Artificial Neural Networks

Course-5

Gavrilut Dragos

rev 1.2

AGENDA FOR TODAY

- PyTorch
  - What is PyTorch
  - Data types
  - Tensors
  - Layers
  - Module base class
  - Datasets & DataLoaders
  - Checkpoints

What is PyTorch

What is PyTorch

- A Python library that provides tensor operations (with support for GPU) and various deep learning out-of-a-box functionalities.

- **Current version**: 2.1.0 (Oct.2023)
- **Site**: https://pytorch.org/
- **GitHub**: https://github.com/pytorch/pytorch
- **Documentation**: https://pytorch.org/docs/stable/index.html

What is PyTorch

- To install PyTorch, use the following command:

  ```
  pip install torch torchvision torchaudio
  ```

  Or with conda:

  ```
  conda install pytorch torchvision torchaudio -c pytorch
  ```

- To use PyTorch with GPU, we need have a graphical card that is compatible (e.g., Nvidia)

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

torch

This is the core module that provides multi-dimensional arrays (tensors) and the necessary operations for their manipulation.

- Tensor operations (e.g., torch.add, torch.mul)
- Mathematical operations (e.g., torch.sin, torch.log)
- Reduction operations (e.g., torch.mean, torch.sum)

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

torch.nn

This module provides the building blocks to create and train neural networks

- Layers (e.g., torch.nn.Linear, torch.nn.Conv2d)
- Activation functions (e.g., torch.nn.ReLU, torch.nn.Sigmoid)
- Loss functions (e.g., torch.nn.CrossEntropyLoss, torch.nn.MSELoss)
- Utilities for building neural network models (torch.nn.Module base class)
- Predefined models for various tasks (e.g., ResNet, VGG)

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

torch.optim

This module provides dynamic computation graph and automatic differentiation.

- Stochastic Gradient Descent (torch.optim.SGD)
- Adam (torch.optim.Adam)
- RMSprop (torch.optim.RMSprop)

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

torch.autograd

This module provides optimization algorithms to train neural networks. It includes various gradient descent-based optimizers.

- Provides the Variable class, which is now mostly integrated directly with tensors via the **requires_grad** attribute.
- Manages the automatic differentiation of operations on tensors.
- Contains the Function class, which forms the basis for the dynamic computation graph.

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

torch.util

This is a utility module containing several sub-modules and tools for various tasks such as data loading, checkpoint, etc)

What is PyTorch

PyTorch consists in multiple modules (with different functionalities):

Besides this, the following modules are also supported:

- **torchvision**, **torchaudio** and **torchtext** (tools, models and utilities for autodion, vision an text processing)
- **jit –** a Just In Time compiler to convert PyTorch code into a form that can be optimized and executed in various environments without a dependency on the Python runtime
- **onnx** - exporting PyTorch models in the Open Neural Network Exchange (ONNX) format, which can then be consumed by various deep learning frameworks and tools.

Data Types

Data types

PyTorch supports the most of the same types as NumPy supports.

- ***Integers***:
  - ***int8*** : 8-bit signed integer (-128 to 127)
  - ***int16*** : 16-bit signed integer (-32768 to 32767)
  - ***int32*** : 32-bit signed integer (-2^{31} to $2^{31}-1$)
  - ***int64*** : 64-bit signed integer (-2^{63} to $2^{63}-1$)
- ***Unsigned Integers***:
  - ***uint8*** : 8-bit unsigned integer (0 to 255)

Data types

PyTorch supports the same types as NumPy supports.

- ***Floating Point***:
  - ***float16***: Half precision float (or **half**)
  - ***float32***: Single precision float (or **float**)
  - ***float64***: Double precision float (or **double**)
- ***Complex Numbers***:
  - ***complex64***: Complex number with two 32-bit floats (real and imaginary components)
  - ***complex128***: Complex number with two 64-bit floats
- ***Boolean***:
  - **bool**: Boolean type storing True and False values

Data types

Most of these types are in fact C/C++ types (or the actual types that are accepted under the current architecture).

Because of this, **numpy** and **pytorch** can share some memory zones with continuous data (arrays).

However, **PyTorch** has to take into consideration that not all CPU types have a similar type on the GPU (with a similar representation).

Tensors

Tensors

- A tensor is a multi-dimensional array. It is very similar to to NumPy's array, but has some additional properties, the most important one being the ability to be used on GPUs (graphics processing units) to accelerate computing.

- A tensor (if runs on a CPU) can share the same memory with a NumPy array (making working with numpy fairly easy).

Tensors

- A tensor can be created in the following ways:
  - From a python list: **torch.tensor**
  - With zeros: **torch.zeros**
  - With ones: **torch.ones**
  - With random values: **torch.rand, torch.randn, torch.randint**
  - With a specific value: **torch.fill**
  - Based on an interval: **torch.arrange**
  - From a numpy array: **torch.from_numpy**
  - Or an empty one: **torch.empty**
- The format is similar to the one from numpy (meaning you have to provide a shape, a type, etc).

Tensors

› Let's see some examples on how to build a tensor.

```python
import torch

t1 = torch.tensor([1,2,3])
t2 = torch.ones((2,3))
t3 = torch.zeros(3,2)
t4 = torch.rand(3,3)
t5 = torch.arange(1,9)

print("t1=",t1)
print("t2=",t2)
print("t3=",t3)
print("t4=",t4)
print("t5=",t5)
```

Output

```
t1= tensor([1, 2, 3])
t2= tensor([[1., 1., 1.],
        [1., 1., 1.]])
t3= tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
t4= tensor([[0.2781, 0.4629, 0.9208],
        [0.7049, 0.7303, 0.2473],
        [0.5844, 0.6176, 0.2701]])
t5= tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

Tensors

› Let's see some examples on how to build a tensor.

```python
import torch

t1 = torch.tensor([1,2,3])
t2 = torch.ones((2,3))
t3 = torch.zeros(3,2)
t4 = torch.rand(3,3)
t5 = torch.arange(1,9)

print("t1=",t1)
print("t2=",t2)
print("t3=",t3)
print("t4=",t4)
print("t5=",t5)
```

Notice that a shape can be provided via a tuple or directly as a parameter

Output

```
t1= tensor([1, 2, 3])
t2= tensor([[1., 1., 1.],
        [1., 1., 1.]])
t3= tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
t4= tensor([[0.2781, 0.4629, 0.9208],
        [0.7049, 0.7303, 0.2473],
        [0.5844, 0.6176, 0.2701]])
t5= tensor([1, 2, 3, 4, 5, 6, 7, 8])
```

Tensors

› Or, if we want to specify a scalar type:

```
import torch

t1 = torch.tensor([1,2,3],dtype = torch.int16)
t2 = torch.ones((2,3),dtype = torch.int8)
t3 = torch.zeros(3,2,dtype = torch.uint8)
t4 = torch.rand(3,3,dtype = torch.float64)

print("t1=",t1)
print("t2=",t2)
print("t3=",t3)
print("t4=",t4)
```

Output

```
t1= tensor([1, 2, 3], dtype=torch.int16)
t2= tensor([[1, 1, 1],
        [1, 1, 1]], dtype=torch.int8)
t3= tensor([[0, 0],
        [0, 0],
        [0, 0]], dtype=torch.uint8)
t4= tensor([[0.0195, 0.6436, 0.0213],
        [0.5023, 0.0195, 0.4338],
        [0.0393, 0.8765, 0.1443]], dtype=torch.float64)
```

Tensors

› As previously said, a **numpy** array can also be used as a parameter to create a tensor.

```
import torch, numpy

a = numpy.array([[1,2,3],[4,5,6]])
print("a=",a)
t = torch.from_numpy(a)
print("t=",t)
```

Output

```
a= [[1 2 3]
 [4 5 6]]
t= tensor([[1, 2, 3],
        [4, 5, 6]],
dtype=torch.int32)
```

Tensors

› What is important to understand in this case is that they share the same memory space (if run on CPU)

```
import torch, numpy

a = numpy.array([[1,2,3],[4,5,6]])
print("a=",a)
t = torch.from_numpy(a)
print("t=",t)

a[1,1] = 100
print(t)
```

Output

```
a= [[1 2 3]
 [4 5 6]]
t= tensor([[1, 2, 3],
        [4, 5, 6]],
dtype=torch.int32)
tensor([[  1,   2,   3],
        [  4, 100,   6]],
dtype=torch.int32)
```

Notice that if we modify variable a, the tensor t is changing as well

Tensors

- The indexing and slicing rules apply in a similar manner as with NumPy.

```python
import torch

t = torch.tensor([[1,2,3],[4,5,6]])
print("t[0,0]    = ",t[0,0])
print("t[0]      = ",t[0])
print("t[-1,-1]  = ",t[-1,-1])
print("t[0,:1]   = ",t[0,:2])
print("t[-1,1:]  = ",t[-1,1:])

v = torch.tensor([1,2,3,4,5])
print("v      = ",v)
print("v[v>3] = ",v[v>3])
```

Output

```
t[0,0]    =  tensor(1)
t[0]      =  tensor([1, 2, 3])
t[-1,-1]  =  tensor(6)
t[0,:1]   =  tensor([1, 2])
t[-1,1:]  =  tensor([5, 6])

v      =  tensor([1, 2, 3, 4, 5])
v[v>3] =  tensor([4, 5])
```

Tensors

› Similarly, for a tensor there are several information that can be provided (such as shape, element size, etc). Additionally, an extra property called .device can provide information on tensor location (CPU or GPU)

```
import torch

t = torch.tensor([[1,2,3],[4,5,6]])
print("Shape    = ",t.shape)
print("Type     = ",t.dtype)
print("Dim      = ",t.ndim)
print("Size     = ",t.element_size())
print("Device   = ",t.device)
```

Output

```
Shape     =   torch.Size([2, 3])
Type      =   torch.int64
Dim       =   2
Size      =   8
Device    =   cpu
```

Tensors

› A reshape method (.reshape) is also provided and can be used similarly as with NumPy to change the shape of an existing tensor.

```
import torch

t = torch.arange(1,9)
print("Vector = ",t)
t = t.reshape(4,2)
print("4x2 matrix = ",t)
t = t.reshape(2,2,2)
print("2x2x2 array = ",t)
```

Output

```
Vector =  tensor([1, 2, 3, 4, 5,
6, 7, 8])
4x2 matrix =  tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
2x2x2 array =  tensor([[[1, 2],
         [3, 4]],

        [[5, 6],
         [7, 8]]])
```

Tensors

› In the previous example we have used a syntax as follows:

```
new_tensor = tensor.reshape(new shape)
```

It is important to understand that this **DOES NOT COPY** the memory, it just returns another tensor that shares the same memory with the original one, but with a different shape.

As a result, a change in the values from the memory will affect both tensors (the original one and the resulted one).

Tensors

› Let's see an example:

```python
import torch

t = torch.arange(1,9)
print("Vector = ",t)
m = t.reshape(4,2)
print("4x2 matrix = ",m)
t[3] = 1000
print("matrix = ",m)
```

Output

```
Vector =  tensor([1, 2, 3, 4, 5, 6, 7, 8])
4x2 matrix =  tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
matrix =  tensor([[   1,    2],
        [   3, 1000],
        [   5,    6],
        [   7,    8]])
```

› As it can be observed, after we change element with index 3 in the tensor t, the change is also observable in matrix m

Tensors

- Scalar and element-wise operation work in a similar manner

```
import torch

t = torch.tensor(([1,2,3],[4,5,6]))
t = t * 2
print("t = ",t)
t = t + 10
print("t = ",t)
v = torch.ones(2,3, dtype=torch.int)
print("v = ",v)
t += v
print("t = ",t)
```

Output

```
t =  tensor([[ 2,  4,  6],
        [ 8, 10, 12]])
t =  tensor([[12, 14, 16],
        [18, 20, 22]])
v =  tensor([[1, 1, 1],
        [1, 1, 1]], dtype=torch.int32)
t =  tensor([[13, 15, 17],
        [19, 21, 23]])
```

Tensors

› There are also several simple statistical functions that can be used (like sum, median, mean, etc). Its important to notice that the result of these function is a tensor (and not necessarily a scalar value) – even if the tensor is a vector with one element (containing just one scalar).

```python
import torch

t = torch.tensor([[1,2,3],[4,5,6]],dtype=torch.float32)
print("sum(t)    = ",torch.sum(t))
print("mean(t)   = ",torch.mean(t))
print("median(t) = ",torch.median(t))
print("min(t)    = ",torch.min(t))
print("max(t)    = ",torch.max(t))
```

Output

```
sum(t)    =  tensor(21.)
mean(t)   =  tensor(3.5000)
median(t) =  tensor(3.)
min(t)    =  tensor(1.)
max(t)    =  tensor(6.)
```

Tensors

› There is also a method called .dot that performs the dot product between two tensors.

```
import torch

v1 = torch.tensor([1,2,3])
v2 = torch.tensor([10,20,30])
result = v1.dot(v2)
print("v1 . v2 = ", result)
```

Output

```
v1 . v2 =  tensor(140)
```

› In this case the dot product is: 1x10 + 2x20 + 3x30 = 10+40+90 = 140

Tensors

› However, while the method .dot can be used in NumPy to multiply matrixes as well:

```
import numpy
v1 = numpy.array([1,2,3])
v2 = numpy.array([[10,20],[30,40],[5,6]])
result = v1.dot(v2)
print("v1 . v2 = ", result)
```

Output

```
v1 . v2 =  [ 85 118]
```

in PyTorch it only works with vectors:

```
import torch
v1 = torch.tensor([1,2,3])
v2 = torch.tensor([[10,20],[30,40],[5,6]])
result = v1.dot(v2)
print("v1 . v2 = ", result)
```

Error

```
Traceback (most recent call last):
  File "e:\Lucru\RN\teste\a.py", line 5, in <module>
    result = v1.dot(v2)
RuntimeError: 1D tensors expected, but got 1D and 2D tensors
```

Tensors

› We can multiply two matrixes using torch.mm(M_1,M_2) method (mm stands for **m**atrix **m**ultiplication). Its important to highlight that both parameters M_1 and M_2 must be matrixes – bi-dimensional arrays).

```
import torch, numpy

m1 = torch.tensor([[1,2,3]])
m2 = torch.tensor([[10,20],[30,40],[5,6]])
result = torch.mm(m1,m2)
print("m1 x m2 = ", result)
```

Output

```
m1 x m2 =  tensor([[ 85, 118]])
```

Tensors

- Finally, a tensor can be moved to a device (if present) using .to method.

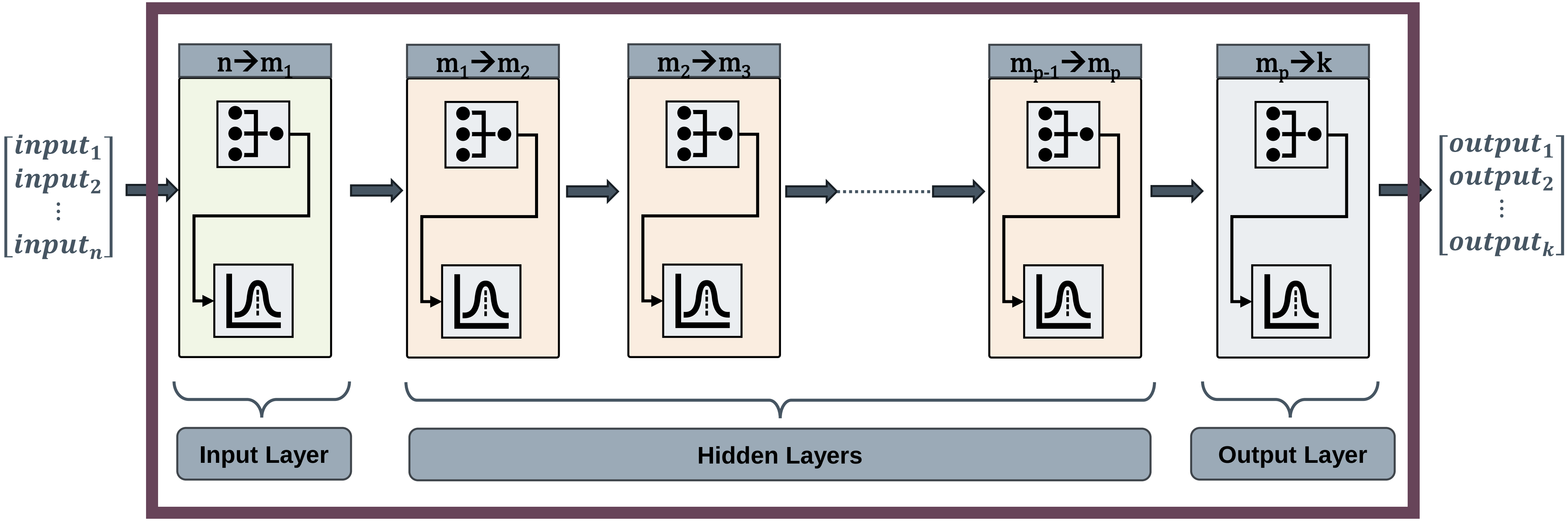
```
import torch

tensor = torch.tensor([1, 2, 3, 4])
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    tensor_gpu = tensor.to(device)
```

- It is recommended to check if GPU are present first.

Layers

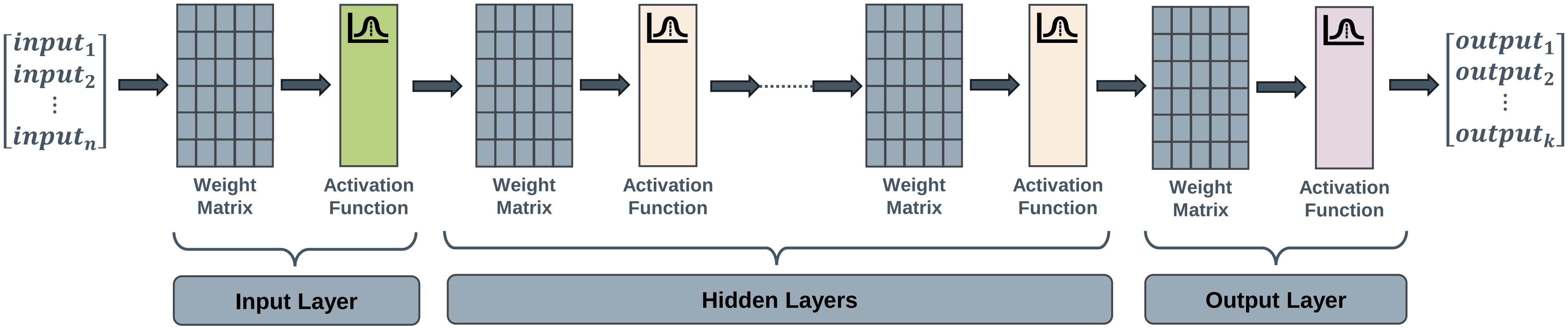
Layers

› In a previous course we have defied a simple neuronal network as follows:



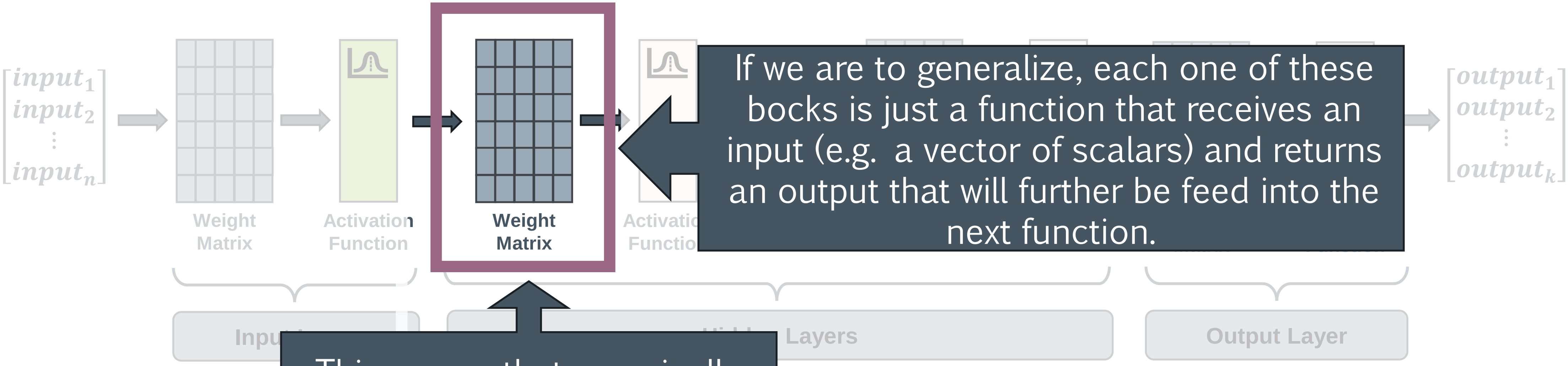
Layers

› In practice we can describe the same architecture in more details:



Layers

› In practice we can describe the same architecture in more details:



If we are to generalize, each one of these bocks is just a function that receives an input (e.g. a vector of scalars) and returns an output that will further be feed into the next function.

This means that generically, we can define these building blocks as follows:

$$\boldsymbol{f(tensor) = tensor}$$

Layers

- PyTorch has multiple such functions defined (that receive a tensor and return another one: $f(tensor) = tensor$
- These functions are referred to by PyTorch as **layers** and can be grouped in the following cathegories:
  - Containers
  - Non-linear activations
  - Linear layers
  - Sparse layers
  - Shuffle layers
  - Normalization layers
  - …
- There are more than 100+ such layers defined in the latest version of PyTorch.

Layers

- Let’s see some examples of most common activation functions:
  - softmax (torch.nn.Softmax)
  - ReLU (torch.nn.RelU)
  - Sigmoid (torch.nn.Sigmoid)
  - Tanh (torch.nn.Tanh)
  - Threshold (torch.nn.Threshold)

Layers

› Let's see some examples on how activation functions can be used.

```
import torch

tensor = torch.tensor([-1, 0, 1, 2],dtype=float)

softmax = torch.nn.Softmax(dim=0)
relu = torch.nn.ReLU()
sigmoid = torch.nn.Sigmoid()
tanh = torch.nn.Tanh()

print("Softmax = ",softmax(tensor))
print("ReLU    = ",relu(tensor))
print("Sigmoid = ",sigmoid(tensor))
print("tanh    = ",tanh(tensor))
```

Output

```
Softmax =  tensor([0.0321, 0.0871, 0.2369, 0.6439],
dtype=torch.float64)
ReLU    =  tensor([0., 0., 1., 2.], dtype=torch.float64)
Sigmoid =  tensor([0.2689, 0.5000, 0.7311, 0.8808],
dtype=torch.float64)
tanh    =  tensor([-0.7616,  0.0000,  0.7616,  0.9640],
dtype=torch.float64)
```

Layers

› Let’s see some examples on how activation functions can be used.

```
import torch

te                                    2],dtype=float)

so
re
sigmoid = torch.nn.Sigmoid()
tanh = torch.nn.Tanh()

print("Softmax = ",softmax(tensor))
print("ReLU    = ",relu(tensor))
print("Sigmoid = ",sigmoid(tensor))
print("tanh    = ",tanh(tensor))
```

For example, in this case, the ReLU function is called on every element in the vector. ReLU(x) = max(0,x) so,

ReLU([-1,0,1,2]) → [0,0,1,2]

Output

```
Softmax =  tensor([0.0321, 0.0871, 0.2369, 0.6439], dtype=torch.float64)
ReLU    =  tensor([0., 0., 1., 2.], dtype=torch.float64)
Sigmoid =  tensor([0.2689, 0.5000, 0.7311, 0.8808], dtype=torch.float64)
tanh    =  tensor([-0.7616,  0.0000,  0.7616,  0.9640], dtype=torch.float64)
```

Layers

› If we want to use them with only one value, we can use a tensor with one value:

```
import torch

tensor = torch.tensor([0.6],dtype=float)

threshold = torch.nn.Threshold(0.5,0)
relu = torch.nn.ReLU()
sigmoid = torch.nn.Sigmoid()
tanh = torch.nn.Tanh()

print("Threshold = ",threshold(tensor))
print("ReLU      = ",relu(tensor))
print("Sigmoid   = ",sigmoid(tensor))
print("tanh      = ",tanh(tensor))
```

Output

```
Threshold =  tensor([0.6000], dtype=torch.float64)
ReLU      =  tensor([0.6000], dtype=torch.float64)
Sigmoid   =  tensor([0.6457], dtype=torch.float64)
tanh      =  tensor([0.5370], dtype=torch.float64))
```

Layers

- If we want to use them with only one value, we can use a tensor with one value:

```
import torch

tensor = torch.tensor([0.6],dtype=float)

threshold = torch.nn.Threshold(0.5,0)
relu = torch.nn.ReLU()
sigmoid = torch.nn.Sigmoid()
```

Threshold function is initialized with two values: threshold and default value. The logic is as follow:

$$threshold(x) = \begin{cases} x, & x > threshold \\ value, & x \leq threshold \end{cases}$$

Layers

- Now let's see some example of matrix multiplication layers:
  - Linear (torch.nn.Linear)
  - Bilinear (torch.nn.Bilinear)

Layers

- torch.nn.Linear is pretty much our component for matrix multiplication, that is organized as follows:

$$\begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}$$

x

	1	2	…	n
1	$W_{1,1}$	$W_{1,2}$	…	$W_{1,n}$
2	$W_{2,1}$	$W_{2,2}$	…	$W_{2,n}$
3	$W_{3,1}$	$W_{3,2}$	…	$W_{2,n}$
4	$W_{4,1}$	$W_{4,2}$	…	$W_{3,n}$
5	$W_{5,1}$	$W_{5,2}$	…	$W_{4,n}$
…				
k	$W_{k,1}$	$W_{k,2}$	…	$W_{k,n}$

+

	1
1	$Bias_1$
2	$Bias_2$
3	$Bias_3$
4	$Bias_4$
5	$Bias_5$
…	
k	$Bias_k$

$$\rightarrow \begin{bmatrix} result_1 \\ result_2 \\ \vdots \\ result_k \end{bmatrix}$$

Layers

- torch.nn.Linear is pretty much our component for matrix multiplication, that is organized as follows:
- The initialization format is:
  - ***torch.nn.Linear*** (number of weights/features,
    size of output vector,
    has_bias)

- Implicitly, all weights are initialized with a random value between $-\sqrt{\frac{1}{number\ of\ weights}}$ and $+\sqrt{\frac{1}{number\ of\ weights}}$

Layers

› Let's see an example:

```python
import torch

tensor = torch.tensor([1.0,2.0,3.0])
linear = torch.nn.Linear(3,2,False)
result = linear(tensor)
print("Linear  =",linear)
print("Output  =",result)
print("weights =",linear.weight)
```

Output

```
Linear= Linear(in_features=3, out_features=2, bias=False)
Output= tensor([-1.9759, -0.9411],
grad_fn=<SqueezeBackward4>)
weights= Parameter containing:
tensor([[-0.5071, -0.0566, -0.4519],
        [-0.3147, -0.4348,  0.0810]], requires_grad=True)
```

› The actual operation behind this code is as follows

$$\begin{bmatrix}1.0\\2.0\\3.0\end{bmatrix} \times$$

	1	2	3
1	-0.5071	-0.0566	-0.4519
2	-0.3147	-0.4348	0.0810

$$\rightarrow \begin{bmatrix}-1.9759\\-0.9411\end{bmatrix}$$

Layers

- In practice, the input tensor does not have to be a vector (it can be an array where every row is a sample. The output will be an array as well with the same number of rows as the input, each row being a vector with the outputted values that correspond to a specific input.

```python
import torch

input = torch.tensor([[1.0,2.0,3.0],
                      [4.0,5.0,6.0],
                      [7.0,8.0,9.0],
                      [1.0,3.0,5.0],
                      [2.0,4.0,6.0]])
linear = torch.nn.Linear(3,2,False)
output = linear(input)
print("Linear  =",linear)
print("Output  =",output)
print("weights =",linear.weight)
```

Output

```
Linear  = Linear(in_features=3,
                 out_features=2,
                 bias=False)
Output  = tensor([[0.5216, 1.1902],
                  [2.1431, 2.4498],
                  [3.7645, 3.7093],
                  [0.5027, 1.9607],
                  [1.0432, 2.3805]],
                  grad_fn=<MmBackward0>)
weights = Parameter containing:
tensor([[ 0.4380,  0.2238, -0.1214],
        [-0.1401,  0.3494,  0.2105]],
        requires_grad=True)
```

Layers

- In practice, the input tensor does not have to be a vector (it can be an array where every row is a sample. The output will be an array as well with the same number of rows as the input, each row being a vector with th... espond to a specific input.

```
import torch

input = torch.tensor([[1.0,2.0,3.0],
                      [4.0,5.0,6.0],
                      [7.0,8.0,9.0],
                      [1.0,3.0,5.0],
                      [2.0,4.0,6.0]])
linear = torch.nn.Linear(3,2,False)
output = linear(input)
print("Linear  =",linear)
print("Output  =",output)
print("weights =",linear.weight)
```

In this case, the output for the sample [7,8,9] from the input matrix (the 3rd row) is the 3rd raw from the output matrix [3.7645, 3.7098]

Output

```
Linear  = Linear(in_features=3,
                 out_features=2,
                 bias=False)
Output  = tensor([[0.5216, 1.1902],
                  [2.1431, 2.4498],
                  [3.7645, 3.7093],
                  [0.5027, 1.9607],
                  [1.0432, 2.3805]],
                  grad_fn=<MmBackward0>)
weights = Parameter containing:
tensor([[ 0.4380,  0.2238, -0.1214],
        [-0.1401,  0.3494,  0.2105]],
        requires_grad=True)
```

Layers

- You can also use torch.nn.Parameter(…) to set up your own weights (for example if you have some values that are already precomputed).

```
import torch

input = torch.tensor([[1.0,2.0,3.0],
                      [4.0,5.0,6.0],
                      [7.0,8.0,9.0],
                      [1.0,3.0,5.0],
                      [2.0,4.0,6.0]])
linear = torch.nn.Linear(3,2,False)
linear.weight = torch.nn.Parameter(torch.zeros(2,3,dtype=torch.float))
output = linear(input)
print("Linear  =",linear)
print("Output  =",output)
print("weights =",linear.weight)
```

Output

```
Linear  = Linear(in_features=3, out_features=2,
bias=False)
Output  = tensor([[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]], grad_fn=<MmBackward0>)
weights = Parameter containing:
tensor([[0., 0., 0.],[0., 0., 0.]], requires_grad=True)
```

- In this example we set up all weights to 0 (as such the output will be a zeroed matrix).

Layers

- Another important type of Layers are containers. A container is essentially a group of other layers that are connected (meaning that the output of one layer is the input of another layer).
- The most commonly used containers are:
  - Sequential
  - ModuleList
  - ModuleDict

Layers

- The **torch.nn.Sequantial** module is initialized with a list of layers (or an OrderedDict) and it chains them (meaning that when it has to compute an input, it passes the input to the first from the list, then the output of that layer is passed to the next one and so on until it reaches the final layer). The output of the final layer will be the output of the Sequantial layer.

Layers

› Let's see an example that uses: **torch.nn.Sequantial**

```
import torch

input = torch.tensor([[1.0,2.0,3.0],
                      [4.0,5.0,6.0],
                      [2.0,4.0,6.0]])
net = torch.nn.Sequential(
    torch.nn.Linear(3,2,False),
    torch.nn.ReLU(),
    torch.nn.Linear(2,1,False),
    torch.nn.Sigmoid()
)
output = net(input)
print("net      =",net)
print("Output   =",output)
```

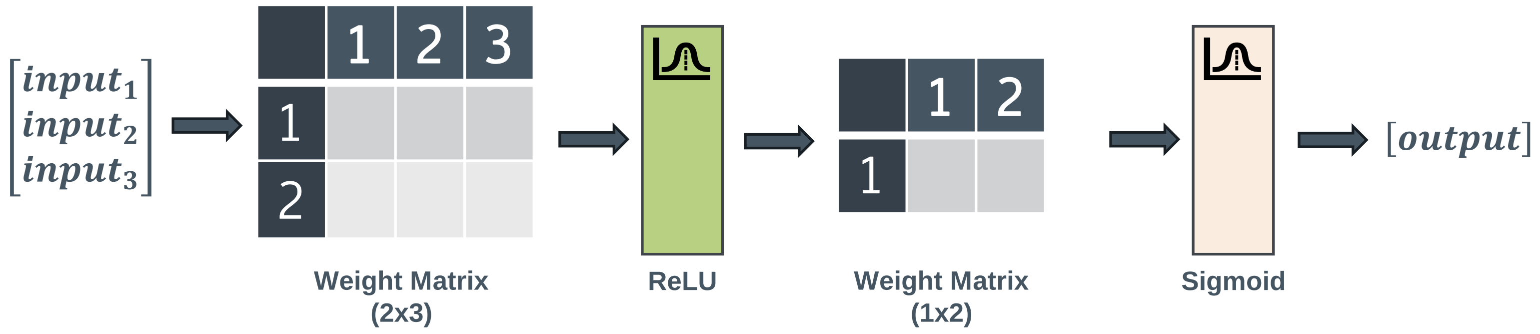
Output

```
net      = Sequential(
  (0): Linear(in_features=3, out_features=2, bias=False)
  (1): ReLU()
  (2): Linear(in_features=2, out_features=1, bias=False)
  (3): Sigmoid()
)
Output   = tensor([[0.4891],
        [0.4728],
        [0.4783]], grad_fn=<SigmoidBackward0>)
```

Layers

› In reality, the Sequential layout is in fact a small neuronal network:

```
net = torch.nn.Sequential
(
    torch.nn.Linear(3,2,False),
    torch.nn.ReLU(),
    torch.nn.Linear(2,1,False),
    torch.nn.Sigmoid()
)
```



Module base class

Module base class

- A Module (torch.nn.Module) is the base class that can be used to implement a neuronal network.
- Usually, you have to do the following:
  - Derive your class from **torch.nn.Module**
  - Add different **layers** and activation functions within your class (you can add them directly as parameters, or within a Sequence layer, …)
  - Implement a constructor **__init__** method
  - Implement a function **forward** that receives an input an returns the output

Module base class

› Let's see an example

```
import torch

class MyNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = torch.nn.Linear(3,1)
        self.activation = torch.nn.ReLU()

    def forward(self, input):
        x = self.layer(input)
        return self.activation(x)

input = torch.tensor([[1.0,2.0,3.0],
                      [4.0,5.0,6.0],
                      [2.0,4.0,6.0]])
net = MyNN()
output = net(input)
print("net     =",net)
print("Output  =",output)
```

Output

```
net      = MyNN(
  (layer): Linear(in_features=3, out_features=1, bias=True)
  (activation): ReLU()
)
Output   = tensor([[0.1091],
        [0.6575],
        [0.0000]], grad_fn=<ReluBackward0>)
```

Module base class

- Let's see an example

```python
import torch

class MyNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer = torch.nn.Linear(3,1)
        self.activation = torch.nn.ReLU()

    def forward(self, input):
        x = self.layer(input)
        return self.activation(x)

input = torch.tensor([[1.0,2.0,3.
```

Alternatively, you can use a sequence parameter to achieve the same result:

```python
class MyNN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(3,1),torch.nn.ReLU())
    def forward(self, input):
        return self.layers(input)
```

Datasets & Dataloaders

Datasets

- Every machine learning algorithm essentially relies on a dataset (for either training or testing).
- However, data can be presented in different forms and as such we will need a common interface that can be use to access different type of data.

Datasets

- PyTorch has an interface/class named: **torch.utils.data.Dataset** that should be used to create a dataset.
- The interface implies 3 methods:
  - An **__init__** function (that loads the dataset)
  - A **__get_item__()** method to access one element from the dataset
  - A **__len__()** method to provide the number of elements from the dataset

Datasets

› PyTorch has an interface/class named: **torch.utils.data.Dataset** that should be used to create a dataset.

```
class <Name>(torch.utils.data.Dataset):
    def __init__(self, ...):
        # instantiate the data set based on parameters

    def __len__(self):
        return 0 # returns the number of records from the dataset

    def __getitem__(self, idx):
        # do some processing if needed
        # return the sample with index 'idx' and its label
        return (sample, label)
```

Datasets

› Let's see an example:

```
import torch

class First100Numbers(torch.utils.data.Dataset):
    def __init__(self):
        self.list = [(i,i%2) for i in range(1,101)]


    def __len__(self):
        return len(self.list)

    def __getitem__(self, idx):
        return (self.list[idx][0],self.list[idx][1])


d = First100Numbers()
print(d[2])
print(d[5]);
print(len(d))
```

Output

```
(3, 1)
(6, 0)
100
```

Datasets

- This method allows one to use 3rd party libraries that can read different type of other datasets, such as:
  - A database (e.g., an SQL database)
  - A CSV/TSV file (e.g., with NumPy)
  - An XML file
  - A JSON file
  - A stream …
- It also allows one to make some transformations on the data before it gets sent to the neuronal network (e.g., convert some strings into number, picture into pixels, etc)

Datasets

Pytorch also has a large set of predefined datasets that can be used to out of the box (**torchvision.datasets.***):

- **MNIST**: Handwritten digit dataset with 60,000 training samples and 10,000 test samples in 10 classes (digits 0-9).
- **Fashion-MNIST**: A dataset comprising of 28x28 grayscale images of 70,000 fashion products from 10 categories, with 7,000 images per category.
- **CIFAR10/1000**: A dataset consisting of 60,000 32x32 color images in 10/100 classes, with 6,000 images per class.
- **ImageNet**: A large dataset designed for use in visual object recognition software research, with more than 14 million images and thousands of classes.
- **COCO** (Common Objects in Context): A large-scale object detection, segmentation, and captioning dataset.
- **VOC** (PASCAL Visual Object Classes): A dataset for object detection, image classification, object segmentation, and person layout.
- **Kinetics**: A large-scale, high-quality dataset of YouTube video URLs which include a diverse range of human-focused actions
- … and many other …

Datasets

The following example downloads the MNIST dataset and stores it locally.

```
import torch
import torchvision
import torchvision.transforms as transforms

image_to_tensor = transforms.Compose([transforms.ToTensor()])

trainset = torchvision.datasets.MNIST(
      root='./training',
      download=True,
      train=True,
      transform= image_to_tensor )
testset = torchvision.datasets.MNIST(
      root='./tests',
      download=True,
      train=False,
      transform= image_to_tensor )
```

Output

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./training\MNIST\raw\train-images-idx3-ubyte.gz
Extracting ./training\MNIST\raw\train-images-idx3-ubyte.gz to ./training\MNIST\raw
....
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./training\MNIST\raw\train-labels-idx1-ubyte.gz
Extracting ./training\MNIST\raw\train-labels-idx1-ubyte.gz to ./training\MNIST\raw
....
```

Datasets

The following example downloads the MNIST dataset and stores it locally.

```
import torch
import torchvision
import torchvision.transforms as transforms

image_to_tensor = transforms.Compose([transforms.ToTensor()])

trainset = torchvision.datasets.MNIST(

testset
        download=True,
        train=False,
        transform= image_to_tensor )
```

This is required as images have to be transformed into a sequence of numbers. **torchvision** library provides several mechanisms to do this, via transforms (including converting to gray scale or other image processing functionalities).

Datasets

The following example downloads the MNIST dataset and stores it locally.

```
import torch
import torchvision
import torchvision.transforms as transforms

image_to_tensor = transforms.Compose([transforms.ToTensor()])

trainset = torchvision.datasets.MNIST(
      root='./training',
      download=True,
      train=True,
      transform= image_to_tensor )
testset = torchvision.datasets.MNIST(
      root='./tests',
      download=True,
      train=False,
      transform= image_to_tensor )
```

The training dataset is going to be downloaded into the folder **./training** after it is converted using the provided transformer.

Datasets

To load the dataset that was downloaded locally, you can use the following code:

```
import torchvision
import torch

trainset = torchvision.datasets.MNIST(root='./training', download=False,
train=True)
print(trainset)
print("Type = ",type(trainset))
print("Is Dataset = ",issubclass(type(trainset), torch.utils.data.Dataset))
```

Notice that trainset is in fact a Dataset object.

Output

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: ./training
    Split: Train
Type =  <class
'torchvision.datasets.mnist.MNIST'>
Is Dataset =  True
```

Dataloaders

Having access to a dataset is not enough. In many cases, the training implies building batches, shuffling the data, and other data sampling procedures.

These procedures work independently from the actual data set (they are agnostic to the content).

As such, PyTorch provide another class: **torch.utils.data.DataLoader** that can facilitate this procedures.

Dataloaders

Having access to a dataset is not enough. In many cases, the training implies building batches, shuffling the data, and other data sampling procedures.

These procedures work independently from the actual data set (they are agnostic to the content).

As such, PyTorch provide another class: **torch.utils.data.DataLoader** that can facilitate this procedures.

Dataloaders

Let's see an example that uses Dataloaders:

```python
import torch
from torch.utils.data import Dataset, DataLoader

class SimpleDataset(Dataset):
    def __init__(self):
        self.samples = torch.randn(9, 4)
        self.label = torch.randn(9)
    def __len__(self):
        return len(self.samples)
    def __getitem__(self, idx):
        return self.samples[idx],self.label[idx]

dataset = SimpleDataset()
dataloader = DataLoader(dataset, batch_size=3, shuffle=True)

for inputs,labels in dataloader:
    print("input = ",inputs)
    print("label = ",labels)
```

Output

```
input =  tensor([[-0.4845, -0.7876, -0.8023,  0.7854],
        [-1.1448, -0.2380, -1.1686,  0.0655],
        [ 0.0174,  0.4290, -0.5761, -0.1557]])
label =  tensor([-0.0152, -0.0988,  0.2331])
----------------------------------------------------
input =  tensor([[ 2.0446, -1.4992, -1.5805, -0.5482],
        [ 0.3754,  0.1475,  0.0598,  0.4752],
        [ 0.0241,  0.7597, -0.1890,  0.7708]])
label =  tensor([-1.1440,  1.4728,  2.3310])
----------------------------------------------------
input =  tensor([[ 0.2800,  0.5915, -1.0798, -1.0840],
        [-1.6834, -0.6283, -0.4241,  0.7376],
        [ 0.4244, -0.4434, -1.0497,  0.2332]])
label =  tensor([-0.4950, -2.0799,  0.8264])
```

Checkpoints

Checkpoints

Checkpoints are a form of model persistence that saves the state of your training process at certain intervals so that you can resume or analyze the training from that point.

In particular for debugging, checkpoints are essential. Furthermore, it is recommended that if a training process takes a long time, checkpoints are gathered to check the state of the training (if some issues happen with the model – for example a vanishing gradient scenario, this can be observed in a checkpoint and corrected).

Checkpoints

A checkpoint typically includes the following information:

- **Model State Dictionary**: The model's parameters or weights.
- **Optimizer State Dictionary:** The state of the optimizer, including the current learning rate, momentum, etc.
- **Epoch**
- **Loss**: The loss value at the checkpoint. This is useful for monitoring progress over time.
- Any other relevant information

Checkpoints

Let's see an example:

```
import torch

checkpoint = {
    'epoch': epoch,
    'model_state': model.state_dict(),
    'optimizer': optimizer.state_dict(),
    'loss': loss,
    # ... any other relevant data ...
}

torch.save(checkpoint, 'my_checkpoint.pth')
```

To load a checkpoint, use torch.load(...). It is important to store in a checkpoint all information needed to resume the training from that point.

Q & A