



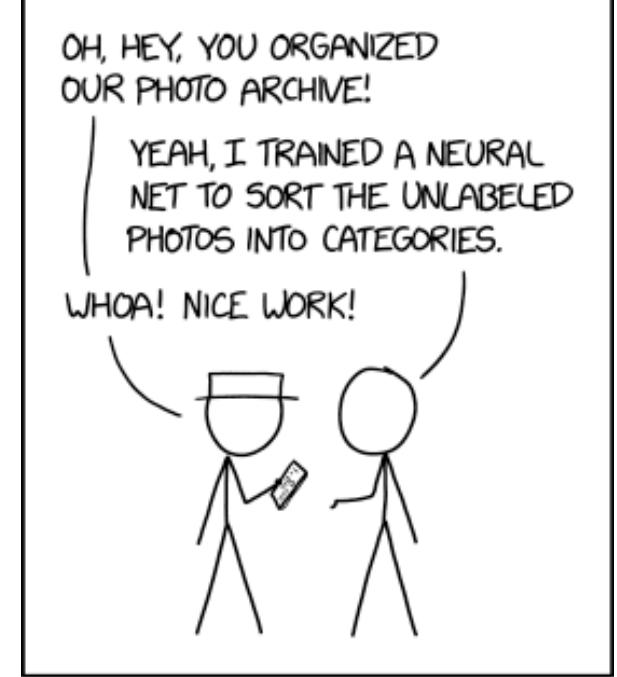
# ➤ Refresher on Neural Networks and Philosophy of pytorch

Alberto TONDA, Ph.D. (Senior permanent researcher, DR)

*UMR 518 MIA-PS, INRAE, AgroParisTech, Université Paris-Saclay  
UAR 3611, Institut des Systèmes Complexes de Paris Île-de-France*

# > Outline

- What is a neural network?
- Forward pass
- Backward pass
- Philosophy of pytorch
- Tensors
- Modules
- Optimizers



OH, HEY, YOU ORGANIZED OUR PHOTO ARCHIVE!  
YEAH, I TRAINED A NEURAL NET TO SORT THE UNLABELED PHOTOS INTO CATEGORIES.

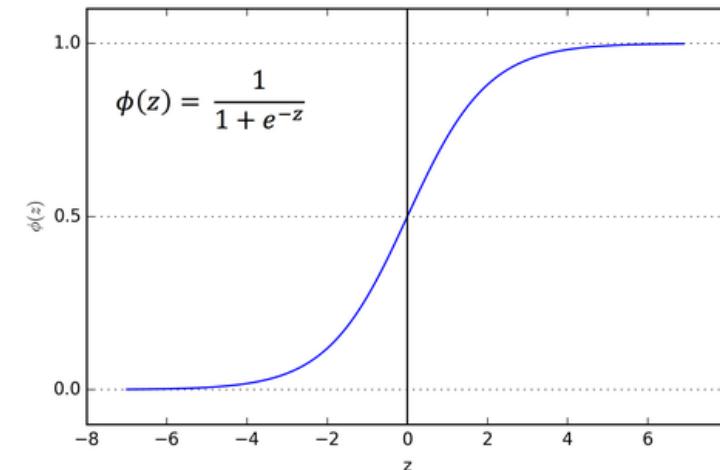
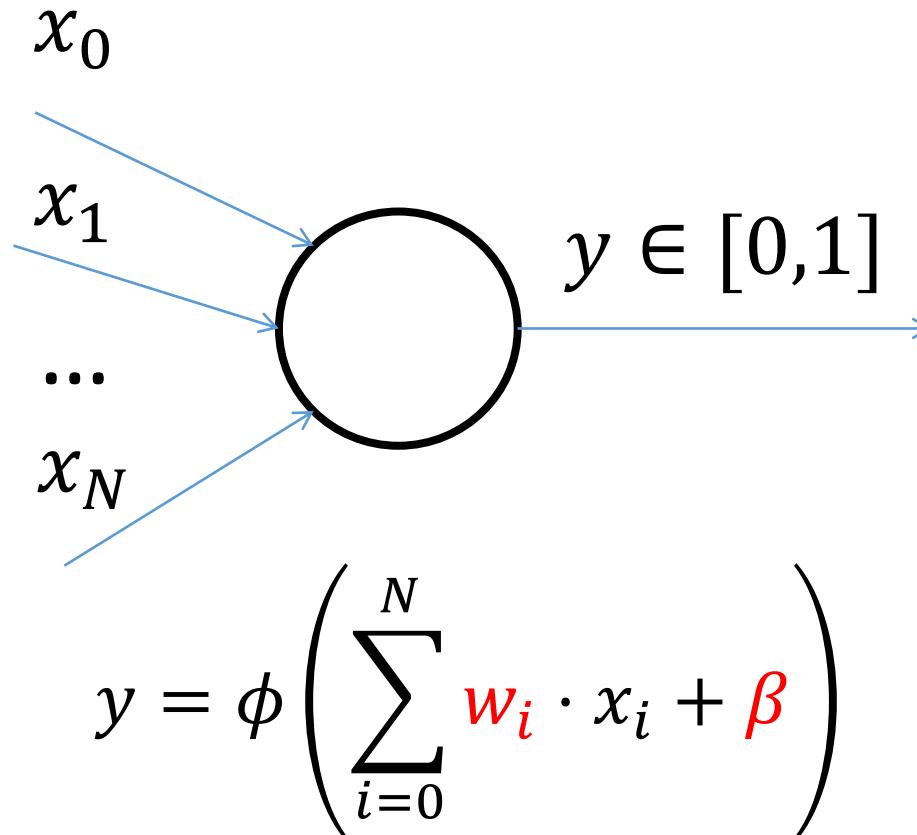
WHOA! NICE WORK!



ENGINEERING TIP:  
WHEN YOU DO A TASK BY HAND,  
YOU CAN TECHNICALLY SAY YOU  
TRAINED A NEURAL NET TO DO IT.

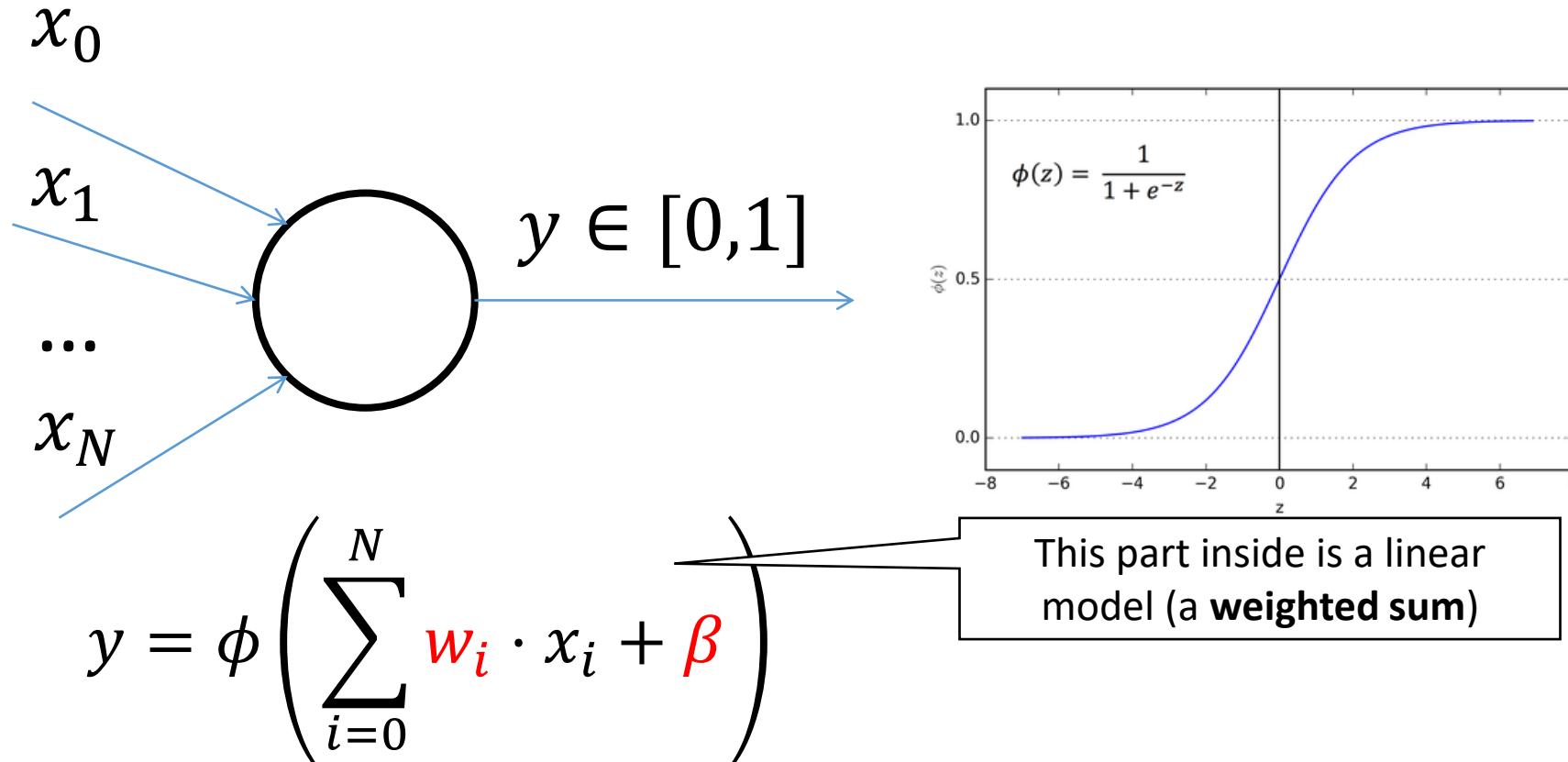
# > What is a neural network?

- Everything starts from the (wrong) model of a neuron



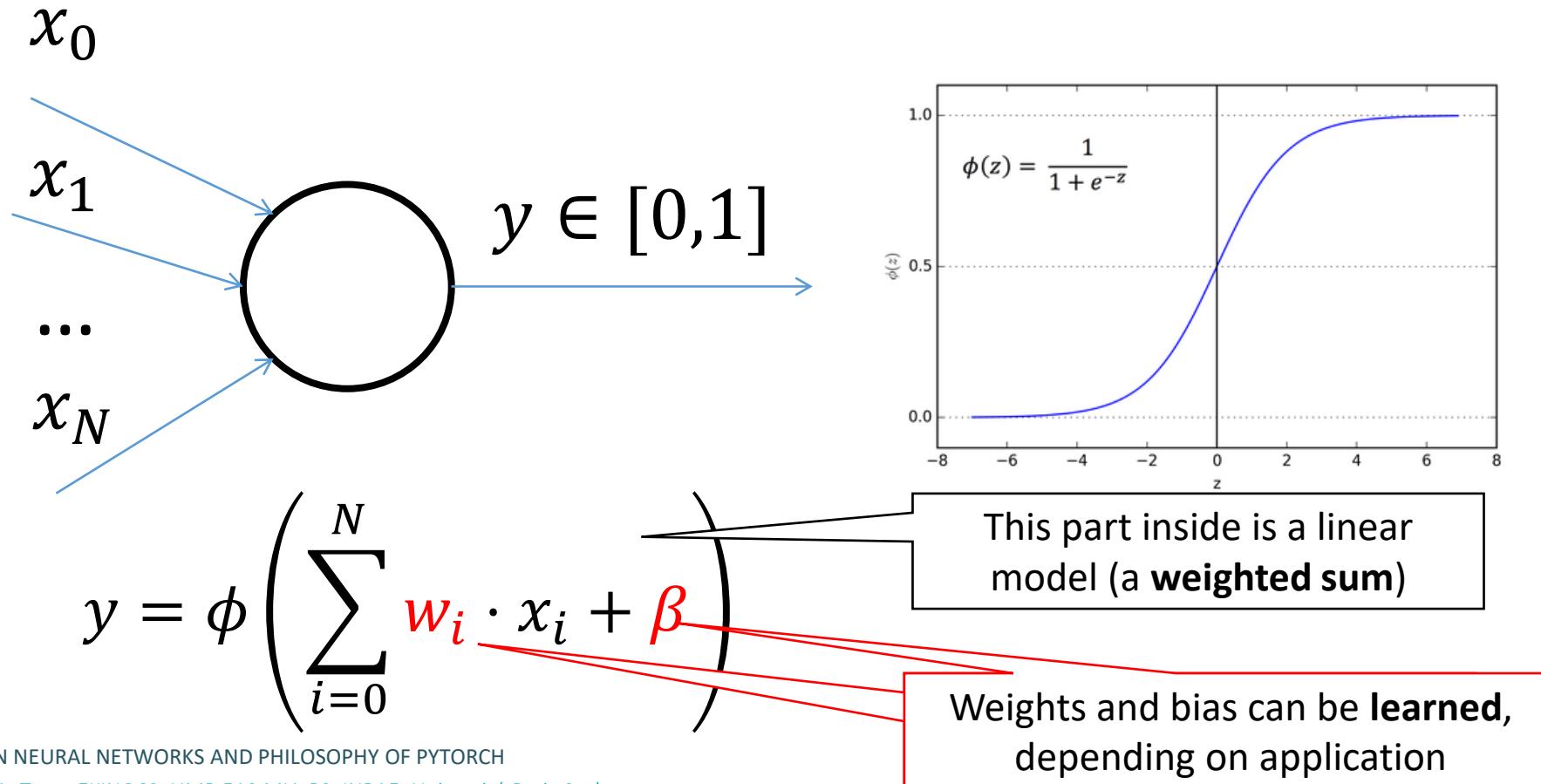
# > What is a neural network?

- Everything starts from the (wrong) model of a neuron



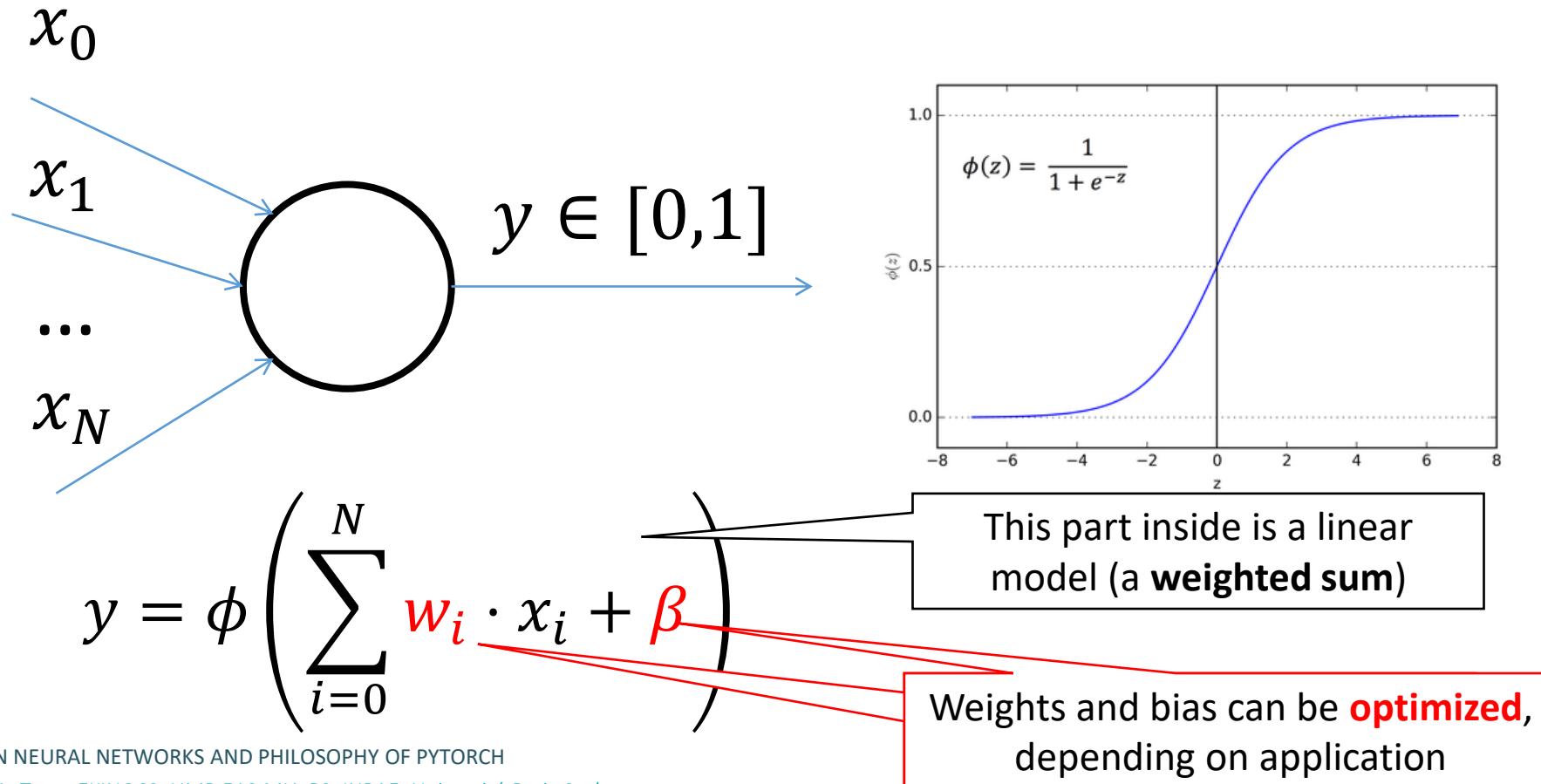
# > What is a neural network?

- Everything starts from the (wrong) model of a neuron



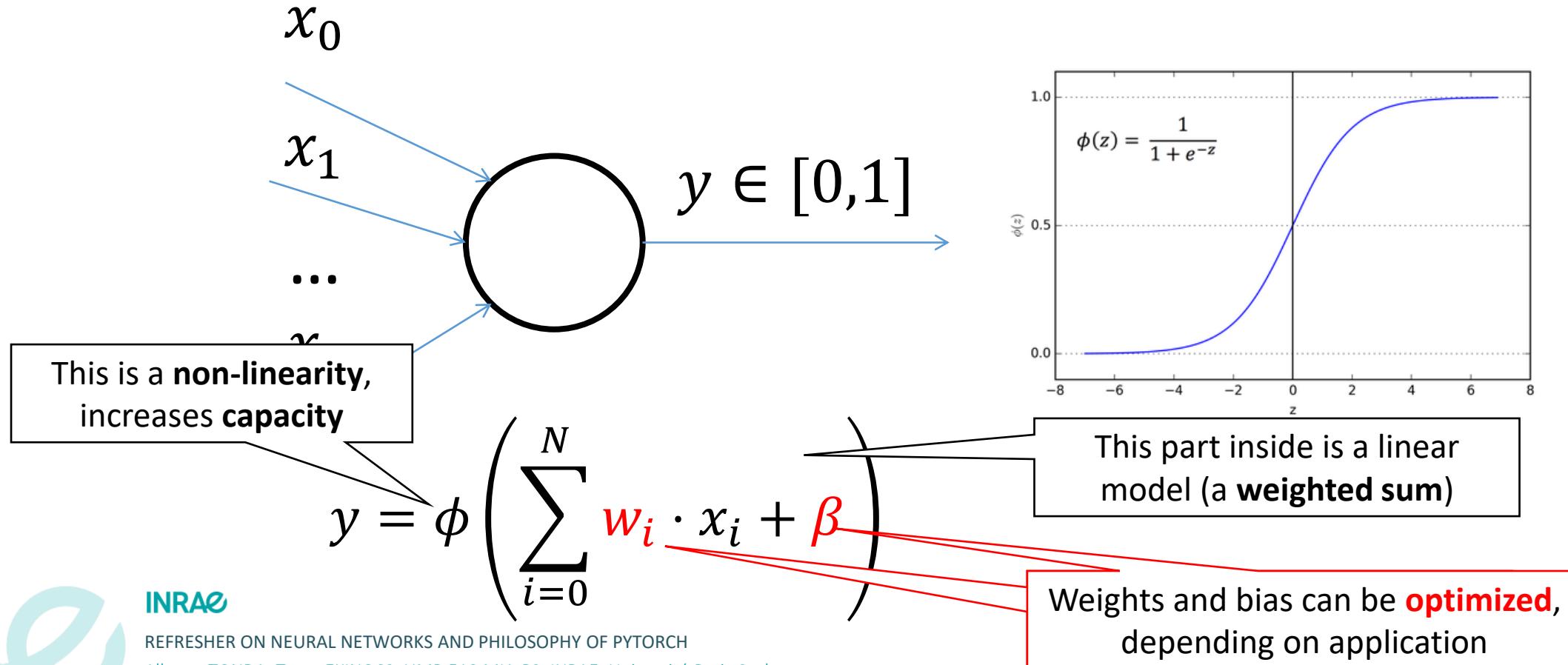
# > What is a neural network?

- Everything starts from the (wrong) model of a neuron



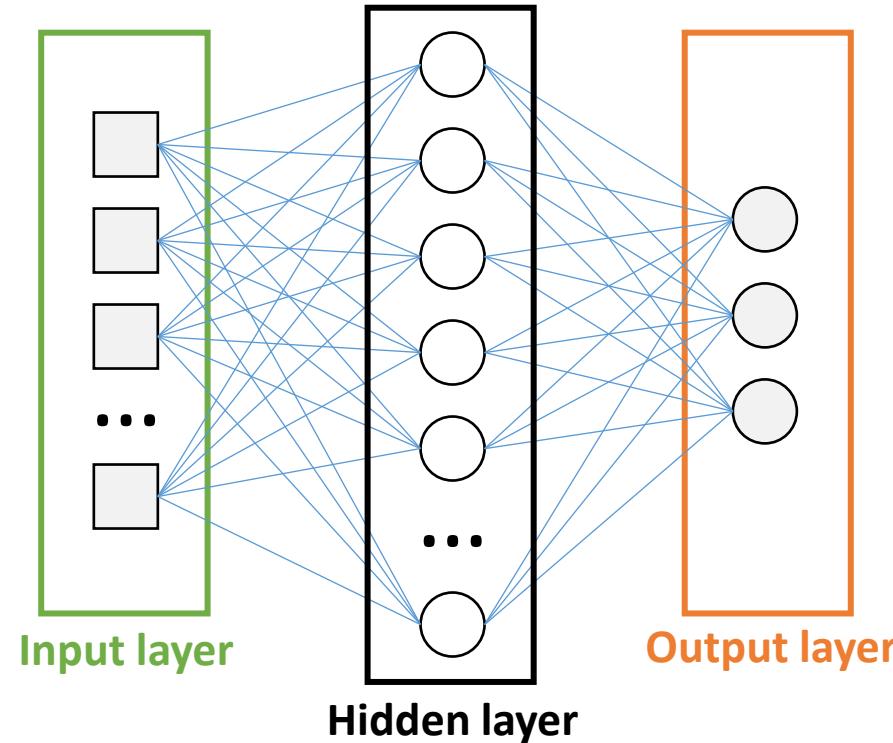
# > What is a neural network?

- Everything starts from the (wrong) model of a neuron



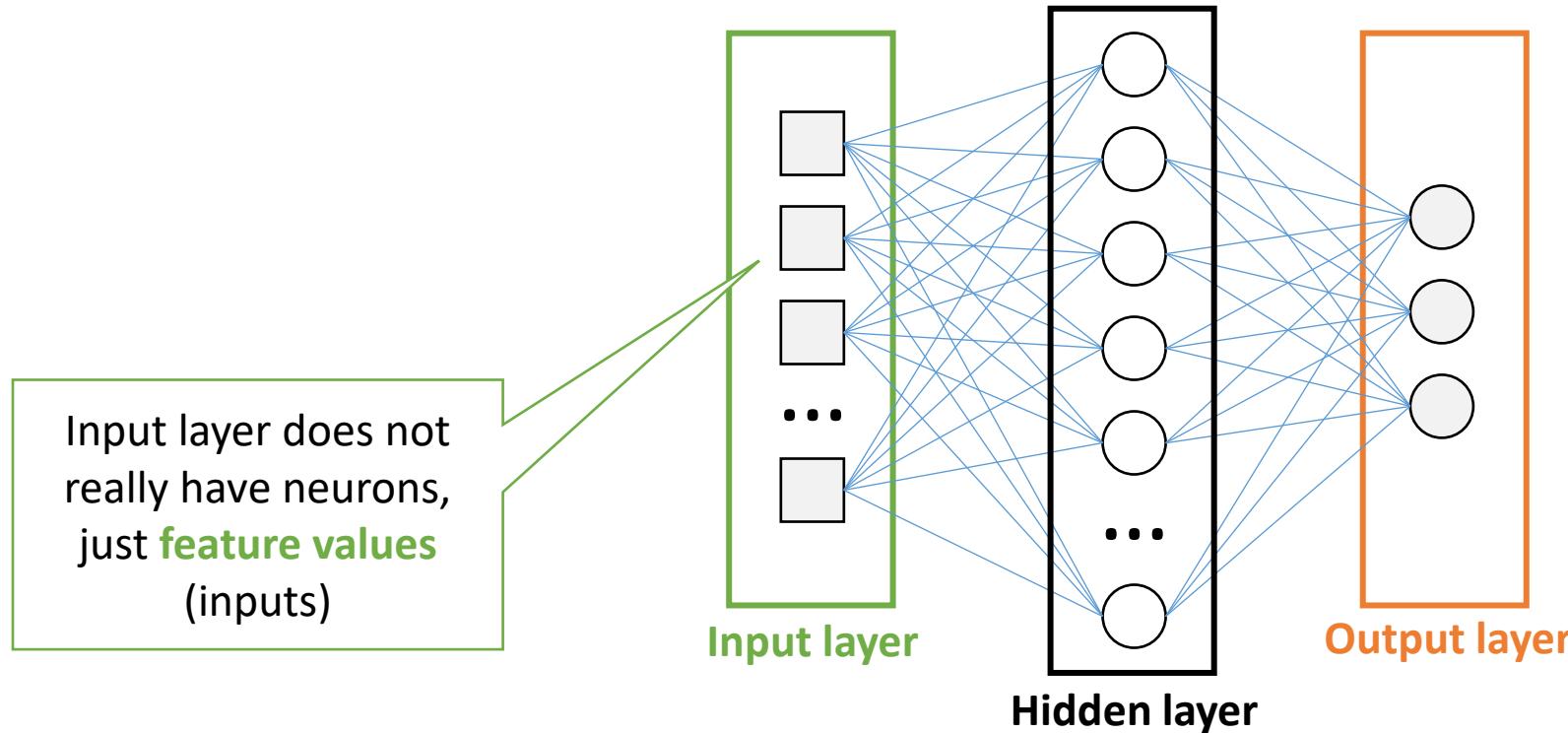
# > What is a neural network?

- Neurons are used in subsequent sets, called *layers*



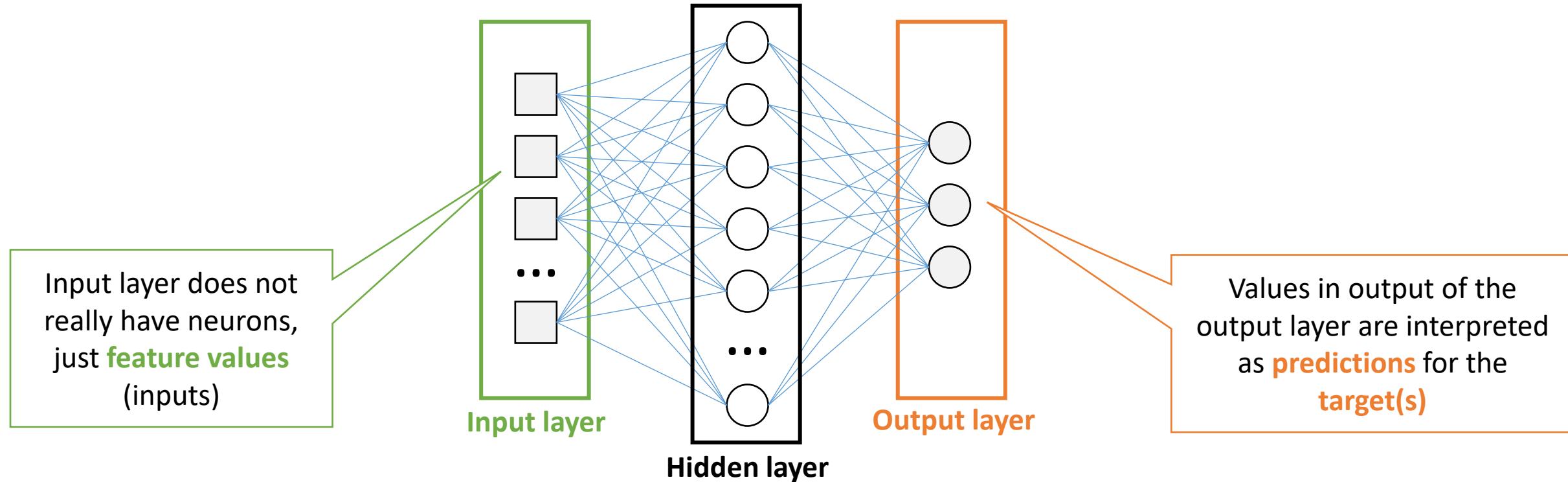
# > What is a neural network?

- Neurons are used in subsequent sets, called *layers*



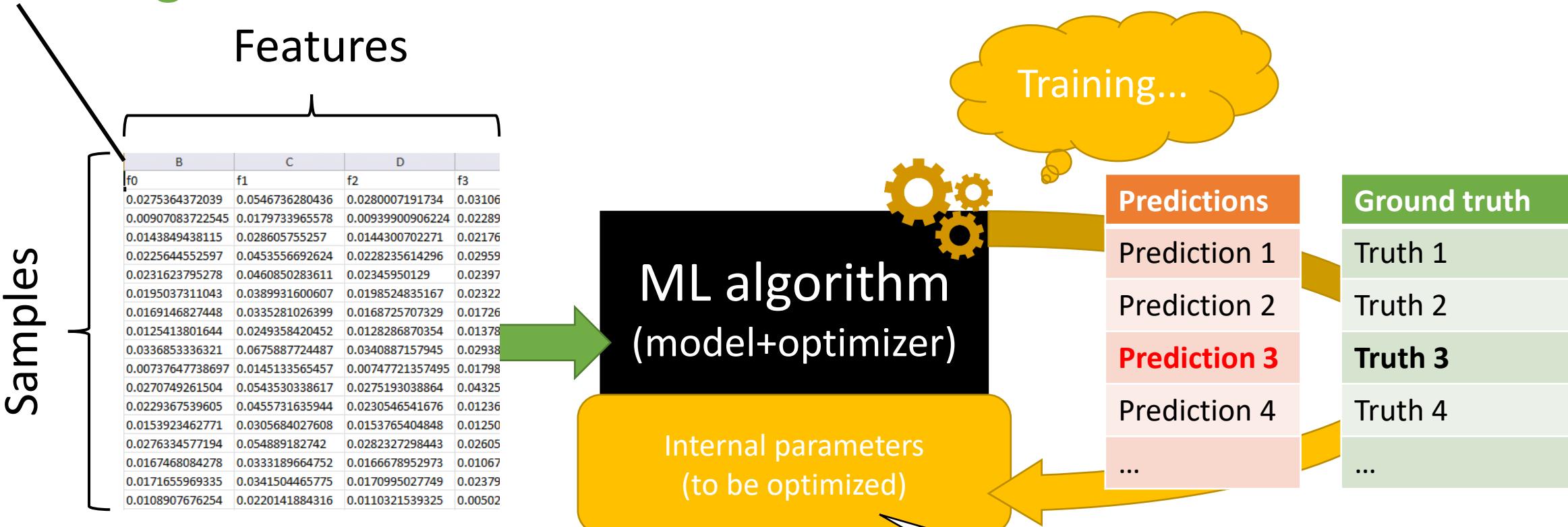
# > What is a neural network?

- Neurons are used in subsequent sets, called *layers*



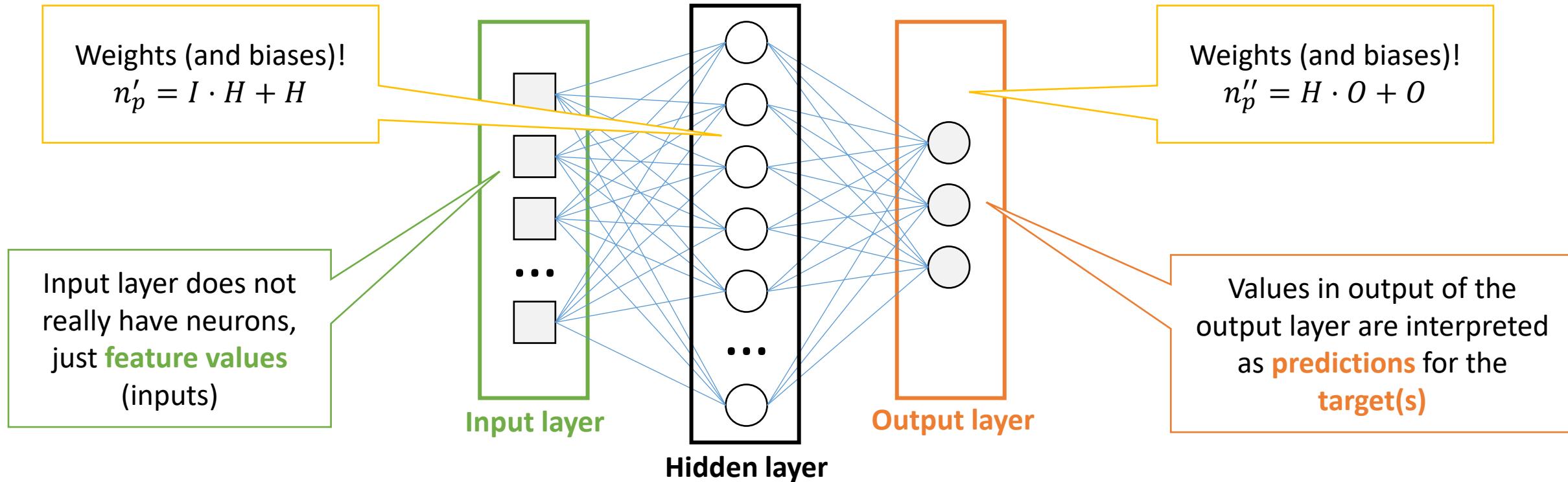
# > Machine learning (supervised)

## Training data



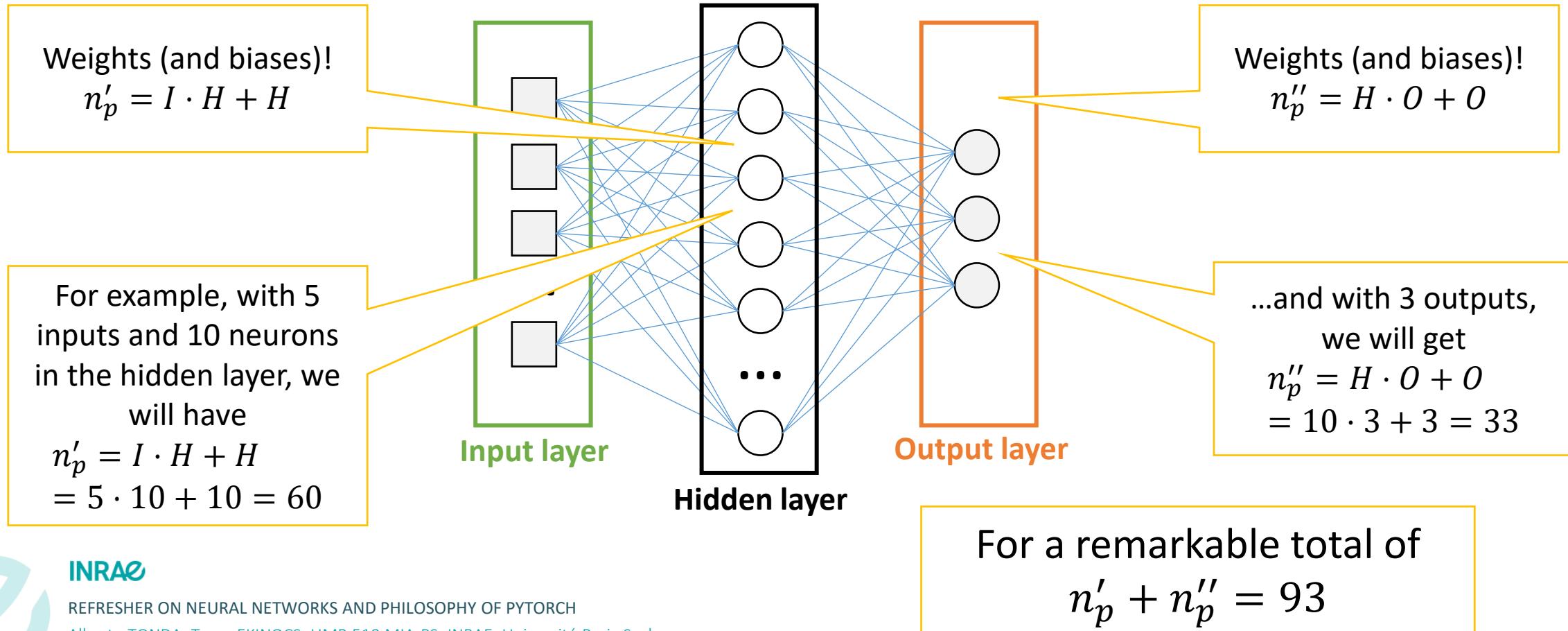
# > What is a neural network?

- Neurons are used in subsequent sets, called *layers*



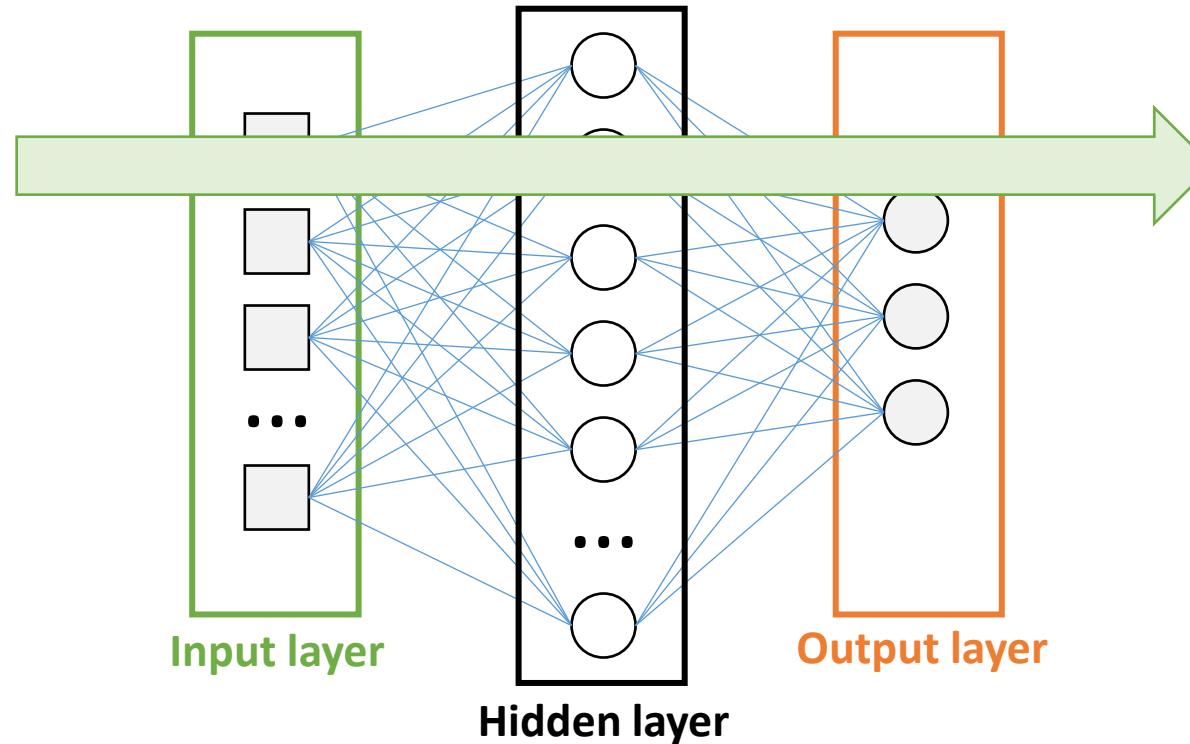
# > What is a neural network?

- Neurons are used in subsequent sets, called *layers*



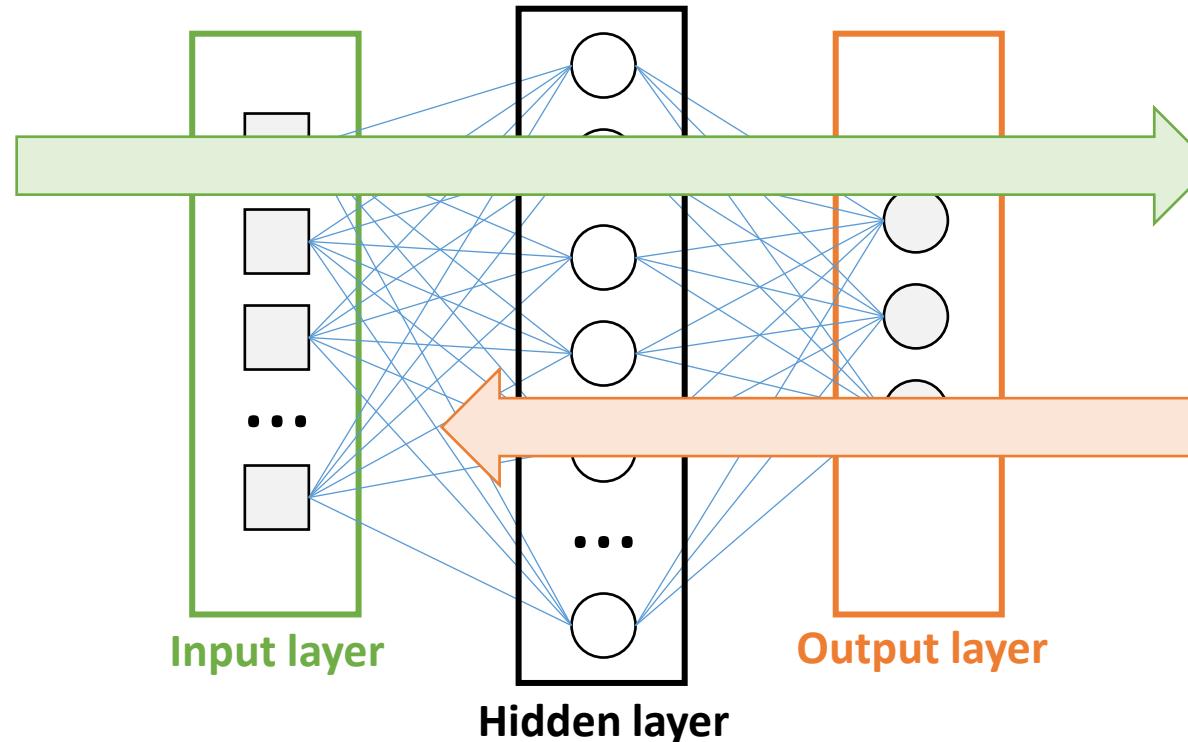
# > What is a neural network?

- Prediction is performed with a **forward pass (inference)**

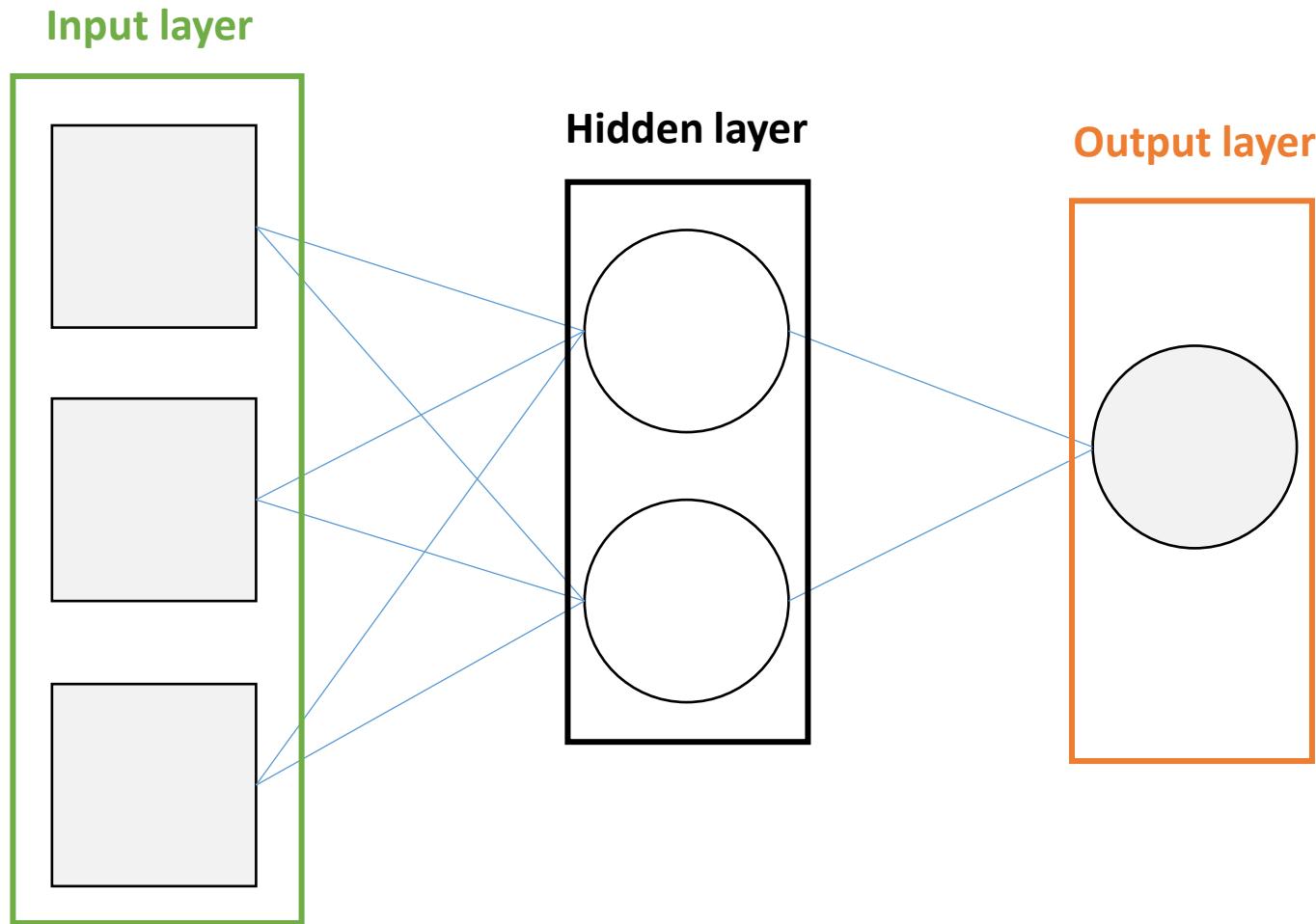


# > What is a neural network?

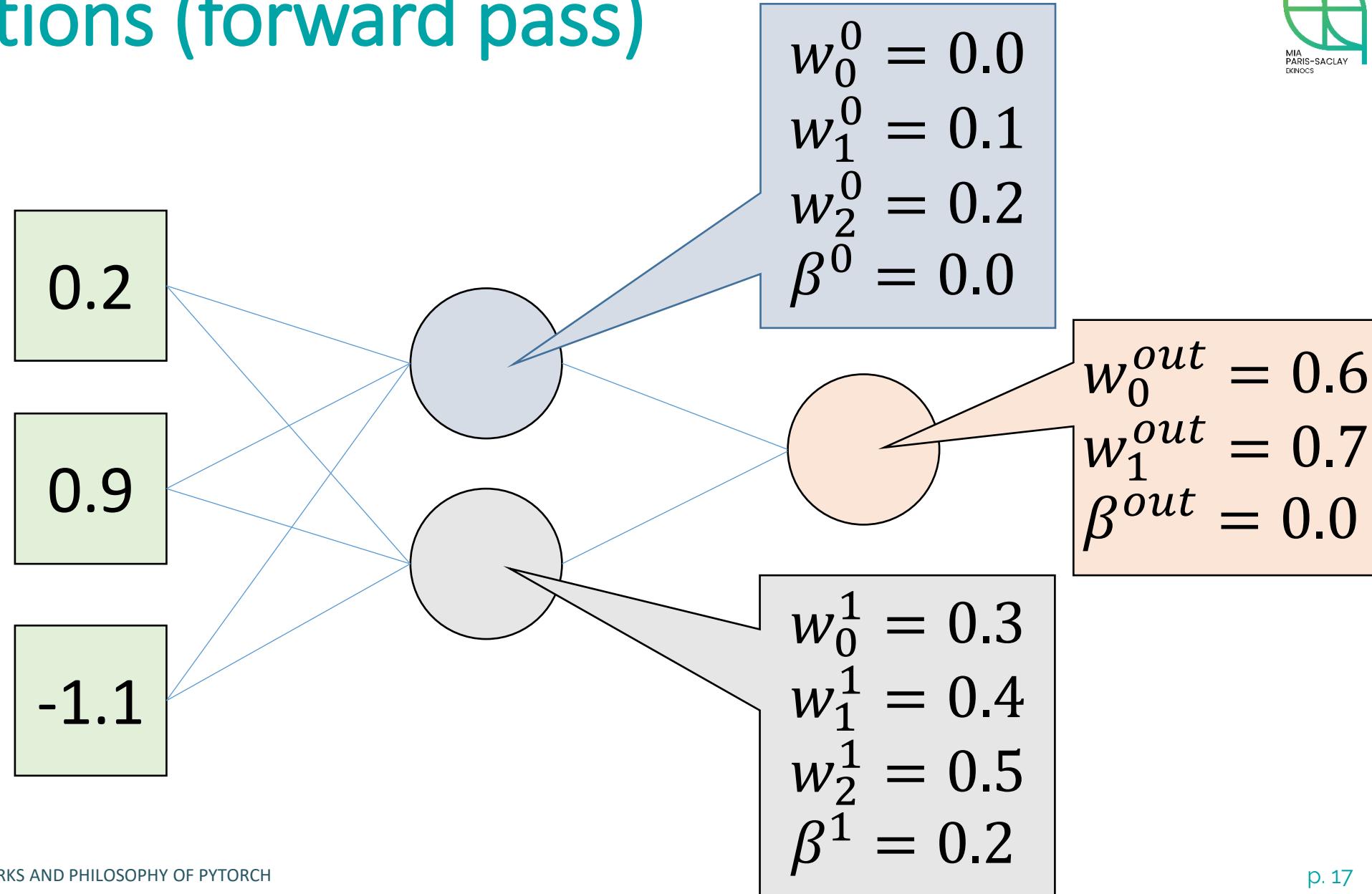
- Learning is performed with a **backward pass**



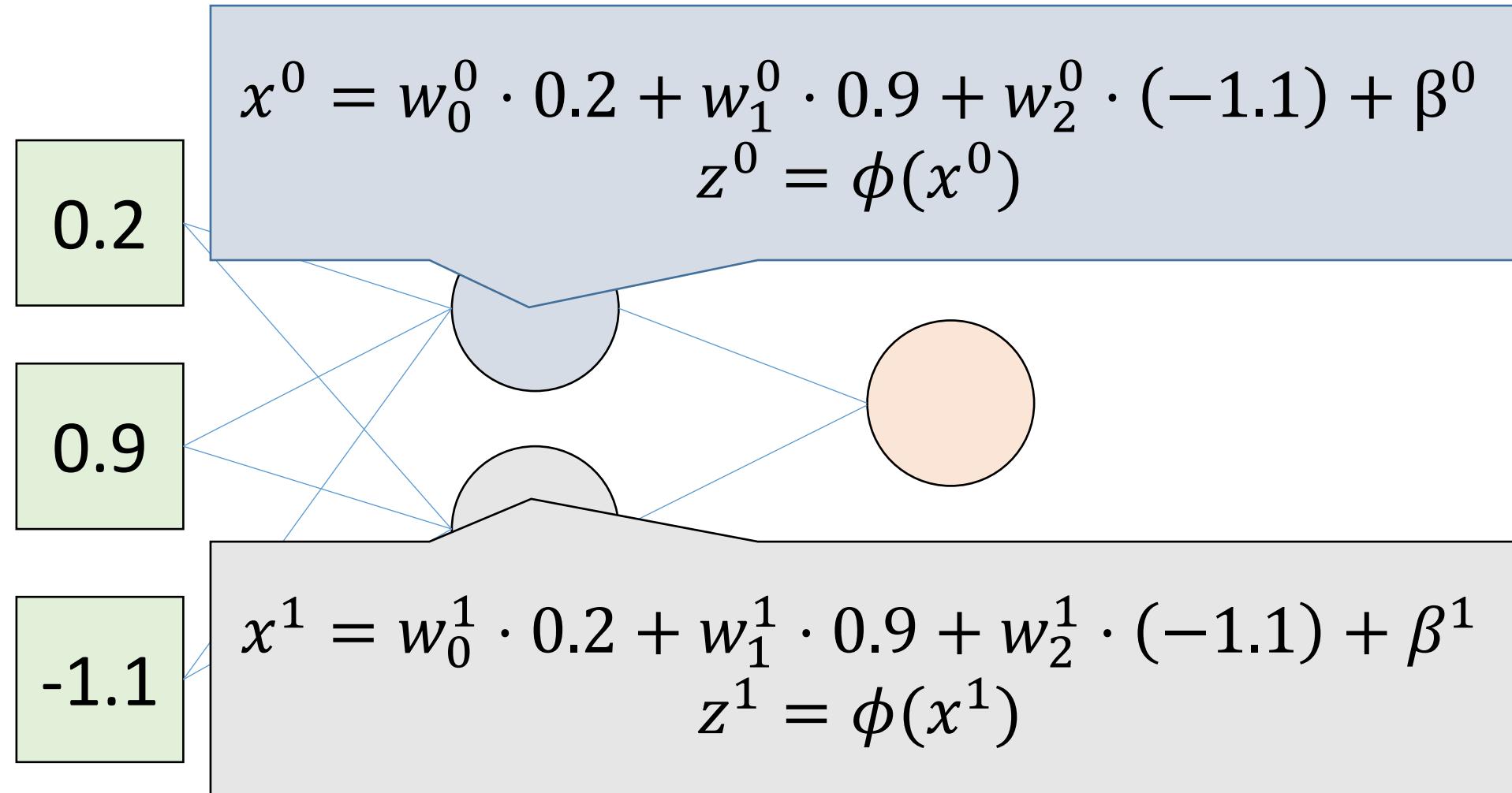
# > Predictions (forward pass)



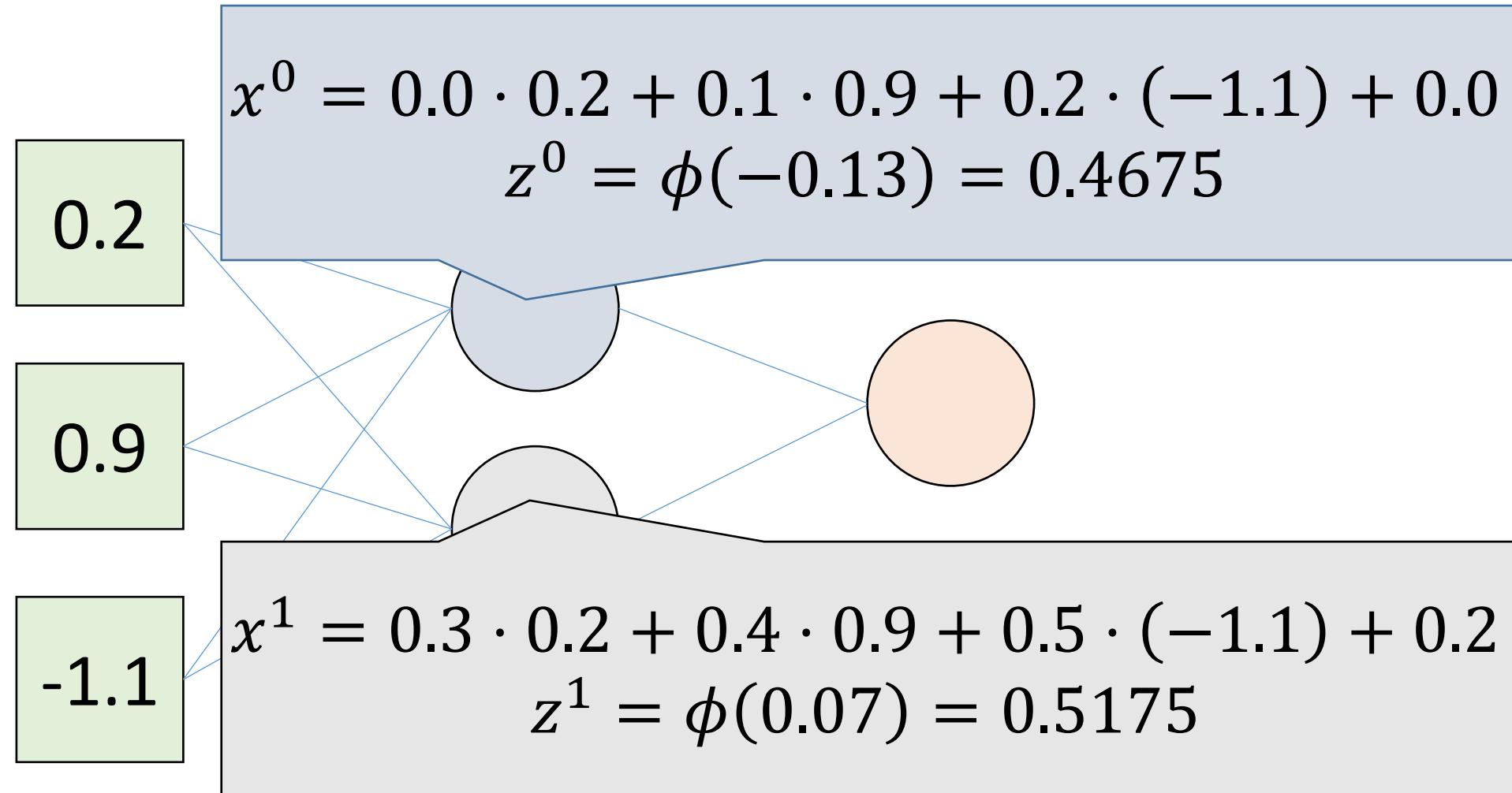
# > Predictions (forward pass)



# > Predictions (forward pass)

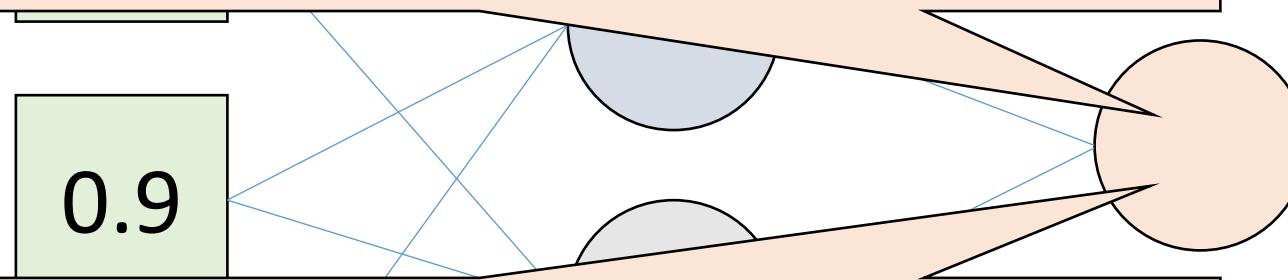


# > Predictions (forward pass)



# > Predictions (forward pass)

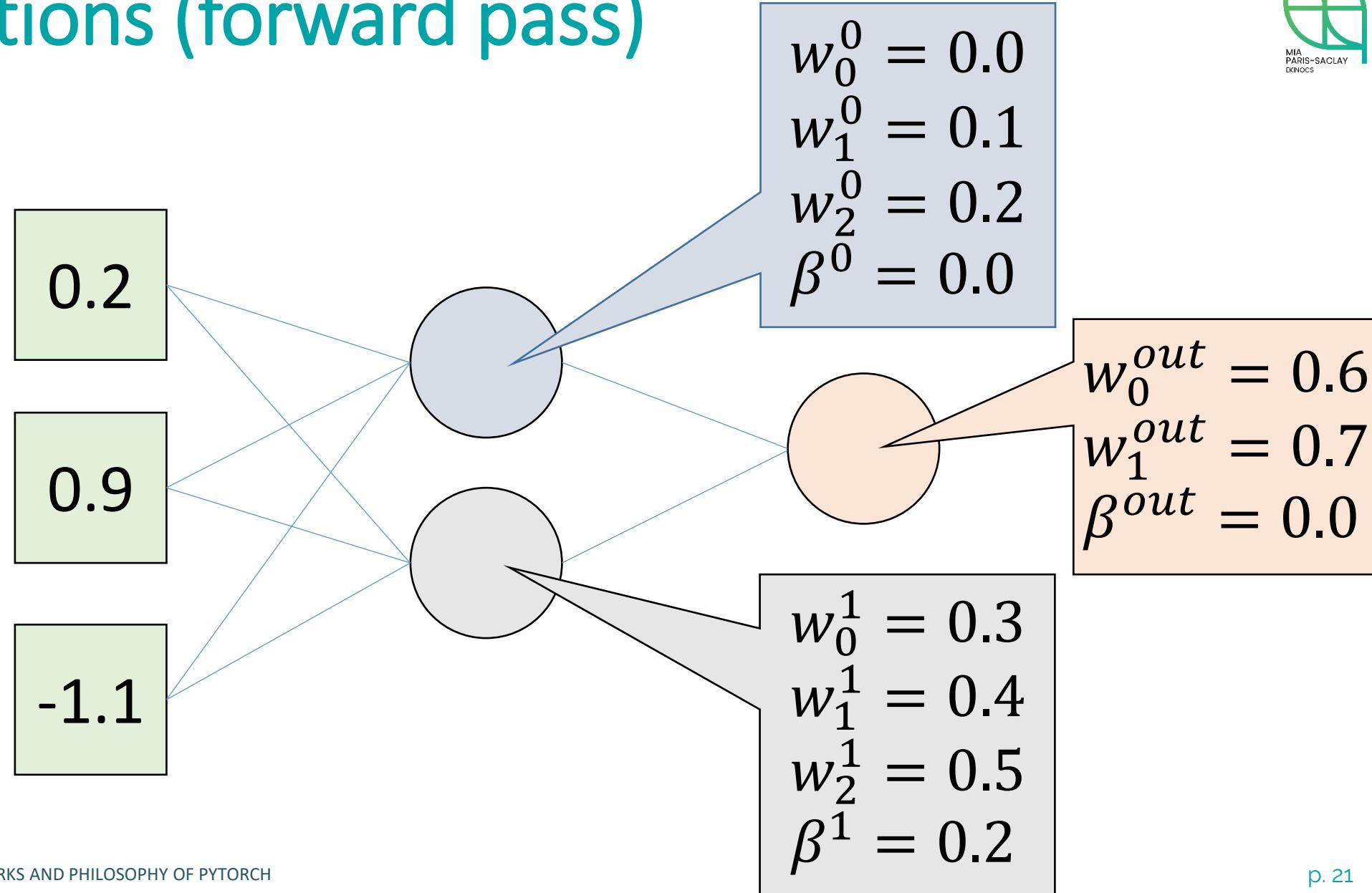
$$\hat{y} = w_0^{out} \cdot z^0 + w_1^{out} \cdot z^1 + \beta^{out}$$



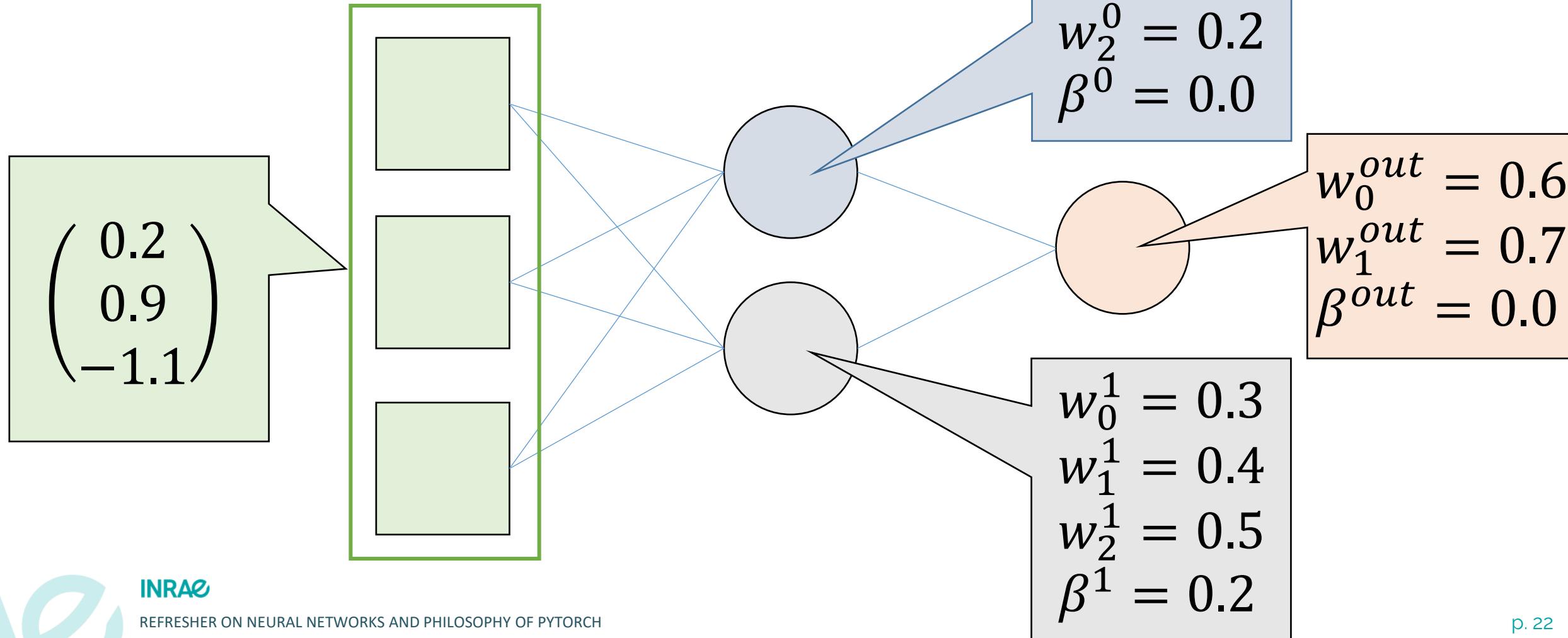
$$\hat{y} = 0.6 \cdot 0.4675 + 0.7 \cdot 0.5175 + 0.0$$

$$\hat{y} = 0.6428$$

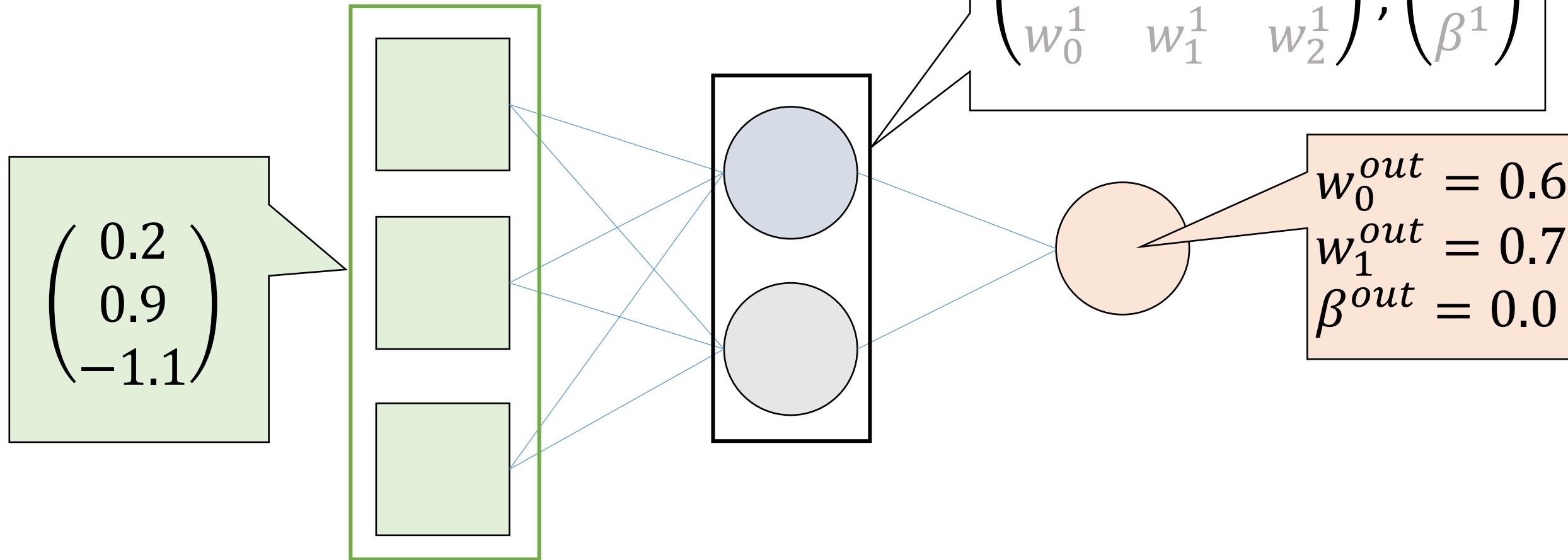
# > Predictions (forward pass)



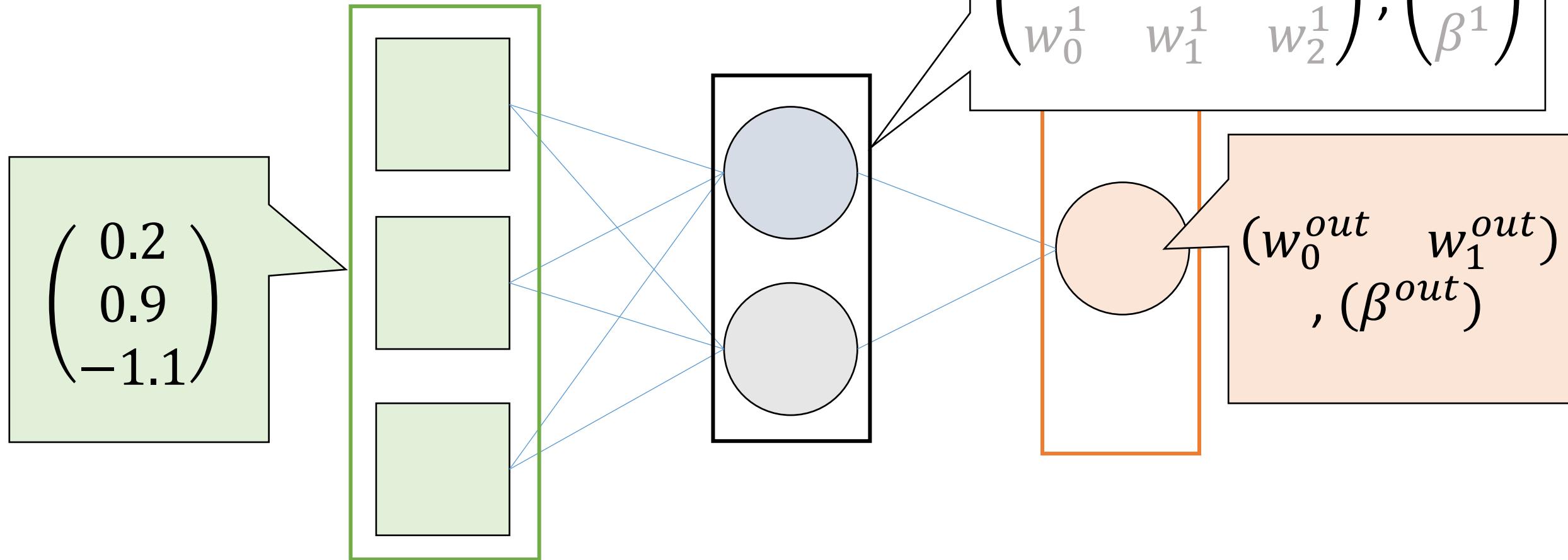
# > Predictions (forward pass)



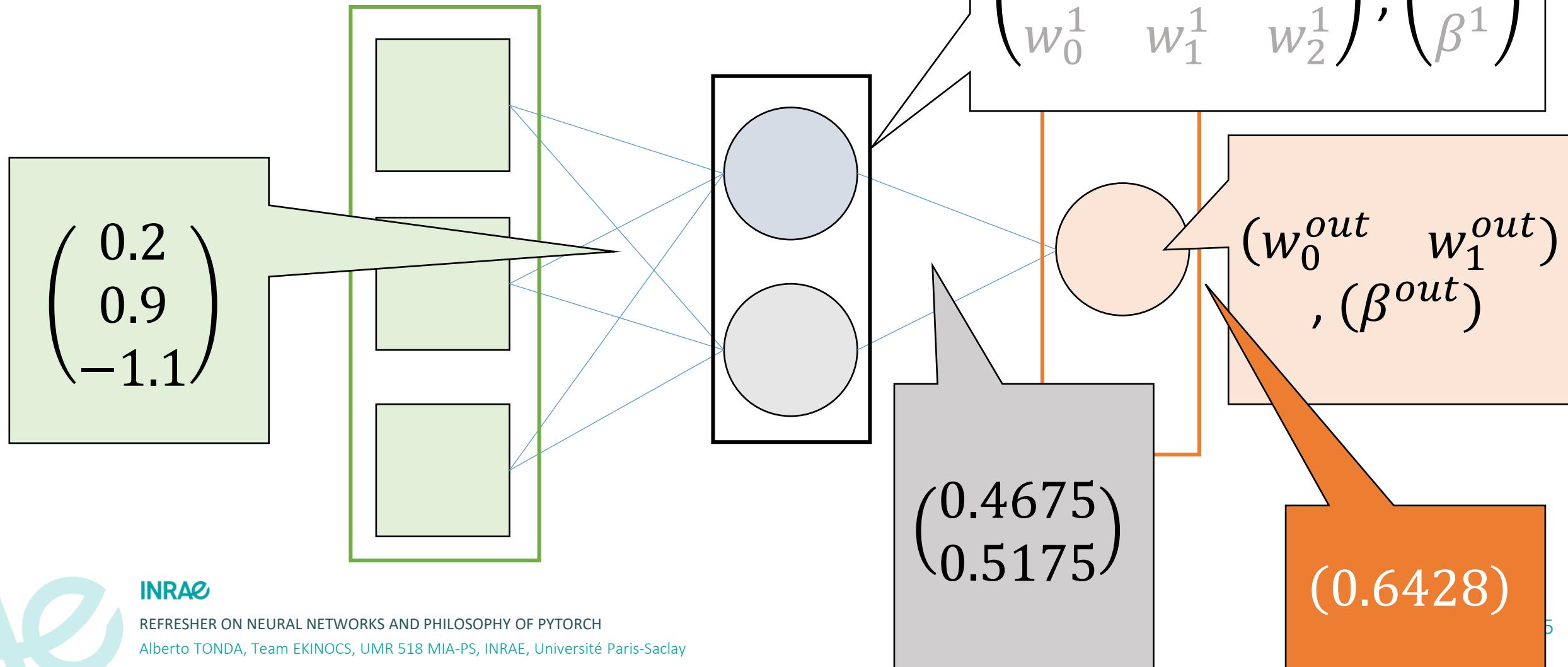
# > Predictions (forward pass)



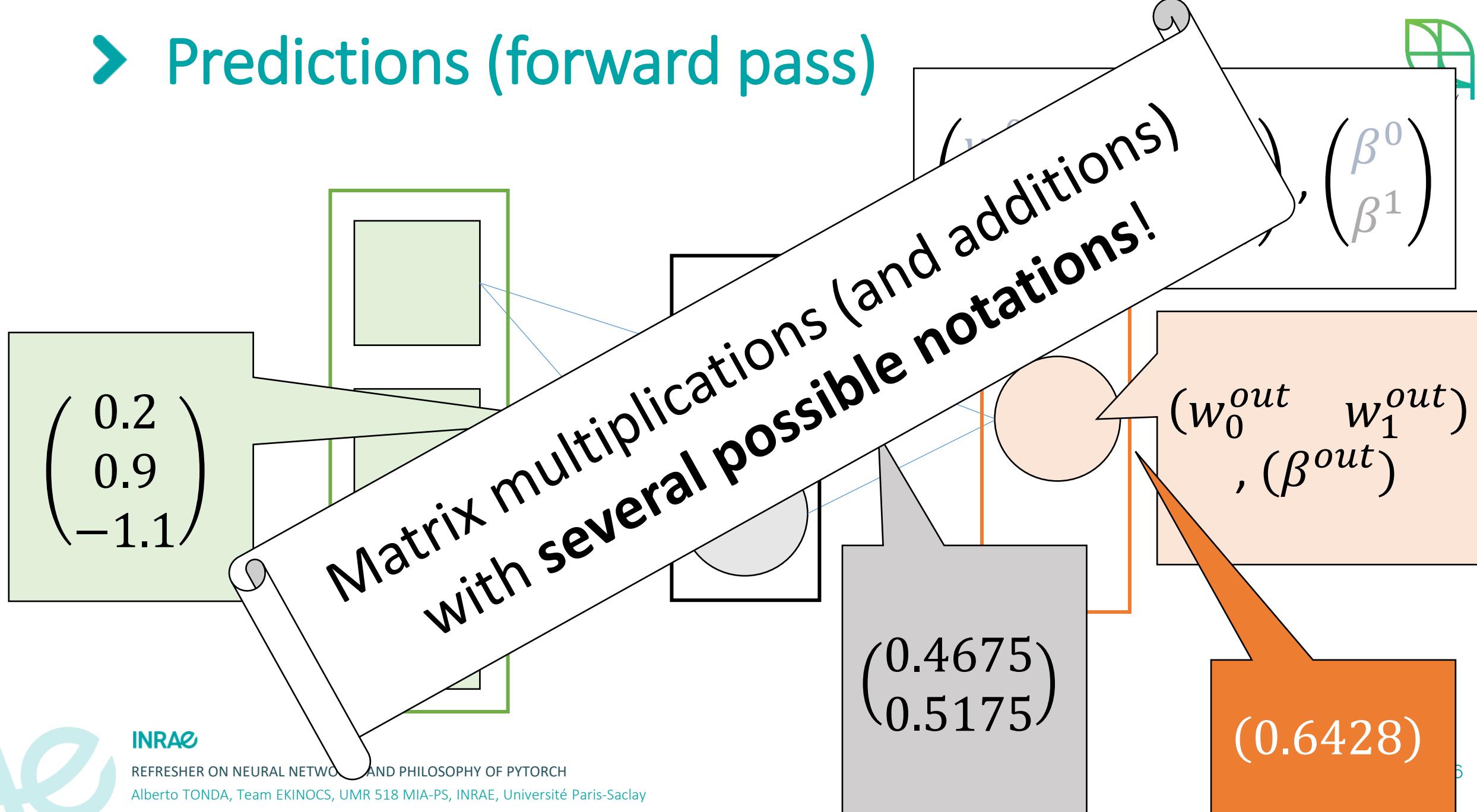
# > Predictions (forward pass)



# > Predictions (forward pass)



## > Predictions (forward pass)



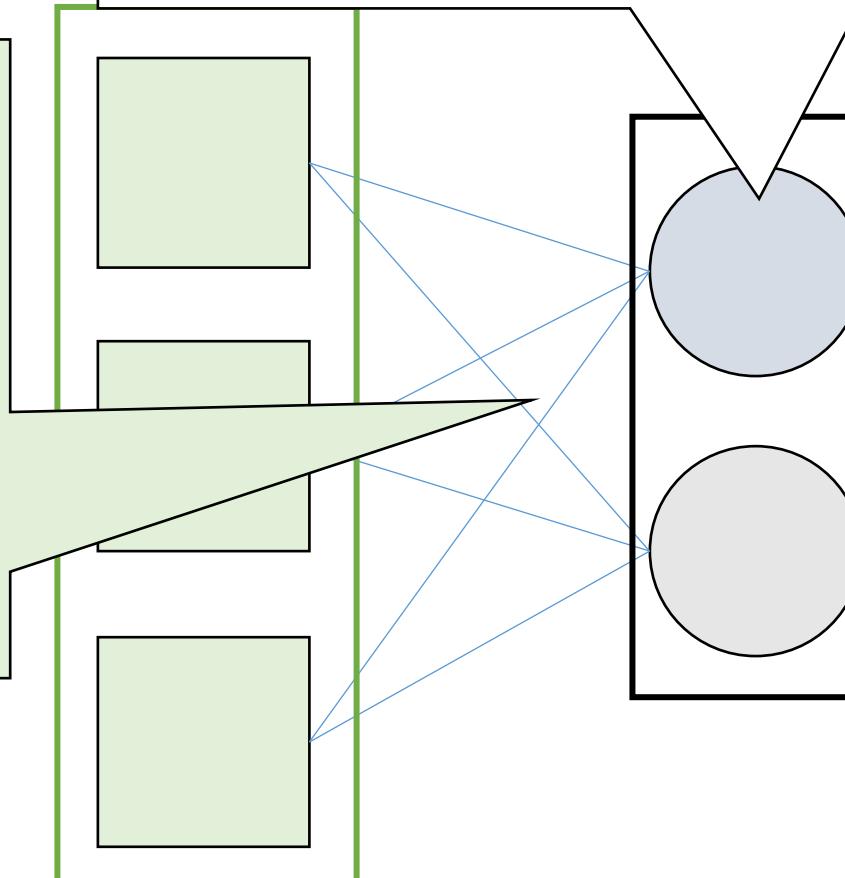
# > Predict

Multiplication by a matrix of shape=(2,3)

$$z = \phi(Wx + B)$$

$$\begin{pmatrix} 0.2 \\ 0.9 \\ -1.1 \end{pmatrix}$$

Column vector  
shape=(3,1)



$(0.4675)$   
 $(0.5175)$   
Output is a column vector of shape=(2,1)

$$\begin{pmatrix} w_0^0 & w_1^0 & w_2^0 \\ w_0^1 & w_1^1 & w_2^1 \end{pmatrix}, \begin{pmatrix} \beta^0 \\ \beta^1 \end{pmatrix}$$

$$(w_0^{out} & w_1^{out}) \\ , (\beta^{out})$$

$$(0.6428)$$



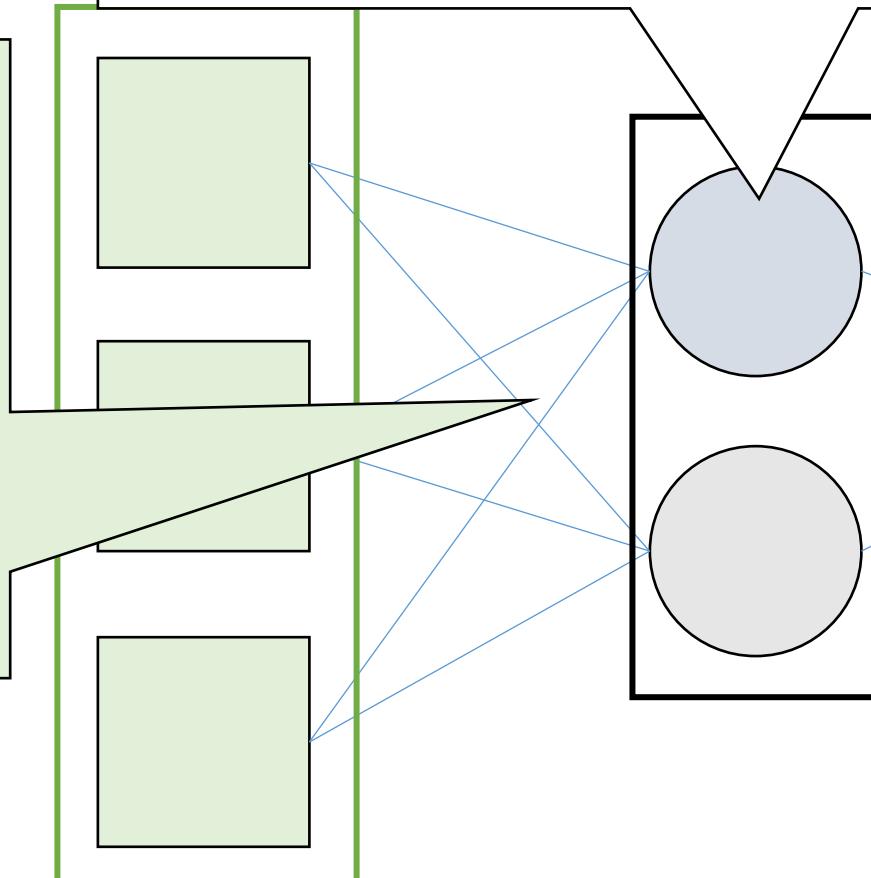
# Predict

Multiplication by a matrix of shape=(2,3)

$$z = \phi(Wx + B)$$

$$\begin{pmatrix} 0.2 \\ 0.9 \\ -1.1 \end{pmatrix}$$

Column vector  
shape=(3,1)



Multiplication by a “matrix”  
of shape=(1,2)

$$\hat{y} = W_{out} \cdot z + B_{out}$$

$$(w_0^{out}, w_1^{out}), (\beta^{out})$$

$(0.4675)$   
 $(0.5175)$   
Output is a  
column  
vector of  
shape=(2,1)

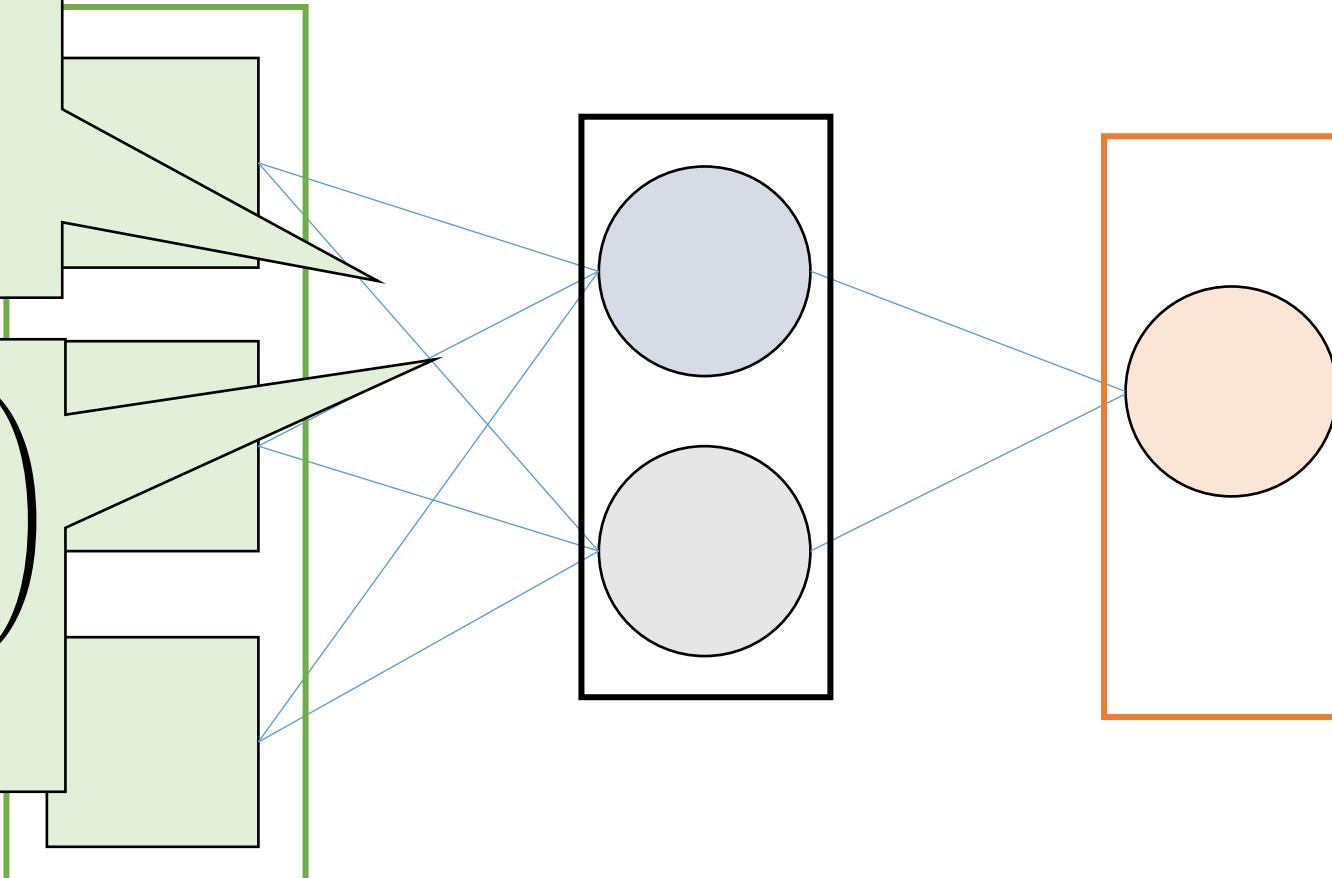
$(0.6428)$   
Final vector of  
shape=(1,1)

# > Predictions (forward pass)

What would happen if we had **more samples**?

$$\begin{pmatrix} \dots & 0.2 & \dots \\ \dots & 0.9 & \dots \\ \dots & -1.1 & \dots \end{pmatrix}$$

shape=(3,N)

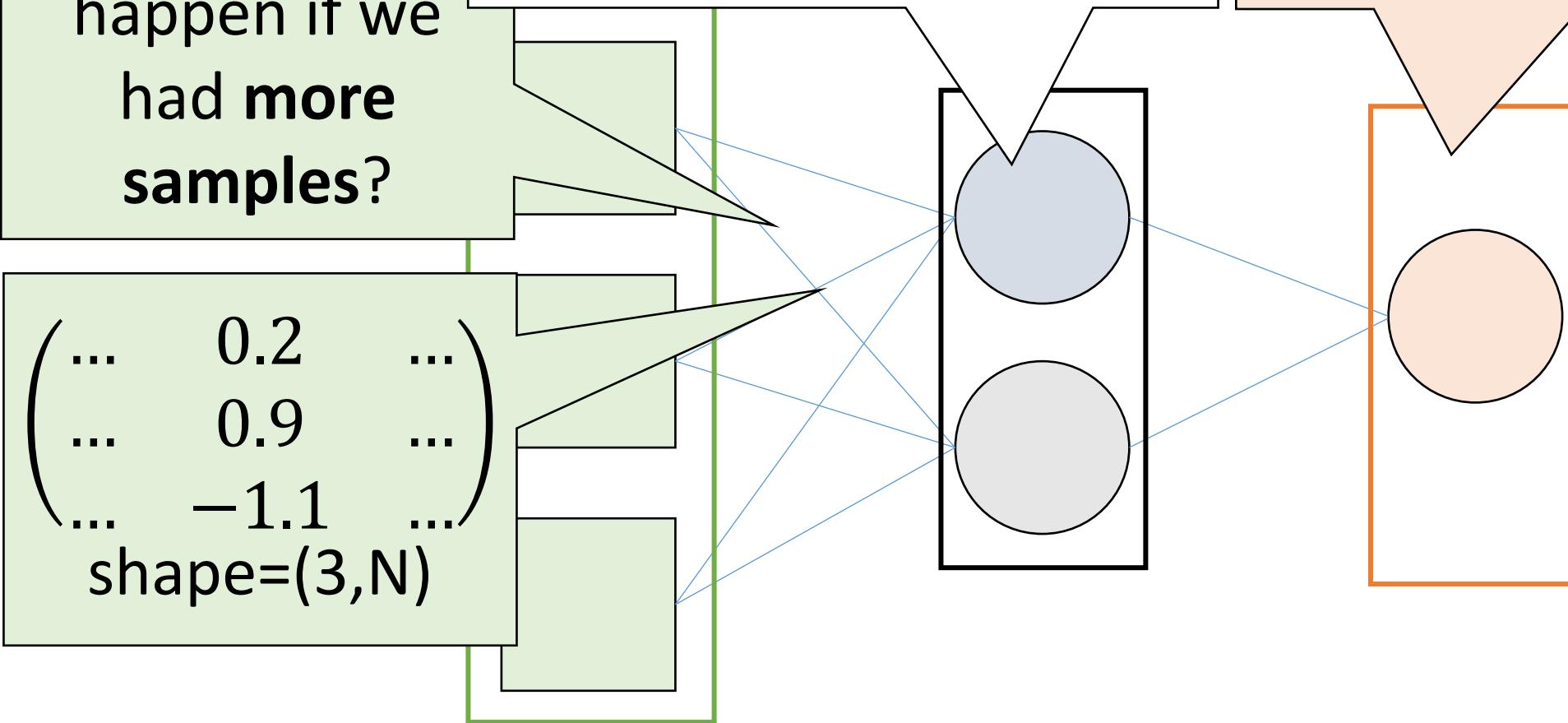


## > Prediction

What would happen if we had **more samples**?

Multiplication by a matrix of shape=(2,3)  
 $z = \phi(Wx + B)$

Multiplication by a “matrix” of shape=(1,2)  
 $\hat{y} = W_{out} \cdot z + B_{out}$

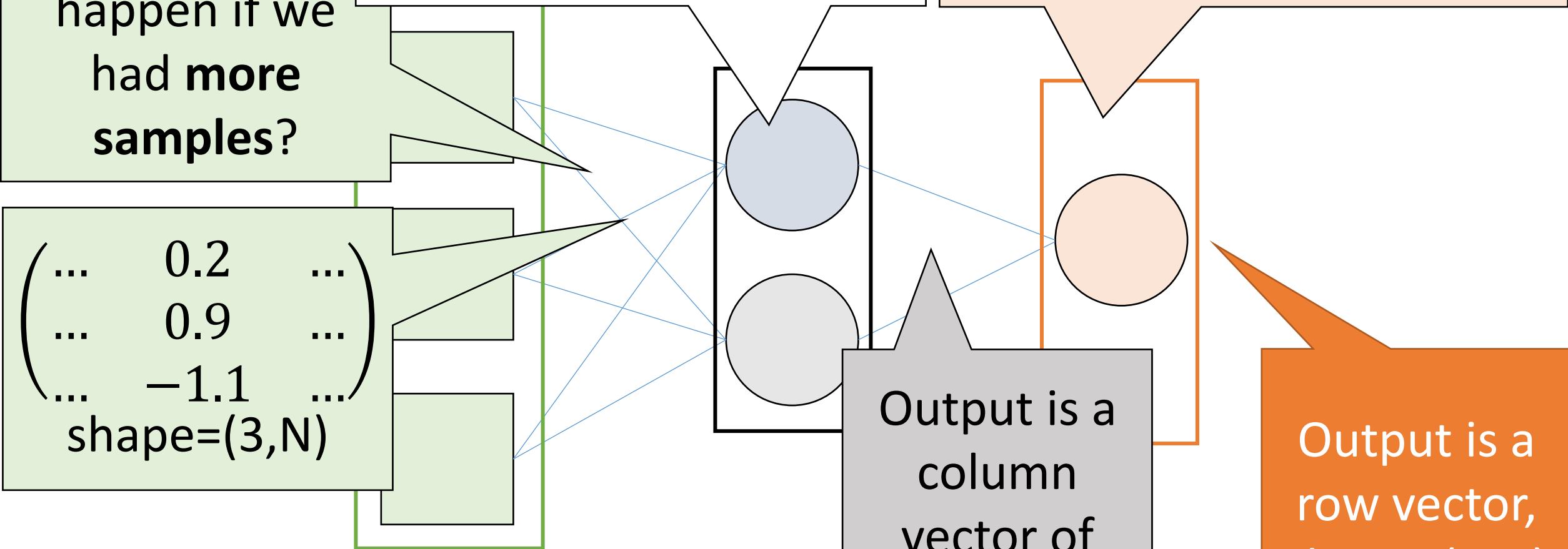


## > Prediction

What would happen if we had **more samples**?

Multiplication by a matrix of shape=(2,3)  
 $z = \phi(Wx + B)$

Multiplication by a “matrix” of shape=(1,2)  
 $\hat{y} = W_{out} \cdot z + B_{out}$



# > A flow of tensors through modules

- The generic name for n-dimensional arrays is **tensors**
- The high-level view of NNs (shared by multiple libraries)
  - Each network is a sequence of **layers** (or better, **modules**)
  - Input and output of each layer/module: **tensors** of a certain shape
  - The **shape** of the tensors is modified as data flows through NN
  - The **output tensor** is interpreted in a human-readable way



# > Notations?

- Other possible notations to describe matrix multiplications
- $z = W^T x + \beta$ 
  - Both  $x$  and the weights of a neuron are seen as column vectors
  - It is thus necessary to transpose  $W$  before multiplying
- $z = xW + \beta$ 
  - Both  $x$  and the weights are seen as row vectors

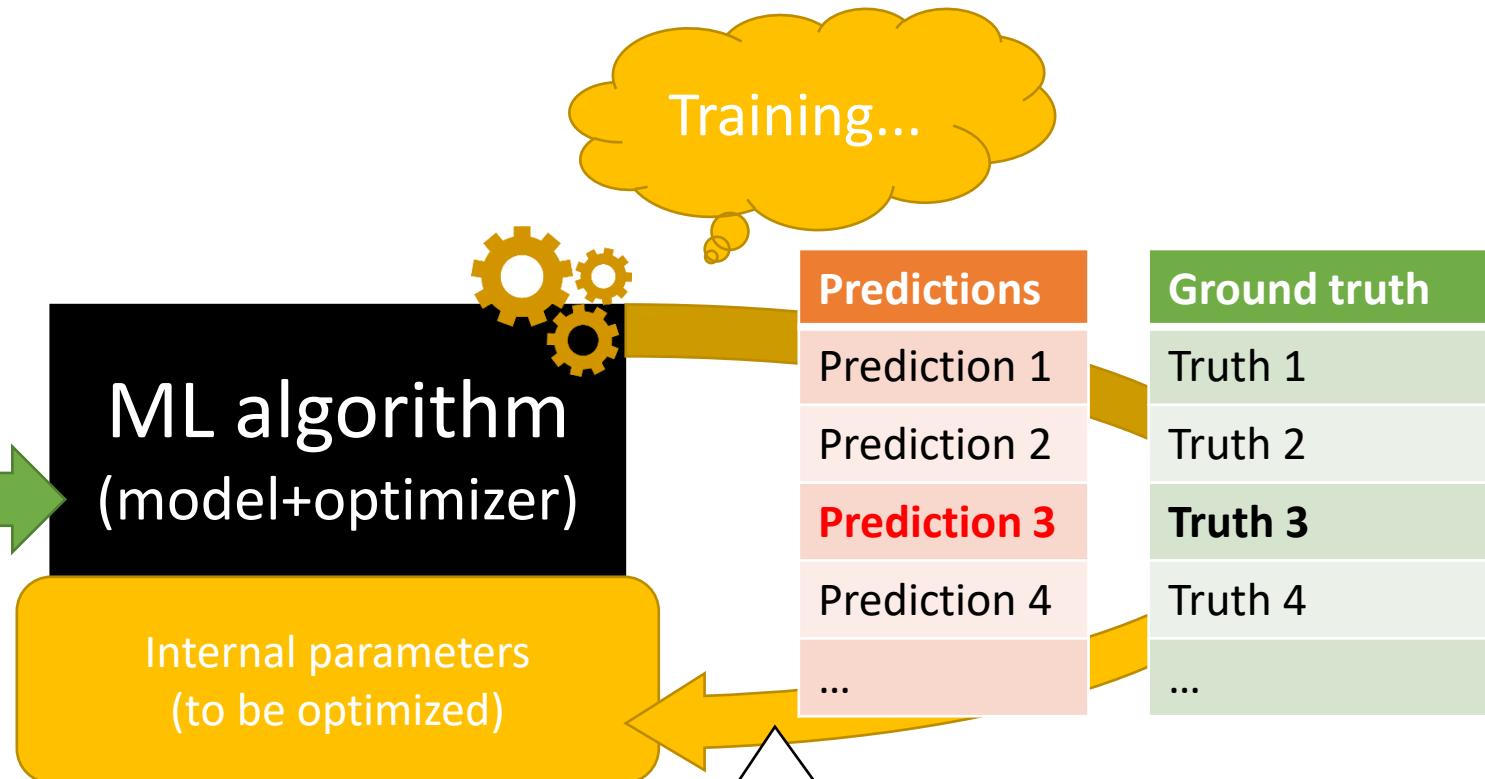
# > Machine learning (supervised)

## Training data

Features

	B	C	D	f3
f0	0.0275364372039	0.0546736280436	0.0280007191734	0.03106
0.00907083722545	0.0179733965578	0.00939900906224	0.02289	
0.0143849438115	0.028605755257	0.0144300702271	0.02176	
0.0225644552597	0.0453556692624	0.0228235614296	0.02959	
0.0231623795278	0.0460850283611	0.02345950129	0.02397	
0.0195037311043	0.0389931600607	0.0198524835167	0.02322	
0.0169146827448	0.0335281026399	0.0168725707329	0.01726	
0.0125413801644	0.0249358420452	0.0128286870354	0.01378	
0.0336853336321	0.0675887724487	0.0340887157945	0.02938	
0.00737647738697	0.0145133565457	0.00747721357495	0.01798	
0.0270749261504	0.0543530338617	0.0275193038864	0.04325	
0.0229367539605	0.0455731635944	0.0230546541676	0.01236	
0.0153923462771	0.0305684027608	0.0153765404848	0.01250	
0.0276334577194	0.054889182742	0.0282327298443	0.02605	
0.0167468084278	0.0333189664752	0.01666678952973	0.01067	
0.0171655969335	0.0341504465775	0.0170995027749	0.02379	
0.0108907676254	0.0220141884316	0.0110321539325	0.00502	

Samples



If we are **learning**, how do we get the feedback to optimize the parameters?

# > Machine learning with neural networks

- First, a **loss function**  $L(\hat{y}, y)$  has to be defined
  - $\hat{y}$  is the **prediction** of the model;  $y$  is the **ground truth**
  - Typical choice for regression is mean/sum of squared errors
  - For classification is the scarily-named *categorical cross-entropy*
  - We want to **minimize**  $L(\hat{y}, y)$ ; if zero, perfect predictions
- Since  $\hat{y} = f(x, \theta)$  where  $\theta$  are parameters of the network
  - Modify  $\theta$  to minimize  $L(\hat{y}, y) = L(f(x, \theta), y)$
  - How do we do that?

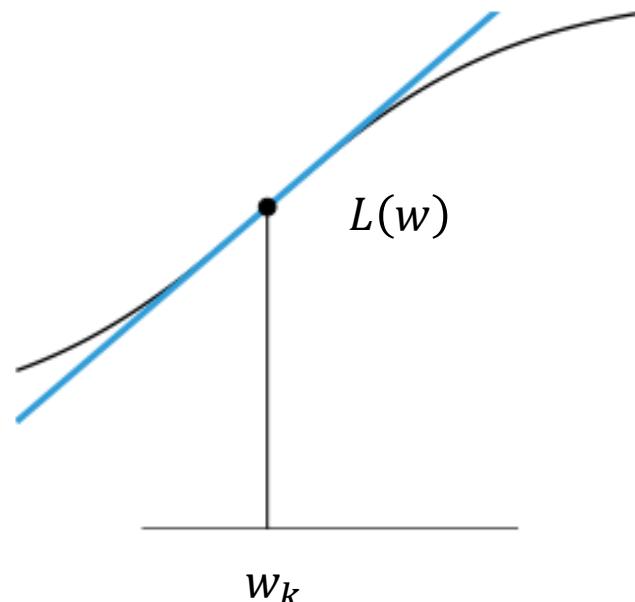
# > Gradient descent

- A classic optimization algorithm is **gradient descent**
  - If the derivative of the target function can be computed
  - Compute partial derivative w.r.t. each weight
  - Push each weight in the “right direction” by a bit (**learning rate**)
  - Evaluate loss function again, compute derivative, iterate

$$\forall i, j, \quad \frac{\partial L(\hat{y}, y)}{\partial w_{ij}} \quad w_{ij} \leftarrow w_{ij} - \underbrace{\varepsilon}_{\text{Learning rate}} \frac{\partial L(\hat{y}, y)}{\partial w_{ij}}$$

# > Gradient descent

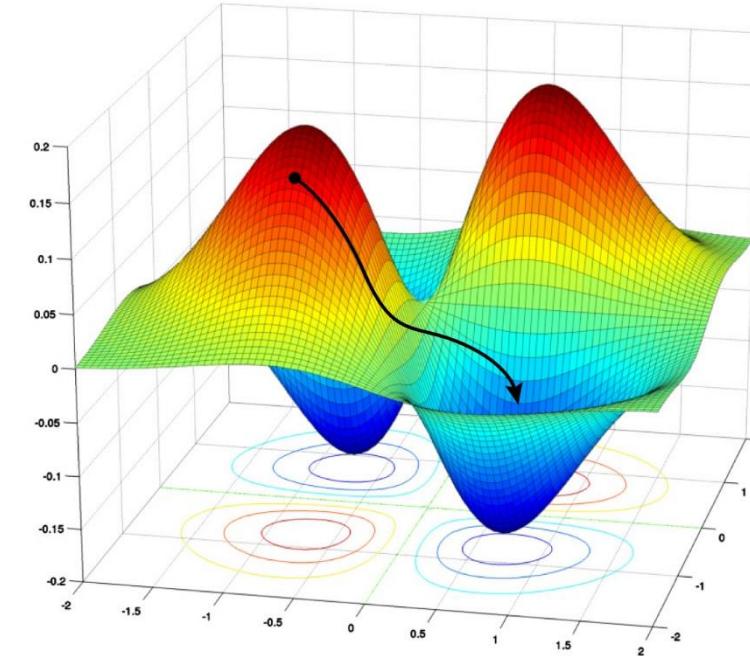
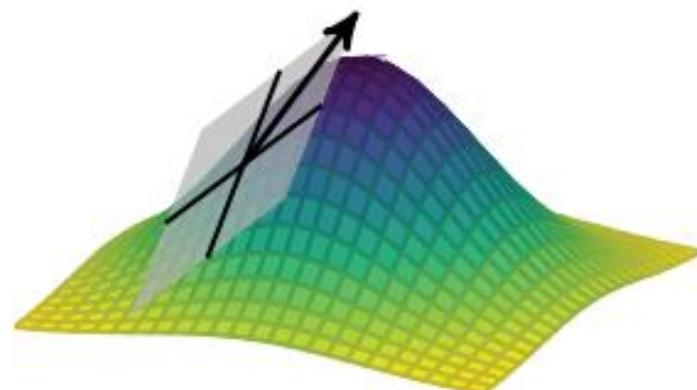
- Derivative  $\frac{\partial L}{\partial w}$  of a function  $L(w)$  in a single point  $w_k$ 
  - Rate at which the value of  $L(w)$  changes at  $w_k$
  - Can be visualized as the slope of a tangent line



# > Gradient descent

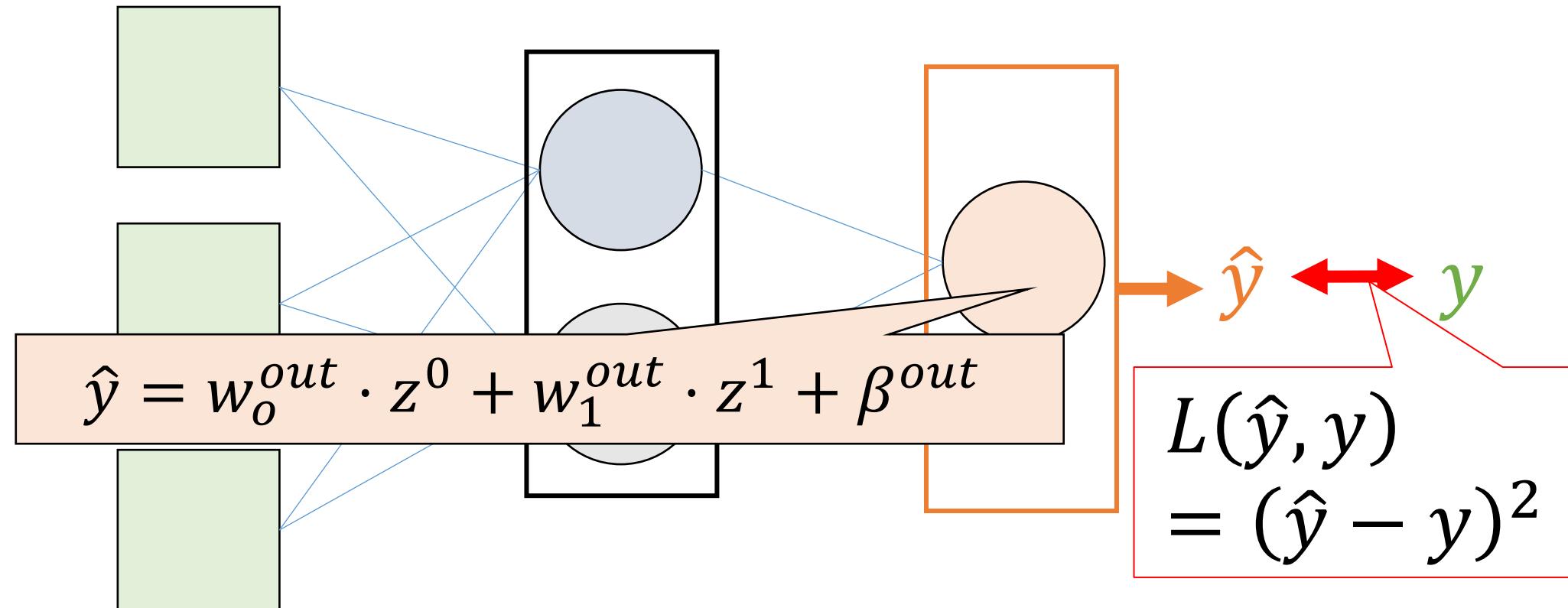
- Derivative in multiple dimensions is the **gradient (Jacobian)**

$$\nabla L(\theta_k) = \left( \frac{\partial L(\theta_k)}{\partial w_0}, \dots, \frac{\partial L(\theta_k)}{\partial w_N} \right)$$



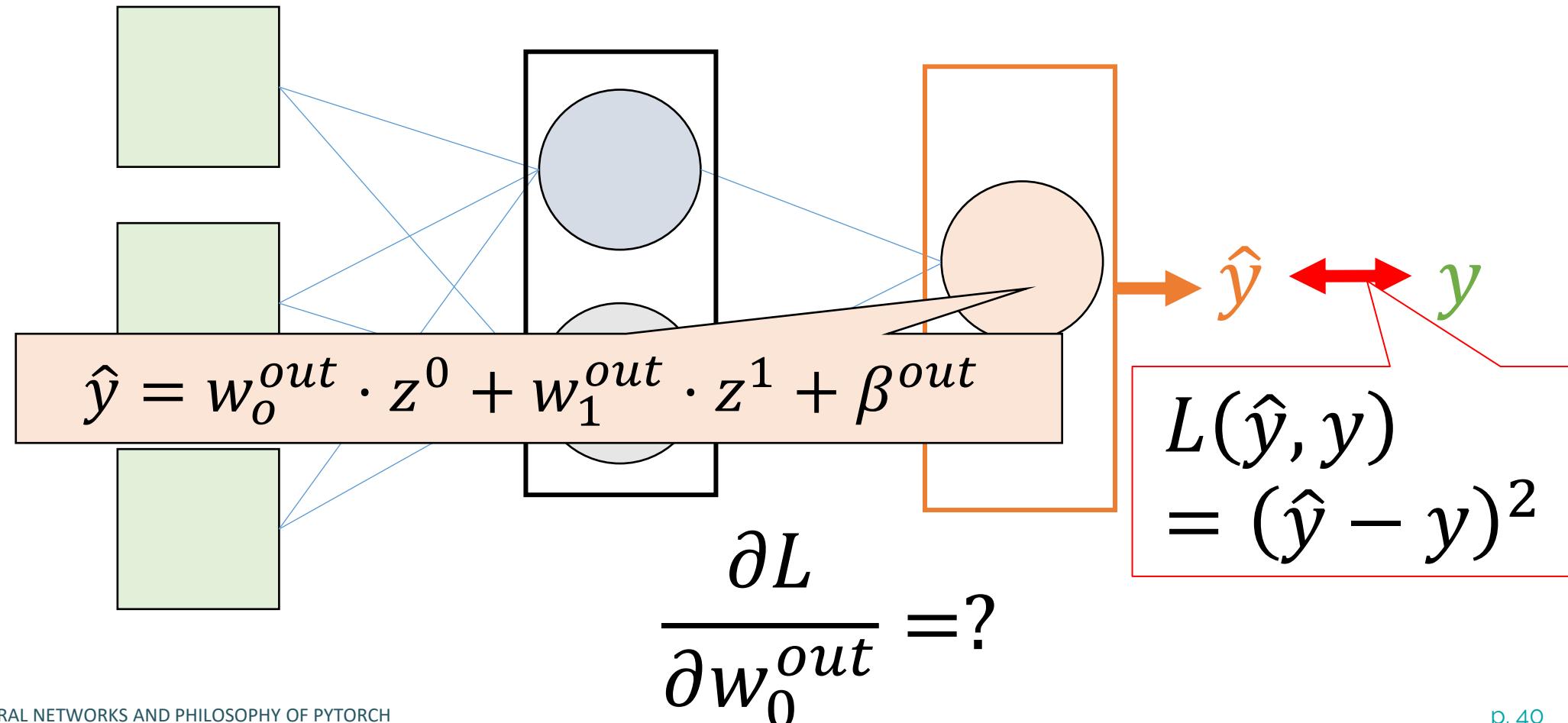
# > Computing derivatives: backward pass

- Derivatives of output w.r.t last weights are straightforward



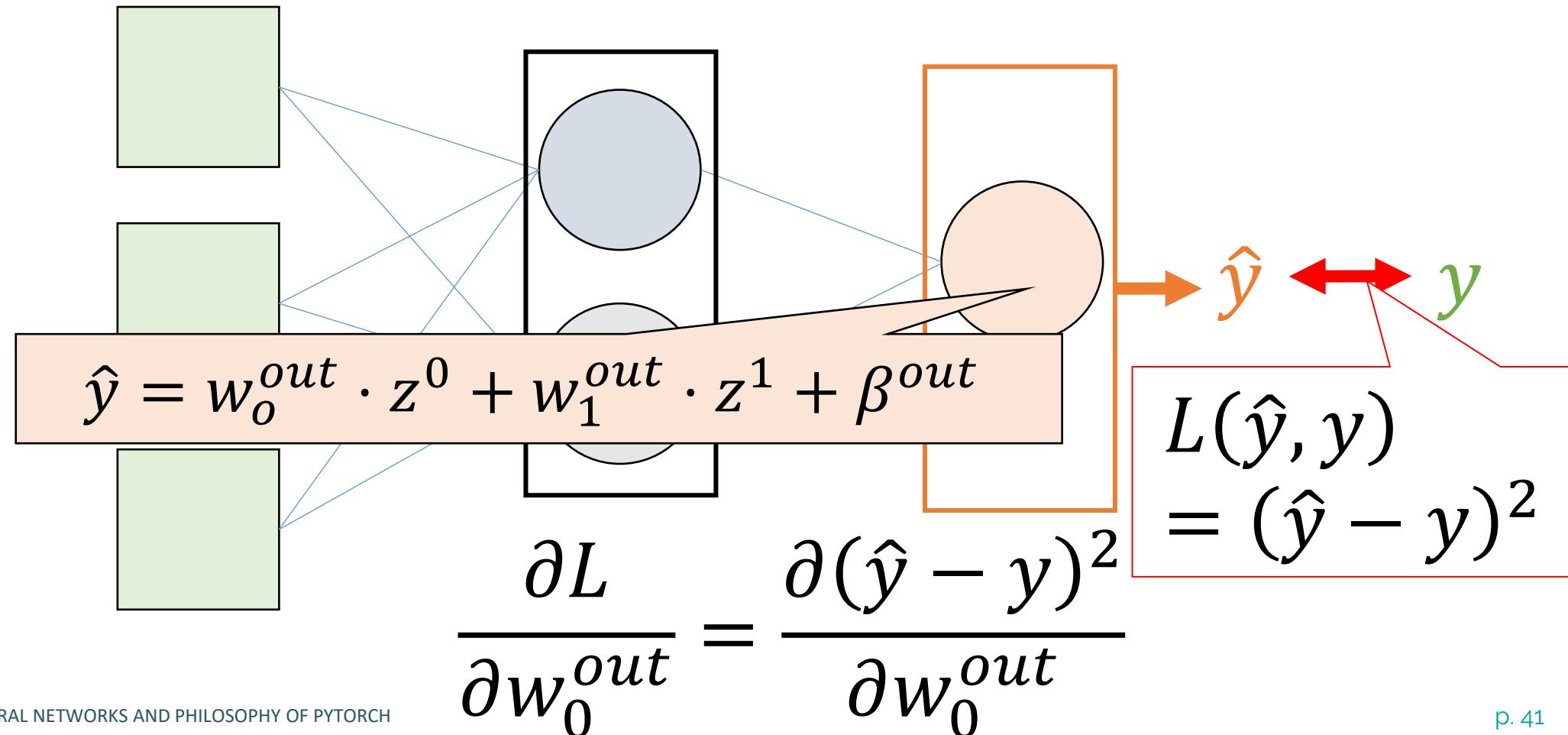
# > Computing derivatives: backward pass

- Derivatives of output w.r.t last weights are straightforward



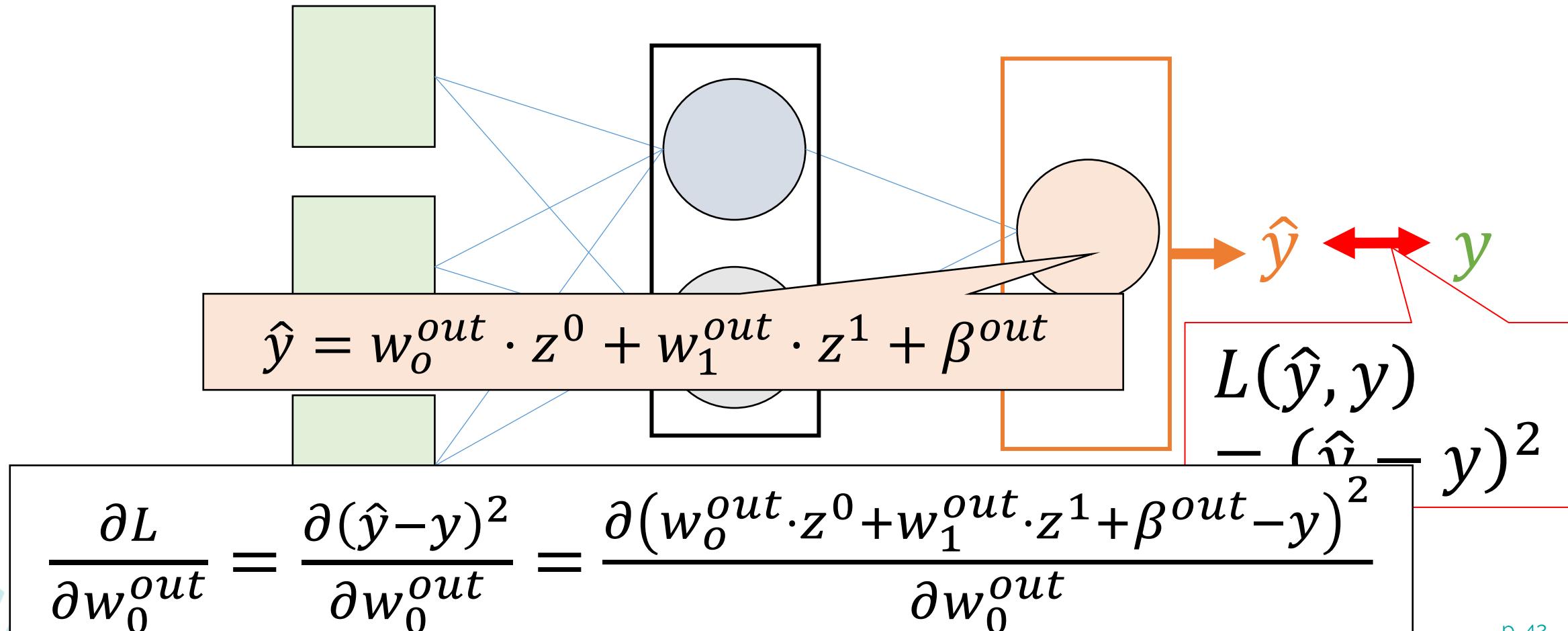
# > Computing derivatives: backward pass

- Derivatives of output w.r.t last weights are straightforward



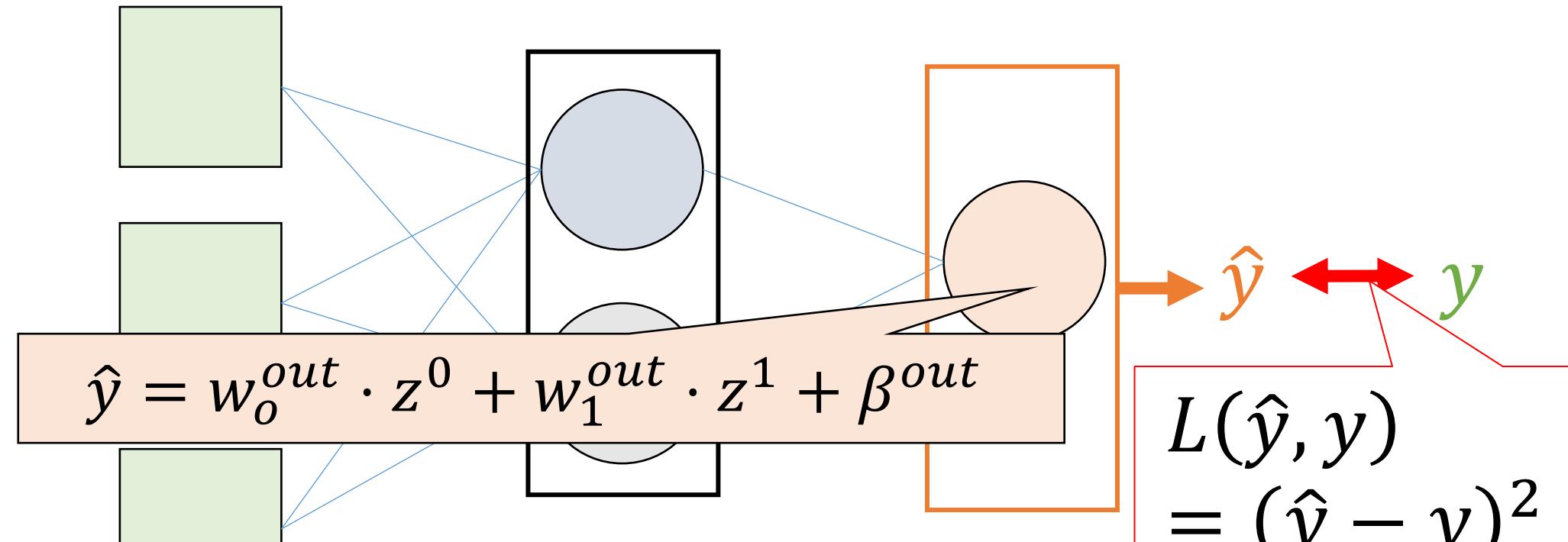
# > Computing derivatives: backward pass

- Derivatives of output w.r.t last weights are straightforward



# > Computing derivatives: backward pass

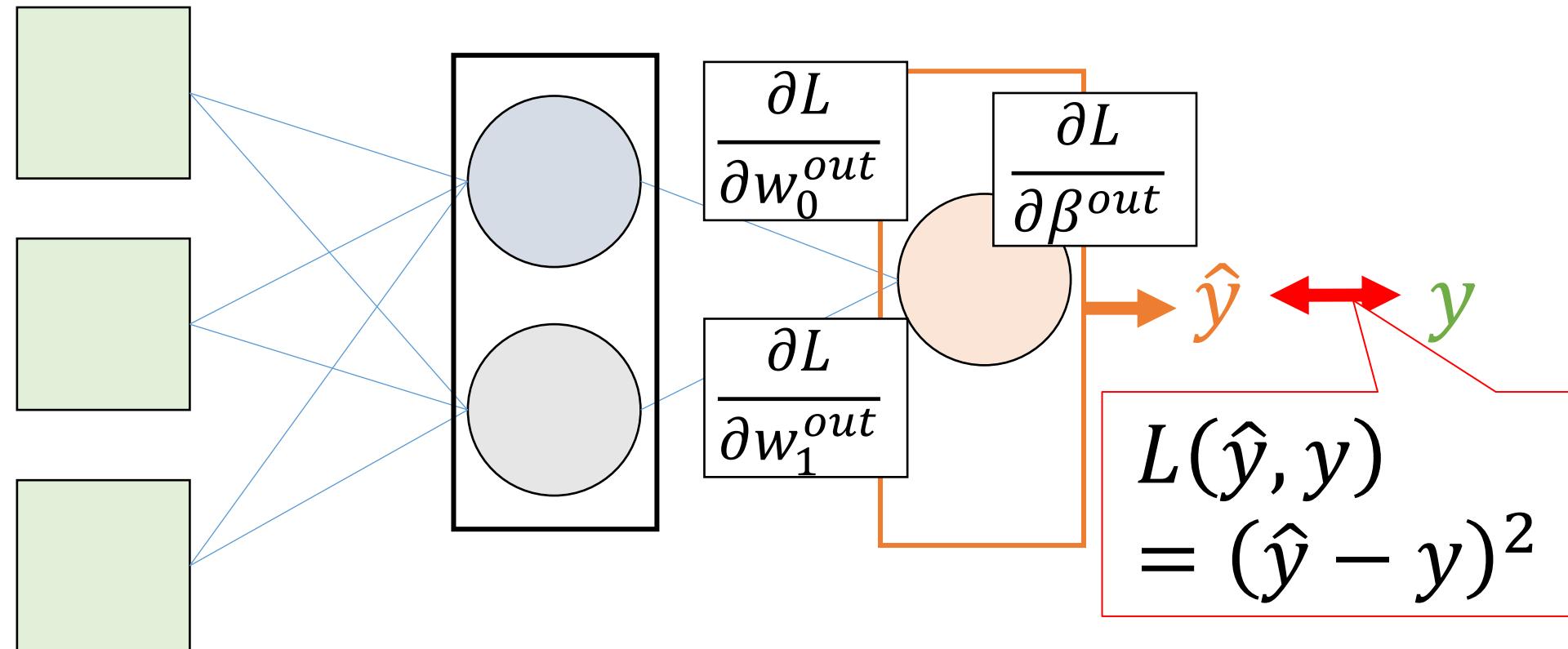
- Derivatives of output w.r.t last weights are straightforward



$$\frac{\partial L}{\partial w_0^{out}} = 2(w_0^{out} \cdot z^0 + w_1^{out} \cdot z^1 + \beta^{out} - y) \cdot z^0$$

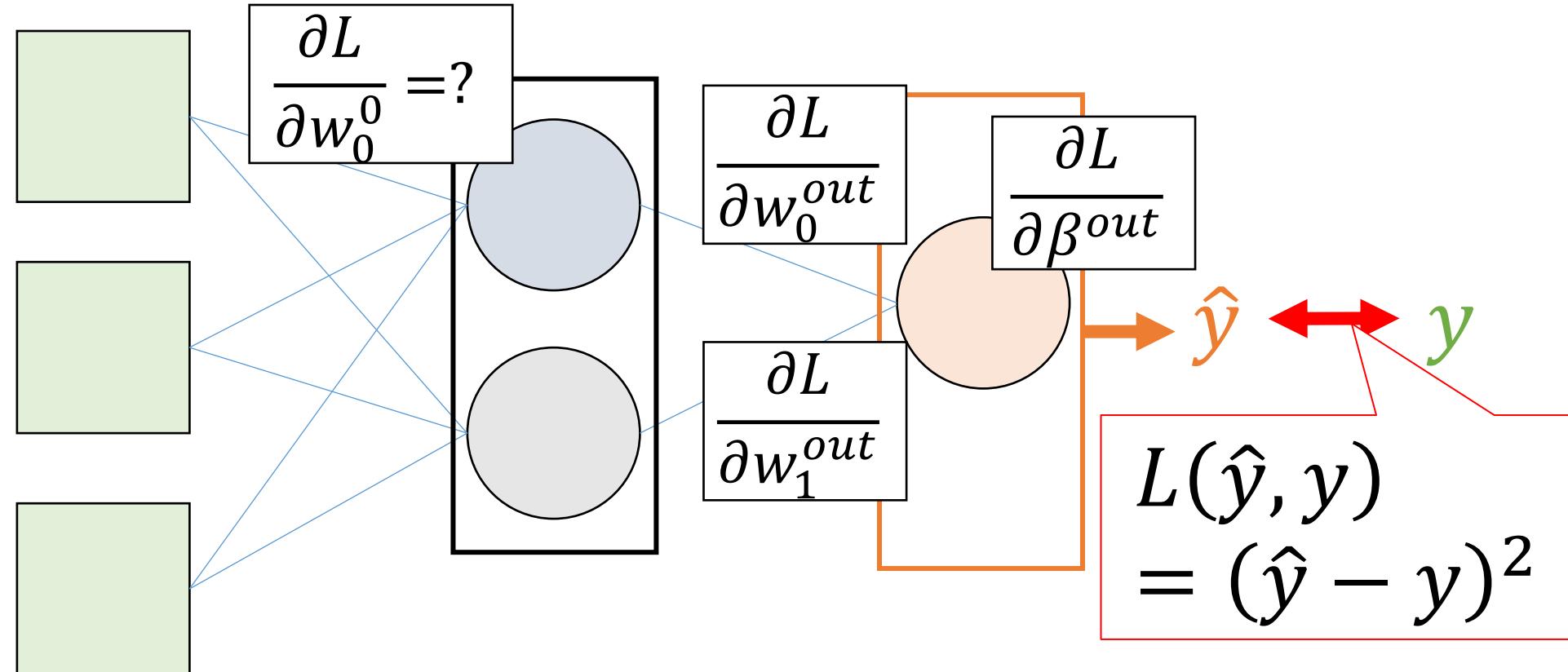
# > Computing derivatives: backward pass

- Derivatives of output w.r.t last weights are straightforward



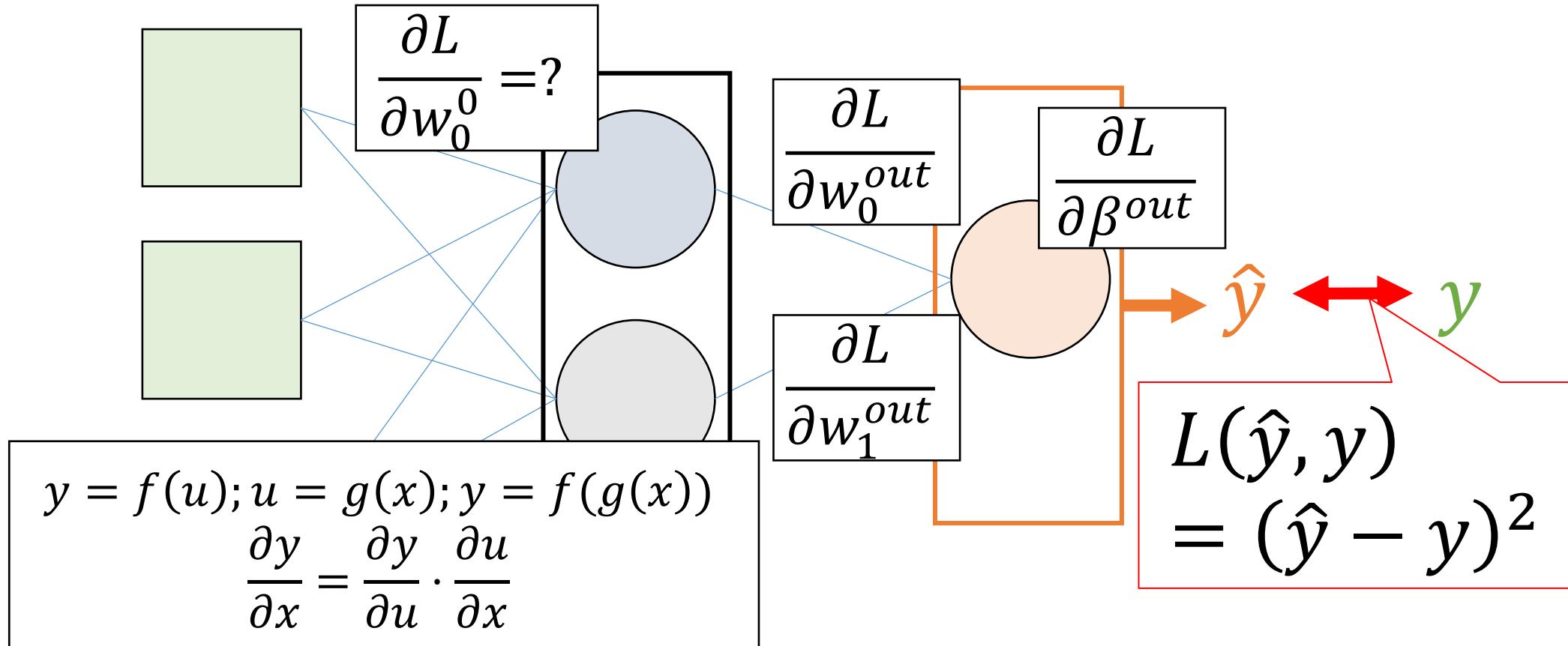
# > Computing derivatives: backward pass

- But what about weights in the first layer?



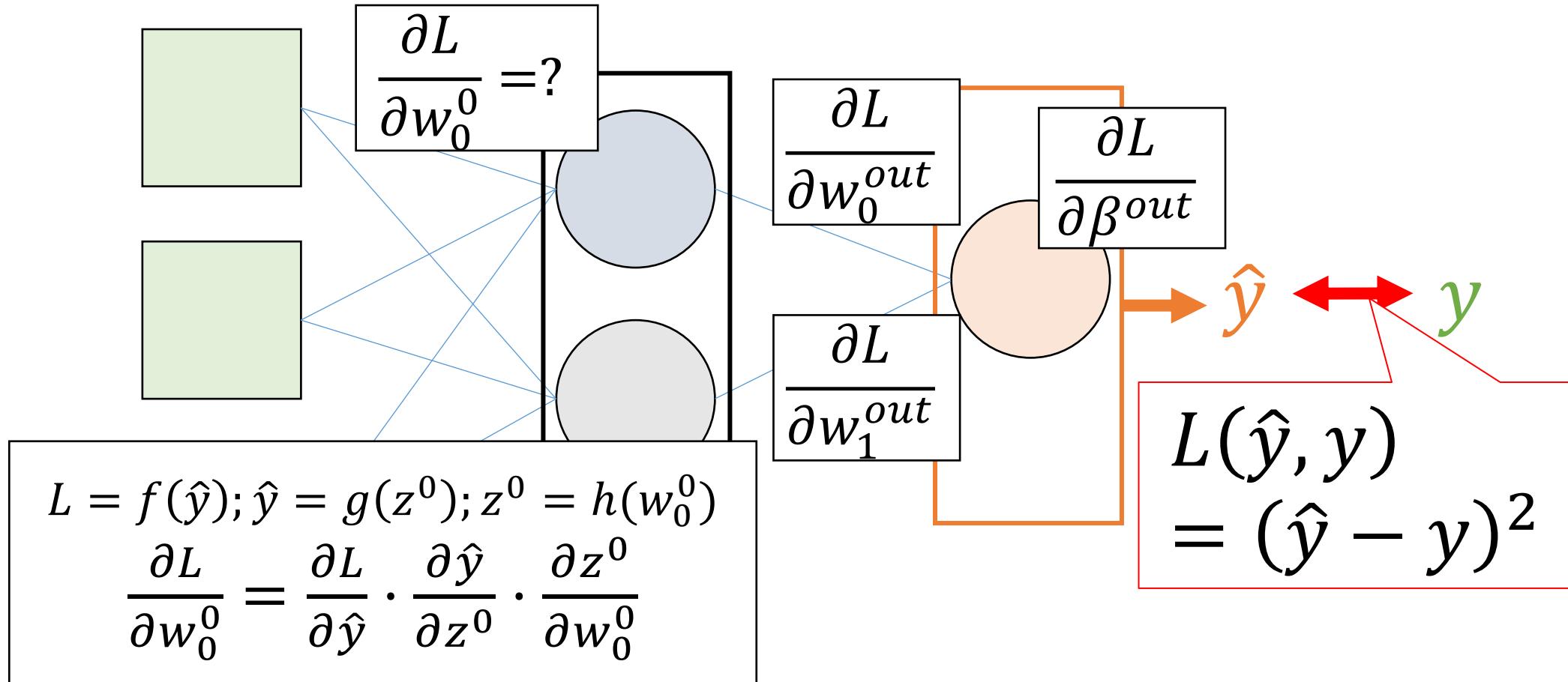
# > Computing derivatives: backward pass

- Chain rule!



# > Computing derivatives: backward pass

- Chain rule!



# > Computing derivatives: backward pass

- At the end of the backward pass (or backpropagation)
  - We have the gradient  $\nabla L$ , that we can use to update weights
  - $\nabla L$  gives us the direction in which we should go to minimize  $L$
  - We take a step (learning rate) in that direction, updating weights
  - Another forward pass, another  $\hat{y}$ , another  $L$ , another  $\nabla L$
  - Repeat until  $L = 0$  (unlikely), out of time, or other stop condition

# > A few remarks

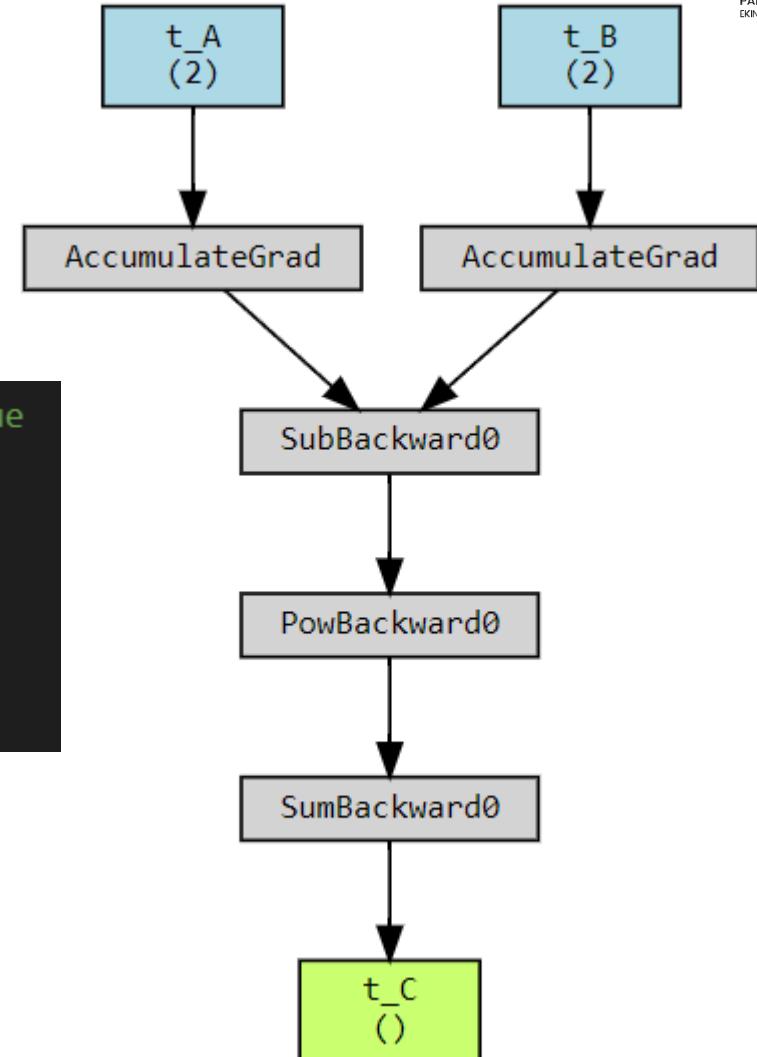
- Computing the gradient is a fundamental operation
  - Luckily, we don't have to do it by hand
  - Automatic computation of gradients
  - This is a *highly parallelizable* operation!
- pytorch offers a **computational graph**

# > Computational graph

## pytorchviz

```
# tensor initialization; note that we specify explicitly requires_grad=True
# to enable the computation of gradients (the default is false)
t_A = torch.tensor([1.,2], requires_grad=True)
t_B = torch.tensor([4.,4], requires_grad=True)

# sum of squared differences between elements of t_A and t_B
t_C = ((t_A - t_B)**2).sum()
```



# > A few remarks

- The NN representation with layers is a bit *inconsistent*
  - What does a *single neuron* represent?
  - Sometimes it's a weighted sum, sometimes + activation function
  - Where are weights and biases stored? On the arcs? Even biases?

# > Philosophy of pytorch

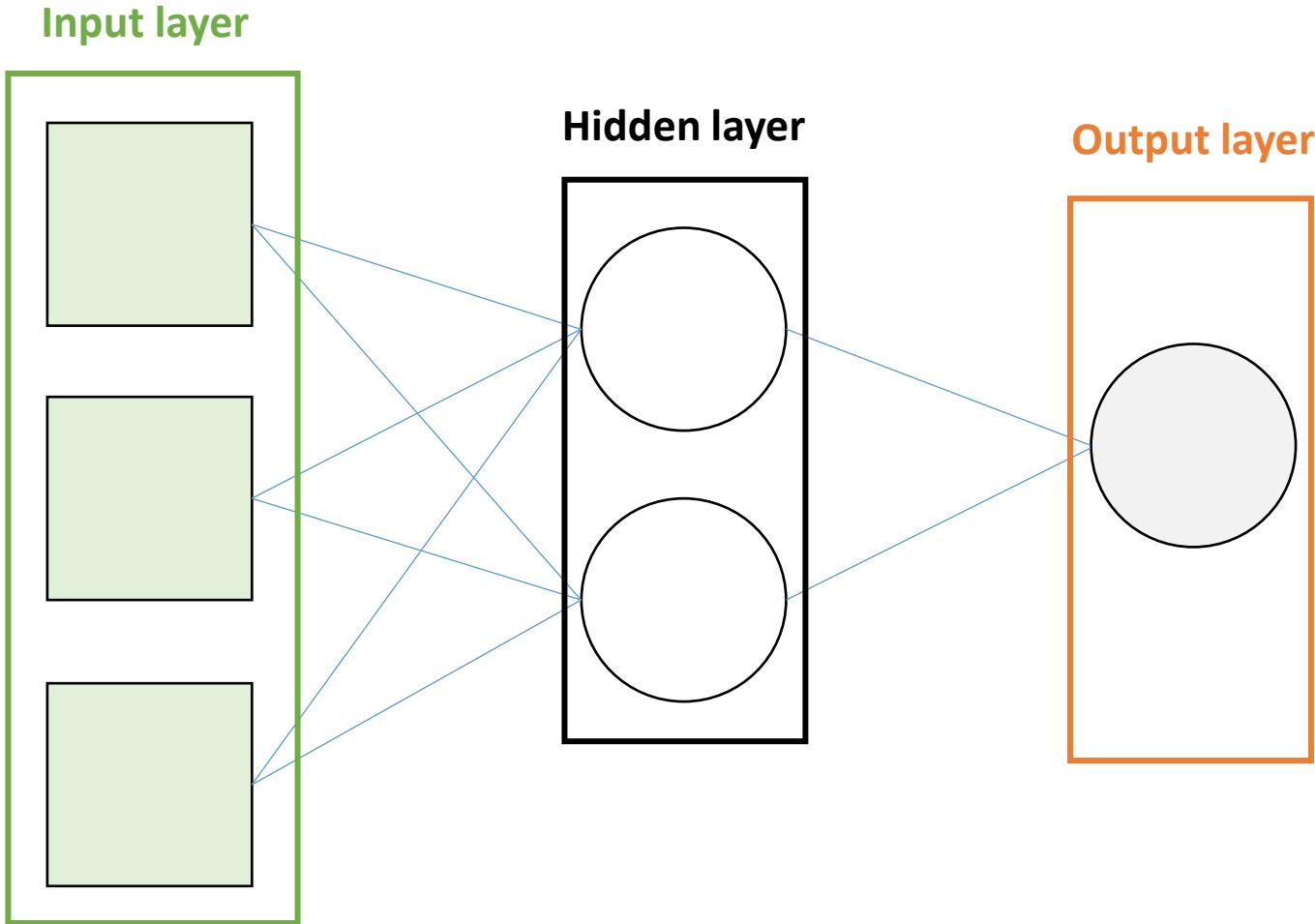
- Like other libraries, **tensors** flow through **modules**
- Tensors (torch.Tensor) are specific data structures
  - Just like numpy arrays, multi-dimensional matrices
  - They can easily be assigned to run on GPUs for quick computations
  - They **store information** about gradients! torch.Tensor.grad
  - Compute a graph of the derivatives in forward pass
  - Context torch.no\_grad() or Tensor.detach() to deactivate

# > Philosophy of pytorch

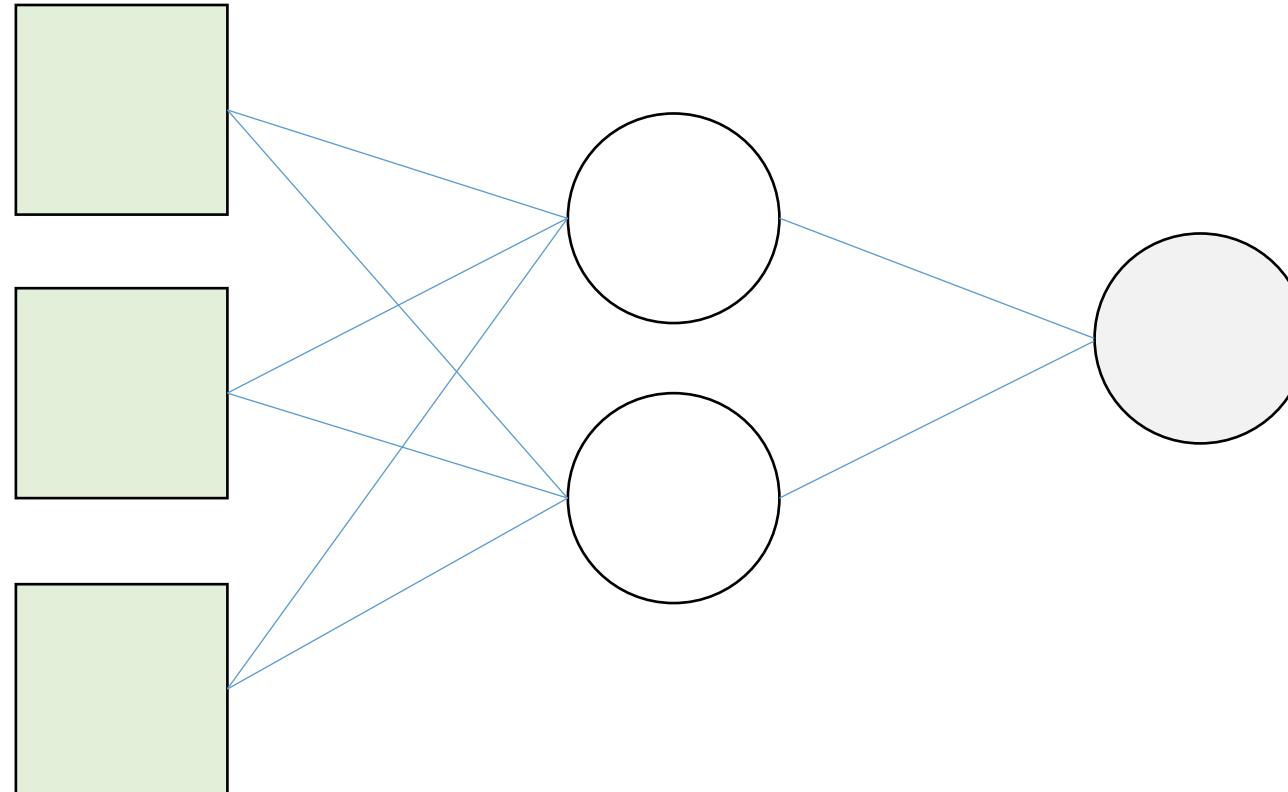
- Modules
  - torch.nn.Module represents a (unitary\*) operation on a tensor
  - Implements both forward() and backward()
  - Expects a **tensor of given shape** in input
  - Gives **tensor of another shape** in output

\*this is true for most Modules found in pytorch

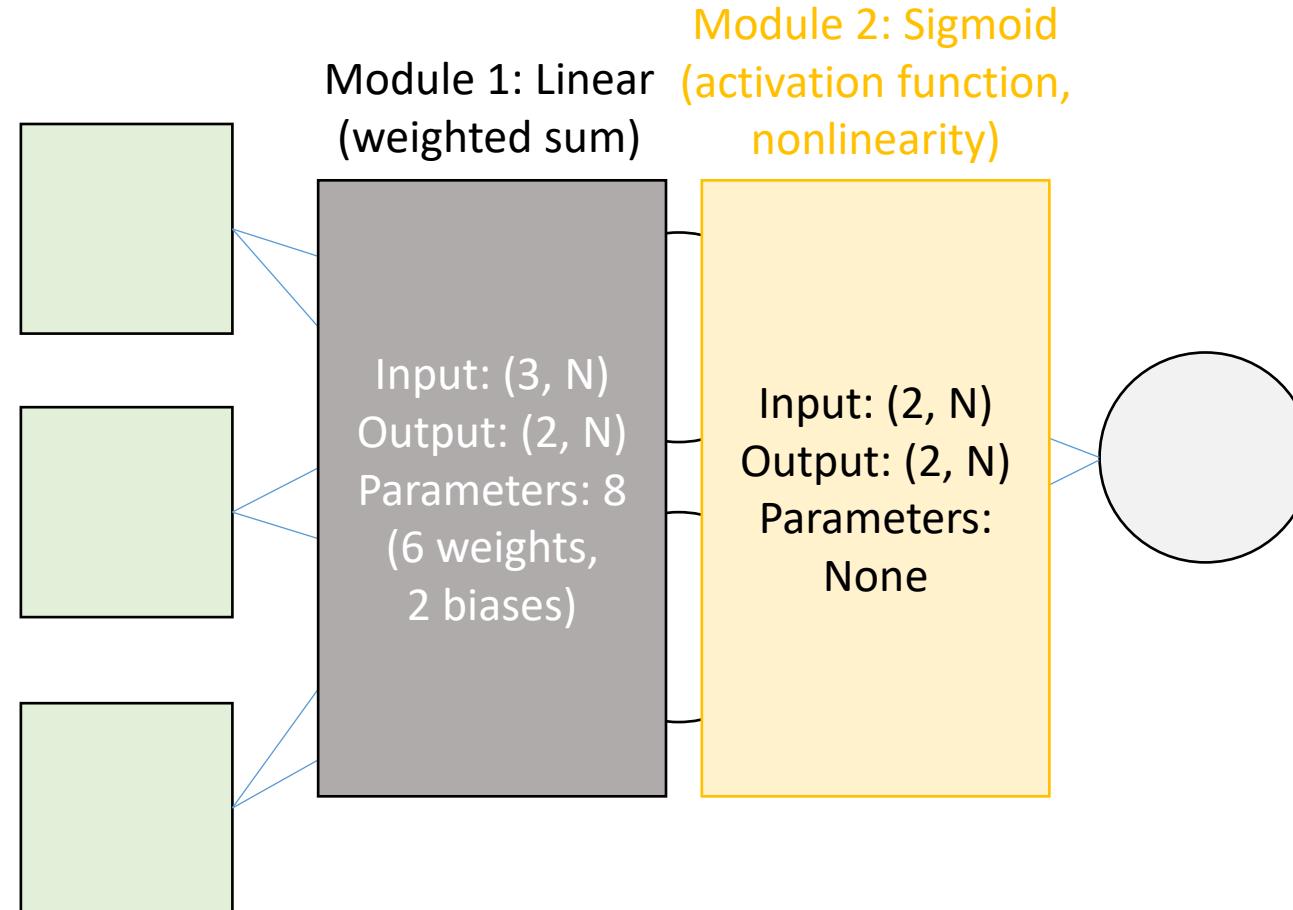
# > Philosophy of pytorch



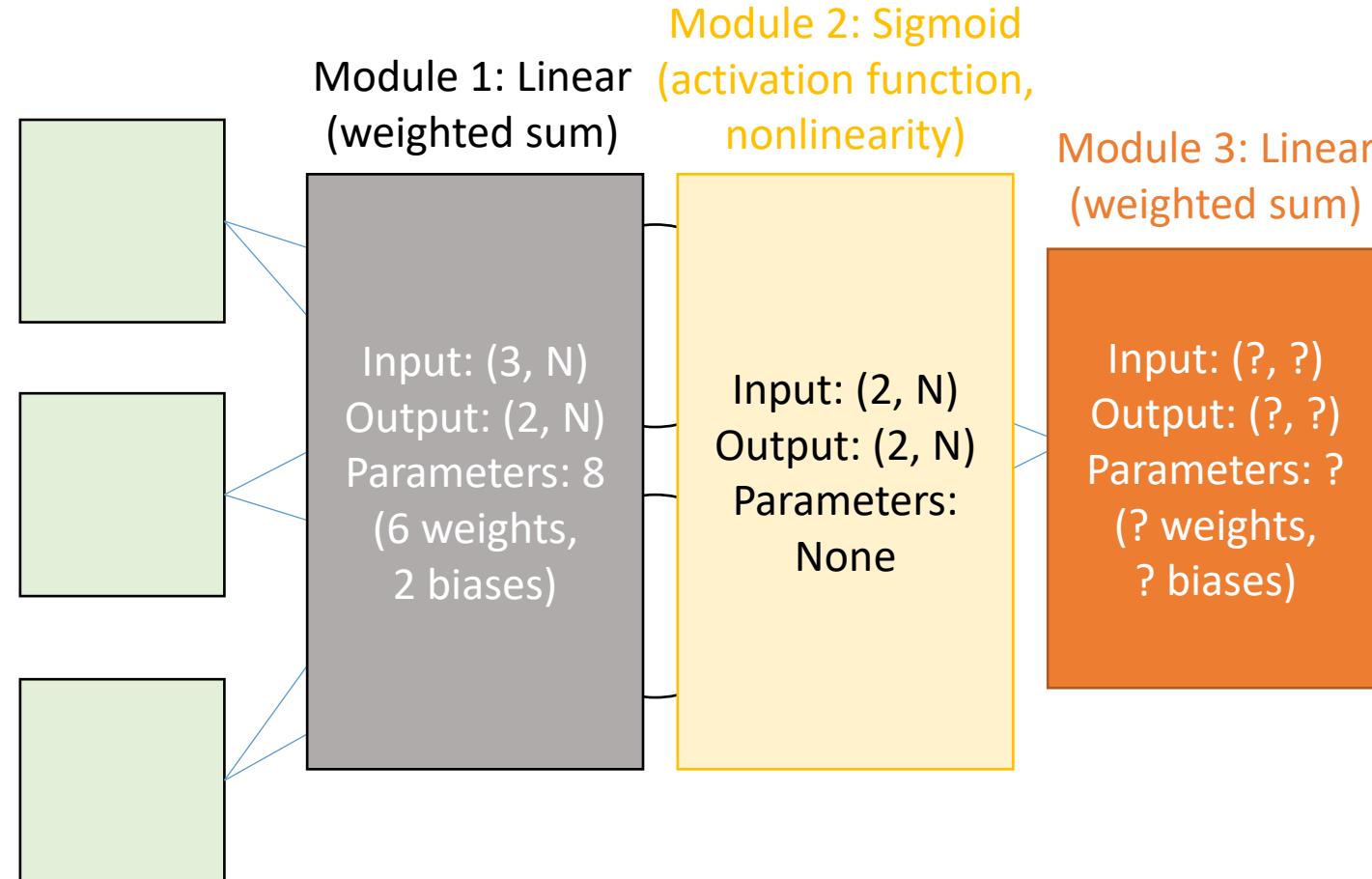
# > Philosophy of pytorch



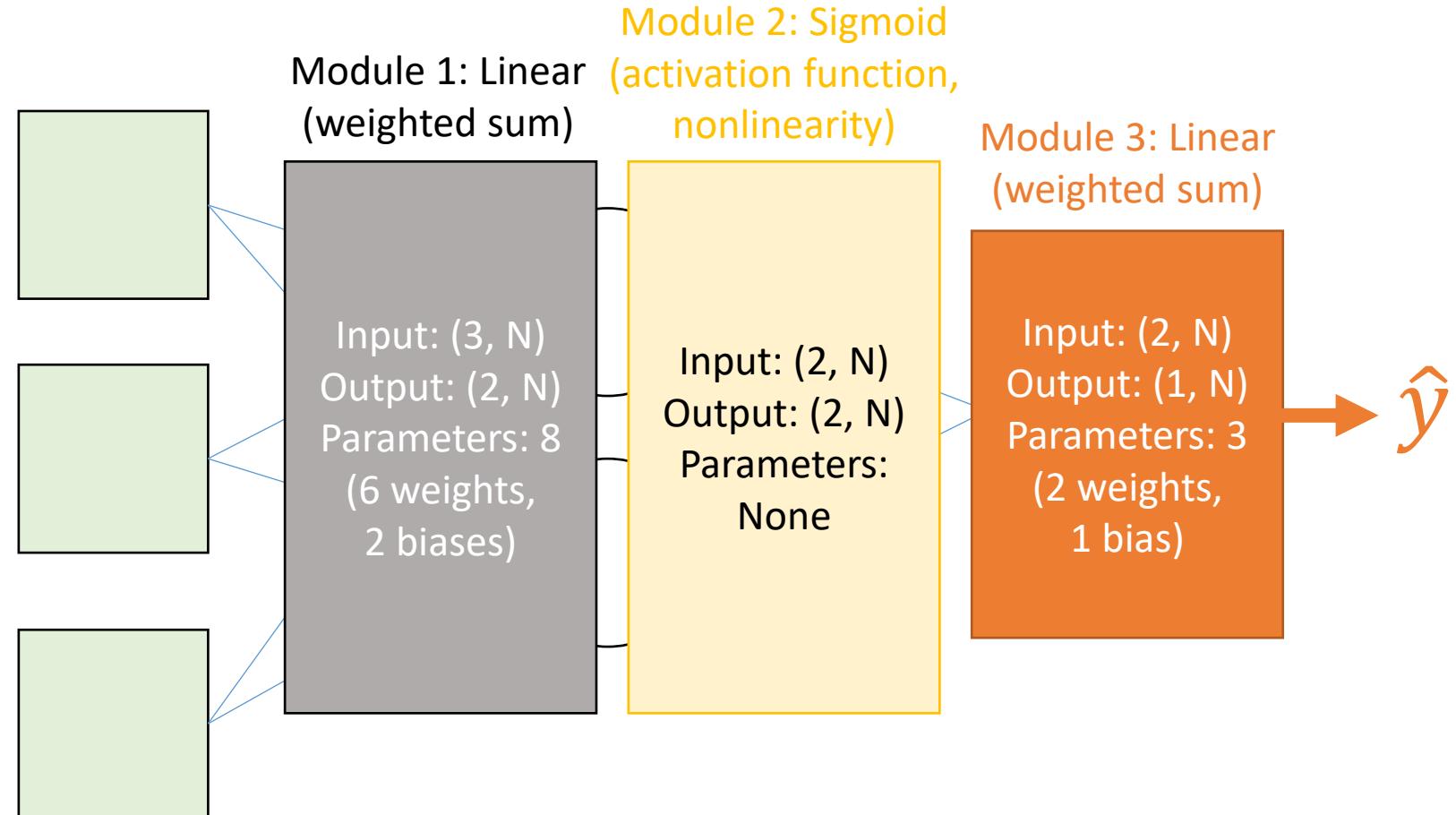
# > Philosophy of pytorch



# > Philosophy of pytorch



# > Philosophy of pytorch



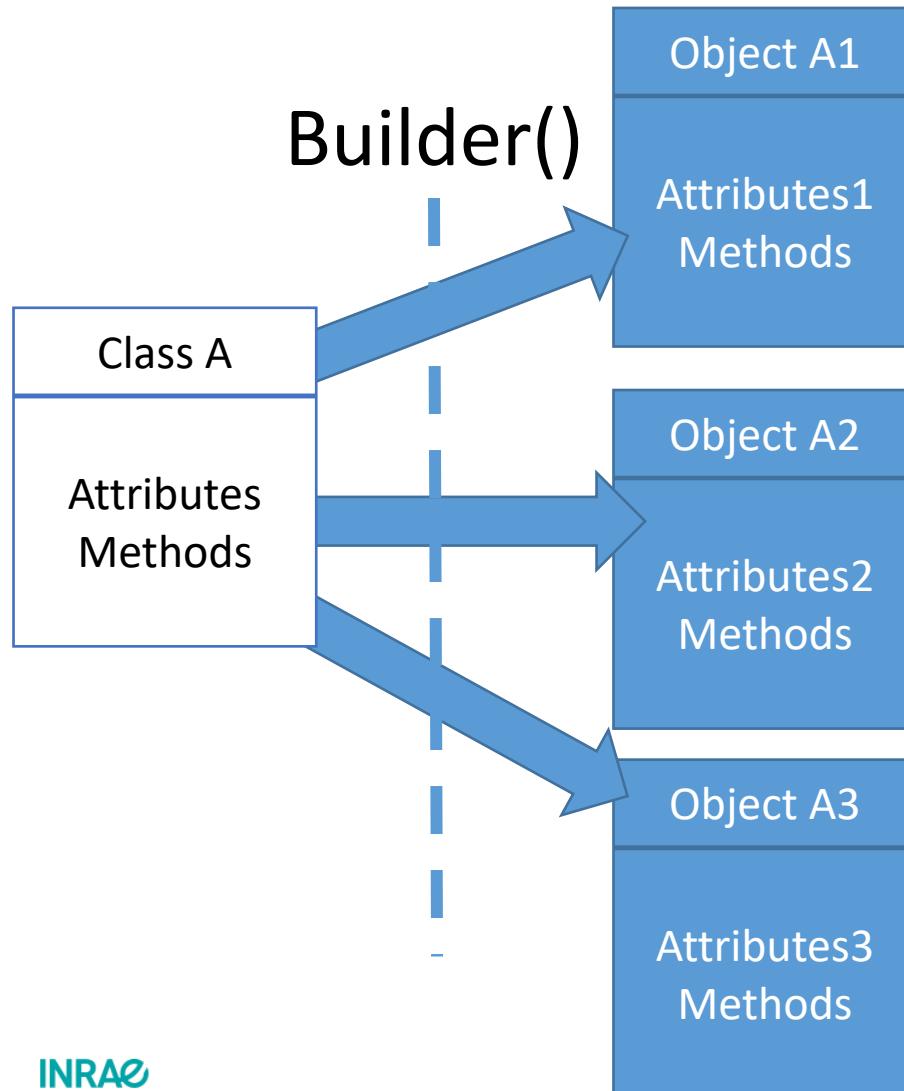
# > pytorch modules

- Considerable number of different Modules
  - Implementing layers of different type (convolutional, recurrent, ...)
  - Different activation functions (Sigmoid, Tanh, ReLU, ...)
- Adding new Modules is relatively easy
  - Need to implement forward() and backward()

# > Philosophy of pytorch: inheritance

- pytorch wants users to create new Classes
  - Classes that **inherit** from existing classes
  - For example, Neural Networks inherit from Module
  - Load data efficiently, creating a Class that inherits from Dataset

# > Philosophy of pytorch: inheritance

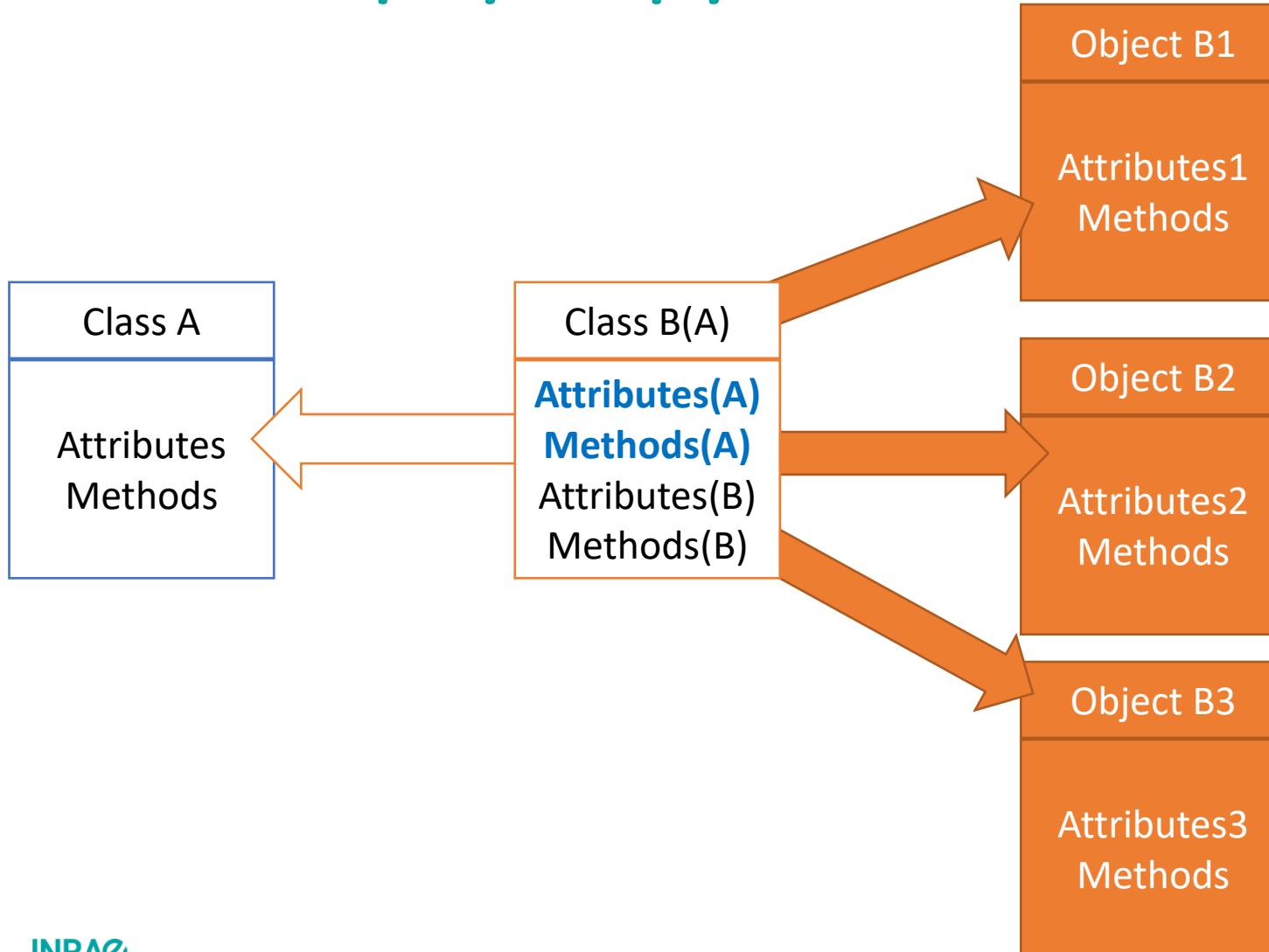


Objects A1...A3 are also called **instances** of Class A

Python syntax

- Attributes: `object.attribute`
- Methods: `object.method()`

# > Philosophy of pytorch: inheritance



Objects B1...B3 are  
**instances** of Class B

Class B **inherits**  
(some) attributes  
and methods from  
Class A

# > Philosophy of pytorch

- Every network is an instance of a `torch.nn.Module` object
- Methods inside a **Module**:
  - `__init__(self)` : builder, used to set up layers (and other stuff)
  - `forward(self, X)` : forward pass of the Module
  - `backward(self, X)` : backward pass of the Module
- `forward()` is called during training, necessary to specify it
- Luckily, often is just passing a tensor through Modules
- If all modules are standard, `backward()` is automatic

# > Philosophy of pytorch

- Need for a Loss function
- Losses are objects instantiated from appropriate classes
  - `torch.nn.MSELoss()` # mean squared error
  - `torch.nn.CrossEntropy()` # categorical cross-entropy, classification

$$CE = - \sum_{i=1}^{i=N} y\_true_i \cdot \log(y\_pred_i)$$

Class labels represented with one-hot encoding

$$CE = - \sum_{i=1}^{i=N} y_i \cdot \log(\hat{y}_i)$$

$$\implies CE = -[y_1 \cdot \log(\hat{y}_1) + y_2 \cdot \log(\hat{y}_2) + y_3 \cdot \log(\hat{y}_3)]$$

# > Computational graph

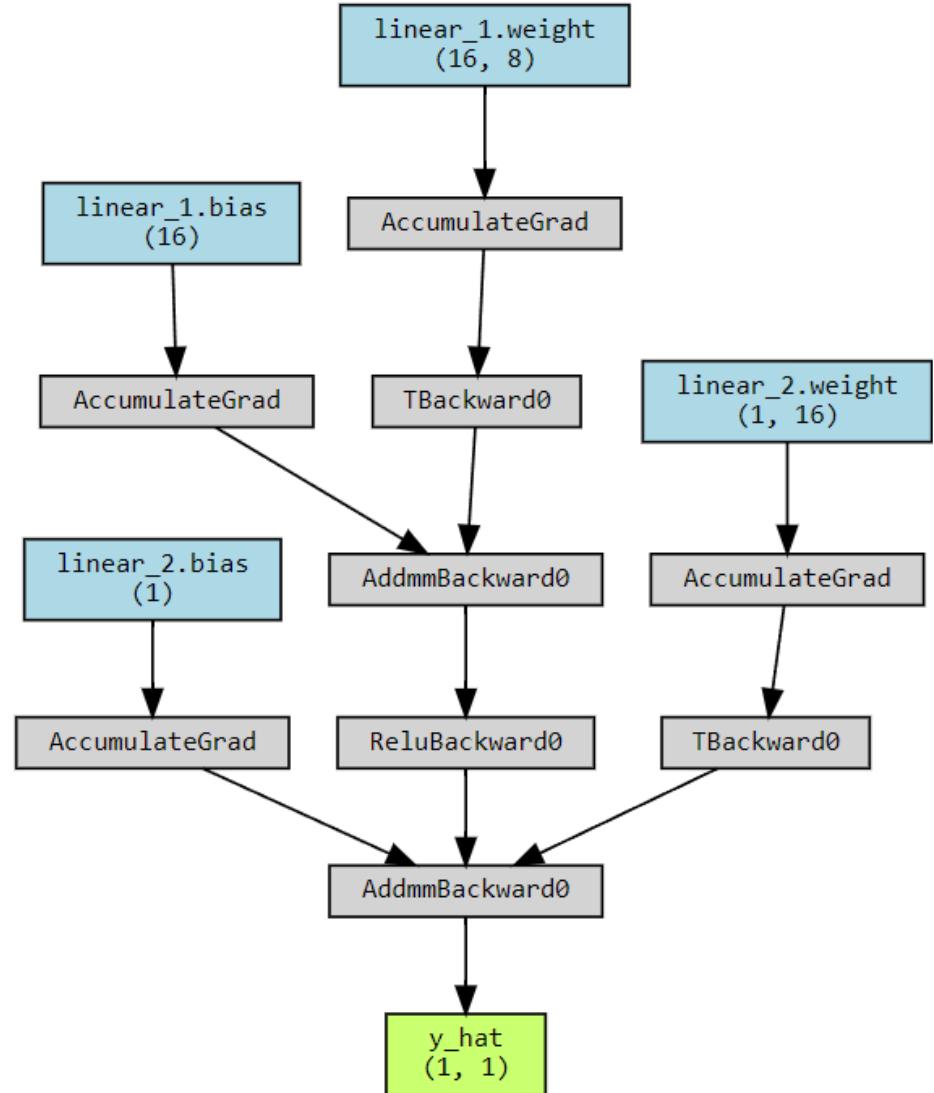
```

class MyMiniMLP(torch.nn.Module) :
    def __init__(self) :
        super(MyMiniMLP, self).__init__()
        self.linear_1 = torch.nn.Linear(8, 16)
        self.activation = torch.nn.ReLU()
        self.linear_2 = torch.nn.Linear(16, 1)

    def forward(self, x) :
        x = self.linear_1(x)
        x = self.activation(x)
        y_hat = self.linear_2(x)

        return y_hat

my_mini_mlp = MyMiniMLP()
x0 = torch.randn(1,8)
y_hat = my_mini_mlp(x0)
  
```



# ➤ Computational graph

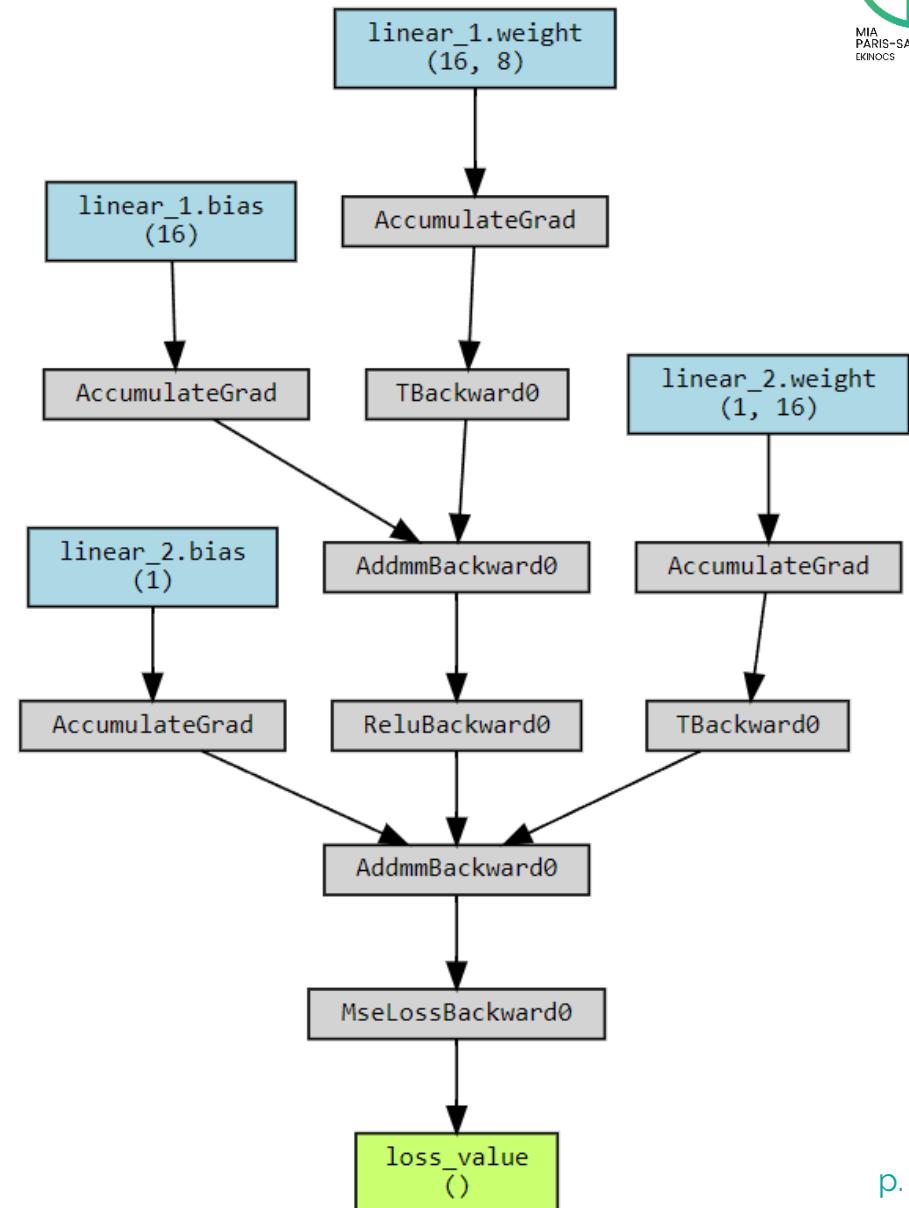


```
class MyMiniMLP(torch.nn.Module) :
    def __init__(self) :
        super(MyMiniMLP, self).__init__()
        self.linear_1 = torch.nn.Linear(8, 16)
        self.activation = torch.nn.ReLU()
        self.linear_2 = torch.nn.Linear(16, 1)

    def forward(self, x) :
        x = self.linear_1(x)
        x = self.activation(x)
        y_hat = self.linear_2(x)

    return y_hat

my_mini_mlp = MyMiniMLP()
x0 = torch.randn(1,8)
y_hat = my_mini_mlp(x0)
loss_function = torch.nn.MSELoss()
loss_value = loss_function(y_hat, torch.tensor([0.0]))
```



# > Philosophy of pytorch

- Optimizer
  - Initialized by passing the parameters to optimize
  - And a learning rate (size of the step of Gradient Descent)
  - Example: `optimizer = torch.optim.SGD(params=neural_network.parameters(), lr=learning_rate)`
- All the rest, we are free to manipulate as we like
  - Number of iterations, stopping condition
  - Even *how* to cumulate gradients (will come in handy later)



## Questions?

### Bibliography

- E. Stevens, L. Antiga. 2019. *Deep learning with pytorch*
- <https://pytorch.org/tutorials/beginner/basics/intro.html>

Images and videos: unless otherwise stated, I stole them from the Internet. I hope they are not copyrighted, or that their use falls under the Fair Use clause, and if not, I am sorry. Please don't sue me.