50.039 Theory and Practice of Deep Learning

W3-S1 Introduction to Deep Learning using the PyTorch framework

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

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)

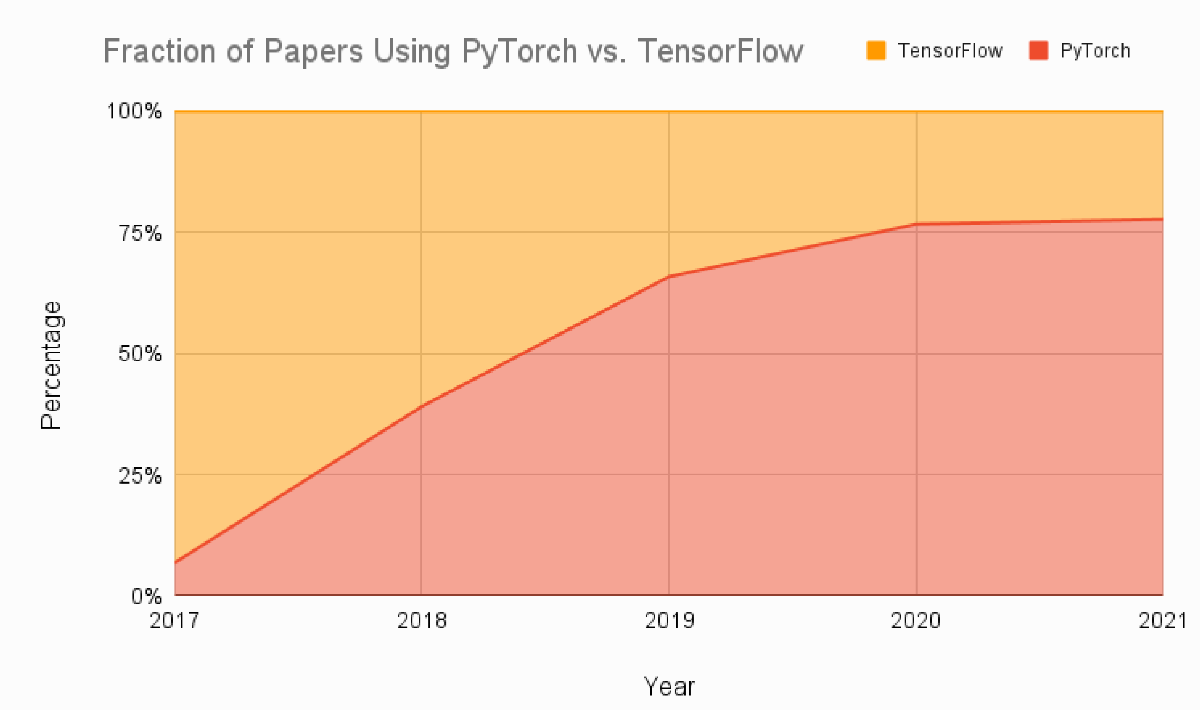
9. What is the **autograd/backprop** module in PyTorch, and how does it use a **computational graph** to **compute all derivatives**?
10. How to use the **autograd** to implement **derivatives** and a **vanilla gradient descent**?
11. How to implement **backprop** in PyTorch for our **shallow Neural 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?

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 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*.

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

```
# 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.]])
```

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.

```python
# 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.

```python
# 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
list_tensor2 = list_tensor.type(torch.float64)
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.

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

```python
# Seeding
torch.manual_seed(17)

# Create a 4D tensor, of size 4 by 2 by 3 by 7, filled with random values
# drawn from a normal distribution with zero mean and variance one.
rand_normal_tensor = torch.randn(size = (4, 2, 3, 7))
print(rand_normal_tensor.shape)

# Calculate mean with method (should be close to 0)
# (With see 17, should be 0.0865)
val = rand_normal_tensor.mean()
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.

```
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**!

```python
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 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;

```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)

# 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 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;

```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)

# Slicing
slice1 = rand_unif_tensor[0:2]
print("Slice1: ", slice1)
slice2 = rand_unif_tensor[:2]
print("Slice2: ", slice2)
slice3 = rand_unif_tensor[1:]
print("Slice3: ", slice3)
slice4 = rand_unif_tensor[0, :, :]
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;

```
# 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 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**.

```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,

```
# 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 addition
c = a + b
print(c)
c = torch.add(a, b)
print(c)
```

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

```
# Element-wise substraction
c = a - b
print(c)
c = torch.sub(a, b)
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,

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

```python
# 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]])
```

```python
# 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,

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

```python
# Transposition
c = a.T
print(a)
print(c)
```

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

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

```
# 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)
```

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

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

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

```python
# Tensors x and y are not broadcastable,
# because x does not have at least 1 dimension
x = torch.ones((0,))
y = torch.ones(2,2)
z = (x+y)
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

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

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

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

<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).

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

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

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

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

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

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

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

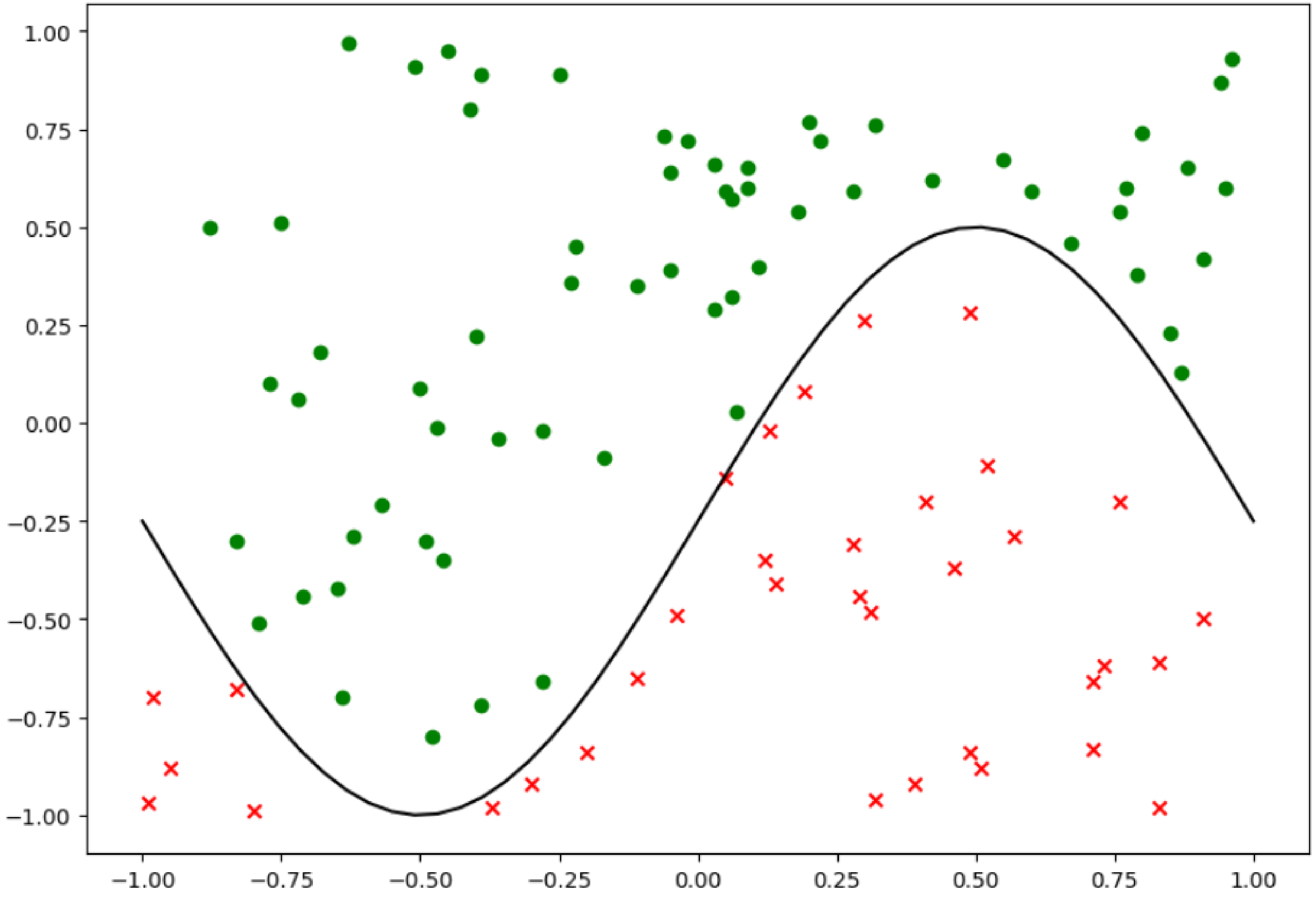
```
# Operations require tensors
# to be on same device
c = torch.ones(2, 3).cpu()
print(c)
print(c.device)
d = torch.ones(2, 3).cuda()
print(d)
print(d.device)
f = c + d
print(f)
```

```
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 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 metric 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**.

```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()
```

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 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.

```python
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()
```

Restricted

Trying out our new neural network

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

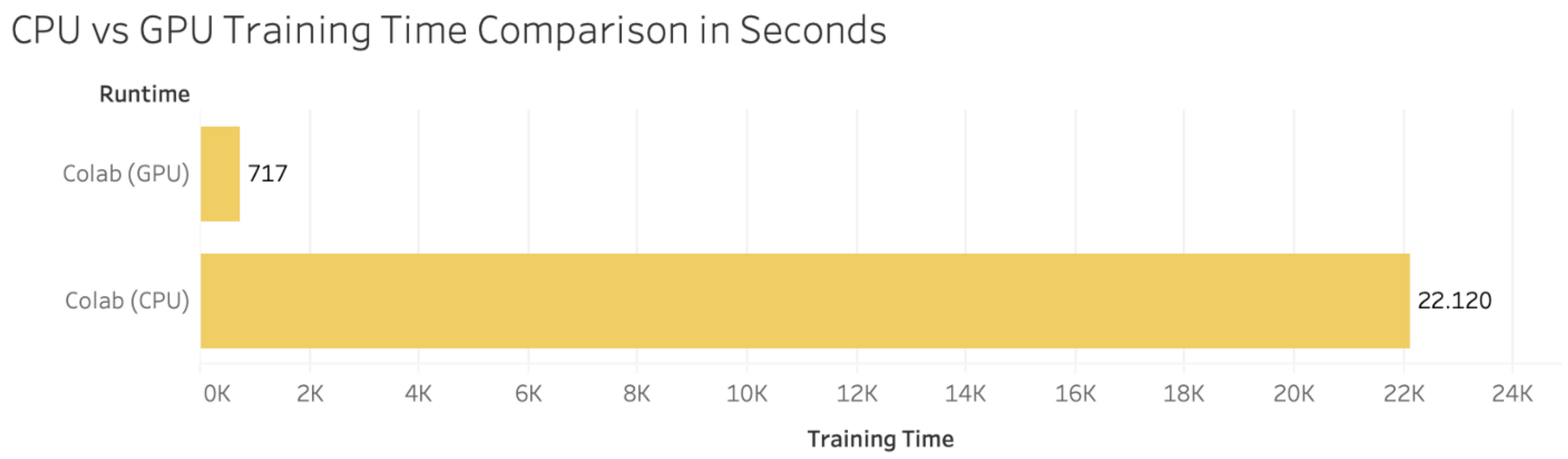
```
train_inputs_pt = torch.from_numpy(train_inputs).to(device)
train_outputs_pt = torch.from_numpy(train_outputs).to(device)
```

```
# 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 1060), 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 bit of practice for you

In order to practice your PyTorch Tensor skills,

- Have a look at Notebook #3,
- It contains three functions (computing the mean of a 2D tensor column-wise, finding indices where the values a 2D tensor are greater than 5, concatenating two 2D tensors, etc.)
- Solutions are given in the Notebook 3 solution folder.

Let us use the remaining time to practice.

A bit of practice for you

In order to continue your practice your PyTorch Tensor skills, you may try to manually implement some typical algorithms using the basic operations on PyTorch tensors:

- Finding the maximum, minimum, average, median values of a given 1D tensor,
- Transposing a given 2D tensor, computing its determinant, eigenvalues/eigenvectors,
- 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 CPU and CUDA. In which scenarios is it slower to implement said functions and run them on GPU?