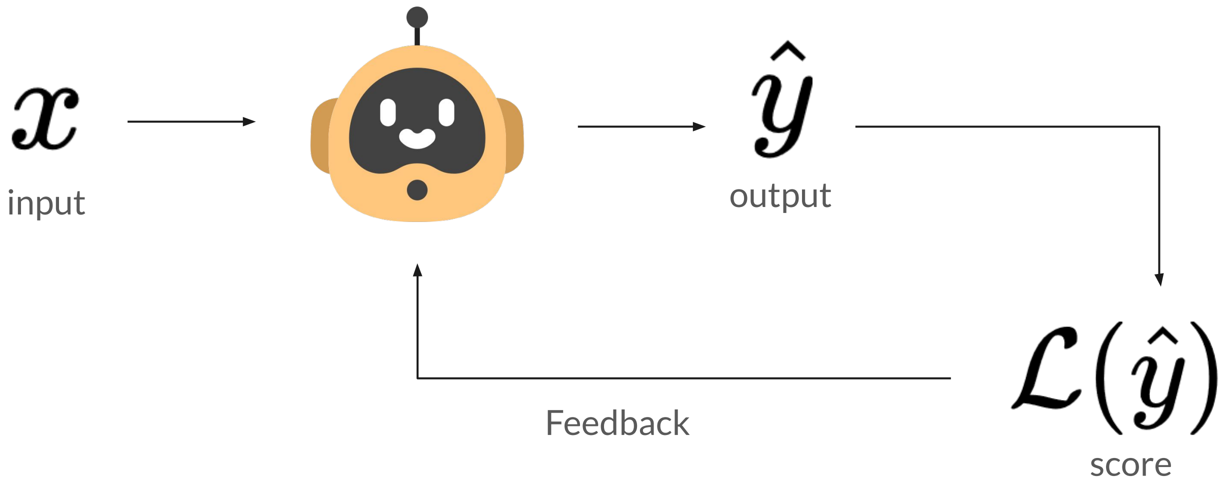
Introductory Seminar of PyTorch for Deep Learning

Daniele Angioni, Cagliari Digital Lab 2024 - Day 1

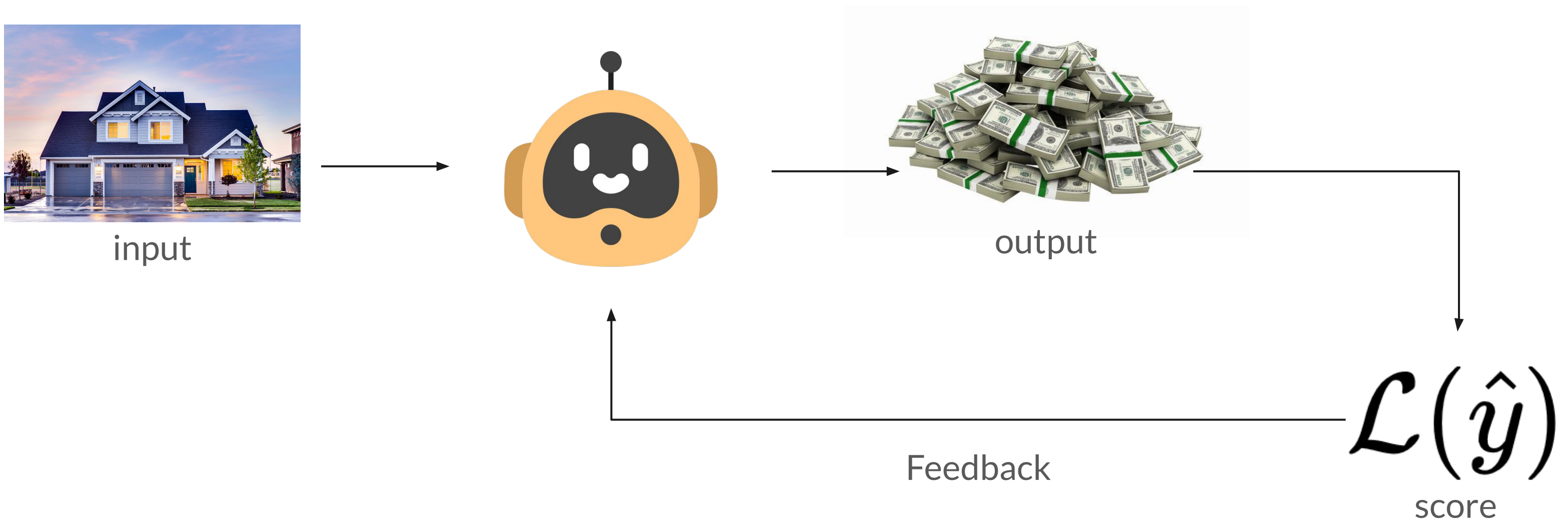
DLAB
CTE • CAGLIARI

Machine Learning Fundamentals through PyTorch lenses

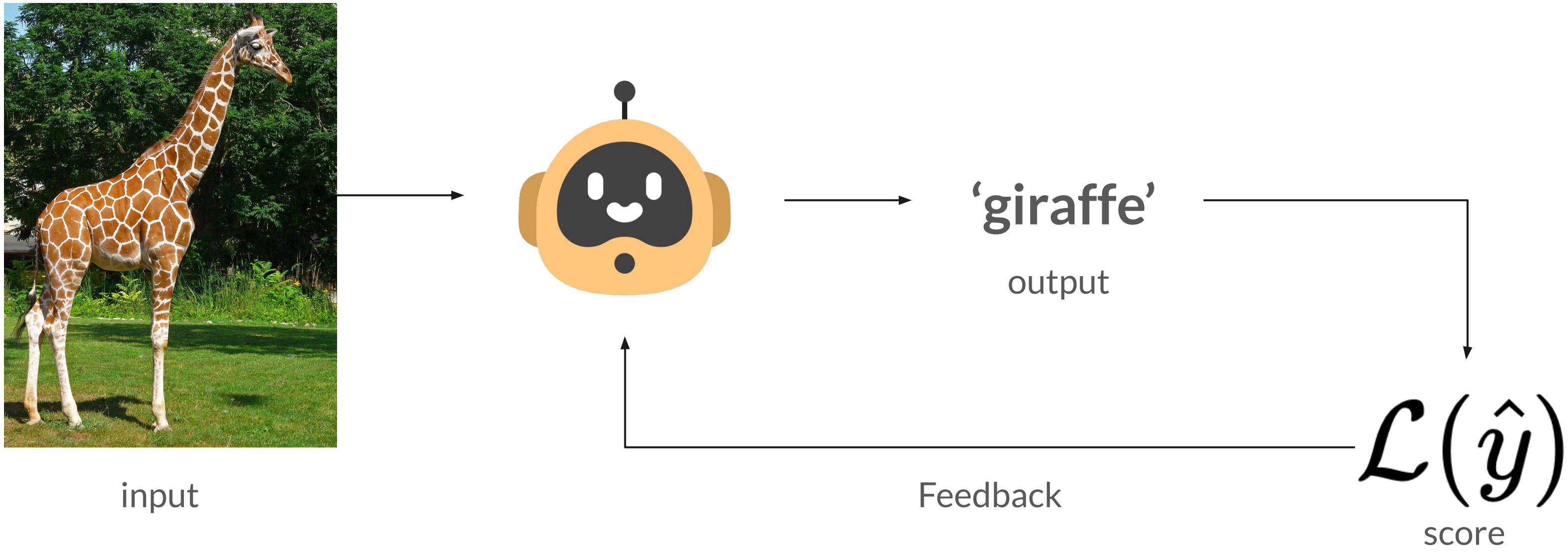
Deep Learning Pipeline



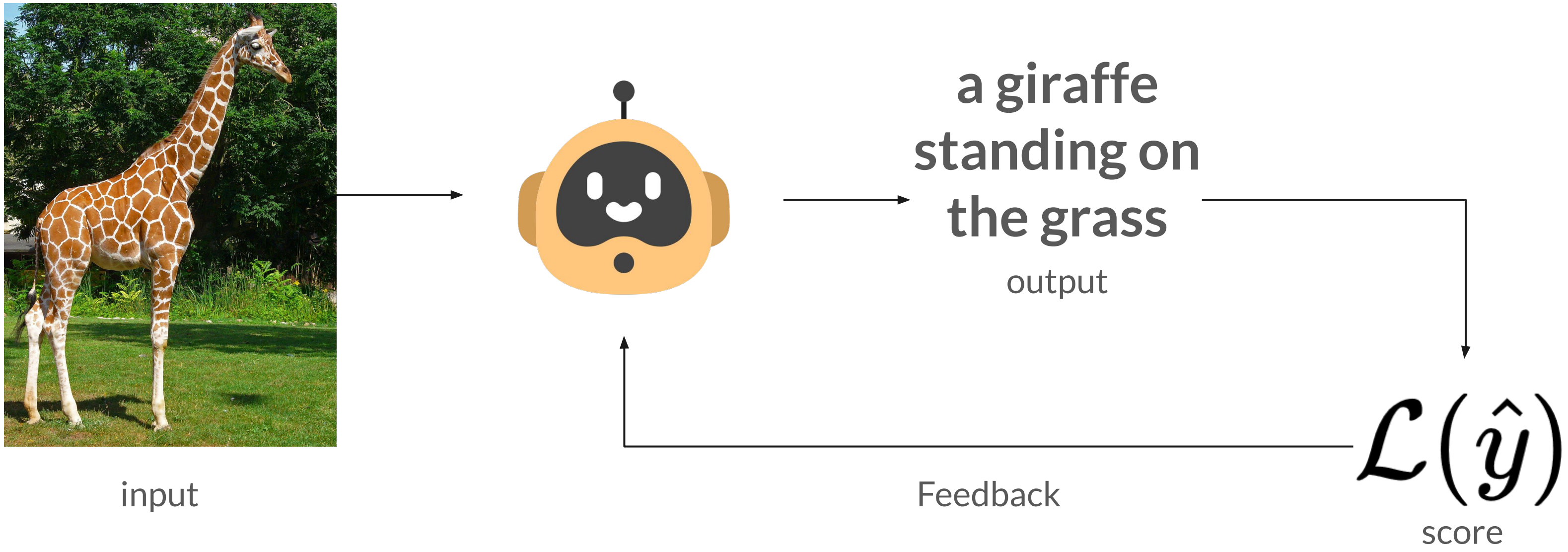
Example



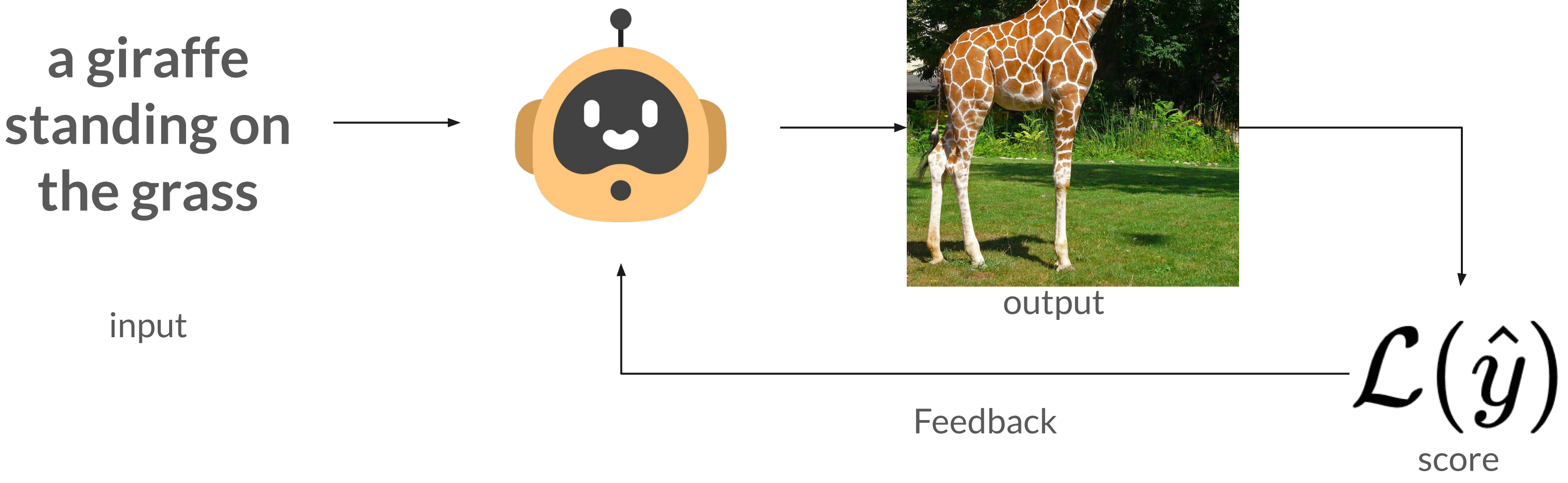
Example



Example

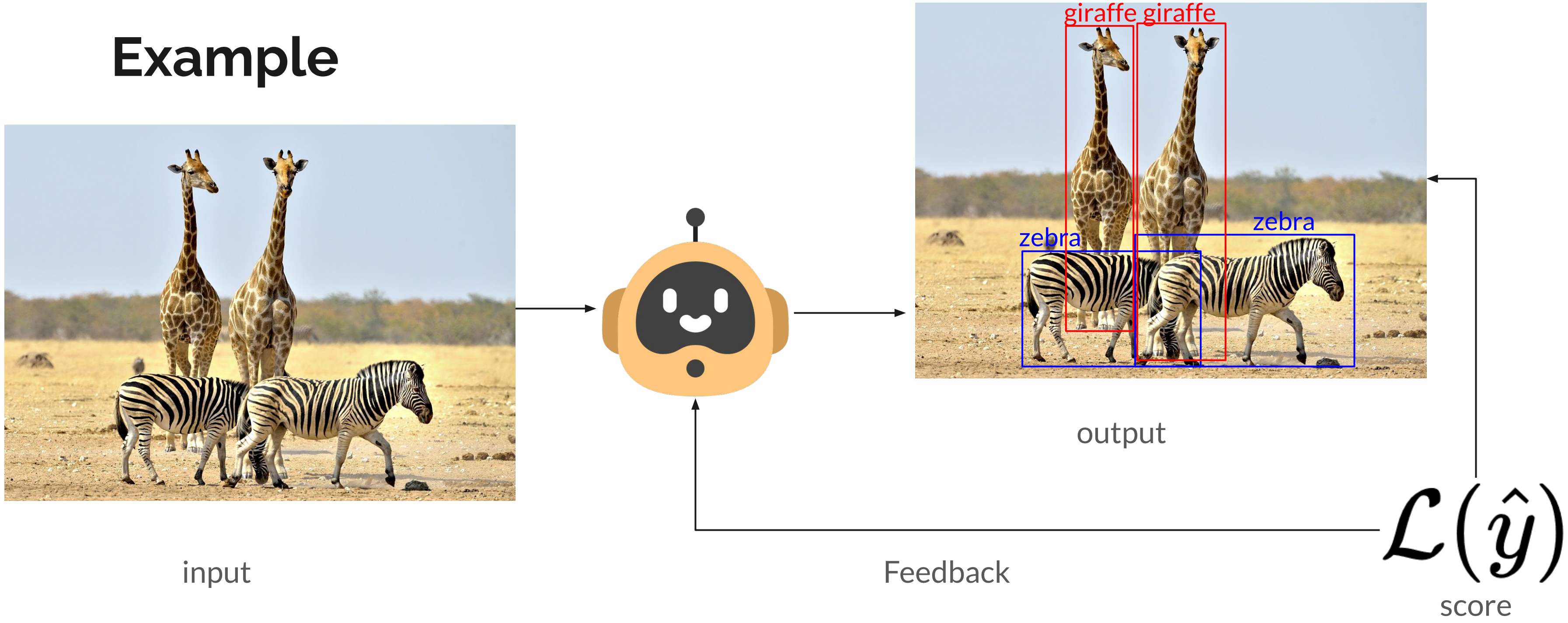

a giraffe standing on the grass
output
input
Feedback
$\mathcal{L}(\hat{y})$
score

Example



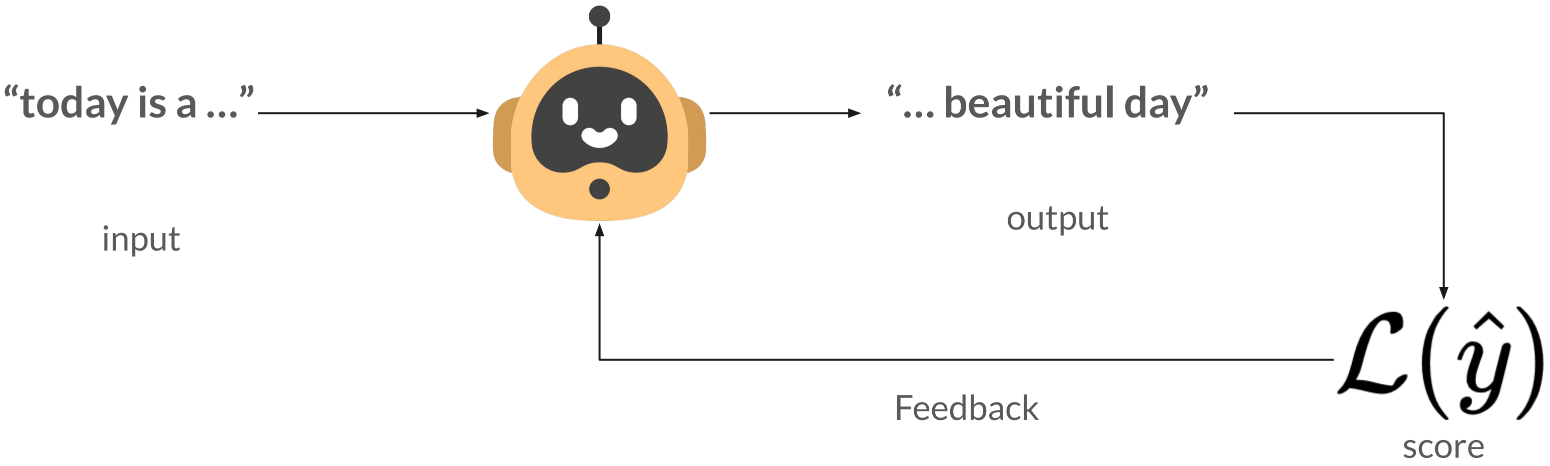
a giraffe standing on the grass
input
output
$\mathcal{L}(\hat{y})$
score
Feedback

Example



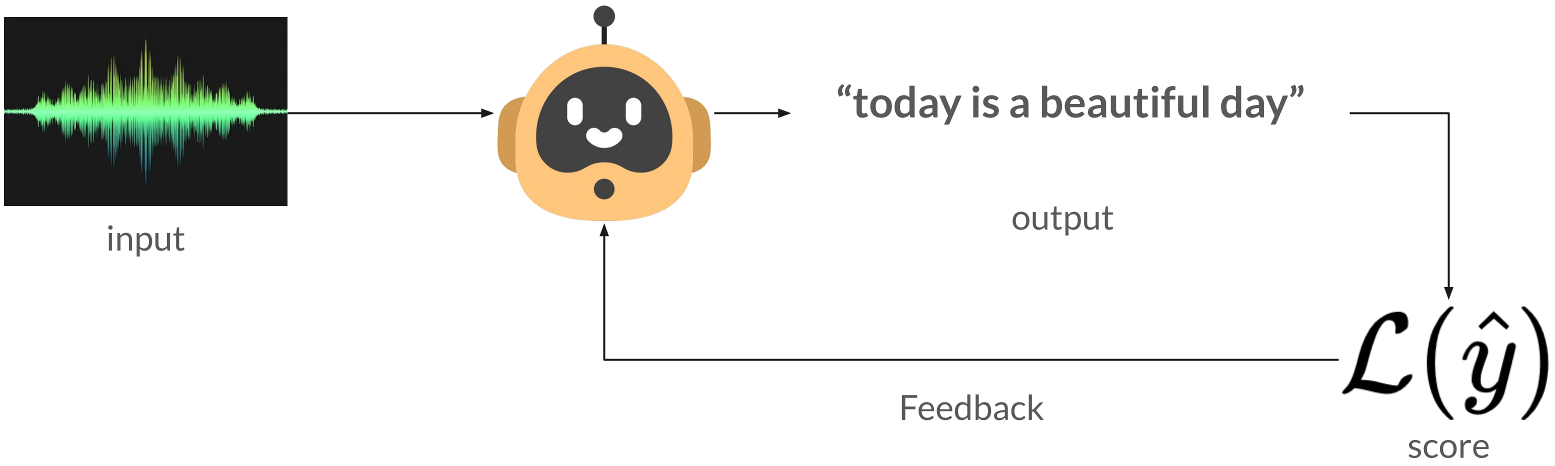
giraffe
giraffe
zebra
zebra
output
input
Feedback
$\mathcal{L}(\hat{y})$
score

Example



"today is a …"
input
"… beautiful day"
output
$\mathcal{L}(\hat{y})$
score
Feedback

Example


input
"today is a beautiful day"
output
$\mathcal{L}(\hat{y})$
score
Feedback

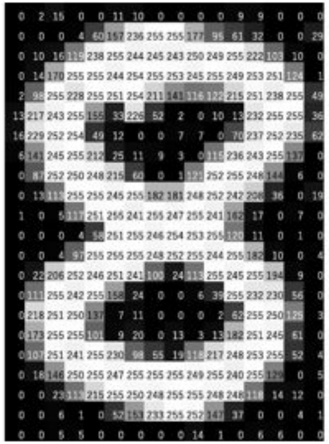
How can machines deal with such different inputs?

Representing data

Deep-learning systems have to be able to map input data to the outputs.

To do so, they usually represent the input data with a specic format.

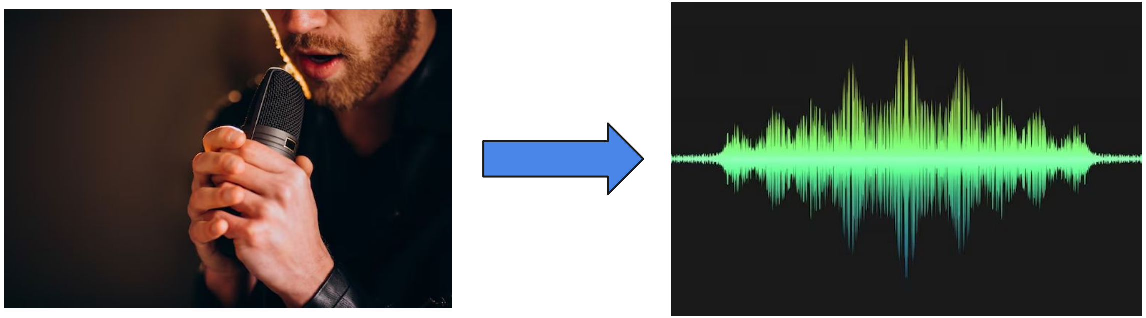
Images



What Computer Sees

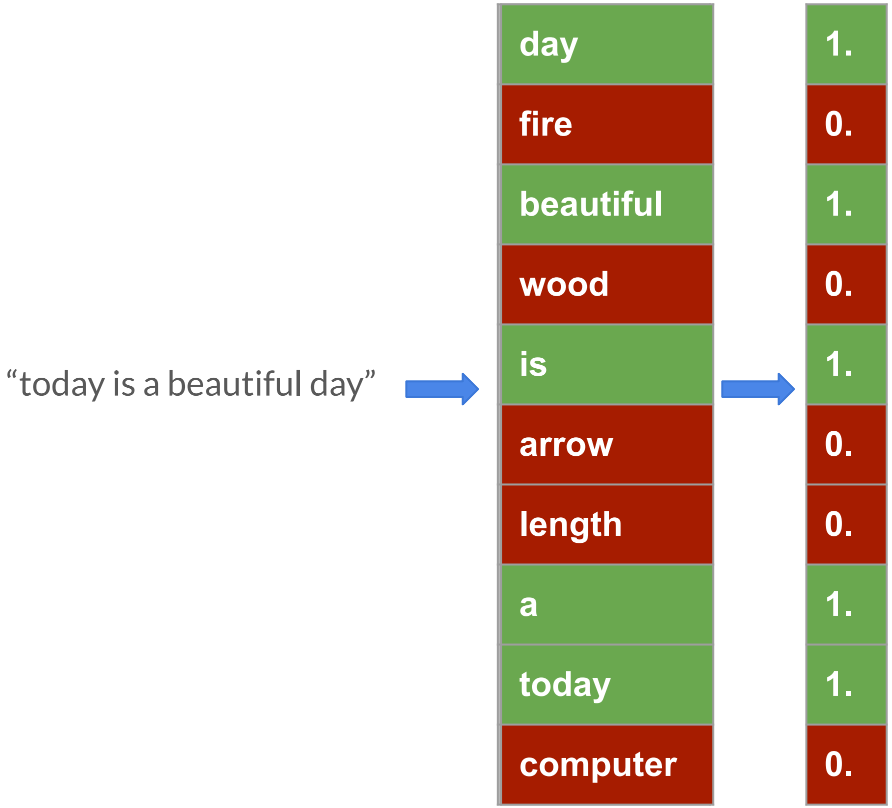
0	2	15	0	0	11	10	0	0	0	0	9	9	0	0	0
0	0	0	4	60	157	236	255	255	177	95	61	32	0	0	29
0	10	16	119	238	255	244	245	243	250	249	255	222	103	10	0
0	14	170	255	255	244	254	255	253	245	255	249	253	251	124	1
2	98	255	228	255	251	254	211	141	116	122	215	251	238	255	49
13	217	243	255	155	33	226	52	2	0	10	13	232	255	255	36
16	229	252	254	49	12	0	0	7	7	0	70	237	252	235	62
6	141	245	255	212	25	11	9	3	0	115	236	243	255	137	0
0	87	252	250	248	215	60	0	1	121	252	255	248	144	6	0
0	13	113	255	255	245	255	182	181	248	252	242	208	36	0	19
1	0	5	117	251	255	241	255	247	255	241	162	17	0	7	0
0	0	0	4	58	251	255	246	254	253	255	120	11	0	1	0
0	0	4	97	255	255	255	248	252	255	244	255	182	10	0	4
0	22	206	252	246	251	241	100	24	113	255	245	255	194	9	0
0	111	255	242	255	158	24	0	0	6	39	255	232	230	56	0
0	218	251	250	137	7	11	0	0	0	2	62	255	250	125	3
0	173	255	255	101	9	20	0	13	3	13	182	251	245	61	0
0	107	251	241	255	230	98	55	19	118	217	248	253	255	52	4
0	18	146	250	255	247	255	255	255	249	255	240	255	129	0	5
0	0	23	113	215	255	250	248	255	255	248	248	118	14	12	0
0	0	6	1	0	52	153	233	255	252	147	37	0	0	4	1
0	0	5	5	0	0	0	0	0	14	1	0	6	6	0	0

Audio



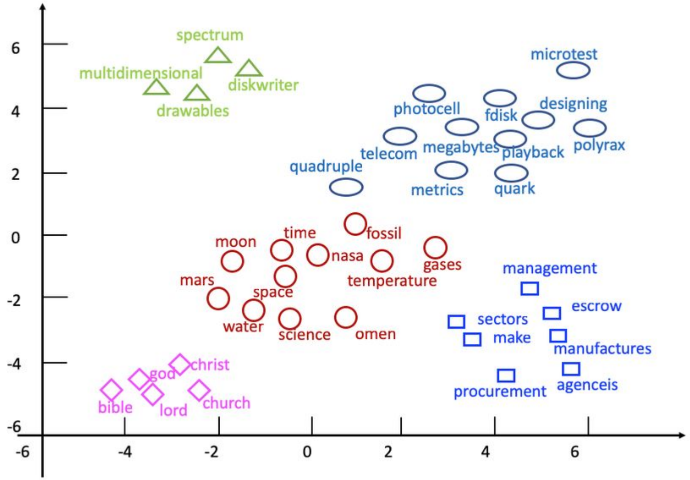
Text

- Vocabulary



Text

- Vocabulary
- Encoding in higher-dimensional space



Tensor Basics

Deep Learning as Floating Point Numbers

All these data are represented as a collection of floating point numbers structured in a particular way depending on the specific application.

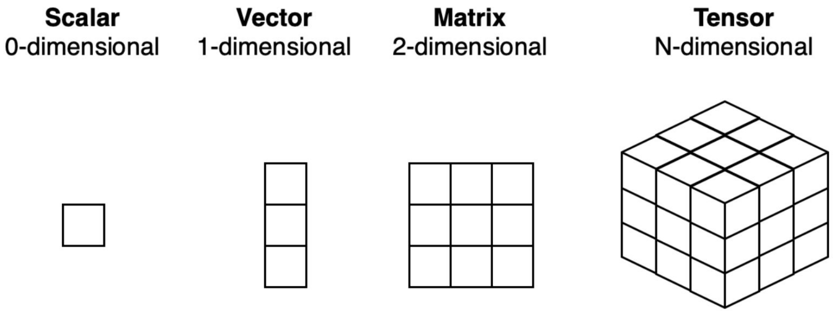
The same can apply to the parameters of a deep learning model, or its output.

These are all collections of floating point numbers!

Tensors

In the context of deep learning, we use the mathematical concept of **tensor**.
A tensor is simply the extension of a vector that has an arbitrary number of dimensions.

tensors == multi-dimensional arrays



Tensors in PyTorch

PyTorch is not the only library that deals with n-dimensional arrays. **NumPy, SciPy, Scikit-learn, Pandas**, and other deep-learning libraries such as **Tensorflow** also support n-dimensional arrays.

Tensors in PyTorch

However, in PyTorch, the Tensor class is more powerful than standard numeric libraries.

- GPU support
- Parallel operations on multiple devices or machines
- Keep track of graph of computations that created them

All these features, especially the last one, are of utmost importance when dealing with deep learning!

We will see why in the next chapters...

Accessing Tensors and their Elements

Tensors are arrays, i.e., **data structures** that store a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices (at most, one index for each dimension).

Tensors vs Python lists

Creating a list and accessing one element

```
[1]  l = [1, 0, 3]
     print(l[0])

1
```

Creating a nested list and accessing one element

```
[ ]  nested_list = [[0, 1, 2],
                    [1, 2, 3]]
     print(nested_list[1][2])

3
```

Tensors vs Python lists

```
[4]
import torch
t = torch.tensor([[0, 1, 2],
                  [1, 2, 3]])
print(t[0, 1]) # indexing tensors (we will see more indexing tricks later)
```

```
tensor(1)
```

Although on the surface this example doesn't differ much from a list of number objects, under the hood things are completely different.

How Tensors are Stored in Memory

- Python lists are collections of objects (also of different types) allocated and stored individually in memory
- PyTorch tensors are allocated contiguously in memory blocks containing C numeric types of 32-bit floats.

Indexing Tensors

```
[4]
import torch
t = torch.tensor([[0, 1, 2],
                  [1, 2, 3]])
print(t[0, 1]) # indexing tensors (we will see more indexing tricks later)
```

```
tensor(1)
```

The indexing operation does not create a new tensor by allocating memory and storing the values in it. That would be very inefficient, especially if we had millions of points.

PyTorch indexing directly references the original tensor.

Fancy Indexing

With tensors, we can use fancy indexing (like NumPy indexing)

```
[ ]
x = torch.tensor([0, 1, 2, 3, 4, 5, 6]) # 1-d tensor
element = x[0] # i-th element
first_elements = x[:3] # from start to element 3
last_elements = x[3:] # from element 3 to the end
some_elements = x[3:5] # from element 1 to element 3
```

Fancy Indexing

Works similarly with 2-d tensors (rows and columns):

```
[ ]
x = torch.tensor([[0, 1, 2], [1, 2, 3]]) # 2-d tensor
element = x[0, 0]
row = x[0, :] # works also with x[0] in this case
column = x [:, 0]
some_rows = x[1:, :] # from row 1 to the end, all columns
some_elements = x[1:2, :1] # from row 1 to 2, from column 0 to 1
```

Same applies to n-d tensors as well!

Tensor element types

Why not using lists or Python numbers?

- The Python interpreter is slow compared to optimized, compiled code.
- PyTorch tensors provide low-level implementations of the data structures and high-level APIs for the operations.
- PyTorch Tensors keep track of the data type in their attribute dtype.
- Possible values of dtype are: torch.float32, torch.float64, torch.int8, torch.uint8, torch.bool*, ...

*particularly useful for indexing!

Handling (and changing) tensor dtypes

```
[12]
double_precision = torch.tensor([0, 1], dtype=torch.double)
print(double_precision.dtype)
short_tensor = double_precision.short()
print(short_tensor.dtype)
bool_tensor = double_precision.bool()
print(bool_tensor.dtype)
```

```
torch.float64
torch.int16
torch.bool
```

Handling (and changing) tensor dtypes

In general you can call the .type method and specify the torch data type (a complete list in the documentation)

```
[20]  1  x = torch.tensor([0, 1], dtype=torch.double)
      2  x.type(torch.uint8)

tensor([0, 1], dtype=torch.uint8)
```

Basic Tensor operations

Creation operations and mutations

```
[21]
a = torch.ones(3, 2) # 3x2 tensor of only ones
b = torch.zeros(3, 1) # 3x1 tensor of only zeros
c = torch.zeros_like(a) # same shape and type as a
a_t = a.t() # 2x3 tensor (transpose of a)
print(a.shape) # prints the shape (i.e., all the sizes of the dimensions)
```

Math operations

```
[22]
absolute_values = torch.abs(a) # pointwise operations
mean_value = torch.mean(a) # reduction operations
s = a + c # element-wise sum
p = a * c # element-wise product
z = torch.mm(a, c.t()) # matrix multiplication (careful with shapes!)
broadcasting = a + torch.tensor([1, 2]) # torch tries to match shapes
```

Boolean indexing

Similarly to NumPy, we can use the boolean tensors to select certain elements of another tensor.

```
[11]
x = torch.tensor([[-4, -1, 2], [1, -2, 3]]) # 2-d tensor

boolean_mask = (x > 0)  # this could be any boolean expression
print(boolean_mask)
print(x[boolean_mask])
```

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

Tensor storage

```
a = torch.tensor([1, 2, 3, 4])
b = a[1] # different Tensor, same storage (points to the same location)
c = a.reshape([2, 2]) # same storage, different stride
print(a.storage())
print(c.storage())
print(a.data_ptr() == c.data_ptr()) # same storage
print(c.stride()) # how many storage items to skip for incrementing each dimension
```

Remember: the underlying memory is allocated only once, which makes the view operation very lightweigth even for large storages.

Modifying stored values: in-place operations

In-place operations are used to modify directly stored values. The most used one is the zero_, that sets to zero all values. They can be recognized by the trailing underscore _ in their name.

```
[ ]
a = torch.ones(3, 2)
a.zero_() # in-place operation, does not create a new tensor
```

The methods that are not in-place, always return a new tensor.

Moving tensors to the GPU

Moving tensors to the GPU can make computations massively parallel and fast.

Then, all the operations will be performed with GPU operations, while the API remains the same.

PyTorch supports all GPUs that have support for **CUDA** (Compute Uniefd Device Architecture), a software layer created by Nvidia.

An accelerated version of PyTorch is also available for Apple Silicon, but it is still not very stable.

Moving tensors to the GPU

Every PyTorch tensor has the attribute device, which says where the tensor data is placed in storage. Tensors can be "moved" (rather, copied) to another device by using the method 'to'.

```
gpu_tensor = torch.zeros(1, device='cuda') # created on the GPU
cpu_tensor = torch.zeros(1)
to_gpu = cpu_tensor.to(device='cuda') # this creates a copy of the tensor!
to_gpu_another = cpu_tensor.cuda() # shorthand for the previous command
again_to_cpu = to_gpu.cpu() # shorthand for copying the tensor to cpu
```

If your machine has more GPUs, you can also specify which one to use, e.g., cuda:0.
Note that operations can be performed only between tensors located on the same device.

Serializing tensors

Until now, we created tensors only in RAM. At some point, we will want to store a tensor in the persistent memory. PyTorch uses pickle to serialize the tensors. Here is how to store a tensor in memory.

```
torch.save(a, 'tensor.pth') # note that the extension is arbitrary
```

And to load back the tensor, a similar API is available.

```
b = torch.load('tensor.pth')
```

PyTorch Autograd Engine

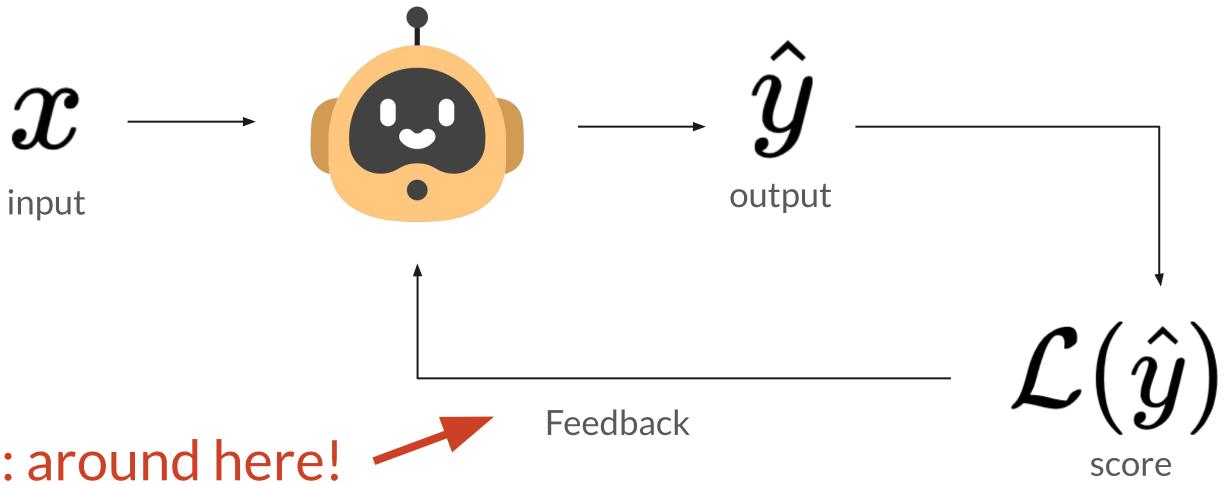
PyTorch Autograd Engine

As we will see in the next chapters, in deep learning we need to obtain the **gradients**.

PyTorch computes the gradient of any differentiable function w.r.t. their inputs by using **automatic differentiation** (even for extremely complex functions!)

This PyTorch component is called **autograd**.

Where DL use gradients?



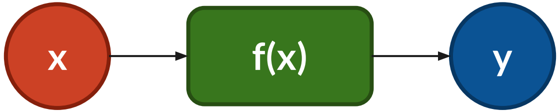
Spoiler: around here!

The Computational Graph

When operating with tensors, PyTorch automatically build the corresponding **computational graph** by keeping track of every interaction between tensors.

In particular **each node** remembers:

- the **parent tensors** that originated it
- the **operation** performed on the parent tensors



Getting the gradients (in theory)

In DL we usually want to modify a variable by changing one of the previous ones.

For example, how does modifying the variable **y** influence the variable **x**?

We can find this information in the **partial derivative** $\dfrac{\partial y}{\partial x}$

Getting the gradients (in PyTorch)

In PyTorch, each operation block f(x) contains the **rule to compute the partial derivative**, and

by calling the .backward() method on y

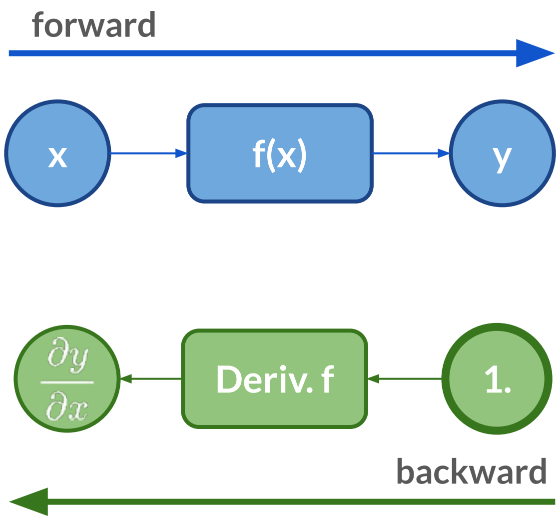
to the gradient field inside x, initialized empty, will be **added** the **value of the derivative.**

Forward and backward

In PyTorch, we call **forward pass** the series of operations that start from the inputs to the output nodes of the computational graph.

Instead, we call **backward pass** the series of multiplications of partial derivatives from the root from which we call the .backward() method to the inputs (i.e. leaves of the computational graph).

This procedure is nothing else than the implementation of the **backpropagation**, the algorithm that made possible to train neural networks!

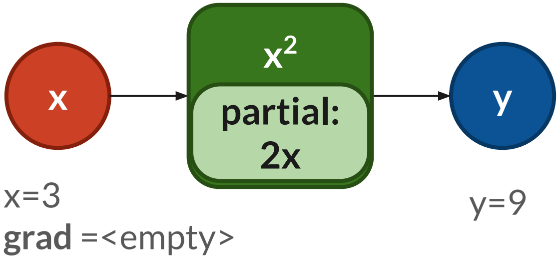


Example

$$y = f(x) = x^2$$

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f(x=3)}{\partial x} = ?$$




```python
x = torch.tensor(3., requires_grad=True)
print(x.grad)
y = x**2     # forward pass: here we define and use the function
print(y)
y.backward()     # backward pass
print(x.grad)
```

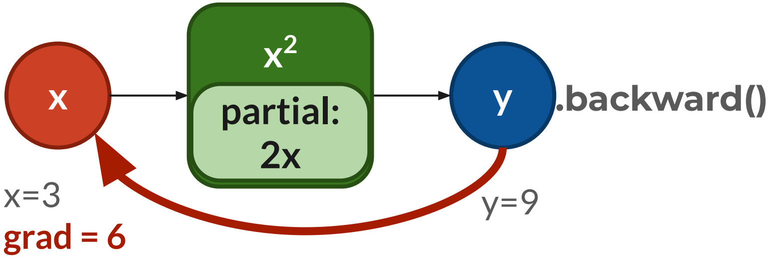
```
None
tensor(9., grad_fn=<PowBackward0>)
```

Example

$$y = f(x) = x^2$$

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f(x=3)}{\partial x} = 6$$




```
x = torch.tensor(3., requires_grad=True)
print(x.grad)
y = x**2    # forward pass: here we define and use the function
print(y)
y.backward()    # backward pass
print(x.grad)
```

```
None
tensor(9., grad_fn=<PowBackward0>)
tensor(6.)
```

Accumulated gradients

Remember that every time we call forward and backward, the gradients are not overwritten but accumulated.

This is an important property needed for complex operations, but if disregarded can lead to wrong results.

To fix this issue is sufficient to call the .zero_() method on the grad attribute

```
[36]
y = x**2
y.backward()
print(x.grad)
```

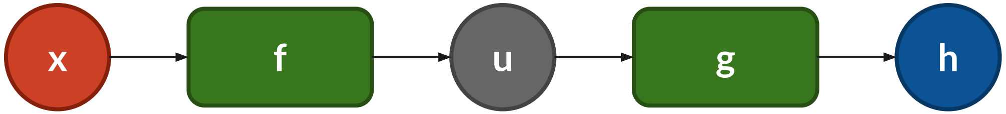
```
tensor(24.)
```

```
y = x**2
y.backward()
print(x.grad)
x.grad.zero_()
```

```
tensor(6.)
tensor(0.)
```

Cascading operations

What if the computational graph starts becoming bigger?



How can we compute the gradient $\frac{\partial h}{\partial x}$?

We know from calculus courses that a cascade of operations can be formally expressed as a **composite function** $h = g(f(x))$

Chain Rule

The derivative of a composite function can be computed using the **chain rule**:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial u}\frac{\partial u}{\partial x}$$

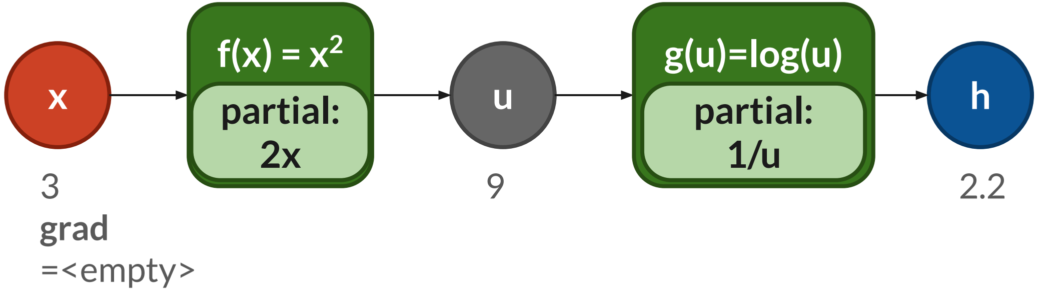
If $u = f(x)$ is the output of $f(x)$, then we can compute the derivative of the composed function as the product of the derivative of the "external" function w.r.t. u and the derivative of the "internal" function w.r.t. x.

Example

$$u = f(x) = x^2$$

$$h = g(u) = log(u)$$

$$\frac{\partial h}{\partial x} = ?$$




```
x = torch.tensor(3., requires_grad=True)
print(x.grad)
y = (x**2).log()     # forward (only part of the code that changed)
print(y)
y.backward()     # backward
print(x.grad)
```

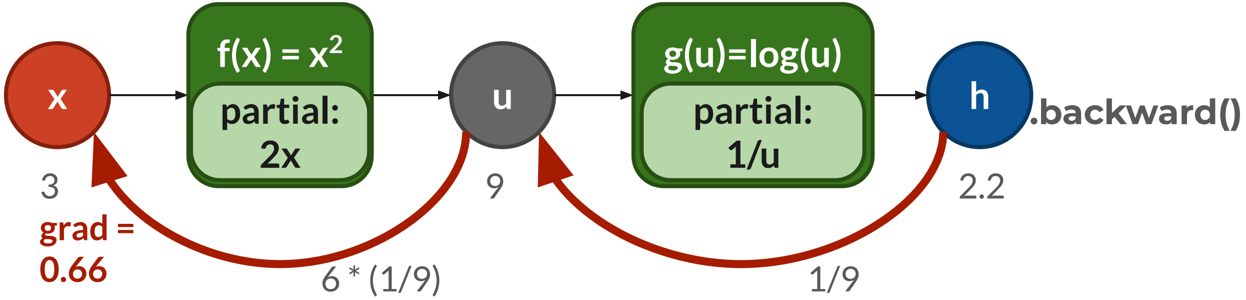
```
None
tensor(2.1972, grad_fn=<LogBackward0>)
```

Example

$$u = f(x) = x^2$$

$$h = g(u) = log(u)$$

$$\frac{\partial h}{\partial x} = 0.66$$



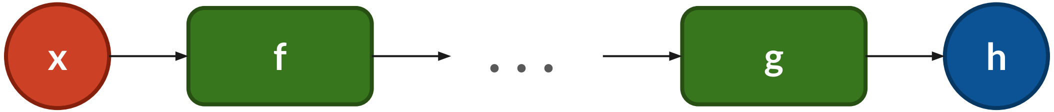

```
x = torch.tensor(3., requires_grad=True)
print(x.grad)
y = (x**2).log()     # forward (only part of the code that changed)
print(y)
y.backward()     # backward
print(x.grad)
```

```
None
tensor(2.1972, grad_fn=<LogBackward0>)
tensor(0.6667)
```

Scaling up...

Thanks to the autograd engine, we can compute gradients from any variable with respect to any other one in an efficient and simple way.

This is of paramount importance when we will deal with large deep learning models!



Tensor API

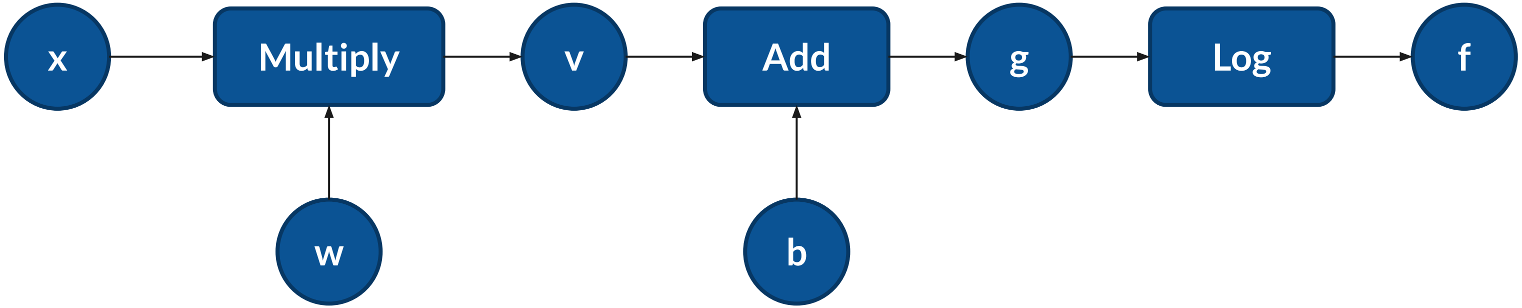
Your first source of information should be the [PyTorch documentation].

- more complete
- more updated
- (it might also say something different than these slides!)

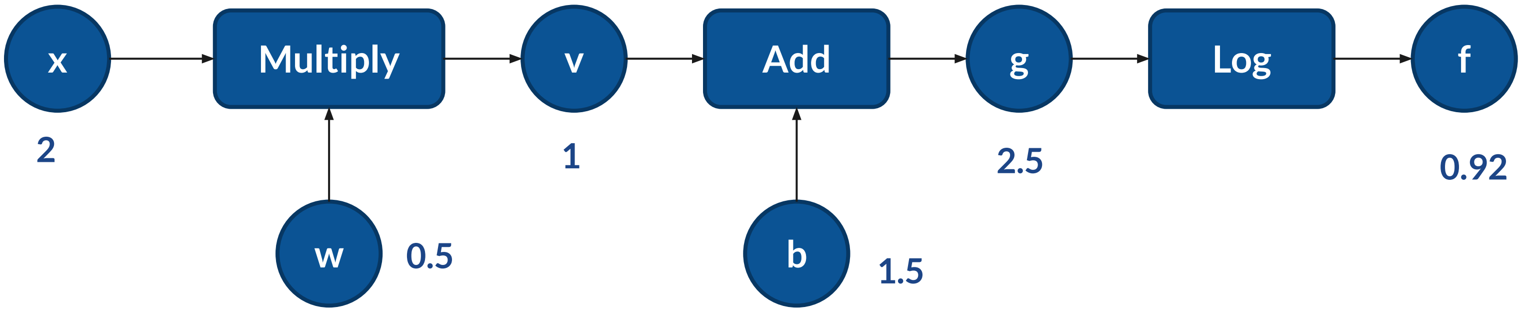
Exercise

1. Write the **computational graph** for the function **f = log(w * x + b)**
2. Compute the **forward** pass setting **x=2**, **w=0.5** and **b=1.5**
3. Compute the **backward** pass **from f to w** (the derivative of **f** with respect to **w**)

1. Computational graph



2. Forward pass



3. Backward pass