallowNeuralNet** class from earlier, 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 metric function**,
- comes with a **display** function, to show **training curves** on both the loss and the accuracy,

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**.

```python
# Our class will inherit from the torch.nn.Module
# used to write all model in PyTorch
class ShallowNeuralNet_PT(torch.nn.Module):

    def __init__(self, n_x, n_h, n_y, device):
        # Super __init__ for inheritance
        super().__init__()

        # Network dimensions (as before)
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y

        # Device
        self.device = device

        # Initialize parameters using the torch.nn.Parameter type (a subclass of Tensors).
        # We immediatly initialize the parameters using a random normal.
        # The RNG is done using torch.randn instead of the NumPy RNG.
        # We add a conversion into float64 (the same float type used by Numpy to generate our data)
        # And send them to our GPU/CPU device
        self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.W1.retain_grad()
        self.b1.retain_grad()
        self.W2.retain_grad()
        self.b2.retain_grad()
```

**Explicitly indicate that the Wx and bx matrices are now tensors, playing the roles of parameters in a Neural Network of some sort.**

# Writing a PyTorch neural network

In PyTorch, **requires_grad** is an attribute of tensors that **allows the gradients to be calculated for a particular tensor**, **during forward operations and 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 these 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 most 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 ahve 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 fre 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 ouputs
    # We will, again, rely as much as possible on the torch methods and functions.
    return ((pred >= 0.5).int() == outputs).float().mean()
```

```python
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y
        self.device = device
        self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
                                    dtype = torch.float64, device = device)*0.1)
        self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
                                    dtype = torch.float64, device = device)*0.1)
        self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
                                    dtype = torch.float64, device = device)*0.1)
        self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
                                    dtype = torch.float64, device = device)*0.1)
        self.W1.retain_grad()
        self.b1.retain_grad()
        self.W2.retain_grad()
        self.b2.retain_grad()

    def forward(self, inputs):
        A1 = torch.sigmoid(torch.matmul(inputs, self.W1) + self.b1)
        return torch.sigmoid(torch.matmul(A1, self.W2) + self.b2)

    def CE_loss(self, pred, outputs):
        eps = 1e-10
        losses = outputs * torch.log(pred + eps) + (1 - outputs) * torch.log(1 - pred + eps)
        return -torch.sum(losses)/outputs.shape[0]

    def accuracy(self, pred, outputs):
        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 our samples to PyTorch Tensor objects and send them to GPU (if available).

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

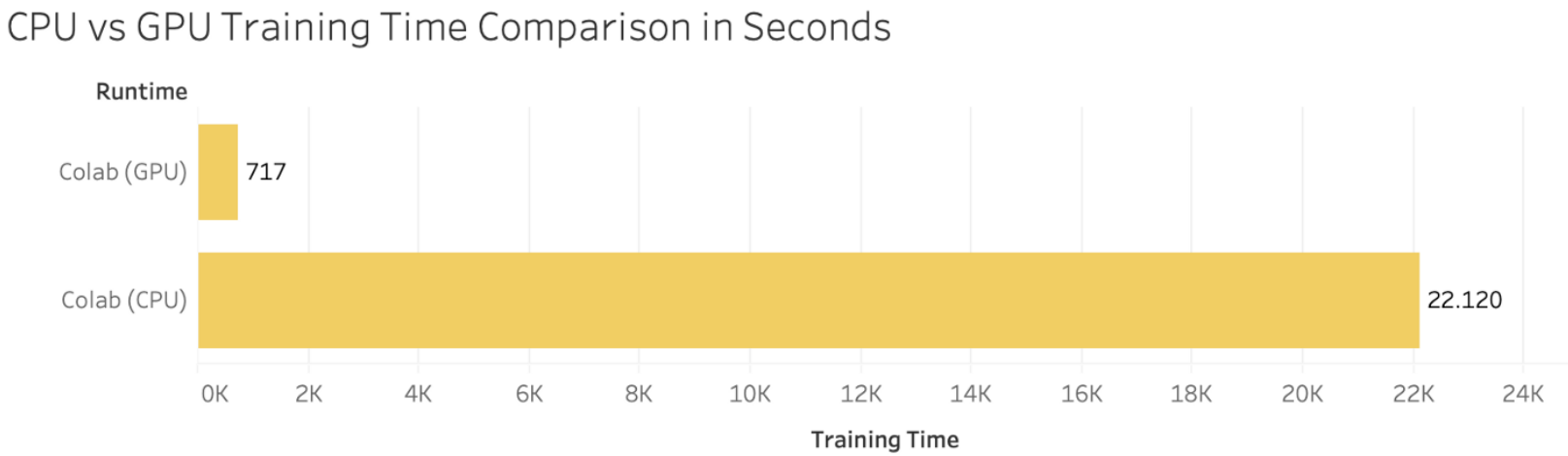
```
# Define a neural network structure
n_x = 2
n_h = 10
n_y = 1
np.random.seed(37)
shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y, device)
train_pred = shallow_neural_net_pt.forward(train_inputs_pt)
acc = shallow_neural_net_pt.accuracy(train_pred, train_outputs_pt)
loss = shallow_neural_net_pt.CE_loss(train_pred, train_outputs_pt)
print(train_pred.shape)
print(train_outputs_pt.shape)
print(acc, loss)
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 4070), the PyTorch model is **roughly 30 times faster**!
- *(Obviously depends on your machine, could be even more on some specific DL GPUs).*



# Computation times comparison

```
# Calculate computation times (NumPy NN)
start = time()
for i in range(1000):
    train_acc = shallow_neural_net.accuracy(train_inputs, train_outputs)
end = time()
time_np = end - start
print(time_np)

# Calculate computation times (PyTorch NN)
start = time()
for i in range(1000):
    train_acc_pt = shallow_neural_net_pt.accuracy(train_inputs_pt, train_outputs_pt)
end = time()
time_pt = end - start
print(time_pt)

# Ratio
ratio = time_np/time_pt
print(ratio)
```

```
2.0732526779174805
0.07289958000183105
28.43984393141093
```

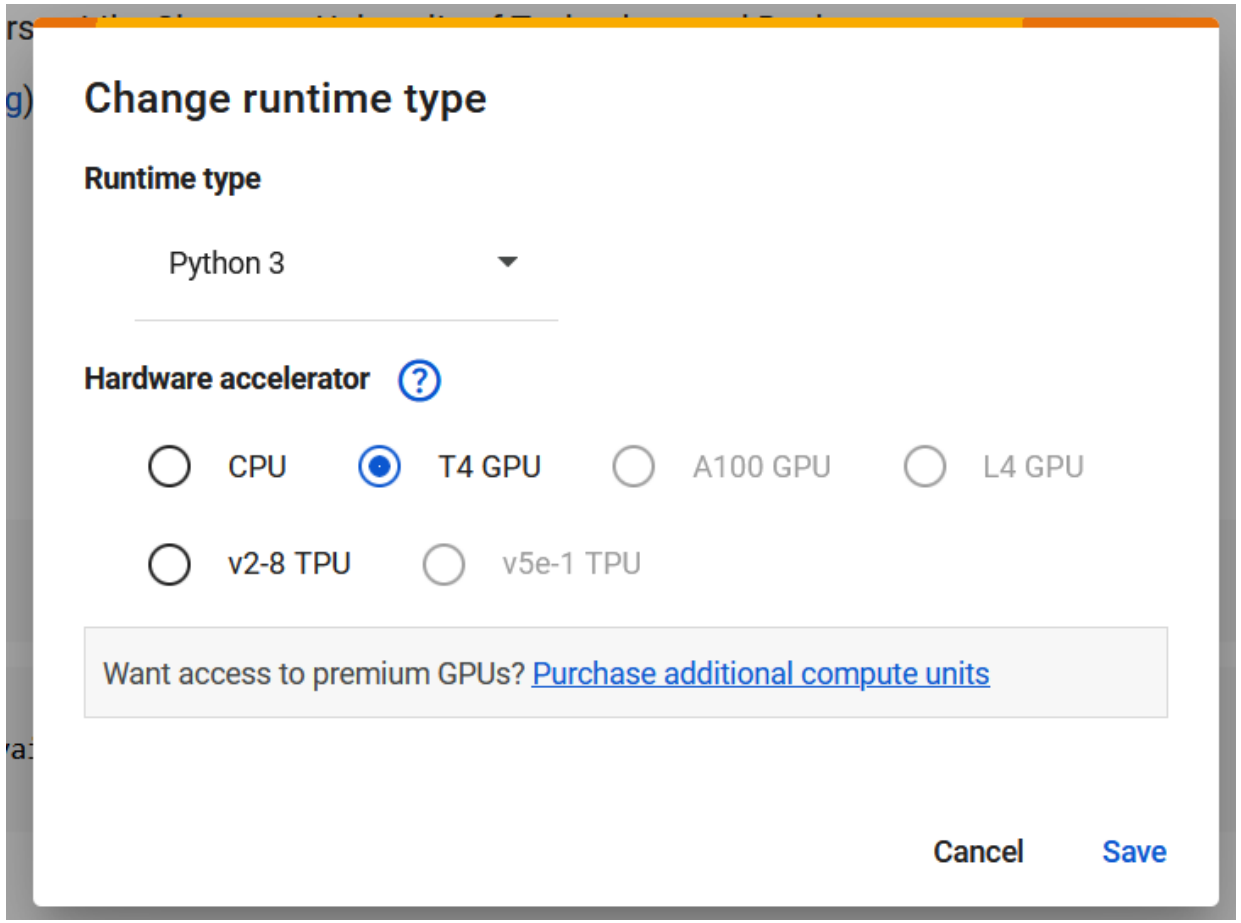
# A note for those of you running on MacOS

Use Google Colab but remember to select GPUs are runtime hardware (by default CPU).

***Also the option of using MPS (but you have to figure it out!)***

***Careful, as of Feb 2025, float64 is not a valid datatype in MPS, might need to use float32 in notebooks instead.***

***Possibly more adjustments needed…***



# Autograd, the PyTorch beast

Another feature that makes PyTorch extremely powerful is the **autograd**, a **computational graph able to provide automatic differentiation for all operations made on Tensors**.

- To demonstrate, let us consider the function $f(x) = (x-2)^2$.
- Let us compute $\frac{\partial f}{dx}$ and then compute $f'(1)$.
- The **automatic differentiation/gradient computation** is done by PyTorch, for any operation involving a tensor, that has been tracked by setting its attribute **requires_grad** to **True**.
- Any operation can then be tracked back by using the **backward()** method on the resulting variable (here, $y = f(1)$).

# Autograd in action

Define our function and its derivative (for reference).

- Create tensor x, containing a single scalar with value 1.
- Requires/Retain grad is set to True for tensor x.
- Compute $y = f(x = 1)$.
- Use backward() to ask PyTorch to compute $\frac{\partial y}{\partial x} = f'(x = 1)$.
- Derivative value stored in **x.grad**.

```python
# Our function f
def f(x):
    return (x-2)**2

# Its differentiation
def fp(x):
    return 2*(x-2)
```

```python
# Let us create a tensor with  single value, 1.0
# We make its attribute requires_grad = True.
# This allows to track all operations on the tensor.
x = torch.tensor([1.0], requires_grad = True)

# Compute the value of f(1)
y = f(x)

# Backward methods will compute the gradient of the operations using x
# Here, this computes the gradient of given tensors y with respect to x.
y.backward()

# Compare the theoretical value of fp(1)...
print('Theoretical f\'(x):', fp(x))
# ... to the one calculated by PyTorch!
print('PyTorch\'s Value of f\'(x):', x.grad)
```

```
Theoretical f'(x): tensor([-2.], grad_fn=<MulBackward0>)
PyTorch's Value of f'(x): tensor([-2.])
```

# Autograd in action

**Whenever backward() is used:**

- Autograd will look at the computational graph,
- And it will compute the partial derivatives of the variable $y$ on which backward() is applied,
- With respect to all the tensors $x$ whose requires_grad parameter was set to True,
- And store these derivatives wrt. to each tensor $x$, in said tensor $x$, under the grad attribute.

```python
# Our function f
def f(x):
    return (x-2)**2

# Its differentiation
def fp(x):
    return 2*(x-2)
```

```python
# Let us create a tensor with  single value, 1.0
# We make its attribute requires_grad = True.
# This allows to track all operations on the tensor.
x = torch.tensor([1.0], requires_grad = True)

# Compute the value of f(1)
y = f(x)

# Backward methods will compute the gradient of the operations using x
# Here, this computes the gradient of given tensors y with respect to x.
y.backward()

# Compare the theoretical value of fp(1)...
print('Theoretical f\'(x):', fp(x))
# ... to the one calculated by PyTorch!
print('PyTorch\'s Value of f\'(x):', x.grad)
```

```
Theoretical f'(x): tensor([-2.], grad_fn=<MulBackward0>)
PyTorch's Value of f'(x): tensor([-2.])
```

# Autograd, in short

The **autograd** is one of the most powerful features of the PyTorch framework, and why we like it so much for training Neural Networks.

- **Thanks to autograd, we do not need to manually calculate the gradients for any of our future gradient descent rules ever again!**
- The counterpart however is that all operation must involve tensors and must use PyTorch functions and methods... (but that is ok).
- You do not need to know how it implements the computation of all gradients in the background, but if you are curious, have a look at: https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/.

# Implementing gradient descent with autograd

Now that we have an automated way to compute gradients, we can use gradient descent to find the (local) minima of any differentiable function. To demonstrate, we use $f(x) = (x-2)^2$.

We find the minimum by using the **gradient descent algorithm**:

- **Define a tensor with retained gradients and starting value $x_0$, learning rate $\alpha$,** and set iteration number $t$ to 0.
- **Forward pass:** compute $f(x_t)$
- **Backward pass:** Using **autograd,** compute $x_{t+1} = x_t - \alpha\, f'(x_t)$.
- Increment $t$ by 1, and **repeat forward-backward passes** for a given number of iterations (or until convergence is observed).

# Implementing gradient descent with autograd

**Gradient descent algorithm steps:**

- **Define a tensor with retained gradients and starting value $x_0$, learning rate $\alpha$,** and set iteration number $t$ to 0.
- **Forward pass:** compute $f(x_t)$
- **Backward pass:** Using **autograd,** compute $x_{t+1} = x_t - \alpha f'(x_t)$.
- Increment $t$ by 1 and **repeat forward-backward passes** for a given number of iterations (or until convergence is observed).

```python
# Initialize x to the value 5 (i.e. x_0 = 5).
# It will be a 1D tensor with a single value again.
x = torch.tensor([5.0], requires_grad = True)

# Set alpha to 0.25
alpha = 0.25

# Repeat iterations of gradient descent over 15 iterations
for i in range(20):
    # Forward pass
    y = f(x)
    # Backward pass
    y.backward()
    # Note x.data consists of all the data in the tensor.
    # Here, this is equivalent to x[0].
    x.data = x.data - alpha*x.grad
    # This basically resets the gradients in x,
    # now that they have been used so that the next forward pass
    # is not getting the gradients mixed with the previous one.
    x.grad.zero_()
print("Local minima found at:", x.item())
```

```
Local minima found at: 2.000002861022949
```

# Rewriting our backpropagation

Let us now steal this autograd and use it to make our backward method simpler! This will be great because:

- There is no need for a **backward()** method as before to compute gradients and gradient descent update rules (built-in method for nn.Module classes, covered automatically by PyTorch autograd, yay!).
- Our **train()** method will now simply consist of several iterations of
  - **forward()** pass and loss calculation,
  - gradient computation with the **backward()** method, used on loss,
  - and **gradient descent updates** on trainable parameters.

# Backpropagation

Our **train()** method will be rewritten and now consists of several iterations of

- **forward()** pass and loss calculation,
- gradient computation with the **backward()** method, used on loss,
- and **gradient descent updates** on trainable parameters.

```
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # History of losses
    self.loss_history = []
    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss = self.CE_loss(pred, outputs)
        self.loss_history.append(loss.item())

        # Backpropagate
        # Compute differentiation of loss with respect to all
        # parameters involved in the calculation that have a flag
        # requires_grad = True (that is W2, W1, b2 and b1)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

# Backpropagation

**Important note:**

Using **with torch.no_grad():** tells PyTorch that the operations should not be tracked by the autograd.

Indeed, when performing gradient update for a trainable parameter, we do not want to update gradients for other trainable parameters!

```
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # History of losses
    self.loss_history = []
    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss = self.CE_loss(pred, outputs)
        self.loss_history.append(loss.item())

        # Backpropagate
        # Compute differentiation of loss with respect to all
        # parameters involved in the calculation that have a flag
        # requires_grad = True (that is W2, W1, b2 and b1)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

# Backpropagation

**Important note #2:**

Also, do not forget to **reset your gradients to zero** before the next iteration!

Otherwise, they will accumulate (and we do not want that!)

This can be simply done with **grad.zero_().**

```
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # History of losses
    self.loss_history = []
    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss = self.CE_loss(pred, outputs)
        self.loss_history.append(loss.item())

        # Backpropagate
        # Compute differentiation of loss with respect to all
        # parameters involved in the calculation that have a flag
        # requires_grad = True (that is W2, W1, b2 and b1)
        loss.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            print("Iteration {} - Loss = {}".format(iteration_number, loss.item()))
```

# Trying our trainer function… Works!

```
# Define a neural network structure
n_x = 2
n_h = 10
n_y = 1
np.random.seed(27)
shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y).to(device)
train_pred = shallow_neural_net_pt.train(train_inputs_pt, train_outputs_pt, N_max = 1500, alpha = 5)
```

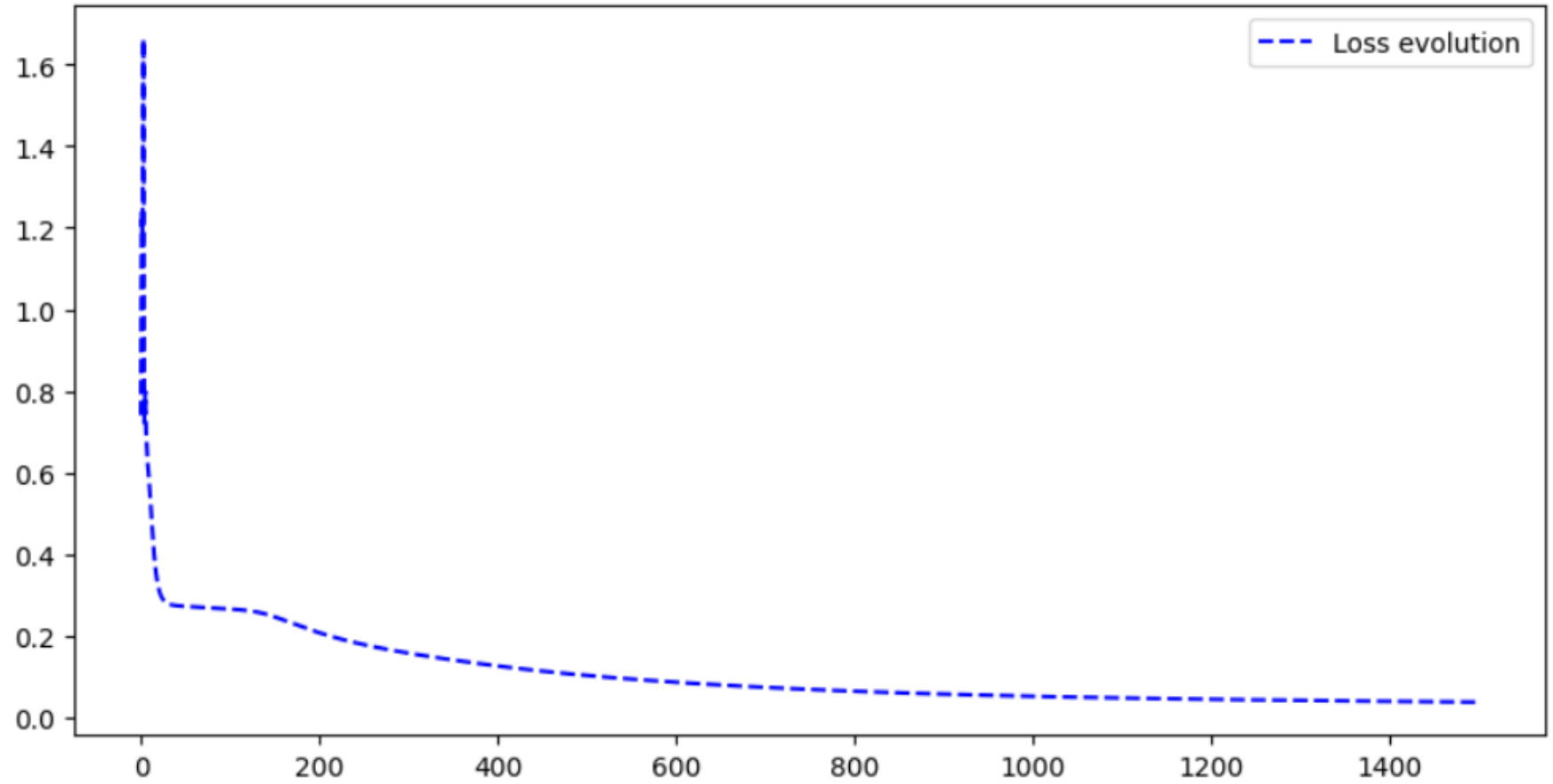
```
Iteration 1 - Loss = 0.740706584204259
Iteration 76 - Loss = 0.26962259292159146
Iteration 151 - Loss = 0.24679984335478158
Iteration 226 - Loss = 0.19185312002269
Iteration 301 - Loss = 0.15782585723135667
Iteration 376 - Loss = 0.13340462157889243
Iteration 451 - Loss = 0.11368678438788624
Iteration 526 - Loss = 0.09846243310305443
Iteration 601 - Loss = 0.08644133092348712
Iteration 676 - Loss = 0.07667983004400601
Iteration 751 - Loss = 0.06878116008852213
Iteration 826 - Loss = 0.062435676044891
Iteration 901 - Loss = 0.05733159760509488
Iteration 976 - Loss = 0.05319045574961357
Iteration 1051 - Loss = 0.049788446489380474
Iteration 1126 - Loss = 0.04695522876594684
Iteration 1201 - Loss = 0.04456387433429782
Iteration 1276 - Loss = 0.04252011654915635
Iteration 1351 - Loss = 0.040753530538752275
Iteration 1426 - Loss = 0.03921094018947538
```

```
# Check accuracy after training
acc = shallow_neural_net_pt.accuracy(shallow_neural_net_pt(train_inputs_pt), train_outputs_pt).item()
print(acc)
```

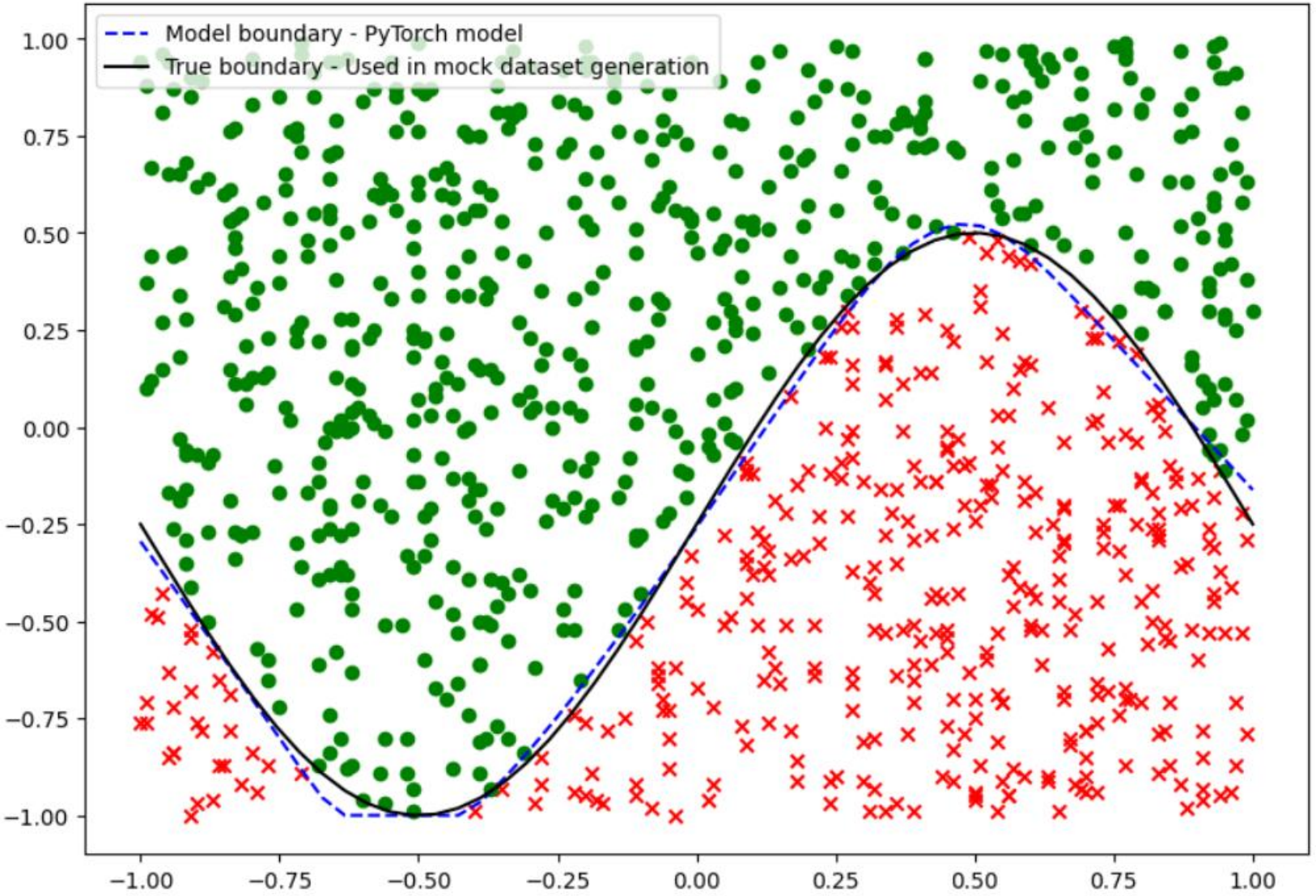
```
0.9910000562667847
```

# Trying our trainer function… Works!

```
plt.figure(figsize = (10, 5))
plt.plot(list(range(len(shallow_neural_net_pt.loss_history))), \
         shallow_neural_net_pt.loss_history, "b--", label = "Loss evolution")
plt.legend(loc = "best")
plt.show()
```



# Trying our trainer function… Works!



# Using PyTorch losses and metrics

In order to make our model even simpler and more efficient, we will use the loss functions and evaluation functions from PyTorch.

Our **CE_loss()** and **accuracy()** methods will therefore be replaced with the built-in **nn.BCELoss()** and the **BinaryAccuracy()** functions from the **PyTorch** and **TorchMetrics** libraries.

Feel free to have a look at the loss functions available in **PyTorch**, here: https://pytorch.org/docs/stable/nn.html#loss-functions.

**Torchmetrics** is a supporting library, which provides a few additional losses and metrics functions, ready to use with **PyTorch**: https://torchmetrics.readthedocs.io/en/stable/all-metrics.html.

```python
class ShallowNeuralNet_PT(torch.nn.Module):

    def __init__(self, n_x, n_h, n_y):
        # Super __init__ for inheritance
        super().__init__()

        # Network dimensions (as before)
        self.n_x = n_x
        self.n_h = n_h
        self.n_y = n_y

        # Initialize parameters using the torch.nn.Parameter type (a subclass of Tensors).
        # We immediatly initialize the parameters using a random normal.
        # The RNG is done using torch.randn instead of the NumPy RNG.
        # We add a conversion into float64 (the same float type used by Numpy to generate our data)
        # And send them to our GPU/CPU device
        self.W1 = torch.nn.Parameter(torch.randn(n_x, n_h, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.b1 = torch.nn.Parameter(torch.randn(1, n_h, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.W2 = torch.nn.Parameter(torch.randn(n_h, n_y, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.b2 = torch.nn.Parameter(torch.randn(1, n_y, requires_grad = True, \
                                     dtype = torch.float64, device = device)*0.1)
        self.W1.retain_grad()
        self.b1.retain_grad()
        self.W2.retain_grad()
        self.b2.retain_grad()

        # Loss and accuracy functions
        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()
```

```
def train(self, inputs, outputs, N_max = 1000, alpha = 1):
    # History of losses
    self.loss_history = []
    # Repeat gradient descent procedure for N_max iterations
    for iteration_number in range(1, N_max + 1):
        # Forward pass
        # This is equivalent to pred = self.forward(inputs)
        pred = self(inputs)
        # Compute loss
        loss_val = self.loss(pred, outputs.to(torch.float64))
        self.loss_history.append(loss_val.item())

        # Backpropagate
        # Compute differentiation of loss with respect to all
        # parameters involved in the calculation that have a flag
        # requires_grad = True (that is W2, W1, b2 and b1)
        loss_val.backward()

        # Update all weights
        # Note that this operation should not be tracked for gradients,
        # hence the torch.no_grad()!
        with torch.no_grad():
            self.W1 -= alpha*self.W1.grad
            self.W2 -= alpha*self.W2.grad
            self.b1 -= alpha*self.b1.grad
            self.b2 -= alpha*self.b2.grad

        # Reset gradients to 0
        self.W1.grad.zero_()
        self.W2.grad.zero_()
        self.b1.grad.zero_()
        self.W2.grad.zero_()

        # Display
        if(iteration_number % (N_max//20) == 1):
            # Compute accuracy for display
            acc_val = self.accuracy(pred, outputs)
            print("Iteration {} - Loss = {} - Accuracy = {}".format(iteration_number, \
                                                                    loss_val.item(), \
                                                                    acc_val.item()))
```

```python
# Define a neural network structure
n_x = 2
n_h = 10
n_y = 1
np.random.seed(37)
shallow_neural_net_pt = ShallowNeuralNet_PT(n_x, n_h, n_y).to(device)
train_pred = shallow_neural_net_pt.train(train_inputs_pt, train_outputs_pt, N_max = 1001, alpha = 5)
```

```
Iteration 1 - Loss = 0.7197759687513233 - Accuracy = 0.37400001287460327
Iteration 51 - Loss = 0.27012799843420826 - Accuracy = 0.8740000128746033
Iteration 101 - Loss = 0.26844662460163987 - Accuracy = 0.8759999871253967
Iteration 151 - Loss = 0.2514640191969197 - Accuracy = 0.878000020980835
Iteration 201 - Loss = 0.20506982178051603 - Accuracy = 0.9070000052452087
Iteration 251 - Loss = 0.17680756703390124 - Accuracy = 0.9160000085830688
Iteration 301 - Loss = 0.1579480030878312 - Accuracy = 0.925000011920929
Iteration 351 - Loss = 0.14387554010094675 - Accuracy = 0.9359999895095825
Iteration 401 - Loss = 0.13283459121849153 - Accuracy = 0.9440000057220459
Iteration 451 - Loss = 0.12385125501615053 - Accuracy = 0.949999988079071
Iteration 501 - Loss = 0.11634485891740215 - Accuracy = 0.9580000042915344
Iteration 551 - Loss = 0.10992562465020217 - Accuracy = 0.9620000123977661
Iteration 601 - Loss = 0.10428099036393183 - Accuracy = 0.968999981880188
Iteration 651 - Loss = 0.09912382760500893 - Accuracy = 0.9700000286102295
Iteration 701 - Loss = 0.09418616337556611 - Accuracy = 0.9729999899864197
Iteration 751 - Loss = 0.08925529750401755 - Accuracy = 0.9739999771118164
Iteration 801 - Loss = 0.084245324853110009 - Accuracy = 0.9760000109672546
Iteration 851 - Loss = 0.0792171201563788 - Accuracy = 0.9760000109672546
Iteration 901 - Loss = 0.07430772516730857 - Accuracy = 0.9760000109672546
Iteration 951 - Loss = 0.06965367567190604 - Accuracy = 0.9789999723434448
Iteration 1001 - Loss = 0.0653535992262251 - Accuracy = 0.9800000190734863
```

```python
# Check accuracy after training
acc = shallow_neural_net_pt.accuracy(shallow_neural_net_pt(train_inputs_pt), train_outputs_pt).item()
print(acc)
```

```
0.9800000190734863
```

# Last but not least, using layers prototypes

A simpler way would be to

- Define the trainable parameters,
- And the operation $WX + b$ to use in a given layer,

All at once, by using the **Linear()** layer available in PyTorch!

- We simply use the Linear() layer and call it as a function in the forward method!
- And remove the Parameter objects in our __init__, as they will now be attributes of the Linear() objects.

```
class ShallowNeuralNet_PT(torch.nn.Module):
    def __init__(self, n_x, n_h, n_y, device):
        super().__init__()
        self.n_x, self.n_h, self.n_y = n_x, n_h, n_y

        # Using the Linear() layer prototype
        self.linear1 = nn.Linear(n_x, n_h, dtype = torch.double)
        self.linear2 = nn.Linear(n_h, n_y, dtype = torch.double)

        self.loss = torch.nn.BCELoss()
        self.accuracy = BinaryAccuracy()

    def forward(self, inputs):
        # Reusing the layers as functions in the forward method
        Z1 = self.linear1(inputs)
        A1 = torch.sigmoid(Z1)
        Z2 = self.linear2(A1)
        A2 = torch.sigmoid(Z2)
        return A2
```

# To summarize

We now have a full Neural Network class, written in PyTorch, with:

- 2 linear layers, sigmoid activation functions,
- Xavier uniform initialization on trainable parameters,
- Forward pass method,
- Autograd backpropagation and trainer method,
- Cross-entropy loss and accuracies,
- Linear layer prototypes

And it runs/trains at the speed of light (almost…) on GPU!

# Let us call it a break for now

We will continue on the next lecture with more PyTorch concepts