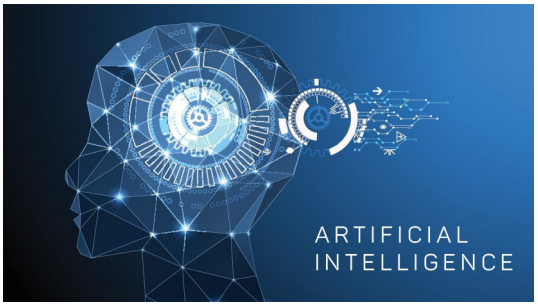
# Chapitre 8 : Réseaux de neurones en pratique pour l'apprentissage automatique




Pr. Mustapha BOURAHLA, Département d'Informatique, Université de M'Sila, Contact : mustapha.bourahla@univ-msila.dz

Mars 2021
v 1.0

# Table des matières

# Introduction

Ce chapitre vous montre comment programmer un réseau de neurones en Python pour réaliser un apprentissage automatique.

# Introduction


On supposera à partir d'ici que vous avez déjà quelques connaissances fondamentales sur le modèle du neurone artificiel et les réseaux de neurones. Dans ce chapitre nous présentons l'algorithme des gradients descendants. Encore une fois, le but ici n'est pas d'expliquer les différentes applications possibles du machine learning, mais plutôt comment implémenter ces algorithmes.

# Couche par Couche

II

Gardons à l'esprit la démarche globale du machine learning :

1. Donner une entrée au modèle.
2. Propager cette entrée à travers le réseau de neurones jusqu'à récupérer la sortie.
3. Une fois la sortie obtenue, nous pouvons la comparer à la sortie voulue et donc calculer une erreur.
4. On ajuste les paramètres du modèle pour diminuer l'erreur précédemment calculée. Pour cela on soustrait à chaque paramètre la dérivée de l'erreur par rapport à lui-même (gradient descendant).
5. On recommence à l'étape 1.

L'étape la plus importante est la 4ième. Nous voulons être capable de créer autant de couches que l'on veut, de n'importe quel type, et d'utiliser n'importe quelle fonction d'activation. Seulement, en changeant l'architecture du réseau de neurones, on change également la formule littérale du calcul de la dérivée de l'erreur par rapport aux paramètres.

Le but est donc de faire une implémentation qui fait abstraction de l'architecture du modèle (comme dans Keras). Pour cela, nous devons implémenter chaque couche séparément.

### *Ce que chaque couche doit faire*

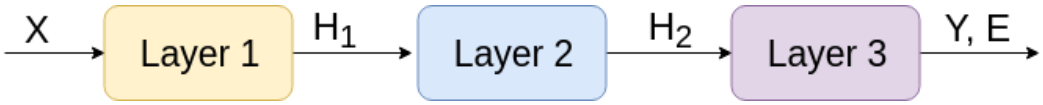
Quelle que soit la couche que nous codons (fully connected, convolutional, maxpooling, dropout, etc.), il y aura toujours au moins deux éléments fondamentaux : une entrée et une sortie.

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

### *Passe avant — forward propagation*

Nous pouvons dès lors préciser une propriété importante : la sortie d'une couche est l'entrée de la couche suivante.

Cette partie est ce qu'on appelle la 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 maintenant diminuer.



# Descente de Gradient

III

Fondamentalement, nous voulons changer un paramètre dans le réseau (appelez-le w) afin que l'erreur totale E diminue. Il existe un moyen intelligent de le faire (sans changer le paramètre au hasard) qui est le suivant:

$$w \leftarrow w - \alpha \frac{\partial E}{\partial w}$$

Où α est un paramètre dans l'intervalle [0,1] que nous fixons et qui est appelé le taux d'apprentissage. Quoi qu'il en soit, l'important ici est ∂E/∂w (la dérivée de E par rapport à w). Nous devons être en mesure de trouver la valeur de cette expression pour n'importe quel paramètre du réseau, quelle que soit son architecture.

*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).

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

L'erreur E est un scalaire (un nombre), et X et Y sont des matrices. La notation ci-dessus (abusive) signifie ceci:

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

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

Laissons de côté ∂E/∂X pour l'instant et concentrons-nous sur ∂E/∂Y. Si une couche a accès à ∂E/∂Y où Y est sa propre sortie, alors nous pouvons très facilement calculer la dérivée de l'erreur par rapport à ses paramètres ∂E/∂W (pour l'étape de l'ajustement), et cela, indépendamment de l'architecture globale du réseau de neurones! Il suffit d'utiliser la règle de dérivation des fonctions composées :

$$\frac{\partial E}{\partial w} = \sum_j \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w}$$

Etant donnée ∂E/∂Y nous pouvons donc calculer ∂E/∂W, et donc ajuster les paramètres de la couche!

### *Pourquoi avons-nous besoin de ∂E/∂X ?*

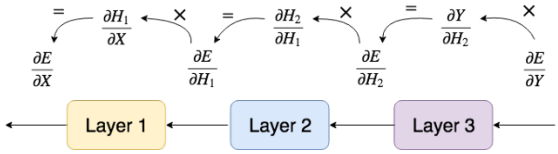
N'oubliez pas, la sortie d'une couche est l'entrée de la couche suivante. Donc ∂E/∂X pour une couche sera ∂E /∂Y pour la couche précédente! Une fois munit de son propre ∂E/∂Y, la couche précédente pourra à son tour ajuster ses paramètres. Pour calculer ∂E/∂X on utilise encore une fois la règle de dérivation des fonctions composées :

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

Cette astuce est la clé de compréhension de la backward propagation! Apres cela, nous pourrons programmer un réseau de neurones convolutif en un rien de temps.

### *Un super diagramme*

C'est ce que j'ai décrit plus tôt. La couche 3 va mettre à jour ses paramètres en utilisant ∂E/∂Y, puis va passer ∂E/∂H2 à la couche précédente, qui est son propre “∂E/∂Y”. La couche 2 va alors faire de même, et ainsi de suite.




Ça peut paraître abstrait maintenant mais deviendra très clair part la suite. Nous pouvons dès lors créer notre première classe en Python qui sera une classe abstraite représentant une couche.

# Classe abstraite : Layer

IV

*Classe abstraite : Layer*

- La classe abstraite Layer, dont les autres couches hériteront, contient les caractéristiques communes à toutes les couches :
    1. une entrée,
    2. une sortie,
    3. une fonction qui fait la passe avant, et une pour la passe arrière.
- Comme vous pouvez le constater, il existe un paramètre supplémentaire dans backward_propagation qu'il n'est pas mentionné, c'est le learning_rate.
- Ce paramètre devrait être une politique de mise à jour, ou un optimiseur, comme ils l'appellent dans Keras, mais par souci de simplicité, nous allons simplement passer le learning rate et mettre à jour nos paramètres en utilisant la descente de gradient.

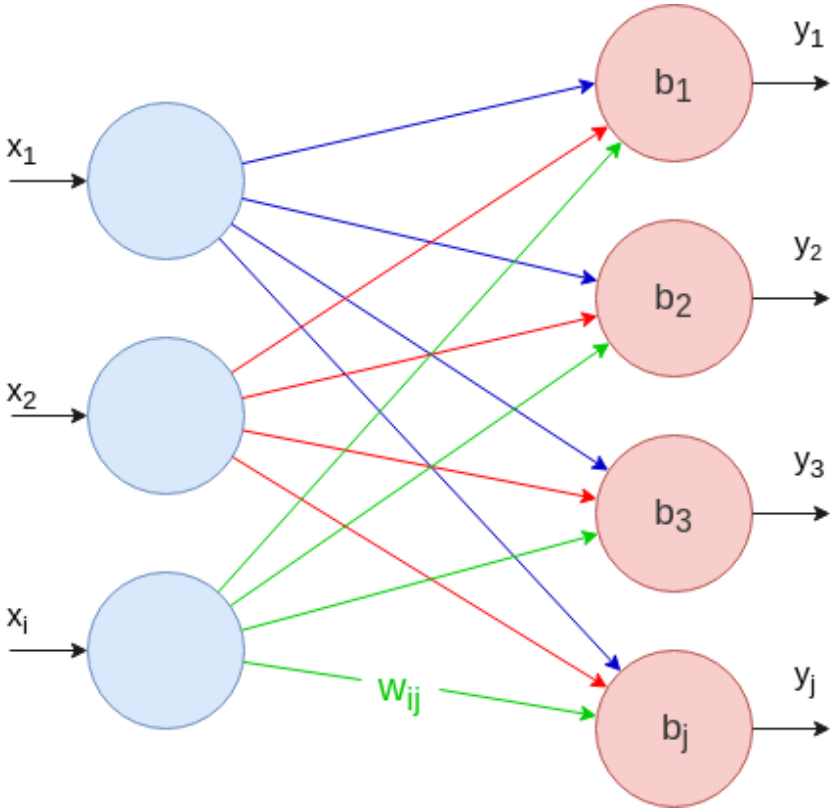
```
1  # Base class
2  class Layer:
3      def __init__(self):
4          self.input = None
5          self.output = None
6
7      # computes the output Y of a layer for a given input X
8      def forward_propagation(self, input):
9          raise NotImplementedError
10
11     # computes dE/dX for a given dE/dY (and update parameters if any)
12     def backward_propagation(self, output_error, learning_rate):
13         raise NotImplementedError
```

# Fully Connected Layer

V

*Fully Connected Layer*

Définissons et implémentons maintenant le premier type de couche: Fully Connected Layer ou FC Layer. Les couches FC sont les couches les plus élémentaires car tous les neurones d'entrée sont connectés à tous les neurones de sortie.




*Forward Propagation*

La valeur de chaque neurone de sortie peut être calculée comme suit :

$$y_j = b_j + \sum_i x_i w_{ij}$$

La notation matricielle permet de simplifier ce calcul :

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

$$Y = XW + B$$

Nous en avons fini avec la passe avant. Traitons maintenant la passe arrière de la couche FC.

Notez que je n'utilise aucune fonction d'activation, c'est parce que nous allons l'implémenter dans une couche à part !

### *Backward Propagation*

Comme nous l'avons dit, supposons que nous ayons une matrice contenant la dérivée de l'erreur par rapport à la sortie de cette couche (∂E/∂Y). Nous avons besoin de :

La dérivée de l'erreur par rapport aux paramètres (∂E/∂W, ∂E/∂B)

La dérivée de l'erreur par rapport à l'entrée (∂E/∂X)

Commençons par ∂E/∂W. Cette matrice doit avoir la même taille que W : ixj où i est le nombre de neurones d'entrée et j le nombre de neurones de sortie. Nous avons besoin d'une dérivée pour chaque paramètre :

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

En utilisant la règle de dérivation des fonctions composées, on écrit :

$$\begin{aligned} \frac{\partial E}{\partial W} &= \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \cdots & \frac{\partial E}{\partial y_j} x_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial y_1} x_i & \cdots & \frac{\partial E}{\partial y_j} x_i \end{bmatrix} \\ &= \begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} \begin{bmatrix} \frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_j} \end{bmatrix} \\ &= X^t \frac{\partial E}{\partial Y} \end{aligned}$$

Nous avons notre première formule qui permet d'ajuster les poids ! Calculons à présent ∂E/∂B.

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

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

$$\begin{aligned} \frac{\partial E}{\partial b_j} &= \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial b_j} + \ldots + \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial b_j} \\ &= \frac{\partial E}{\partial y_j} \end{aligned}$$

D'où,

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

Maintenant que nous avons ∂E/∂W et ∂E/∂B nous pouvons ajuster tout les paramètres de cette couche de sorte à diminuer l'erreur! 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 :

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

Nous pouvons écrire la matrice entière :

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

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

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

## *Coder la couche FC*

Créez un nouveau fichier qui contiendra le code suivant :

```
1 from layer import Layer
2 import numpy as np
3
4 # inherit from base class Layer
5 class FCLayer(Layer):
6     # input_size = number of input neurons
7     # output_size = number of output neurons
8     def __init__(self, input_size, output_size):
9         self.weights = np.random.rand(input_size, output_size) - 0.5
10        self.bias = np.random.rand(1, output_size) - 0.5
11
12    # returns output for a given input
13    def forward_propagation(self, input_data):
14        self.input = input_data
15        self.output = np.dot(self.input, self.weights) + self.bias
16        return self.output
17
18    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns input_error=dE
  /dX.
19    def backward_propagation(self, output_error, learning_rate):
20        input_error = np.dot(output_error, self.weights.T)
21        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
```

# Couche d'activation

VI

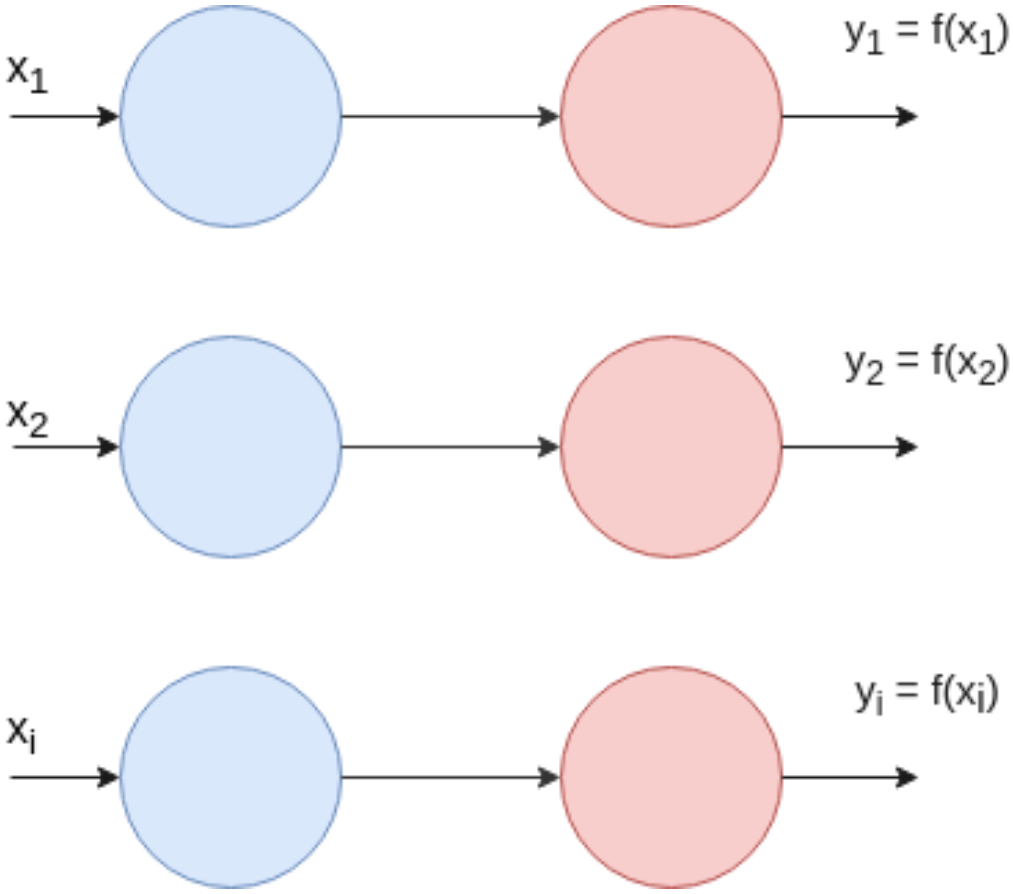
*Couche d'activation*

Tous les calculs que nous avons faits jusqu'à présent étaient complètement linéaires. La machine n'apprendra rien avec ce genre de modèle. Nous devons ajouter une non-linéarité au modèle en appliquant des fonctions non linéaires à la sortie de certaines couches.

Nous devons maintenant refaire tout le processus pour ce nouveau type de couche !

Pas de soucis, ça va être bien plus rapide car il n'y a pas de paramètres entraînable. Il suffit de calculer ∂E/∂X.

Nous appellerons f et f' la fonction d'activation et sa dérivée, respectivement.




*Forward Propagation*

Comme vous le verrez, c'est assez simple. Pour une entrée X donnée, la sortie est simplement la fonction d'activation appliquée à chaque élément de X. Ce qui signifie que l'entrée et la sortie ont la même dimension.

$$
\begin{aligned}
Y &= \begin{bmatrix} f(x_1) & \dots & f(x_i) \end{bmatrix} \\
&= f(X)
\end{aligned}
$$

### *Backward Propagation*

Étant donné ∂E/∂Y, nous voulons calculer ∂E/∂X.

$$
\begin{aligned}
\frac{\partial E}{\partial X} &= \begin{bmatrix} \frac{\partial E}{\partial x_1} & \dots & \frac{\partial E}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1}\frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial x_i} \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} f'(x_1) & \dots & \frac{\partial E}{\partial y_i} f'(x_i) \end{bmatrix} \\
&= \begin{bmatrix} \frac{\partial E}{\partial y_1} & \dots & \frac{\partial E}{\partial y_i} \end{bmatrix} \odot \begin{bmatrix} f'(x_1) & \dots & f'(x_i) \end{bmatrix} \\
&= \frac{\partial E}{\partial Y} \odot f'(X)
\end{aligned}
$$

Attention, nous utilisons ici une multiplication scalaire (élément par élément) entre les deux matrices.

### *Coder la couche d'activation*

Le code pour la couche d'activation est aussi très simple.

```
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
```

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

```
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;
```

# Couche convolution

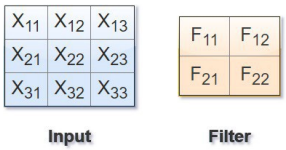
VII

*Fondamental : Propagation avant et arrière dans un réseau neuronal convolutif*

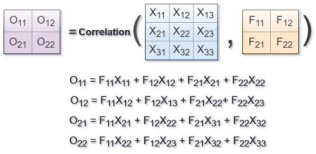
Nous démontrons l'utilisation de l'opération de convolution pour effectuer la propagation arrière dans un CNN

*Différence entre l'opération de convolution et la corrélation.*

Considérons l'entrée et le filtre qui va être utilisé pour effectuer la convolution comme indiqué ci-dessous.

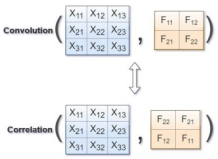



*Ensuite, la corrélation de la matrice de filtre avec la matrice d'entrée est décrite dans la figure ci-dessous*




$$O_{11} = F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22}$$
$$O_{12} = F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23}$$
$$O_{21} = F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32}$$
$$O_{22} = F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33}$$

*Maintenant, la convolution de la matrice de filtre avec l'image d'entrée est la même que la rotation du filtre de 180 degrés et ensuite l'exécution de la corrélation de la matrice de filtre tournée avec la matrice d'entrée.*




Comme on peut le voir sur l'image ci-dessus, l'opération de convolution est la même que celle de l'opération de corrélation mais avec un filtre tourné.

*Propagation vers l'avant et vers l'arrière à l'aide de l'opération de convolution.*

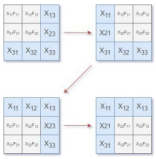
Remarque: Pour dériver l'équation des gradients pour les valeurs de filtre et les valeurs de matrice d'entrée, nous considérerons que l'opération de convolution est la même que l'opération de corrélation, juste pour plus de simplicité.

Par conséquent, l'opération de convolution peut être écrite comme décrit dans la figure ci-dessous.

$$\begin{bmatrix} O_{11} & O_{12} \\ O_{21} & O_{22} \end{bmatrix} = \text{Convolution}\left( \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix} \right)$$

$$O_{11} = F_{11}X_{11} + F_{12}X_{12} + F_{21}X_{21} + F_{22}X_{22}$$
$$O_{12} = F_{11}X_{12} + F_{12}X_{13} + F_{21}X_{22} + F_{22}X_{23}$$
$$O_{21} = F_{11}X_{21} + F_{12}X_{22} + F_{21}X_{31} + F_{22}X_{32}$$
$$O_{22} = F_{11}X_{22} + F_{12}X_{23} + F_{21}X_{32} + F_{22}X_{33}$$

*Il peut être visualisé dans la figure ci-dessous.*



*Maintenant, pour calculer les gradients du filtre «F» par rapport à l'erreur «E», les équations suivantes doivent être résolues.*

$$\frac{\partial E}{\partial F_{11}} = \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{11}}$$
$$\frac{\partial E}{\partial F_{12}} = \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{12}}$$
$$\frac{\partial E}{\partial F_{21}} = \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{21}}$$
$$\frac{\partial E}{\partial F_{22}} = \frac{\partial E}{\partial O_{11}}\frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial E}{\partial O_{12}}\frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial E}{\partial O_{21}}\frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial E}{\partial O_{22}}\frac{\partial O_{22}}{\partial F_{22}}$$

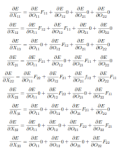
*qui évalue à*

$$\frac{\partial E}{\partial F_{11}} = \frac{\partial E}{\partial O_{11}}X_{11} + \frac{\partial E}{\partial O_{12}}X_{12} + \frac{\partial E}{\partial O_{21}}X_{21} + \frac{\partial E}{\partial O_{22}}X_{22}$$
$$\frac{\partial E}{\partial F_{12}} = \frac{\partial E}{\partial O_{11}}X_{12} + \frac{\partial E}{\partial O_{12}}X_{13} + \frac{\partial E}{\partial O_{21}}X_{22} + \frac{\partial E}{\partial O_{22}}X_{23}$$
$$\frac{\partial E}{\partial F_{21}} = \frac{\partial E}{\partial O_{11}}X_{21} + \frac{\partial E}{\partial O_{12}}X_{22} + \frac{\partial E}{\partial O_{21}}X_{31} + \frac{\partial E}{\partial O_{22}}X_{32}$$
$$\frac{\partial E}{\partial F_{22}} = \frac{\partial E}{\partial O_{11}}X_{22} + \frac{\partial E}{\partial O_{12}}X_{23} + \frac{\partial E}{\partial O_{21}}X_{32} + \frac{\partial E}{\partial O_{22}}X_{33}$$

*Si nous regardons de près cette équation ci-dessus peut être écrite sous la forme de notre opération de convolution.*

$$\begin{bmatrix} \partial E/\partial F_{11} & \partial E/\partial F_{12} \\ \partial E/\partial F_{21} & \partial E/\partial F_{22} \end{bmatrix} = \text{Convolution}\left( \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix} \right)$$

*De même, nous pouvons trouver les gradients de la matrice d'entrée «X» par rapport à l'erreur «E».*

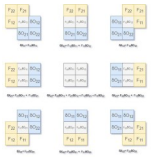


*Maintenant, le calcul ci-dessus peut être obtenu par un autre type d'opération de convolution connue sous le nom de convolution complète.*

Afin d'obtenir les gradients de la matrice d'entrée, nous devons faire pivoter le filtre de 180 degrés et calculer la convolution complète du filtre tourné par les gradients de la sortie par rapport à l'erreur, comme représenté dans l'image ci-dessous.

$$\begin{bmatrix} \partial E/\partial X_{11} & \partial E/\partial X_{12} & \partial E/\partial X_{13} \\ \partial E/\partial X_{21} & \partial E/\partial X_{22} & \partial E/\partial X_{23} \\ \partial E/\partial X_{31} & \partial E/\partial X_{32} & \partial E/\partial X_{33} \end{bmatrix} = \text{Full\_Convolution} \left( \begin{bmatrix} \partial E/\partial O_{11} & \partial E/\partial O_{12} \\ \partial E/\partial O_{21} & \partial E/\partial O_{22} \end{bmatrix}, \begin{bmatrix} F_{22} & F_{21} \\ F_{12} & F_{11} \end{bmatrix} \right)$$

*La convolution complète peut être visualisée comme exécutant la procédure comme représenté dans la figure ci-dessous.*



- Par conséquent, à la fois la propagation vers l'avant et vers l'arrière peut être effectuée en utilisant l'opération de convolution.
- Pour calculer les gradients des couches de regroupement et de Relu, les gradients peuvent être calculés en suivant la même procédure d'utilisation de la règle de chaîne des dérivés.

la couche de convolution

```
from layer import Layer
from scipy import signal
import numpy as np

## Math behind this layer can found at :
## https://medium.com/@2017csm1006/forward-and-backpropagation-in-convolutional-neural-network-4dfa96d7b37e

# inherit from base class Layer
# This convolutional layer is always with stride 1
class ConvLayer(Layer):
    # input_shape = (i,j,d)
    # kernel_shape = (m,n)
    # layer_depth = output_depth
    def __init__(self, input_shape, kernel_shape, layer_depth):
        self.input_shape = input_shape
        self.input_depth = input_shape[2]
        self.kernel_shape = kernel_shape
        self.layer_depth = layer_depth
        self.output_shape = (input_shape[0]-kernel_shape[0]+1, input_shape[1]-kernel_shape[1]+1, layer_depth)
        self.weights = np.random.rand(kernel_shape[0], kernel_shape[1], self.input_depth, layer_depth) - 0.5
        self.bias = np.random.rand(layer_depth) - 0.5

    # returns output for a given input
    def forward_propagation(self, input):
        self.input = input
        self.output = np.zeros(self.output_shape)

        for k in range(self.layer_depth):
            for d in range(self.input_depth):
                self.output[:,:,k] += signal.correlate2d(self.input[:,:,d], self.weights[:,:,d,k], 'valid') + self.bias[k]

        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):
        in_error = np.zeros(self.input_shape)
        dWeights = np.zeros((self.kernel_shape[0], self.kernel_shape[1], self.input_depth, self.layer_depth))
        dBias = np.zeros(self.layer_depth)

        for k in range(self.layer_depth):
            for d in range(self.input_depth):
                in_error[:,:,d] += signal.convolve2d(output_error[:,:,k], self.weights[:,:,d,k], 'full')
                dWeights[:,:,d,k] = signal.correlate2d(self.input[:,:,d], output_error[:,:,k], 'valid')
            dBias[k] = self.layer_depth * np.sum(output_error[:,:,k])

        self.weights -= learning_rate*dWeights
        self.bias -= learning_rate*dBias
        return in_error
```

exemple

```
import numpy as np

from network import Network
from conv_layer import ConvLayer
from activation_layer import ActivationLayer
from activations import tanh, tanh_prime
from losses import mse, mse_prime

# training data
x_train = [np.random.rand(10,10,1)]
y_train = [np.random.rand(4,4,2)]

# network
net = Network()
net.add(ConvLayer((10,10,1), (3,3), 1))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(ConvLayer((8,8,1), (3,3), 1))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(ConvLayer((6,6,1), (3,3), 2))
net.add(ActivationLayer(tanh, tanh_prime))

# train
net.use(mse, mse_prime)
net.fit(x_train, y_train, epochs=1000, learning_rate=0.3)

# test
out = net.predict(x_train)
print("predicted = ", out)
print("expected = ", y_train)
```

# Flatten layer

VIII

- Les couches de convolution 2D traitant des données 2D (par exemple, des images) produisent généralement un tenseur tridimensionnel, les dimensions étant la résolution de l'image (moins la taille du filtre -1) et le nombre de filtres.
- Si vous utilisez un calque avec $N$ filtres de taille s sur des images $L \times H$, le résultat sera de dimension $(L-(s-1), H-(s-1), N)$.
- Vous avez $N$ images de résolution $(L-(s-1), H-(s-1))$.
- Cette structure est importante si vous souhaitez enchaîner des couches de convolution entre elles ou avec d'autres couches qui effectuent un traitement spatial (pooling, upscaling, etc.).
- Mais si vous voulez faire de la classification, dans les dernières étapes de votre réseau, vous voulez généralement utiliser des couches entièrement connectées qui ne prennent en compte aucune structure (spatiale ou autre) pour le traitement, vous voulez donc juste la sortie de dernières couches de convolution à être considérées comme un gros morceau de données non structurées.
- C'est ce que signifie «aplatir». Il brise la structure spatiale des données et transforme votre tenseur tridimensionnel $(L-(s-1), H-(s-1), N)$ en un tenseur monodimensionnel (un vecteur) de taille $(L-(s-1)) \times (H-(s-1)) \times N$.

```python
from layer import Layer

# inherit from base class Layer
class FlattenLayer(Layer):
    # returns the flattened input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = input_data.flatten().reshape((1,-1))
        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 output_error.reshape(self.input.shape)
```

```python
import numpy as np

from network import Network
from fc_layer import FCLayer
from conv_layer import ConvLayer
from flatten_layer import FlattenLayer
from activation_layer import ActivationLayer
from activations import tanh, tanh_prime
from losses import mse, mse_prime

from keras.datasets import mnist
from keras.utils import np_utils

```

```
# load MNIST from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# training data : 60000 samples
# reshape and normalize input data
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_train /= 255
# encode output which is a number in range [0,9] into a vector of size 10
# e.g. number 3 will become [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)

# same for test data : 10000 samples
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

# Network
net = Network()
net.add(ConvLayer((28, 28, 1), (3, 3), 1))  # input_shape=(28, 28, 1)    ;
  output_shape=(26, 26, 1)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FlattenLayer())                     # input_shape=(26, 26, 1)    ;
  output_shape=(1, 26*26*1)
net.add(FCLayer(26*26*1, 100))              # input_shape=(1, 26*26*1)   ;
  output_shape=(1, 100)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(100, 10))                   # input_shape=(1, 100)       ;
  output_shape=(1, 10)
net.add(ActivationLayer(tanh, tanh_prime))

# train on 1000 samples
# as we didn't implemented mini-batch GD, training will be pretty slow if we
  update at each iteration on 60000 samples...
net.use(mse, mse_prime)
net.fit(x_train[0:1000], y_train[0:1000], epochs=100, learning_rate=0.1)

# test on 3 samples
out = net.predict(x_test[0:3])
print("\n")
print("predicted values : ")
print(out, end="\n")
print("true values : ")
print(y_test[0:3])
```

# Fonction de perte

IX

*Fonction de perte*

Jusqu'à présent, pour une couche donnée, nous supposions que ∂E/∂Y était donné (par la couche suivante). Mais qu'advient-il de la dernière couche? Comment obtient-elle ∂E/∂Y? Nous le donnons simplement manuellement, et cela dépend de la façon dont nous définissons l'erreur.

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

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

Où y* et y désignent respectivement la sortie souhaitée et la sortie obtenue. Vous pouvez penser à la perte comme une dernière couche qui regroupe tous les neurones de sortie et les écrases en un seul neurone. Ce dont nous avons besoin maintenant, comme pour toutes les autres couches, c'est de définir ∂E/∂Y. Excepté que maintenant, nous avons enfin “atteint” E!

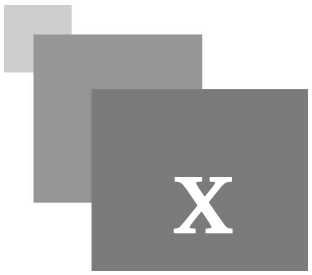
$$\begin{aligned}\frac{\partial E}{\partial Y} &= \begin{bmatrix}\frac{\partial E}{\partial y_1} & \cdots & \frac{\partial E}{\partial y_i}\end{bmatrix} \\ &= \frac{2}{n}\begin{bmatrix}y_1 - y_1^* & \cdots & y_i - y_i^*\end{bmatrix} \\ &= \frac{2}{n}(Y - Y^*)\end{aligned}$$

Et voilà! 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.

Nous pouvons implémenter la fonction de perte dans un fichier séparé. Elle sera utilisée lors de la création du réseau.

```
1 import numpy as np
2
3 # loss function and its derivative
4 def mse(y_true, y_pred):
5     return np.mean(np.power(y_true-y_pred, 2));
6
7 def mse_prime(y_true, y_pred):
8     return 2*(y_pred-y_true)/y_true.size;
```

# Classe Network




*Classe Network*

Bientôt fini, tenez bon ! Maintenant que nous avons tous nos bloque de code prêt à l'emploi, nous allons faire une classe appelée Network qui permettra de construire ces fameux réseaux de neurones !

J'ai commenté presque chaque partie du code, il ne devrait pas être trop compliqué à comprendre si vous avez saisi les étapes précédentes. Néanmoins, laissez un commentaire si vous avez des questions, je répondrai avec plaisir !

```
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

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

        # training loop
```

```
38          for i in range(epochs):
39              err = 0
40              for j in range(samples):
41                  # forward propagation
42                  output = x_train[j]
43                  for layer in self.layers:
44                      output = layer.forward_propagation(output)
45
46                  # compute loss (for display purpose only)
47                  err += self.loss(y_train[j], output)
48
49                  # backward propagation
50                  error = self.loss_prime(y_train[j], output)
51                  for layer in reversed(self.layers):
52                      error = layer.backward_propagation(error, learning_rate)
53
54              # calculate average error on all samples
55              err /= samples
56              print('epoch %d/%d   error=%f' % (i+1, epochs, err))
```

# Construire un réseau de neurone

XI

*Construire un réseau de neurone*

Construire un réseau de neurone

Enfin ! Nous pouvons utiliser notre classe pour créer un réseau de neurones avec autant de couches que nous voulons ! Nous allons construire deux réseaux de neurones: un simple XOR et un solveur MNIST.

### *Exemple : Résoudre XOR*

Commencer par un XOR est toujours important car c'est un moyen simple de savoir si le réseau apprend quelque chose.

```
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)
```

```
$ python xor.py
epoch 1/1000 error=0.322980
epoch 2/1000 error=0.311174
epoch 3/1000 error=0.307195
...
```

```
epoch 998/1000 error=0.000243
epoch 999/1000 error=0.000242
epoch 1000/1000 error=0.000242[
    array([[ 0.00077435]]),
    array([[ 0.97760742]]),
    array([[ 0.97847793]]),
    array([[-0.00131305]])
]
```

### *Remarque*

Je ne pense pas avoir besoin d'insister sur beaucoup de choses. Faites attention avec les données d'entrainement, vous devriez toujours avoir la dimension de l'échantillon en premier. Par exemple, avec le problème xor, la dimension des données d'entrées devrait être (4,1,2).

# exemple MNIST

XII

ddd

```
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

from keras.datasets import mnist
from keras.utils import np_utils

# load MNIST from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# training data : 60000 samples
# reshape and normalize input data
x_train = x_train.reshape(x_train.shape[0], 1, 28*28)
x_train = x_train.astype('float32')
x_train /= 255
# encode output which is a number in range [0,9] into a vector of size 10
# e.g. number 3 will become [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)

# same for test data : 10000 samples
x_test = x_test.reshape(x_test.shape[0], 1, 28*28)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

# Network
net = Network()
net.add(FCLayer(28*28, 100))                # input_shape=(1, 28*28)    ;
output_shape=(1, 100)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(100, 50))                   # input_shape=(1, 100)      ;
output_shape=(1, 50)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(50, 10))                    # input_shape=(1, 50)       ;
output_shape=(1, 10)
net.add(ActivationLayer(tanh, tanh_prime))

# train on 1000 samples
# as we didn't implemented mini-batch GD, training will be pretty slow if we
update at each iteration on 60000 samples...
net.use(mse, mse_prime)
net.fit(x_train[0:1000], y_train[0:1000], epochs=35, learning_rate=0.1)
```

```
43
44 # test on 3 samples
45 out = net.predict(x_test[0:3])
46 print("\n")
47 print("predicted values : ")
48 print(out, end="\n")
49 print("true values : ")
50 print(y_test[0:3])
51
```

# Le deep learning

XIII

cc

```python
import numpy as np

from network import Network
from fc_layer import FCLayer
from conv_layer import ConvLayer
from flatten_layer import FlattenLayer
from activation_layer import ActivationLayer
from activations import tanh, tanh_prime
from losses import mse, mse_prime

from keras.datasets import mnist
from keras.utils import np_utils

# load MNIST from server
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# training data : 60000 samples
# reshape and normalize input data
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_train = x_train.astype('float32')
x_train /= 255
# encode output which is a number in range [0,9] into a vector of size 10
# e.g. number 3 will become [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
y_train = np_utils.to_categorical(y_train)

# same for test data : 10000 samples
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
x_test = x_test.astype('float32')
x_test /= 255
y_test = np_utils.to_categorical(y_test)

# Network
net = Network()
net.add(ConvLayer((28, 28, 1), (3, 3), 1))  # input_shape=(28, 28, 1)    ;
  output_shape=(26, 26, 1)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FlattenLayer())                     # input_shape=(26, 26, 1)    ;
  output_shape=(1, 26*26*1)
net.add(FCLayer(26*26*1, 100))              # input_shape=(1, 26*26*1)   ;
  output_shape=(1, 100)
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(100, 10))                   # input_shape=(1, 100)       ;
  output_shape=(1, 10)
net.add(ActivationLayer(tanh, tanh_prime))

# train on 1000 samples
```

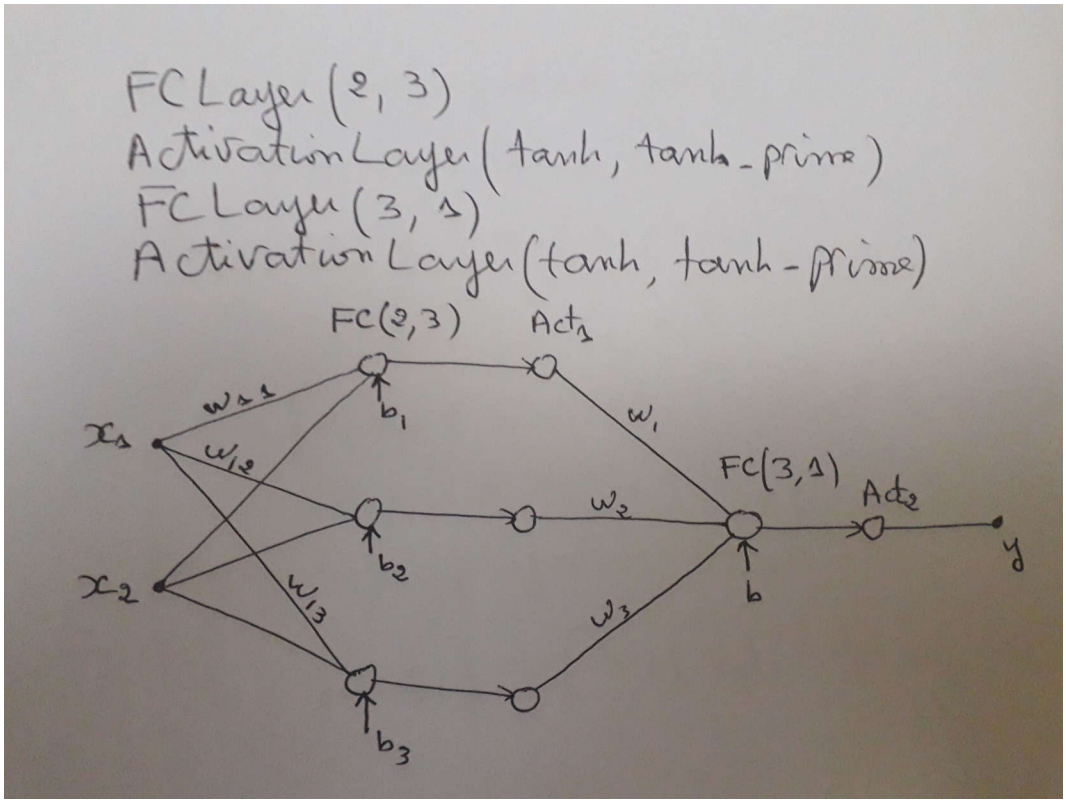
```
# as we didn't implemented mini-batch GD, training will be pretty slow if we update at each iteration on 60000 samples...
net.use(mse, mse_prime)
net.fit(x_train[0:1000], y_train[0:1000], epochs=100, learning_rate=0.1)

# test on 3 samples
out = net.predict(x_test[0:3])
print("\n")
print("predicted values : ")
print(out, end="\n")
print("true values : ")
print(y_test[0:3])
```

# Exercice :

XIV

Dessiner le schéma de réseau de neurones de l'exemple XOR et explique comment l'apprentissage et la prédiction se font



forward propagation FC(3,1) :

$$Y = XW + B$$

$$y = [x_1, x_2, x_3] \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} + [b] =$$

$$x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$

forward propagation $Act_2$ :

$$y = Act_2(x_1 w_1 + x_2 w_2 + x_3 w_3 + b)$$

forward propagation FC(2,3) :

$$Y = XW + B$$

$$[y_1, y_2, y_3] = [x_1, x_2] \cdot \begin{bmatrix} w_{11}, w_{12}, w_{13} \\ w_{21}, w_{22}, w_{23} \end{bmatrix} + [b_1, b_2, b_3]$$

$$= [x_1 w_{11} + x_2 w_{21} + b_1,\ x_1 w_{12} + x_2 w_{22} + b_2 + x_1 w_{13} + x_2 w_{23} + b_3]$$

forward propagation $Act_1$ :

$$[y_1, y_2, y_3] = [Act_1(y_1), Act_1(y_2), Act_1(y_3)]$$

backward propagation FC(3,1):

$$\frac{\partial E}{\partial B} = error = 2 * (y - y\_true) = \frac{\partial E}{\partial Y}$$

$$\frac{\partial E}{\partial x} = input\_error = [error] * \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

$$= [error * w_1 + error * w_2 + error * w_3]$$

$$= \frac{\partial E}{\partial x} = \frac{\partial E}{\partial Y} * W^T$$

$$\frac{\partial E}{\partial w} = X^T \cdot \frac{\partial E}{\partial Y} = weight\ error =$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} * error = \begin{bmatrix} x_1 * error \\ x_2 * error \\ x_3 * error \end{bmatrix}$$

update parameters:

weights = weights − learning-rate * weight-error

$= [w_1, w_2, w_3] - learning\text{-}rate * [x_1 * error,$

$x_2 * error, x_3 * error]$

$= [w_1 - \alpha * x_1 * error, w_2 - \alpha * x_2 * error,$

$w_3 - \alpha * x_3 * error]$

bias $= \alpha * output\_error$

$= \alpha * 2 * (y - y\_true)$

Return $input\_error\left(\frac{\partial E}{\partial x}\right) \Rightarrow error$

# Exercice :

XV

**Question**

Les couches de convolution 2D traitant des données 2D (par exemple, des images) produisent généralement un tenseur tridimensionnel, les dimensions étant la résolution de l'image (moins la taille du filtre -1) et le nombre de filtres.

Si vous utilisez un calque avec $N$ filtres de taille $s$ sur des images $L \times H$, le résultat sera de dimension $(L-(s-1), H-(s-1), N)$.

Vous avez $N$ images de résolution $(L-(s-1), H-(s-1))$.

Cette structure est importante si vous souhaitez enchaîner des couches de convolution entre elles ou avec d'autres couches qui effectuent un traitement spatial (pooling, upscaling, etc.).

Si L = H = 4, N = 2 alors nous avons 2 images de résolution 3 x 3

La taille du filtre est s = 2

$$x_{train}[0](3,3,2) = \left[\begin{bmatrix}1 & 1.5\\ 2 & 2.5\\ 3 & 3.5\end{bmatrix}, \begin{bmatrix}4 & 4.5\\ 5 & 5.5\\ 6 & 6.5\end{bmatrix}, \begin{bmatrix}7 & 7.5\\ 8 & 8.5\\ 9 & 9.6\end{bmatrix}\right]$$

$$y_{train}(2,2) = \left[\begin{bmatrix}90\\ 112\end{bmatrix}, \begin{bmatrix}156\\ 177\end{bmatrix}\right]$$

Comme nous avons deux images, il y a deux filtres. Le filtre est 2 x 2

$$weights = \left[\left[\begin{bmatrix}1\\ 1.5\end{bmatrix}, \begin{bmatrix}2\\ 2.5\end{bmatrix}\right], \left[\begin{bmatrix}3\\ 3.5\end{bmatrix}, \begin{bmatrix}4\\ 4.5\end{bmatrix}\right]\right]$$

Nous avons une seule sortie (profondeur de la couche = 1) :

$$B = \left[2\right]$$

Forme de output = (3-2+1, 3-2+1, 1) = (2,2,1)

$$x_{train}[0][:,:,0] = \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

$$weights[:,:,0,0] = \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$$

$$output1 = correlation(\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}, \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}) + B = \begin{bmatrix}39 & 49\\ 69 & 79\end{bmatrix}$$

$$x_{train}[0][:,:,1] = \begin{bmatrix} 1.5 & 2.5 & 3.5 \\ 4.5 & 5.5 & 6.5 \\ 7.5 & 8.5 & 9.5 \end{bmatrix}$$

$$weights[:,:,1,0] = \begin{bmatrix} 1.5 & 2.5 \\ 3.5 & 4.5 \end{bmatrix}$$

$$output2 = correlation(\begin{bmatrix} 1.5 & 2.5 & 3.5 \\ 4.5 & 5.5 & 6.5 \\ 7.5 & 8.5 & 9.5 \end{bmatrix}, \begin{bmatrix} 1.5 & 2.5 \\ 3.5 & 4.5 \end{bmatrix}) + B = \begin{bmatrix} 51 & 63 \\ 87 & 99 \end{bmatrix}$$

$$output = output1 + output2 = \begin{bmatrix} 90 & 112 \\ 156 & 178 \end{bmatrix}$$

**Calcul de l'erreur E du réseau (MSE) :**

$$E = \frac{1}{n}\sum_{i=1}^{n}(y_i^* - y_i)^2 = \frac{1}{4}(\begin{bmatrix} 90 & 112 \\ 156 & 177 \end{bmatrix} - \begin{bmatrix} 90 & 112 \\ 156 & 178 \end{bmatrix})^2 = 0.25$$

calcul des erreurs de sortie (dérivée de MSE) :

$$\frac{\partial E}{\partial Y} = \frac{2}{n}(Y - Y^*) = \frac{2}{4}(\begin{bmatrix} 90 & 112 \\ 156 & 178 \end{bmatrix} - \begin{bmatrix} 90 & 112 \\ 156 & 177 \end{bmatrix}) = 0.5$$

$$\frac{\partial E}{\partial W}[:,:,0,0] = correlation(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0.5 \end{bmatrix}) = \begin{bmatrix} 2.5 & 3 \\ 4 & 4.5 \end{bmatrix}$$

$$\frac{\partial E}{\partial W}[:,:,1,0] = correlation(\begin{bmatrix} 1.5 & 2.5 & 3.5 \\ 4.5 & 5.5 & 6.5 \\ 7.5 & 8.5 & 9.5 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0.5 \end{bmatrix}) = \begin{bmatrix} 2.75 & 3.25 \\ 4.25 & 4.75 \end{bmatrix}$$

$$\frac{\partial E}{\partial W} = \left[\left[\begin{bmatrix} 2.5 \\ 2.75 \end{bmatrix}, \begin{bmatrix} 3 \\ 3.25 \end{bmatrix}\right], \left[\begin{bmatrix} 4 \\ 4.25 \end{bmatrix}, \begin{bmatrix} 4.5 \\ 4.75 \end{bmatrix}\right]\right]$$

$$\frac{\partial E}{\partial X}[:,:,0] = convolution(\begin{bmatrix} 0 & 0 \\ 0 & 0.5 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.5 & 1 \\ 0 & 1.5 & 2 \end{bmatrix}$$

$$\frac{\partial E}{\partial X}[:,:,1] = convolution(\begin{bmatrix} 0 & 0 \\ 0 & 0.5 \end{bmatrix}, \begin{bmatrix} 1.5 & 2.5 \\ 3.5 & 4.5 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.75 & 1.25 \\ 0 & 1.75 & 2.25 \end{bmatrix}$$

$$\frac{\partial E}{\partial X} = \left[\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0.5 & 0.75 \\ 1 & 1.25 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1.5 & 1.75 \\ 2 & 2.25 \end{bmatrix}\right]$$

Mise à jour pour $\alpha = 0.1$ :

$$weights = weights - \alpha * \frac{\partial E}{\partial W} = \left[\left[\begin{bmatrix} 1 \\ 1.5 \end{bmatrix}, \begin{bmatrix} 2 \\ 2.5 \end{bmatrix}\right], \left[\begin{bmatrix} 3 \\ 3.5 \end{bmatrix}, \begin{bmatrix} 4 \\ 4.5 \end{bmatrix}\right]\right] - 0.1 \times \left[\left[\begin{bmatrix} 2.5 \\ 2.75 \end{bmatrix}, \begin{bmatrix} 3 \\ 3.25 \end{bmatrix}\right], \left[\begin{bmatrix} 4 \\ 4.25 \end{bmatrix}, \begin{bmatrix} 4.5 \\ 4.75 \end{bmatrix}\right]\right] =$$

$$bias = bias - \alpha * \frac{\partial E}{\partial B} = 2 - 0.1 \times 0.5 =$$

# Conclusion

Ce chapitre vous a montré comment programmer un réseau de neurones en Python pour réaliser un apprentissage automatique.