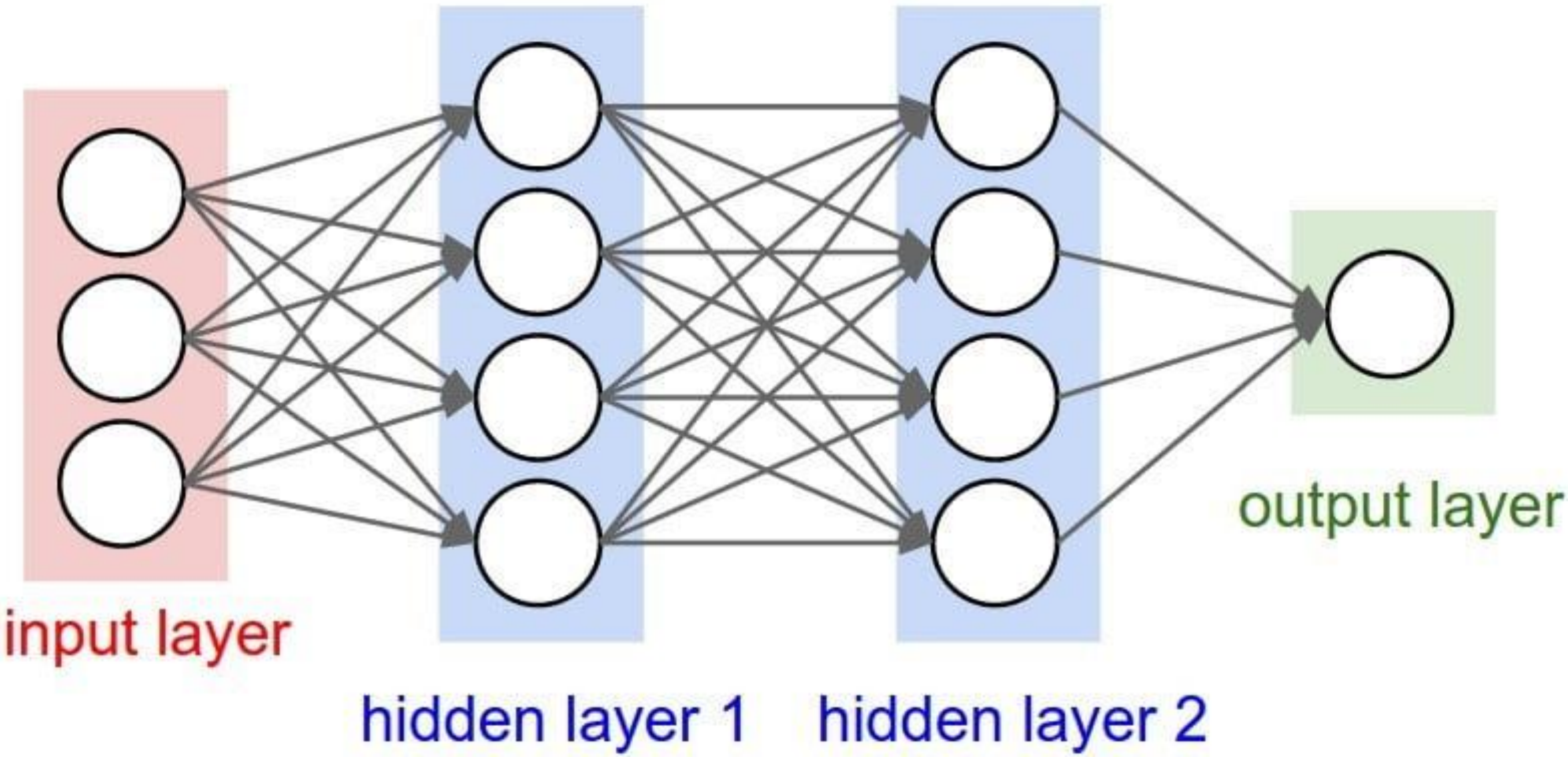
RÉSEAUX DE NOUERONS

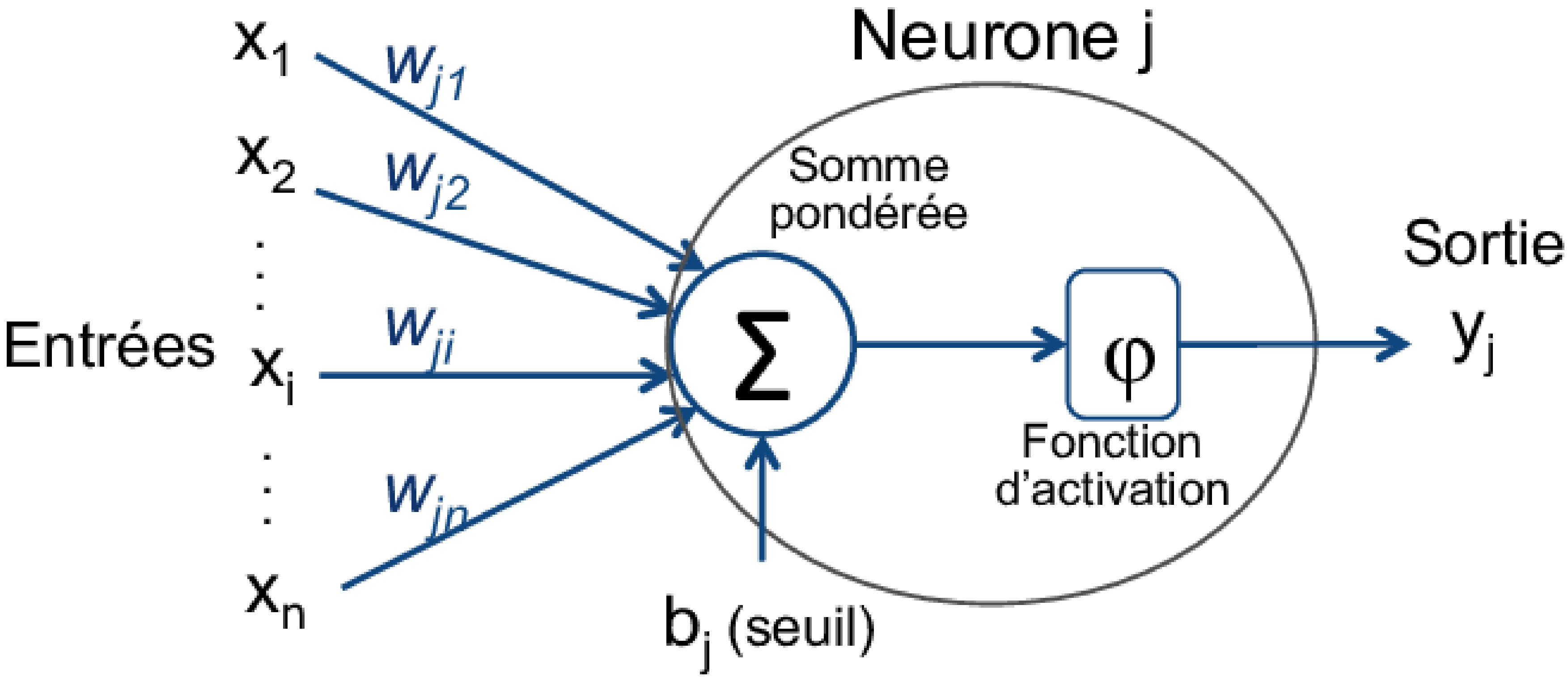
Les bases

Implémentation from scratch

Principe

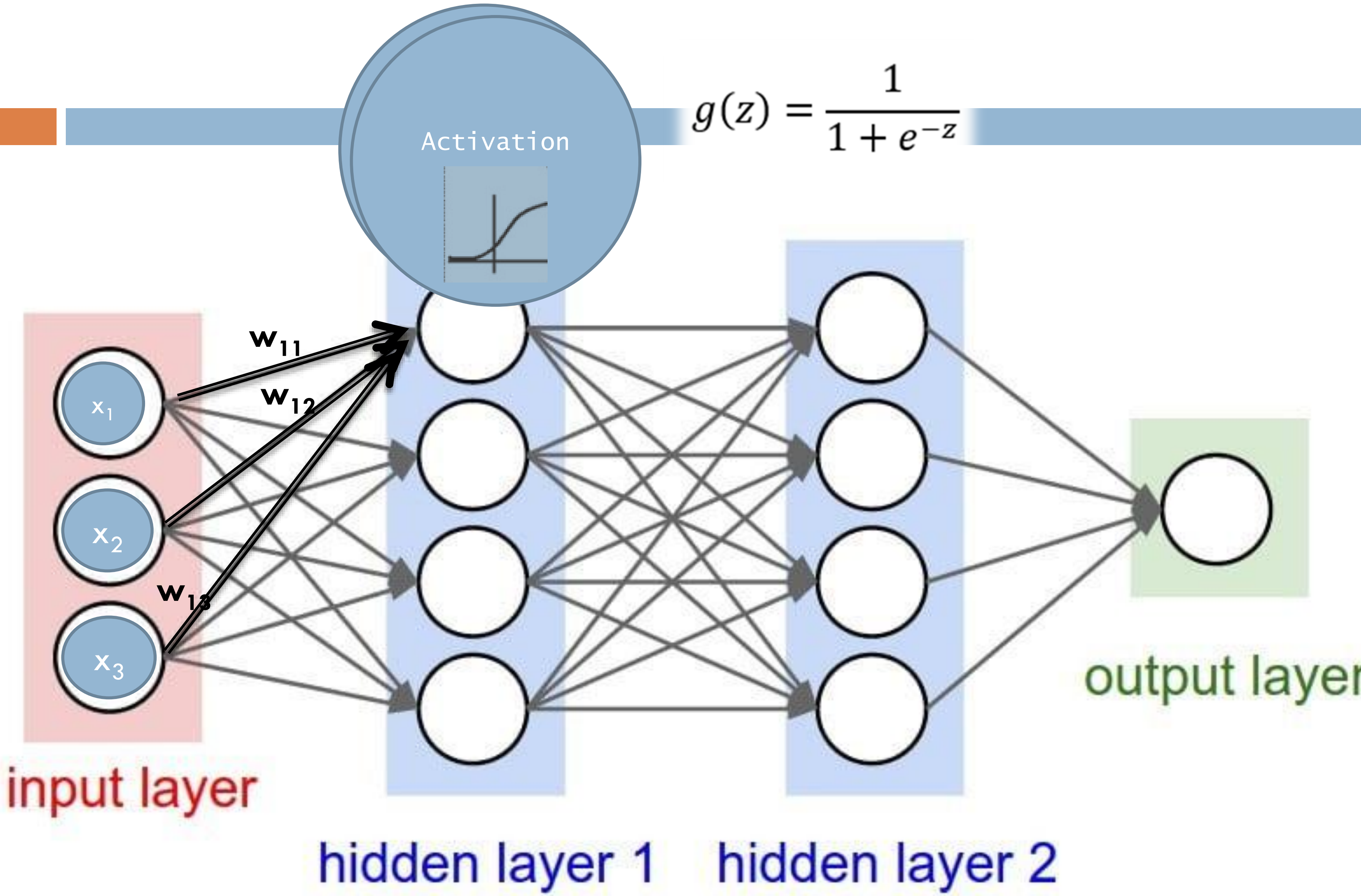


Perceptron



Neurone j
x_1
w_{j1}
x_2
w_{j2}
Somme pondérée
Sortie
Entrées
x_i
w_{ji}
Σ
φ
y_j
Fonction d'activation
w_{jn}
x_n
b_j (seuil)

$$g(z) = \frac{1}{1 + e^{-z}}$$


Activation
w_{11}
w_{12}
w_{13}
x_1
x_2
x_3
input layer
hidden layer 1
hidden layer 2
output layer

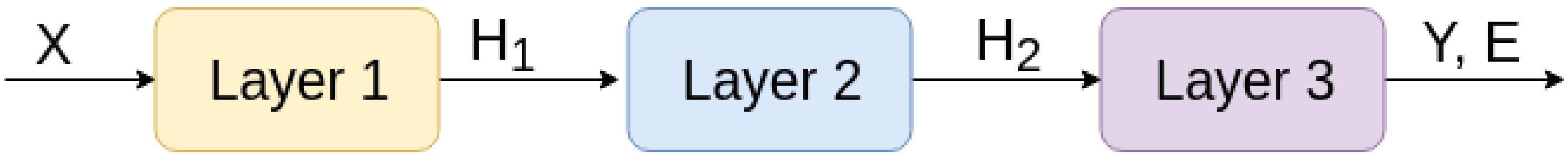
Réseau se compose de:

- Des couches
- Deux opérations
  - Propagation en avant
  - Propagation en arrière
- Des couches d’activation
- Une couche full connected
- Fonction de perte
- Opération d’apprentissage
- Opération de prédiction/classification

Couche (Layer)

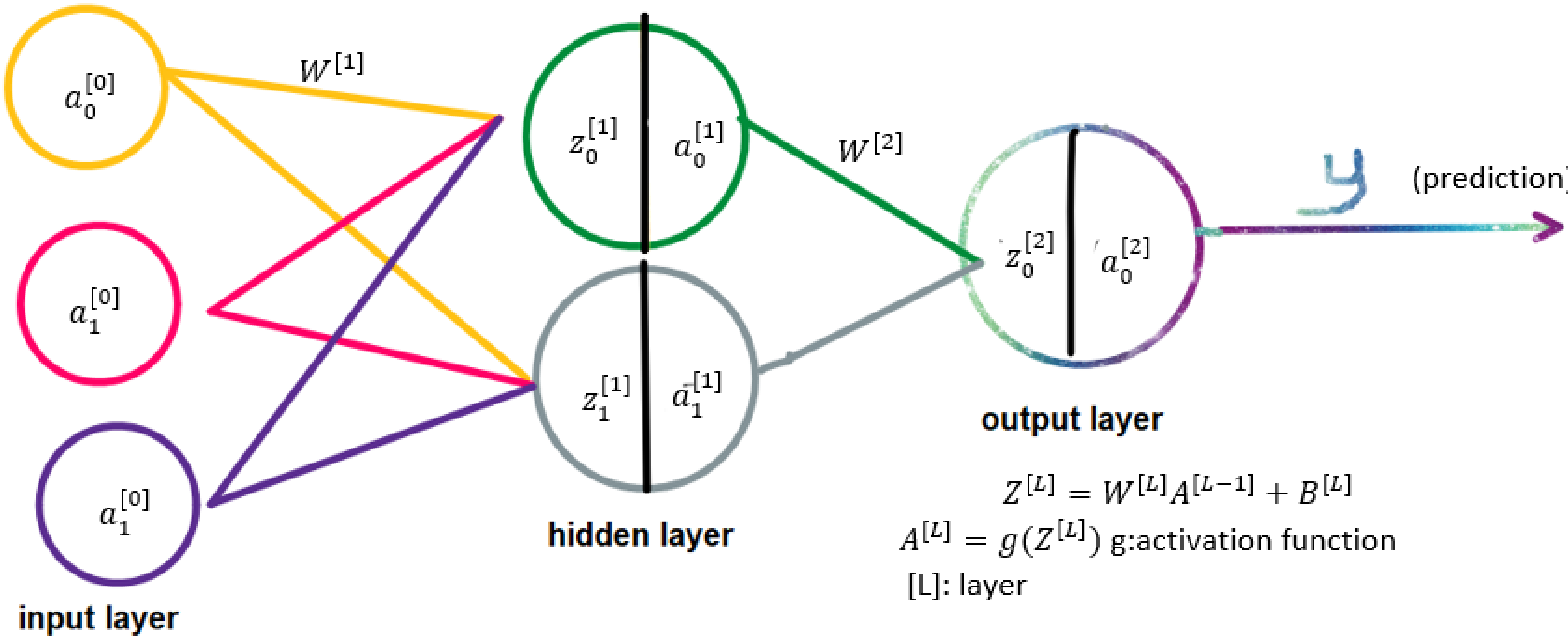
```python
# Base class
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    # computes the output Y of a layer for a given input X
    def forward_propagation(self, input):
        raise NotImplementedError

    # computes dE/dX for a given dE/dY (and update parameters if any)
    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError
```

Passe avant — Forward propagation



- On propage l'entrée **X** (image, son, texte, etc.) dans le réseau de neurones jusqu'à obtenir la sortie **Y**. Puis, on observe une erreur **E** qu'il faut diminuer.

Principe (propagation en avant)



$$Z^{[L]} = W^{[L]}A^{[L-1]} + B^{[L]}$$

$A^{[L]} = g(Z^{[L]})$ g:activation function

[L]: layer

Notation matricielle

$$X = \begin{bmatrix} x_1 & \dots & x_i \end{bmatrix} \quad W = \begin{bmatrix} w_{11} \dots w_{1j} \\ \vdots \quad \ddots \quad \vdots \\ w_{i1} \dots w_{ij} \end{bmatrix} \quad B = \begin{bmatrix} b_1 & \dots & b_j \end{bmatrix}$$

$$Y = X * W + B$$

Fonction de perte

- L'erreur du réseau, qui mesure le degré de performance du modèle pour une entrée donnée, doit être défini. Il existe de nombreuses façons de définir l'erreur, et l'une des plus connues est appelée **MSE — Mean Squared Error.**

$$E = \frac{1}{n} \sum_{i}^{n} (y_i^* - y_i)^2$$

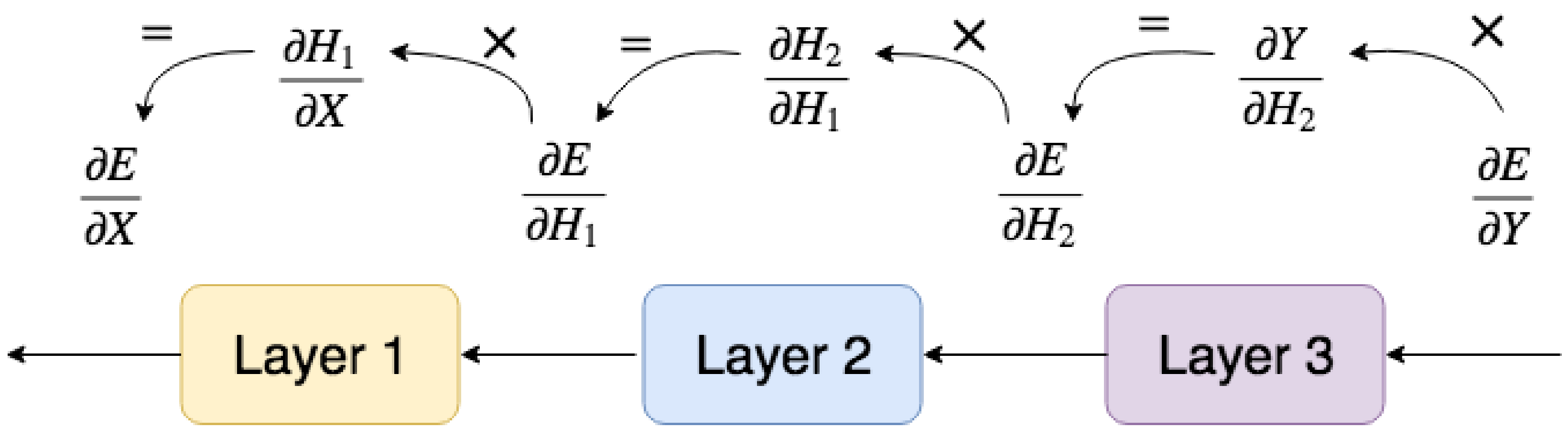
- Pour démunie la valeur de E, il faut ajuster les valeurs des poids Wi, pour cela il faut déterminer le degré de l'influence de Wi sur l'erreur.

Passe arrière — backward propagation

- Supposons que l'on donne à une couche la **dérivée de l'erreur** par rapport à **sa sortie** (∂E/∂Y), alors elle doit être capable de donner la **dérivée de l'erreur** par rapport à **son entrée** (∂E/∂X).

$$\frac{\partial E}{\partial X} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial Y}$$

Passe arrière — Backward propagation



Notation matricielle

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \cdots & \frac{\partial E}{\partial w_{1j}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{i1}} & \cdots & \frac{\partial E}{\partial w_{ij}} \end{bmatrix}$$

Notation matricielle

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_1} * \frac{\partial y_1}{\partial w_{ij}} + \cdots + \frac{\partial E}{\partial y_j} * \frac{\partial y_j}{\partial w_{ij}}$$

$$= \frac{\partial E}{\partial y_j} * x_i$$

Notation matricielle

$$\frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} * x_1 & \cdots & \frac{\partial E}{\partial y_j} * x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1} * x_i & \cdots & \frac{\partial E}{\partial y_j} * x_i \end{bmatrix}$$

$$= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} * \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$= X^t * \frac{\partial E}{\partial Y}$$

Notation matricielle

- Première formule qui permet d'ajuster les poids ! Calculons à présent $\partial E/\partial B$.

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} & \frac{\partial E}{\partial b_2} & \cdots & \frac{\partial E}{\partial b_j} \end{bmatrix}$$

Notation matricielle

- Encore une fois, ∂E/∂B doit-être de la même dimension que B, une dérivée par *bias*.

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_1} * \frac{\partial y_1}{\partial b_j} + \cdots + \frac{\partial E}{\partial y_j} * \frac{\partial y_j}{\partial b_j}$$
$$= \frac{\partial E}{\partial y_j}$$

- D'où,

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \frac{\partial E}{\partial y_2} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix}$$
$$= \frac{\partial E}{\partial Y}$$

Notation matricielle

- Il nous reste simplement à calculer ∂E/∂X pour que la couche précédente puisse faire les mêmes calculs.

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} & \frac{\partial E}{\partial x_2} & \cdots & \frac{\partial E}{\partial x_i} \end{bmatrix}$$

- Encore une fois, dérivation de fonctions composées :

$$\begin{aligned} \frac{\partial E}{\partial x_i} &= \frac{\partial E}{\partial y_1} * \frac{\partial y_1}{\partial x_i} + \ldots + \frac{\partial E}{\partial y_j} * \frac{\partial y_j}{\partial x_i} \\ &= \frac{\partial E}{\partial y_1} * w_{i1} + \ldots + \frac{\partial E}{\partial y_j} * w_{ij} \end{aligned}$$

Notation matricielle

- Nous pouvons écrire la matrice entière :

$$
\begin{aligned}
\frac{\partial E}{\partial X} &= \begin{bmatrix} (\frac{\partial E}{\partial y_1} * w_{11} + \ldots + \frac{\partial E}{\partial y_j} * w_{1j}) & \ldots & (\frac{\partial E}{\partial y_1} * w_{i1} + \ldots + \frac{\partial E}{\partial y_j} * w_{ij}) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \ldots & \frac{\partial E}{\partial y_j} \end{bmatrix} * \begin{bmatrix} w_{11} & \ldots & w_{i1} \\ \vdots & \ddots & \vdots \\ w_{1j} & \ldots & w_{ij} \end{bmatrix} \\
&= \frac{\partial E}{\partial Y} * W^t
\end{aligned}
$$

La couche FC

- Nous avons obtenu ces trois formules fondamentale pour la couche FC !

$$\frac{\partial E}{\partial X} = \frac{\partial E}{\partial Y} * W^t$$

$$\frac{\partial E}{\partial W} = X^t * \frac{\partial E}{\partial Y}$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

Fully Connected Layer

```python
from layer import Layer
import numpy as np

# inherit from base class Layer
class FCLayer(Layer):
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.weights = np.random.rand(input_size, output_size) - 0.5
        self.bias = np.random.rand(1, output_size) - 0.5

    # returns output for a given input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.input, self.weights) + self.bias
        return self.output

    # computes dE/dW, dE/dB for a given output_error=dE/dY.
    #Returns input_error=dE/dX.
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(output_error, self.weights.T)
        weights_error = np.dot(self.input.T, output_error)
        # dBias = output_error

        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * output_error
        return input_error
```

la couche d’activation

```python
from layer import Layer

# inherit from base class Layer
class ActivationLayer(Layer):
    def __init__(self, activation, activation_prime):
        self.activation = activation
        self.activation_prime = activation_prime

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # learning_rate is not used because there is no "learnable" parameters.
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

la couche d'activation

- Nous pouvons également écrire certaines fonctions d'activation et leurs dérivées dans un fichier séparé. Elles seront utilisées plus tard pour créer une couche d'activation.

```python
import numpy as np

# activation function and its derivative
def tanh(x):
    return np.tanh(x);

def tanh_prime(x):
    return 1-np.tanh(x)**2;
```

Fonction de perte

- L'erreur du réseau, qui mesure le degré de performance du modèle pour une entrée donnée, doit être défini. Il existe de nombreuses façons de définir l'erreur, et l'une des plus connues est appelée **MSE — Mean Squared Error.**

$$E = \frac{1}{n}\sum_{i}^{n}(y_i^* - y_i)^2$$

Sa dérivée

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_i} \end{bmatrix}$$

$$= \frac{2}{n} \begin{bmatrix} y_1 - y_1^* & \dots & y_i - y_i^* \end{bmatrix}$$

$$= \frac{2}{n}(Y - Y^*)$$

Il suffira de donner cette valeur à la **dernière** couche lors de la passe arrière, ce qui lui permettra d'ajuster ses paramètres, puis elle calculera **∂E/∂X** qu'elle passera à la couche d'avant, qui fera le même procédé à son tour, etc.

Fonction de perte

```
import numpy as np

# loss function and its derivative
def mse(y_true, y_pred):
    return np.mean(np.power(y_true-y_pred, 2));

def mse_prime(y_true, y_pred):
    return 2*(y_pred-y_true)/y_true.size;
```

Classe Network

```python
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)

        return result
```

Classe Network

```
    # train the network
    def fit(self, x_train, y_train, epochs, learning_rate):
        # sample dimension first
        samples = len(x_train)

        # training loop
        for i in range(epochs):
            err = 0
            for j in range(samples):
                # forward propagation
                output = x_train[j]
                for layer in self.layers:
                    output = layer.forward_propagation(output)

                # compute loss (for display purpose only)
                err += self.loss(y_train[j], output)

                # backward propagation
                error = self.loss_prime(y_train[j], output)
                for layer in reversed(self.layers):
                    error = layer.backward_propagation(error, learning_rate)

            # calculate average error on all samples
            err /= samples
            print('epoch %d/%d   error=%f' % (i+1, epochs, err))
```

Construire le réseau de neurone

Résoudre XOR

```python
import numpy as np

from network import Network
from fc_layer import FCLayer
from activation_layer import ActivationLayer
from activations import tanh, tanh_prime
from losses import mse, mse_prime

# training data
x_train = np.array([[[0,0]], [[0,1]], [[1,0]], [[1,1]]])
y_train = np.array([[[0]], [[1]], [[1]], [[0]]])

# network
net = Network()
net.add(FCLayer(2, 3))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(tanh, tanh_prime))

# train
net.use(mse, mse_prime)
net.fit(x_train, y_train, epochs=1000, learning_rate=0.1)

# test
out = net.predict(x_train)
print(out)
```