

# Reconnaissance vocale lors d'appel d'urgence grâce à un réseau de neurones

TRAN-THUONG Tien-Thinh [n°30903]

2021-2022

# Sommaire

## 1 Introduction

- Présentation de la problématique
- Reconnaissance vocale

## 2 Généralités sur les réseaux de neurones

- Du perceptron au réseau de neurones
- Rétropropagation
- Amélioration de la rétropropagation

## 3 Réalisations concrètes

- Reproduction du XOR
- Reconnaissance de chiffres écrits
- Convolution d'image
- Reconnaissance vocale de mots-clés

## 4 Annexe

- Mes classes
- Mes codes
- Compléments

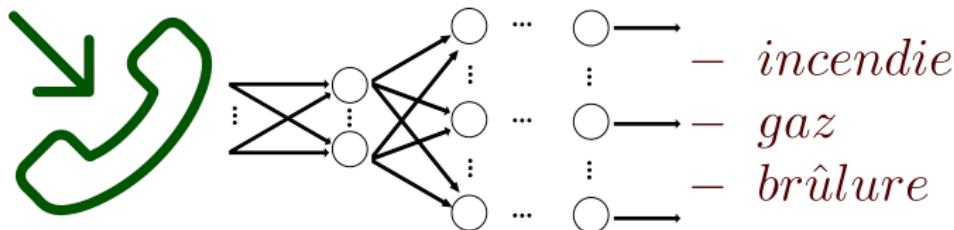
# La problématique

D'après le ministère de la Santé :

- **31 millions** d'appels d'urgence en 2018
- Seuls **69%** des appels décrochés dans la minute

## Objectif

Utiliser la reconnaissance vocale par réseau de neurones pour aider à classifier rapidement l'objet d'un appel.



# Les données

## 12 Mots-clés

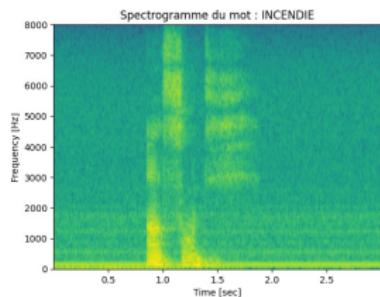
Le site du gouvernement recense ces mots-clés pour les numéros d'appel d'urgence :

n°15 malaise, hémorragie, brûlure, intoxication

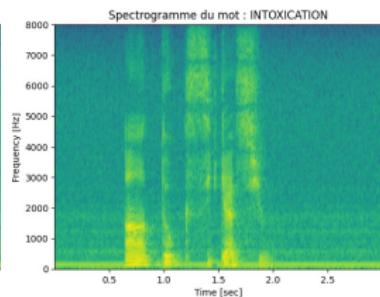
n°17 violences, agression, vol, cambriolage

n°18 incendie, gaz, effondrement, électrocution

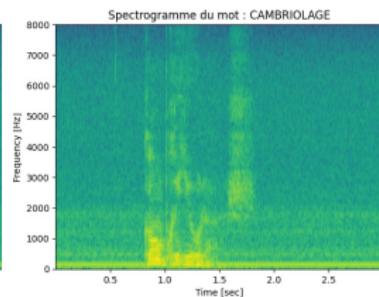
# Le traitement audio



(a) INCENDIE



(b) INTOXICATION

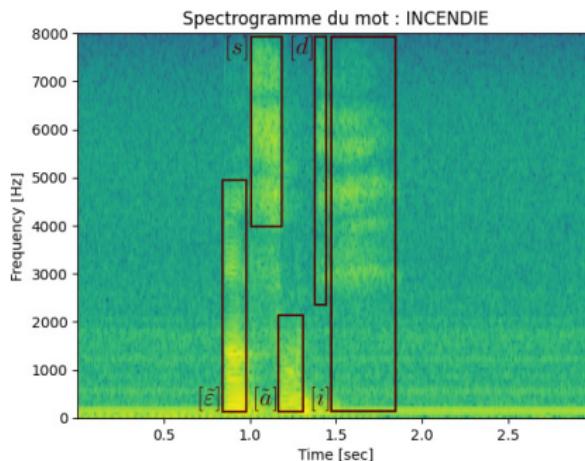


(c) CAMBRIOLAGE

# Le découpage en formant

## Formant - Définition Larousse

Fréquence de résonance du conduit vocal.



[ $\tilde{\varepsilon}$ ]    [s]    [ $\tilde{a}$ ]    [d]    [i]  
↓  
[ $\tilde{\varepsilon} \tilde{s} \tilde{a} \tilde{d} \tilde{i}$ ]

**INCENDIE**

Figure – Analyse de formant du mot "INCENDIE"  
avec l'aide de Mme.Voisin en formation d'orthophonie à la Sorbonne

# Sommaire

## 1 Introduction

- Présentation de la problématique
- Reconnaissance vocale

## 2 Généralités sur les réseaux de neurones

- Du perceptron au réseau de neurones
- Rétropropagation
- Amélioration de la rétropropagation

## 3 Réalisations concrètes

- Reproduction du XOR
- Reconnaissance de chiffres écrits
- Convolution d'image
- Reconnaissance vocale de mots-clés

## 4 Annexe

- Mes classes
- Mes codes
- Compléments

# Le neurone

## Présentation du modèle du perceptron

En 1943, McCulloch et Pitts introduisent le modèle du perceptron.  
Ce modèle est basé sur le fonctionnement du neurone humain.

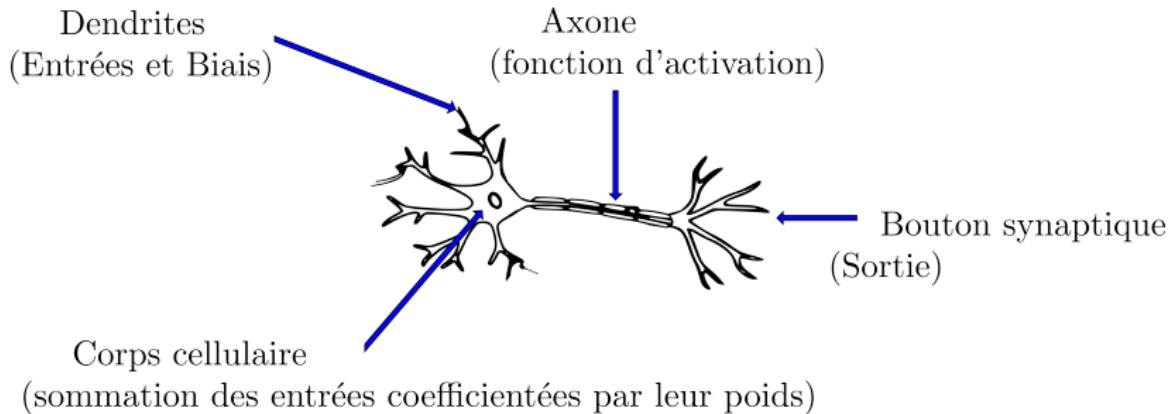


Figure – Schéma d'un neurone humain

# Le perceptron

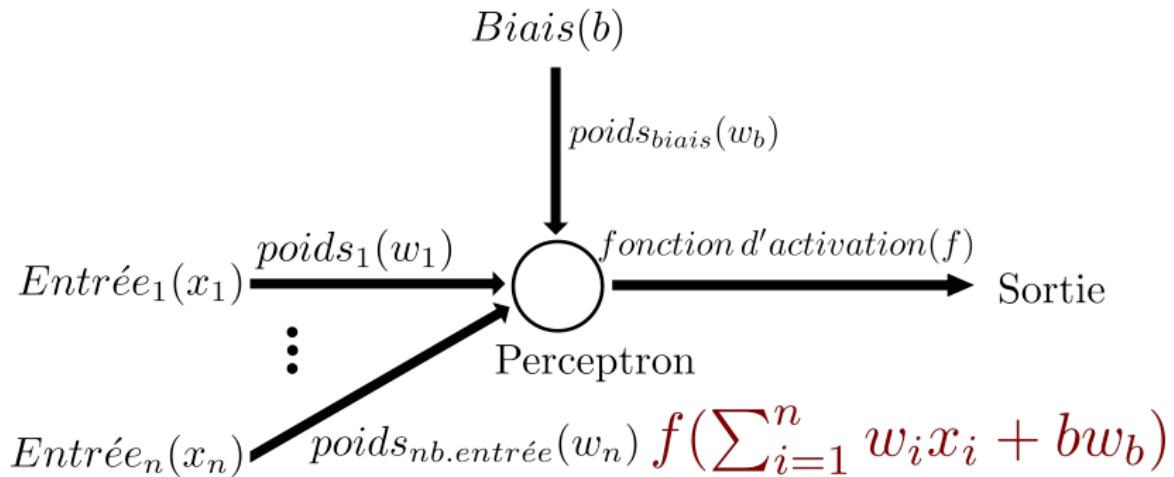
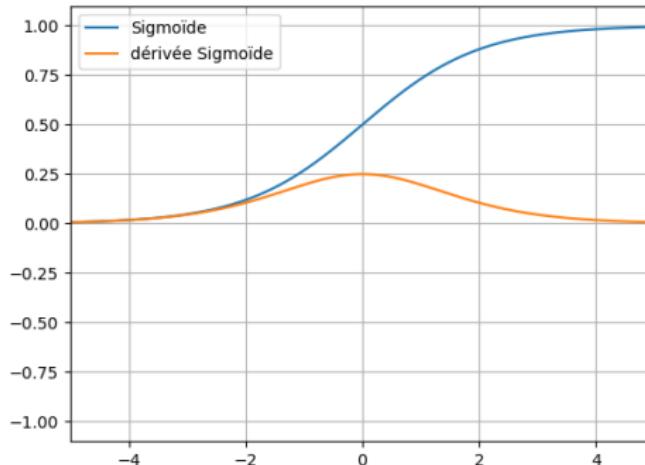


Figure – Schéma d'un perceptron

# La fonction d'activation Sigmoïde



$$\text{Sigmoïde} : \frac{1}{1 + e^{-x}}$$

$$\text{Dérivée} : f(x) \times (1 - f(x))$$

# Sa représentation informatique

$$f \left( \begin{pmatrix} x_1 & \dots & x_n & b \end{pmatrix} \times \begin{pmatrix} w_1 \\ \vdots \\ w_n \\ w_b \end{pmatrix} \right)$$

```
1 def calcul(activation, X, W):
2     # Ajout du biais
3     X = np.concatenate((
4         X,
5         np.ones((len(X), 1))
6     ),
7     axis=1
8 )
9     # Calcul de la sortie
10    z = activation(X@W)
11    return z
```

# Le réseau de neurones

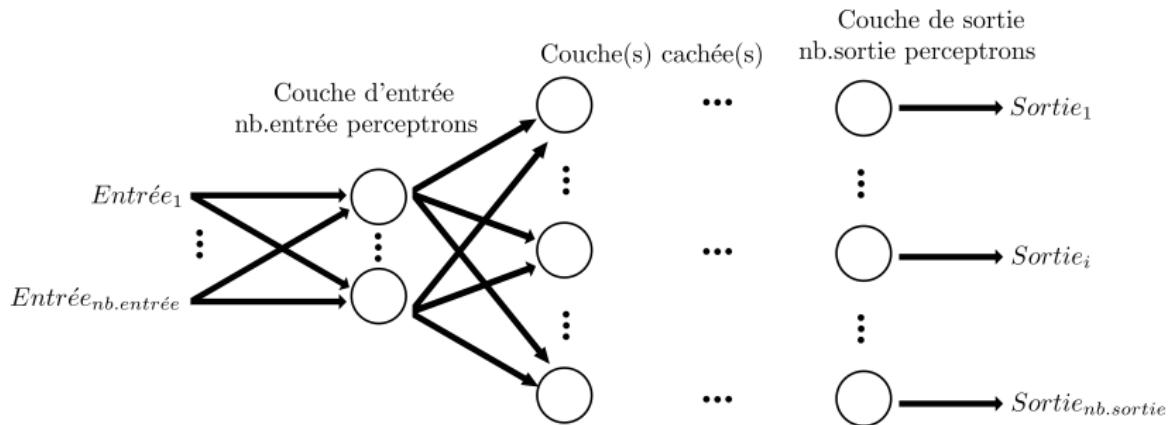


Figure – Schéma d'un réseau de neurones

# La descente de gradient

## Descente de gradient

Algorithme permettant de converger vers un minimum local d'une fonction.

Elle est utilisée pour trouver le minimum d'une fonction évaluant l'erreur entre la valeur de sortie du réseau de neurones et celle attendue :

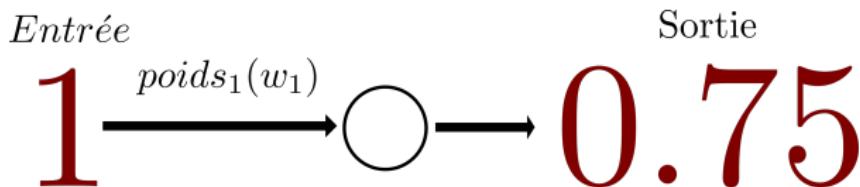
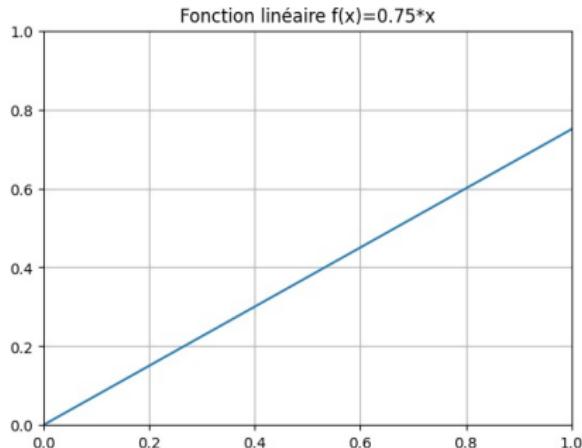
$$f : s \mapsto (s - s_{\text{attendue}})^2$$

- $s$  la sortie donnée par le réseau de neurones
- $s_{\text{attendue}}$  la sortie cible.

## Objectif

Trouver les paramètres pour annuler la fonction d'erreur revient à résoudre le problème qu'elle évalue.

## Un exemple d'un perceptron seul



## Figure – Schéma d'un perceptron à une entrée

# La descente de gradient

## Algorithme du gradient

$f$  une fonction différentiable de  $\mathbb{R} \rightarrow \mathbb{R}$ .

Soit  $x_0$  une valeur initiale aléatoire,  $t$  le taux d'apprentissage.

Supposons  $x_0, \dots, x_k$  construits.

- Si  $\|\nabla f(x_k)\| \leq \varepsilon$ , on s'arrête.
- Sinon on pose  $x_{k+1} = x_k - t \nabla f(x_k)$

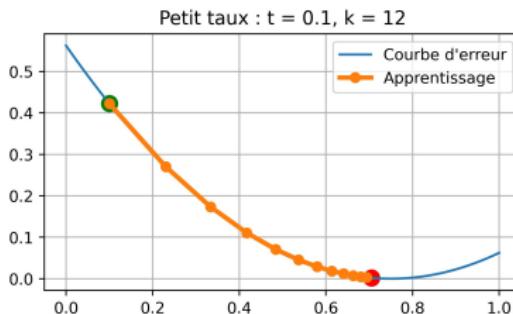
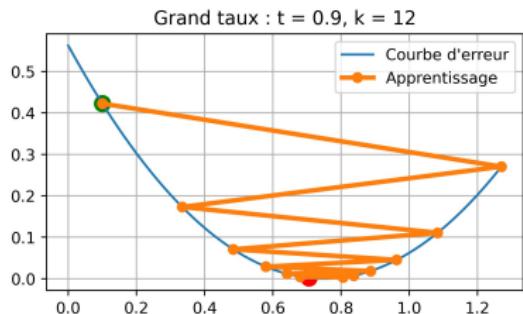


Figure – Descente de Gradient pour  $f(x) = (x - 0.75)^2$ ;  $x_0 = 0.1$  et  $\varepsilon = 0.1$

# La quantité de mouvement (momentum)

## Descente de gradient avec momentum

$x_0$  aléatoire, momentum  $\omega_0 = 0$ .  $x_0, \dots, x_k$  et  $\omega_0, \dots, \omega_k$  construits.

- On pose  $\omega_{k+1} = \gamma\omega_k + t\nabla f(x_k)$
- On pose  $x_{k+1} = x_k - \omega_{k+1}$

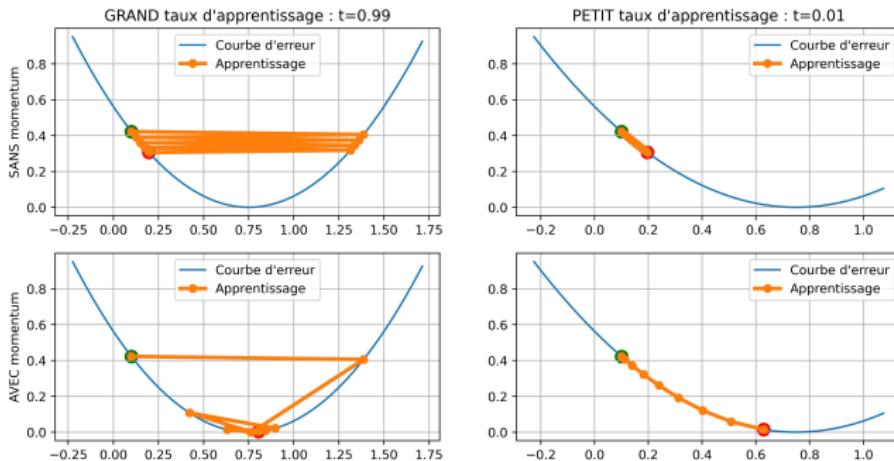


Figure – Descente de gradient SANS/AVEC momentum où  $\gamma = 0.5$

# Les avantages du momentum

## Avantages

- La descente de gradient avec momentum converge plus rapidement
- La descente de gradient avec momentum s'échappe de certains minima locaux

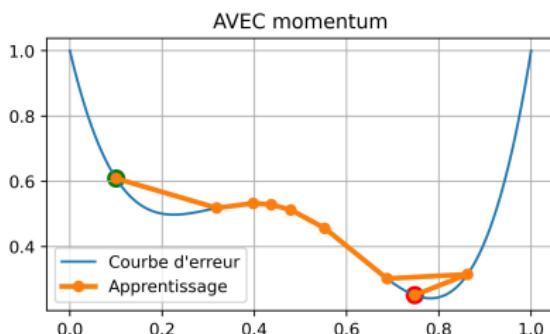
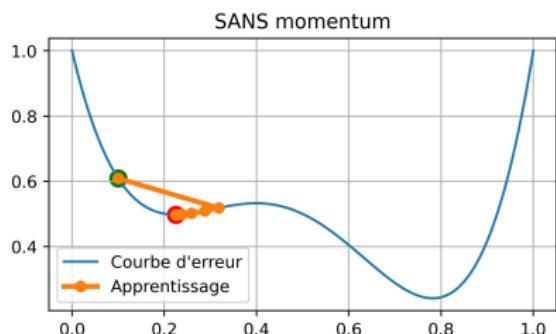


Figure – Descente de gradient SANS/AVEC momentum où  $\gamma = 0.5$

# L'apprentissage stochastique ou par paquet (batch)

## Problème d'incertitude des données

Les données d'apprentissage ne sont pas toujours exactes.

J'ajoute donc aux données théoriques une valeur d'incertitude, pour se rapprocher au plus de la réalité.

## Solution

Il est alors préférable d'appliquer la rétropropagation à notre réseau sur des paquets de données plutôt que donnée par donnée.

$$\left\langle f \left( \begin{pmatrix} x_1^1 & \dots & x_n^1 & b \\ \vdots & \vdots & \vdots & \vdots \\ x_1^D & \dots & x_n^D & b \end{pmatrix} \times \begin{pmatrix} w_1 \\ \vdots \\ w_n \\ w_b \end{pmatrix} \right) \right\rangle$$

# Une illustration de l'apprentissage par batch

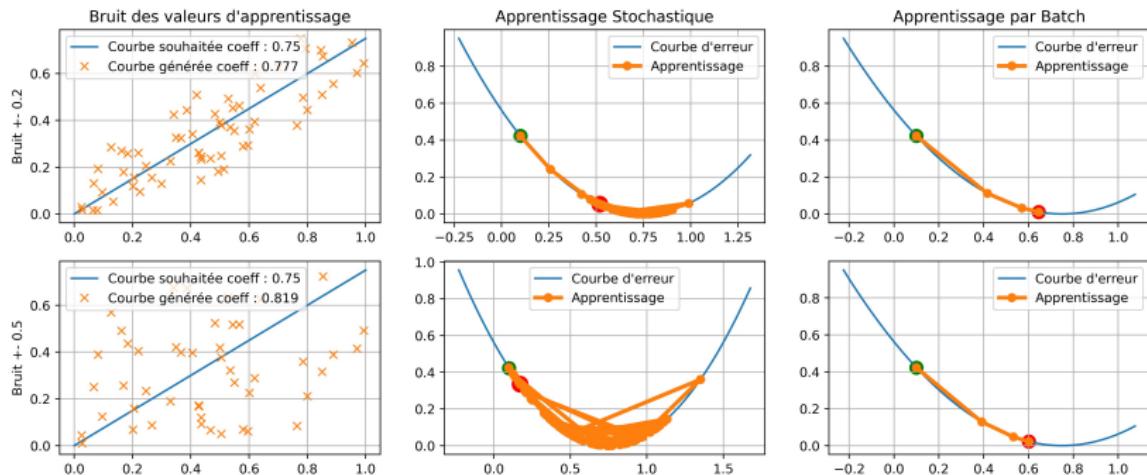


Figure – Comparaison apprentissage stochastique et par paquet

# Sommaire

## 1 Introduction

- Présentation de la problématique
- Reconnaissance vocale

## 2 Généralités sur les réseaux de neurones

- Du perceptron au réseau de neurones
- Rétropropagation
- Amélioration de la rétropropagation

## 3 Réalisations concrètes

- Reproduction du XOR
- Reconnaissance de chiffres écrits
- Convolution d'image
- Reconnaissance vocale de mots-clés

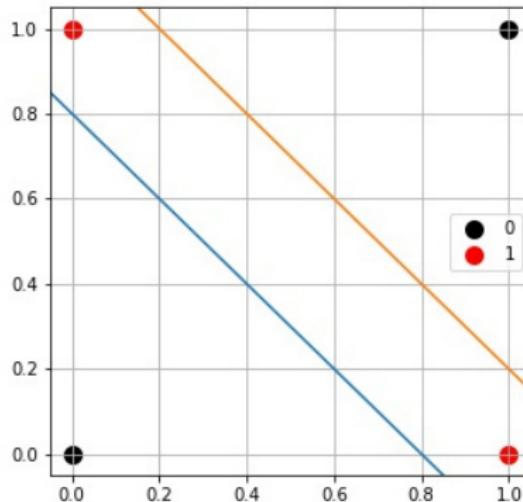
## 4 Annexe

- Mes classes
- Mes codes
- Compléments

# L'opérateur XOR

Le XOR nécessite un réseau

Le XOR, « *ou exclusif* », est un opérateur binaire.



A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

Figure – Schéma de l'opérateur XOR

# Le réseau de neurones reproduisant l'opérateur XOR

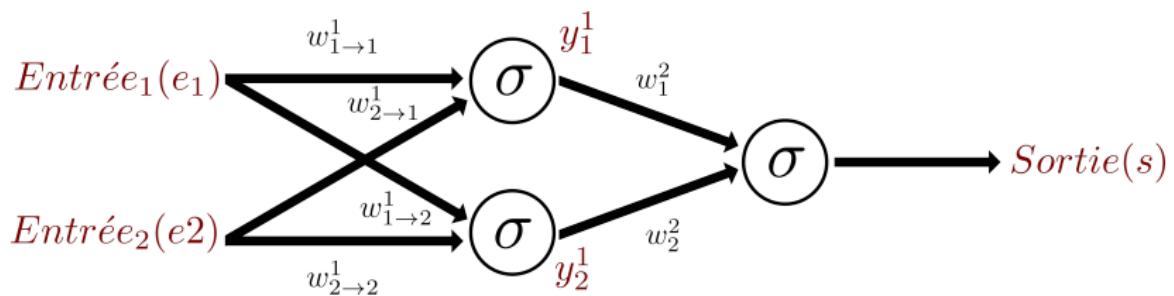
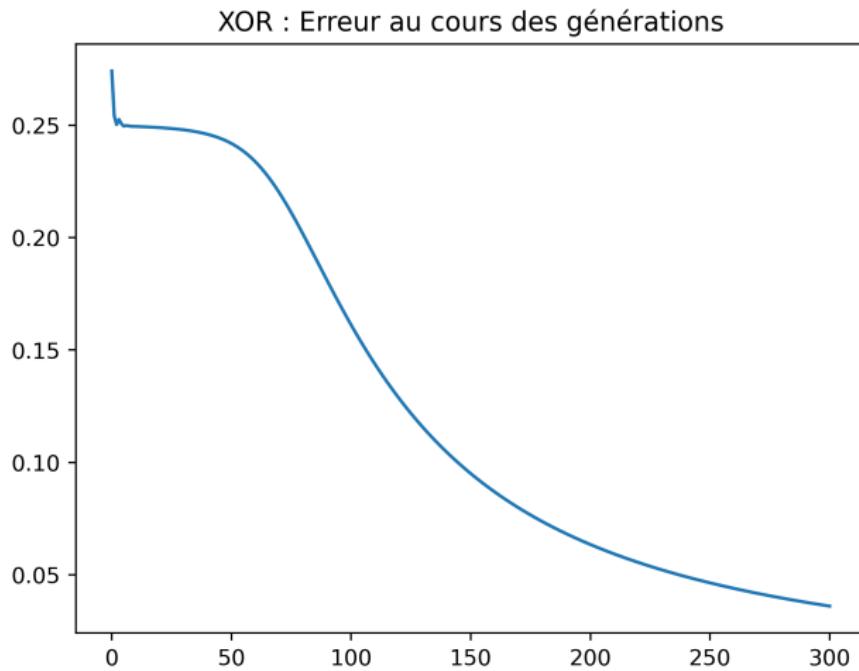


Figure – Schéma du réseau de neurones reproduisant le XOR

# L'apprentissage de la reproduction du XOR



# Mes résultat

## Données

- 4 données
- 300 générations

Entrée :  $\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \rightarrow \sigma_{couche1} \left( \cdot \times \begin{pmatrix} 0.85 & 5.42 \\ 0.85 & 5.40 \end{pmatrix} \right)$

$\rightarrow \sigma_{couche2} \left( \cdot \times \begin{pmatrix} -18.39 \\ 14.42 \end{pmatrix} \right)$

$\rightarrow Sortie : \begin{pmatrix} 0.12 \\ 0.81 \\ 0.81 \\ 0.24 \end{pmatrix}$  Sortieattendue :  $\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$

# La base de données

## Description

Images de taille  $28 \times 28$  pixels en noir et blanc :

- 60 000 images pour l'entraînement.
- 10 000 autres pour la vérification.

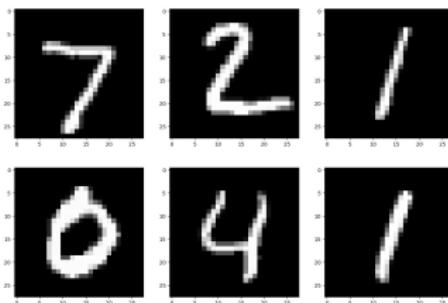


Figure – Exemple d'images

# Un réseau de neurones pour la classification

## Softmax

La fonction d'activation softmax permet d'obtenir en sortie une distribution de probabilité :

- $p_i = \frac{\exp(a_i)}{\sum_{k=1}^n \exp(a_k)}$  la probabilité de la sortie  $a_i$
- $\frac{\partial p_i}{\partial a_j} = p_i \times (\delta_{ij} - p_j)$

# L'utilisation de la fonction Softmax

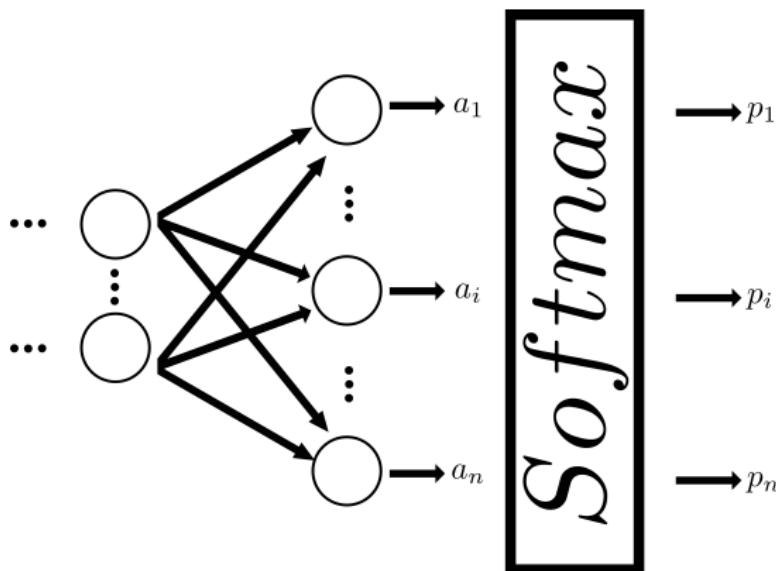


Figure – Schéma d'utilisation du Softmax

# L'évolution de l'apprentissage

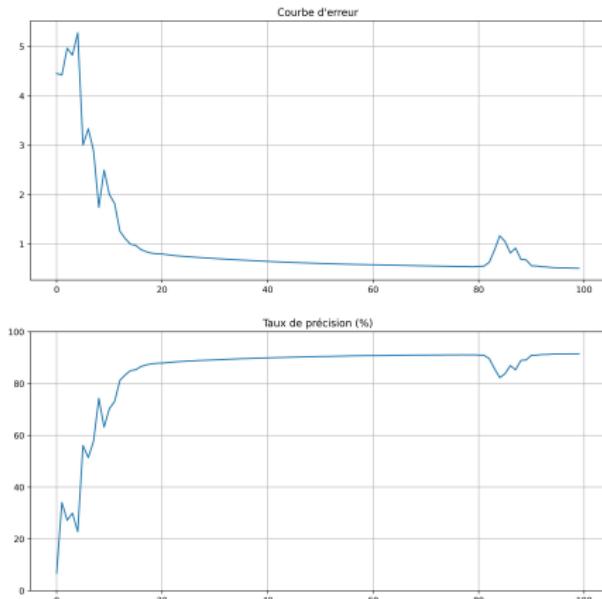


Figure – Courbes d'apprentissage

Taux de bonnes réponses après 100 générations d'entraînement :  
- 91.5% sur les données d'entraînement  
- 91.3% sur les données de validation.

# Mes résultats



Figure – Exemple sur un échantillon de 40 images de validation

# Une base de données plus complexe

Deux façons différentes de faire du traitement d'image

- L'image  $28 \times 28$  pixels est aplatie en un vecteur de taille 784.
- L'image 2D est directement prise en entrée du réseau de neurones.

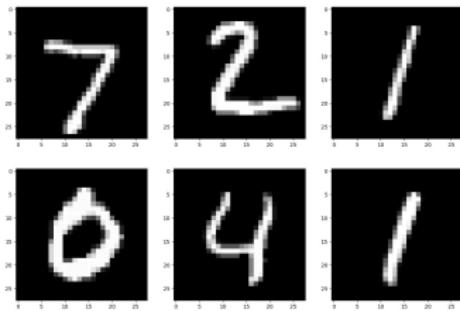


Figure – Chiffres écrits à la main

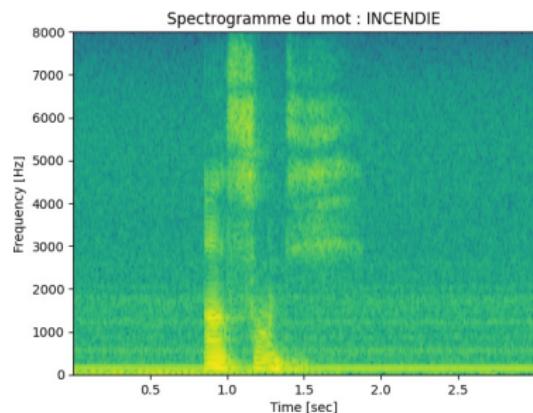
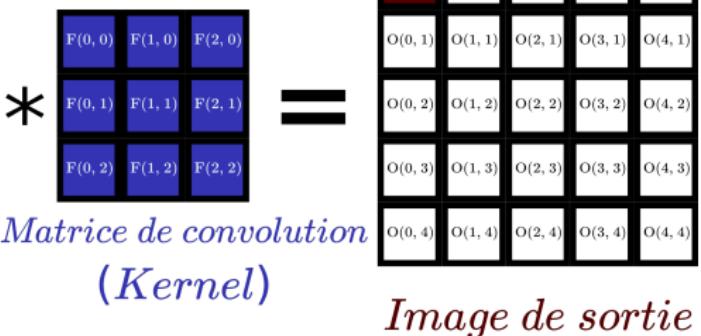


Figure – Spectrogramme

# La convolution d'image

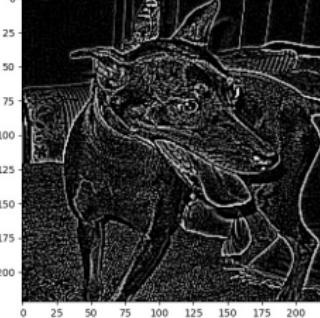
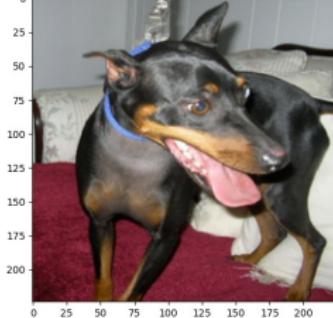
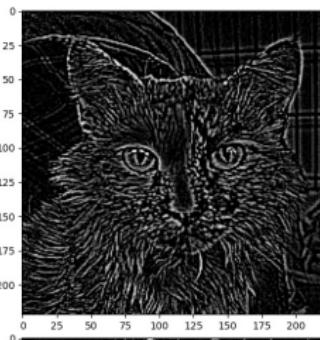
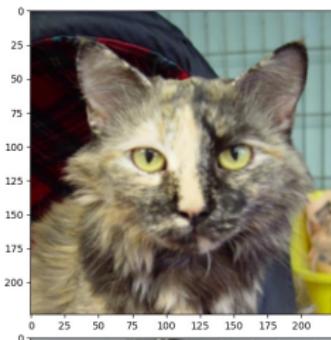
I(0, 0)	I(1, 0)	I(2, 0)	I(3, 0)	I(4, 0)	I(5, 0)	I(6, 0)
I(0, 1)	I(1, 1)	I(2, 1)	I(3, 1)	I(4, 1)	I(5, 1)	I(6, 1)
I(0, 2)	I(1, 2)	I(2, 2)	I(3, 2)	I(4, 2)	I(5, 2)	I(6, 2)
I(0, 3)	I(1, 3)	I(2, 3)	I(3, 3)	I(4, 3)	I(5, 3)	I(6, 3)
I(0, 4)	I(1, 4)	I(2, 4)	I(3, 4)	I(4, 4)	I(5, 4)	I(6, 4)
I(0, 5)	I(1, 5)	I(2, 5)	I(3, 5)	I(4, 5)	I(5, 5)	I(6, 5)
I(0, 6)	I(1, 6)	I(2, 6)	I(3, 6)	I(4, 6)	I(5, 6)	I(6, 6)

*Image d'entrée*



**Figure – Schéma de la convolution d'image**  
 $O(0, 0) = \sum_{i=0}^2 \sum_{j=0}^2 I(i, j) \times F(i, j)$

# Un exemple de convolution d'image



## Détection de contour

$$F = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Figure – Exemple de convolution d'image

# Le schéma du réseau de neurones convolutif

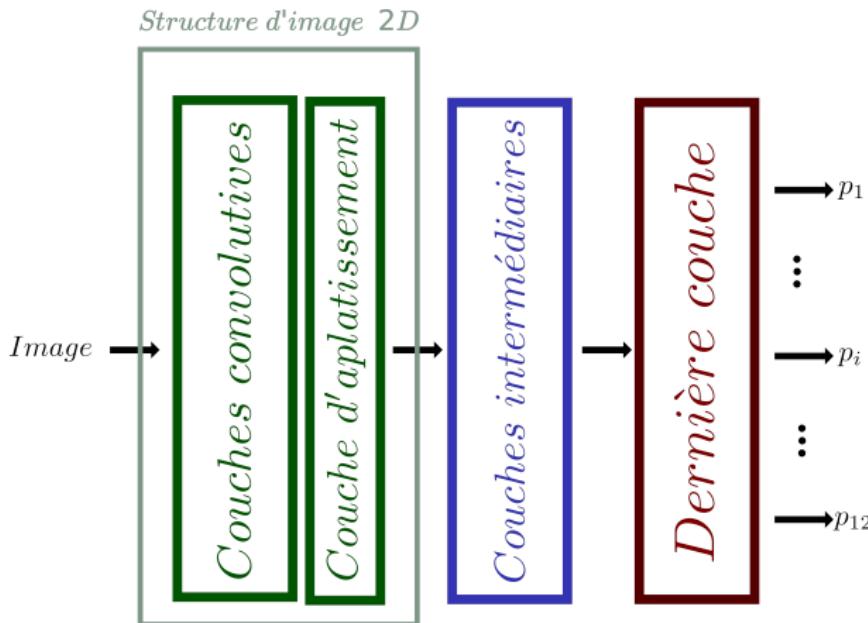


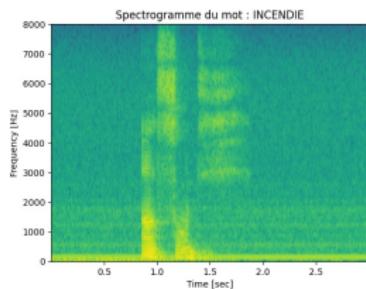
Figure – Schéma de réseau de neurones convolutif

# La reconnaissance de mots-clés

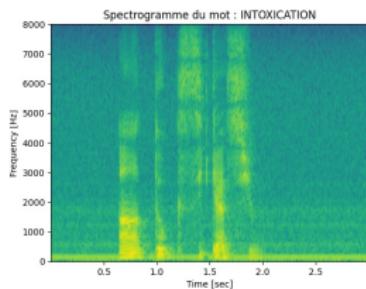
n°15 malaise, hémorragie, brûlure, intoxication

n°17 violences, agression, vol, cambriolage

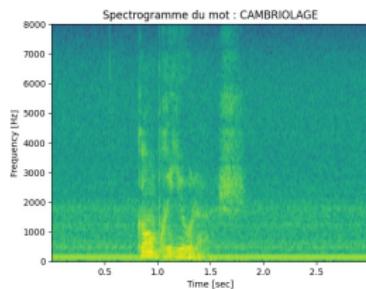
n°18 incendie, gaz, effondrement, électrocution



(a) INCENDIE



(b) INTOXICATION



(c) CAMBRIOLAGE

# La création de la base de données

Données d'apprentissage :  $6 \times 12$

Agathe  $1 \times 12$  données

Florent  $1 \times 12$  données

Tien-Thinh  $4 \times 12$  données

Données d'évaluation :  $3 \times 12$

Agathe  $1 \times 12$  données

Florent  $1 \times 12$  données

Tien-Thinh  $1 \times 12$  données

## Problématiques

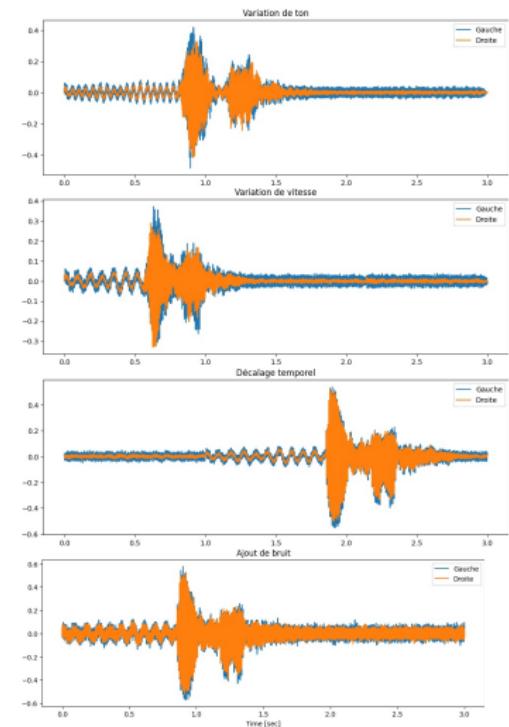
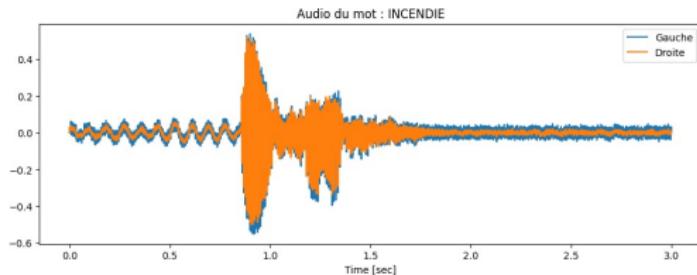
- Les spectrogrammes sont trop grands, de tailles  $374*129$
- Trop peu de données pour la rétropropagation

# L'augmentation de données

## L'augmentation de données

Création de nouvelles données à partir des données existantes :

- Variation de ton
- Variation de vitesse
- Décalage temporel
- Ajout de bruit



# Le transfert d'apprentissage à partir d'un réseau modèle

Le réseau de neurones modèle, juste à 84%

Il est entraîné sur 8 000 données pour reconnaître les 8 mots :  
["no", "stop", "up", "right", "yes", "left", "go", "down"]

## L'adaptation

- La durée des audios est différente
- Le nombre de sorties est différent

# Un schéma du transfert d'apprentissage

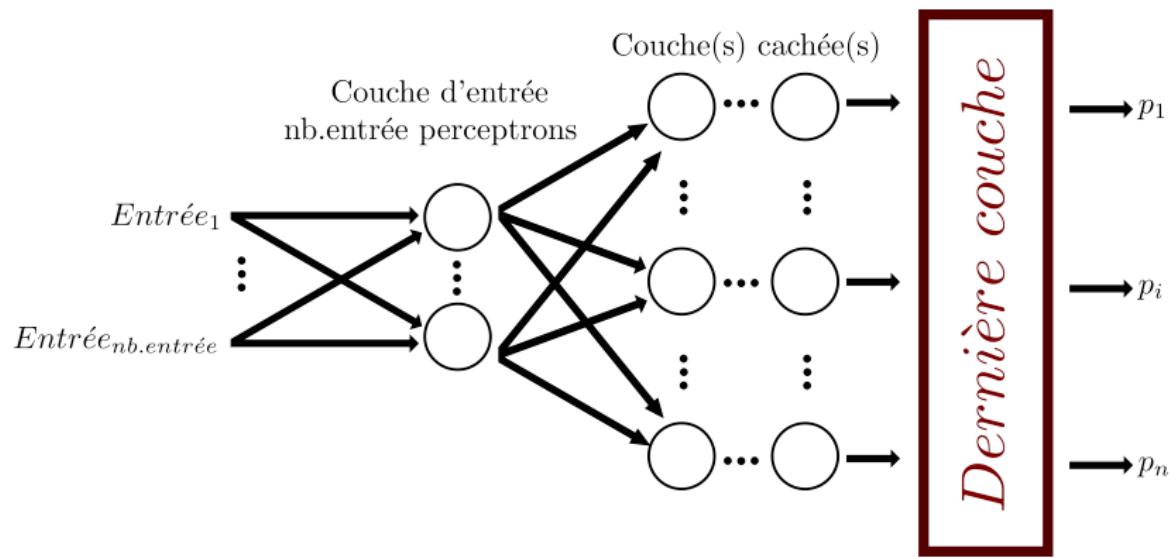
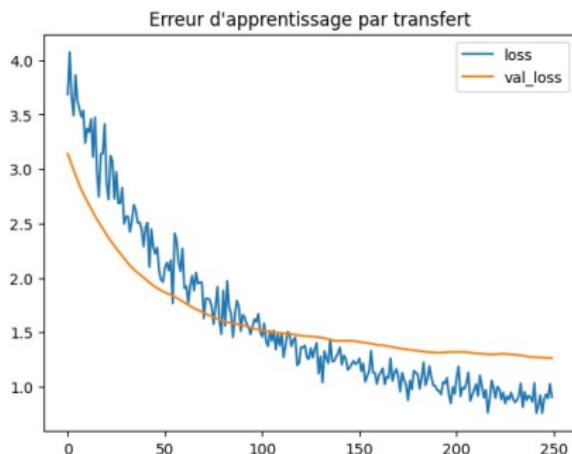
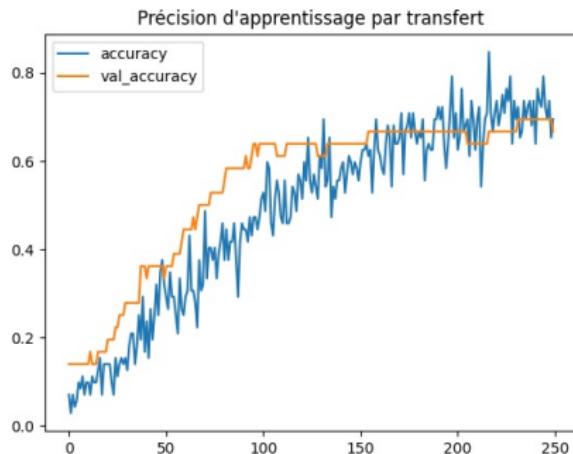


Figure – Schéma de fonctionnement du transfert d'apprentissage

# Mes résultats



(a) Erreur au cours de l'apprentissage



(b) Taux de bonnes réponses

## Précision finale

Agathe 50% accuracy

Florent 75% accuracy

Tien-Thinh 75% accuracy

# Conclusion : Schéma de la réalisation du réseau de neurones

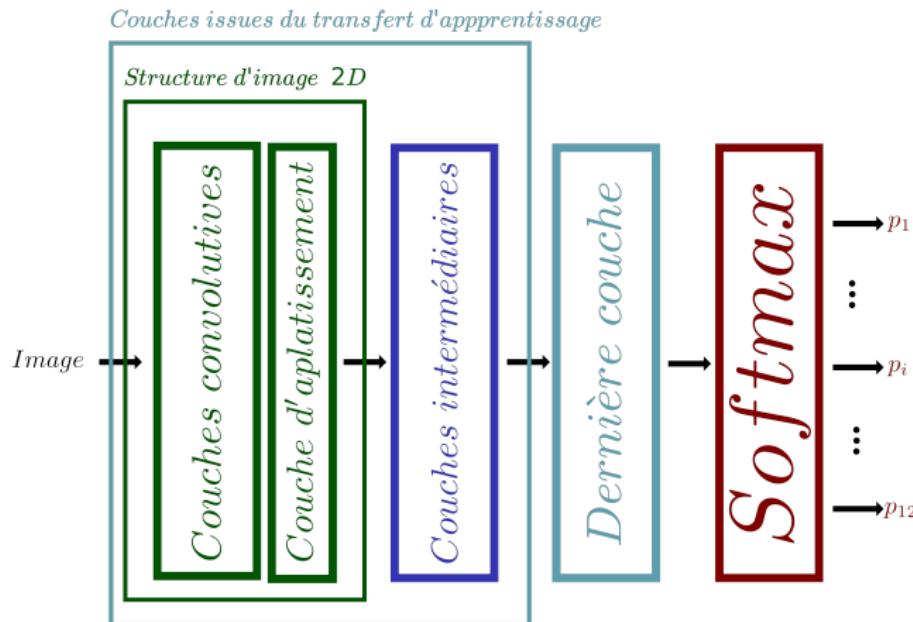


Figure – Structure de mon réseau de neurones

# Conclusion : Les moteurs de reconnaissance vocale

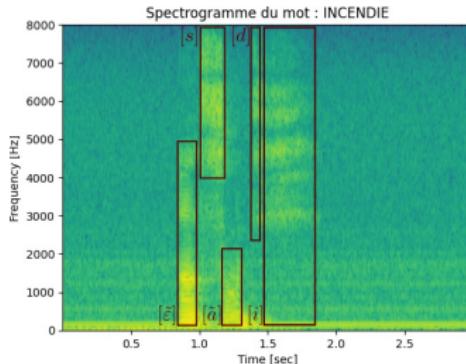


Figure – Analyse du mot "INCENDIE"

## Les performances moyennes des moteurs de reconnaissance vocale

- Textes lus (dictée vocale, système monolocuteur) : 95 %
- Journaux radio et TV : 90 %
- Conversations téléphoniques informelles : 60 %

# Sommaire

## 1 Introduction

- Présentation de la problématique
- Reconnaissance vocale

## 2 Généralités sur les réseaux de neurones

- Du perceptron au réseau de neurones
- Rétropropagation
- Amélioration de la rétropropagation

## 3 Réalisations concrètes

- Reproduction du XOR
- Reconnaissance de chiffres écrits
- Convolution d'image
- Reconnaissance vocale de mots-clés

## 4 Annexe

- Mes classes
- Mes codes
- Compléments

# Annexe

```
1 # Les librairies utilisées
2 import numpy as np
3 import scipy
4 import matplotlib.pyplot as plt
5 import librosa
```

# NeuroneLineaire.py |

```
1 class NeuroneLineaire:  
2     def __init__(self, lr=0.8, cible=0.5):  
3         self.w = np.random.random()  
4         self.lr = lr # le learning rate  
5         self.liste_w = [] # l'historique des poids  
6         self.liste_e = [] # l'historique des erreurs  
7         self.cible = cible # le poids ciblé par l'entraînement  
8  
9  
10    def calcul(self, x):  
11        return self.w * x # prédiction du résultat  
12  
13  
14    @staticmethod  
15    def erreur(y_, y): # sortie calculé, sortie souhaitée  
16        return (y - y_) ** 2  
17  
18  
19    @staticmethod
```

# NeuroneLineaire.py ||

```
20     def d_erreur(x, y_, y): # sortie calculée, sortie souhaitée
21         return (2*(y_-y) * 1 * x).mean()
22
23
24     def validation(self):
25         e = self.erreur(self.w, self.cible)
26         self.liste_w.append(self.w)
27         self.liste_e.append(e)
28
29
30     def calcul_dw(self, x, y):
31         if type(x) == int:
32             x, y = np.array([x]), np.array([y])
33         y_ = self.calcul(x)
34         dw = self.d_erreur(x, y_, y)
35         return dw
36
37
38     def retropropagation(self, x, y):
39         dw = self.calcul_dw(x, y)
```

# NeuroneLineaire.py III

```
40         self.w -= self.lr * dw # Mise à jour du poids
41
42
43     def plot(self, ax, title, mini, maxi):
44         cible = self.cible
45         x = np.linspace(min(mini, min(self.liste_w)),
46                         max(maxi, max(self.liste_w)),
47                         1_000)
48         y = self.erreur(x, self.cible)
49         ax.plot(x, y, label="Courbe d'erreur")
50         ax.plot(
51             self.liste_w, self.liste_e, 'o-', lw=3,
52             label="Apprentissage"
53         )
54         ax.set_title(title)
55         ax.grid()
56         ax.legend()
```

# Activation.py

```
1 # Linéaire
2 def lineaire(x):
3     return x
4 def d_lineaire(x):
5     return np.ones(x.shape)
6
7 # Sigmoid
8 def sigmoid(x):
9     return 1 / (1 + np.exp(-np.clip(x, -10, 10)))
10 def d_sigmoid(x):
11     f_x = sigmoid(x)
12     return f_x * (1-f_x)
13
14 # Tanh
15 def tanh(x):
16     return np.tanh(x)
17 def d_tanh(x):
18     return 1-tanh(x)**2
19
20 # ReLU
21 def relu(x):
22     return np.clip(x, 0, np.inf)
23 def d_relu(x):
24     return 1*(x>0)
25
26 # Softmax
27 def softmax(x):
28     exp = np.exp(x-x.max(axis=-1, keepdims=True))
29     return exp / exp.sum(axis=-1, keepdims=True)
30 def d_softmax(x):
31     return np.ones(x.shape)
```

# Loss.py

```
1 # Mean Square Error
2 def mse(predicted_output, target_output):
3     return ((predicted_output - target_output) ** 2).mean()
4
5 def d_mse(predicted_output, target_output):
6     return predicted_output - target_output
7
8
9 # Cross-Entropy Loss
10 def cross_entropy(predicted_output, target_output):
11     val_log = np.log(np.clip(np.abs(predicted_output), 1e-3, 1))
12     return -(target_output * val_log).sum(axis=-1).mean()
13
14 def d_cross_entropy(predicted_output, target_output):
15     return predicted_output - target_output
```

# Layer.py |

```
1 class Layer:
2     def __init__(self, input_n:int, output_n:int, lr:float,
3                  activation, d_activation, bias:bool=True, mini:float=0,
4                  maxi:float=1):
5         """
6             Crée un layer de neurones
7         """
8         # input_n le nombre d'entrée du neurones
9         # output_n le nombre de neurone de sortie
10        self.weight = np.random.rand(
11            input_n+1,
12            output_n
13        )*(maxi-mini)+mini
14        self.bias = bias
15        self.input_n = input_n
16        self.output_n = output_n
17        self.lr = lr # learning rate (taux d'apprentissage)

18        self.predicted_output_ = 0 # sortie avant activation
```

# Layer.py ||

```
18         self.predicted_output = 0 # sortie
19         self.input_data = 0
20
21     # Fonction d'activation
22     self.activation = activation
23     self.d_activation = d_activation
24
25
26 def calculate(self, input_data:np.ndarray):
27     """
28     Calcule la sortie
29     """
30
31     # Ajout du biais
32     if self.bias:
33         self.input_data = np.concatenate(
34             (input_data, np.ones((len(input_data), 1))), 
35             axis=1
36         )
37     else:
38         self.input_data = np.concatenate(
```

# Layer.py |||

```
38             (input_data, np.zeros((len(input_data), 1))),  
39             axis=1  
40         )  
41     y1 = self.input_data@self.weight  
42     z1 = self.activation(y1)  
43     self.predicted_output_ = y1  
44     self.predicted_output = z1  
45     return y1, z1  
46  
47  
48     def learn(self, e_2:np.ndarray):  
49         """  
50             Permet de mettre à jour les poids "weight"  
51         """  
52         e1 = e_2 / (self.input_n+1)  
53         e1 = e1 * self.d_activation(self.predicted_output)  
54         # e_0 est gardé pour entraîner la couche précédente  
55         e_0 = np.dot(e1, self.weight.T)[:, :-1]  
56         dw1 = np.dot(e1.T, self.input_data)  
57         self.weight -= dw1.T * self.lr
```

# Layer.py |V

```
58         return e_0
59
60
61 class LayerOptimizer(Layer):
62     """
63     On hérite de la class Layer,
64     car toutes les fonctions sont les mêmes
65     Sauf l'apprentissage
66     qui invoque un taux d'apprentissage variable
67     on utilise la variable gamma
68     """
69     def __init__(self, input_n:int, output_n:int, lr:float,
70                  activation, d_activation, bias:bool=True, mini:float=0,
71                  maxi:float=1, gamma:float=0.5):
72         # classe héritée
73         super().__init__(
74             input_n, output_n,
75             lr, activation, d_activation, bias, mini, maxi
76         )
77         self.gamma = gamma
```

# Layer.py V

```
76         self.dw_moment = np.zeros((input_n+1, output_n))  
77  
78     def learn(self, e_2:np.ndarray):  
79         """  
80             Permet de mettre à jour les poids weigth  
81             en prenant en compte le momentum  
82         """  
83         e1 = e_2 / (self.input_n+1)  
84         e1 = e1 * self.d_activation(self.predicted_output)  
85         # e_0 est pour l'entraînement de la couche précédente  
86         e_0 = np.dot(e1, self.weight.T)[:, :-1]  
87         dw1 = np.dot(e1.T, self.input_data)  
88         # La différence est ci-dessous  
89         self.dw_moment = self.gamma * self.dw_moment  
90         self.dw_moment += dw1.T * self.lr  
91         self.weight -= self.dw_moment  
92         return e_0
```

# Convolutional.py |

```
1 class Convolutional(Layer):
2     """
3         On hérite de la class Layer,
4         Au lieu de prendre la representation
5             de kernels appliqués aux images.
6
7         Je représente les kernels sous la forme d'un Layer avec :
8             - input_n = kernel_size**2
9             - output_n = nb_kernel
10
11    def __init__(self, img_size:np.ndarray, kernel_size:int,
12                 nb_kernel:int, lr:float, activation, d_activation, bias:
13                 bool=True, mini:float=0, maxi:float=1):
14         # classe héritée
15         input_n = kernel_size**2
16         output_n = nb_kernel
17         super().__init__(
```

# Convolutional.py II

```
18
19     )
20     self.kernel_size = kernel_size
21     self.nb_kernel = nb_kernel # nombre total de filtre
22     self.img_size = img_size
23     self.output_size = max(0, img_size-kernel_size+1)
24
25 def transform(self, imgs:np.ndarray):
26     """
27         Transforme une liste d'image d'entrée
28         en input pour le perceptron
29     """
30     n = self.output_size
31     k = self.kernel_size
32     t_imgs = np.array([
33         img[lig:lig+k, col:col+k].flatten()
34         for img in imgs
35         for lig in range(n)
36         for col in range(n)
37     ]) 
```

# Convolutional.py III

```
38         return t_imgs
39
40
41     def transform_k(self, imgs:np.ndarray):
42         """
43             Transforme la liste d'image d'intermédiaire de calcul
44             en entrée pour l'apprentissage
45         """
46
47         n = self.output_size
48         nk = self.nb_kernel
49
50         A = imgs
51         A = A.reshape((-1, nk, n**2))
52         A = A.transpose([0, 2, 1])
53         A = A.reshape((-1, nk))
54         return A
55
56     def detransform_k(self, t_imgs:np.ndarray, nb_donnee:np.
57         ndarray, nb_images:int):
```

# Convolutional.py IV

```
57     """
58     Transforme la sortie
59     en une liste d'image
60     """
61     n = self.output_size
62     nk = self.nb_kernel
63
64     A = t_imgs
65     A = A.reshape((nb_donnee*nb_images, n**2, nk))
66     A = A.transpose([0, 2, 1])
67     imgs = A.reshape((nb_donnee, nb_images*nk, n, n))
68     return imgs
69
70
71
72     def calculate(self, input_data:np.ndarray):
73         # input est de taille (nb_donne, nb_images, hauteur,
74         largeur)
75         # On suppose (hauteur = largeur) : image carrée
76         """
```

# Convolutional.py V

```
76     Calcule la sortie
77     """
78
79     # Ajout du biais
80     nb_donnee = len(input_data)
81     nb_images = len(input_data[0])
82     input_data_k = np.concatenate([
83         self.transform(input_data[i])
84         for i in range(nb_donnee)
85     ])
86
87     if self.bias:
88         self.input_data = np.concatenate(
89             (input_data_k, np.ones((len(input_data_k), 1)))
90             ,
91             axis=1
92         )
93     else:
94         self.input_data = np.concatenate(
95             (input_data_k, np.zeros((len(input_data_k), 1)))
96         ),
```

# Convolutional.py VI

```
94                 axis=1
95             )
96         y1 = np.dot(self.input_data, self.weight)
97         z1 = self.activation(y1)
98         self.predicted_output_ = y1
99         self.predicted_output = z1
100        y1 = self.detransform_k(y1, nb_donnee, nb_images)
101        z1 = self.detransform_k(z1, nb_donnee, nb_images)
102        return (y1, z1)
103
104
105    def learn(self, e_2:np.ndarray):
106        """
107            Permet de mettre à jour les poids weight
108        """
109        shape = e_2.shape
110        e_2 = self.transform_k(e_2)
111        e1 = e_2 / (self.input_n+1)
112        e1 = e1 * self.d_activation(self.predicted_output)
113        # e_0 est pour l'entraînement de la couche précédente
```



# Convolutional.py VII

```
114         e_0 = np.dot(e1, self.weight.T)[:, :-1]
115         dw1 = np.dot(e1.T, self.input_data)
116         self.weight -= dw1.T * self.lr
117         return e_0
118
119
120 class Flatten:
121     """
122     Cette classe permet de faire le lien
123     entre les couches Convolutional
124     et les couches Layer
125     """
126     def __init__(self, img_size):
127         self.img_size = img_size
128         self.output_n = img_size**2
129
130
131     def calculate(self, imgs):
132         nb_donne = len(imgs)
133         nb_img = len(imgs[0])
```

# Convolutional.py VIII

```
134
135     flat = imgs.reshape((
136         nb_donne,
137         nb_img*self.img_size*self.img_size
138     ))
139     return flat, flat
140
141
142     def learn(self, e_2):
143         nb_donne = len(e_2)
144         e_0 = e_2.reshape((
145             nb_donne,
146             -1,
147             self.img_size, self.img_size
148         ))
149         return e_0
```

# Model.py |

```
1 class Model:
2     """
3         Le Model est une liste de couche
4         il permet d'organiser :
5             - le calcul de la prediction
6             - l'entraînement de l'ensemble du model
7     """
8
9     def __init__(self, layers:list, loss_function,
10                  d_loss_function):
11         self.layers = layers
12         self.loss = []
13         self.lr = 0.1
14         self.loss_function = loss_function
15         self.d_loss_function = d_loss_function
16
17     def predict(self, input_data:np.ndarray):
18         # On calcule la sortie par récurrence
19         predicted_output = input_data # Initialisation
```

# Model.py II

```
19         for layer in self.layers: # Hérité
20             predicted_output_,predicted_output = layer.
21             calculate(
22                 predicted_output
23             )
24             return predicted_output # Sortie attendue
25
26
27     def predict_loss(self, input_data:np.ndarray, target_output
28 :np.ndarray):
29         # Sortie et Erreur
30         predicted_output = self.predict(input_data)
31         loss = self.loss_function(
32             predicted_output, target_output
33         )
34         return predicted_output, loss
35
36
37     def backpropagation(self, input_data:np.ndarray,
38 target_output:np.ndarray):
```

# Model.py III

```
36     # Entrainement par récurrence
37     predicted_output, loss = self.predict_loss(
38         input_data, target_output
39     )
40     # dérivée
41     d_loss = self.d_loss_function(
42         predicted_output, target_output
43     )
44     for i in range(len(self.layers)): # Entrainement
45         d_loss = self.layers[-i-1].learn(d_loss)
46     self.loss.append(loss)
47     return predicted_output, loss
48
49
50 class ModelClassification(Model):
51     """
52         Permet de calculer le taux de bonne réponse
53     """
54     def __init__(self, layers:list, loss_function,
d_loss_function):
```

# Model.py IV

```
55     # classe héritée
56     super().__init__(layers, loss_function, d_loss_function
57 )
58
59     def predict_accuracy(self, input_data:np.ndarray,
60         target_output:np.ndarray):
61         predicted_output, loss = self.predict_loss(
62             input_data, target_output
63         )
64         po = predicted_output
65         to = target_output
66         nb_bonne_rep = (
67             po.argmax(axis=-1) == to.argmax(axis=-1)
68         ).sum()
69         acc = nb_bonne_rep / len(target_output)
70         return predicted_output, loss, acc
71 
```

# Model.py V

```
72     def backpropagation(self, input_data:np.ndarray,  
73                         target_output:np.ndarray):  
74                 # Entrainement par récurrence  
75                 predicted_output, loss, acc = self.predict_accuracy(  
76                             input_data, target_output  
77                         )  
78                 # dérivée  
79                 d_loss = self.d_loss_function(  
80                             predicted_output, target_output  
81                         )  
82                 for i in range(len(self.layers)): # Entrainement  
83                     d_loss = self.layers[-i-1].learn(d_loss)  
84                 self.loss.append(loss)  
85             return predicted_output, loss, acc
```

# AugmentationDonnee.py

```
1 def pitch_manipulate(data, sampling_rate, pitch_factor):
2     return librosa.effects.pitch_shift(data.astype(np.float32), sampling_rate, pitch_factor)
3
4
5 def speed_manipulate(data, speed_factor):
6     n = len(data)
7     augmented_data = librosa.effects.time_stretch(data.astype(np.float32), speed_factor)
8     d = len(augmented_data)
9     q, r = n//d, n%d
10    augmented_data = np.concatenate(
11        [augmented_data]*q +
12        [augmented_data[-r:]])
13    )
14    return augmented_data
15
16
17 def noise_manipulation(data, noise_factor):
18     noise = np.random.randn(len(data))
19     augmented_data = data + noise_factor * noise
20     augmented_data = augmented_data.astype(type(data[0]))
21     return augmented_data
22
23
24 def shift_manipulation(data, shift):
25     augmented_data = np.roll(data, shift)
26     return augmented_data
```

# ReproductionLineaire.py

```
1 # Petit learning rate
2 reseau = NeuroneLineaire(lr=0.1, cible=0.75)
3 reseau.w = 0.1
4 while abs(2*(0.75-reseau.calcul(1))) > 0.1:
5     reseau.validation()
6     reseau.retropropagation(1, 0.75)
7
8 reseau.validation()
9 reseau.plot(
10     ax=plt.axes(), title=f"Petit taux : t = 0.1",
11     mini=0, maxi=1
12 )
13 plt.show()
```

# ReproductionXOR.py

```
1 # Données
2 train_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
3 train_output = np.array([[0] , [1] , [1] , [0] ])
4 # Creation du model
5 model = Model([
6     LayerOptimizer(
7         2, 2, lr=0.5, gamma=0.5, bias=False,
8         activation=sigmoid, d_activation=d_sigmoid
9     ),
10    LayerOptimizer(
11        2, 1, lr=0.5, gamma=0.5, bias=False,
12        activation=sigmoid, d_activation=d_sigmoid
13    ),
14 ],
15 loss_function=mse,
16 d_loss_function=d_mse
17 )
18 # Entrainement
19 losses = []
20 epochs = 300
21 for epoch in range(epochs):
22     y, loss = model.backpropagation(train_input, train_output)
23     losses.append(loss) # Permet l'affichage des courbes
24 plt.plot(losses)
25 plt.title("Erreur au cours de l'apprentissage")
26 plt.show()
```

# ReconnaissanceChiffres.py

```
1 # Traitement des données
2 (X_train, Y_train), (X_test, Y_test) = mnist.load_data()
3 y_train = np.zeros((len(Y_train), 10))
4 y_train[np.arange(len(Y_train)), Y_train] = 1 # to categorical
5 y_test = np.zeros((len(Y_test), 10))
6 y_test[np.arange(len(Y_test)), Y_test] = 1 # to categorical
7 x_train = X_train.reshape(-1, 28*28)/255 # 28*28 = 784
8 x_test = X_test.reshape(-1, 28*28)/255
9
10 # Creation du model
11 model = ModelClassification([
12     LayerOptimizer(784, 10, lr=0.5, gamma=0.5, activation=softmax, d_activation=d_softmax),
13 ],
14     loss_function=cross_entropy,
15     d_loss_function=d_cross_entropy
16 )
17
18 # Entrainement
19 losses = []
20 accs = []
21 epochs = 100
22 for epoch in range(epochs):
23     y, loss, acc = model.backpropagation(x_train, y_train)
24     losses.append(loss) # Permet l'affichage des courbes
25     accs.append(acc*100)
26
27 # Affichage résultat
28 predicted_output, loss, acc = model.predict_accuracy(x_test, y_test)
29 print(f"Après {epochs} générations : le taux de bonnes réponses est {acc*100}%")
```

# ReconnaissanceVetements.py

```
1 # Traitement des données
2 (X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()
3 y_train = np.zeros((len(Y_train), 10))
4 y_train[np.arange(len(Y_train)), Y_train] = 1 # to categorical
5 y_test = np.zeros((len(Y_test), 10))
6 y_test[np.arange(len(Y_test)), Y_test] = 1 # to categorical
7 x_train = X_train.reshape(-1, 1, 28, 28)/255 # 28*28 = 784
8 x_test = X_test.reshape(-1, 1, 28, 28)/255
9 dico = ["T-shirt", "Trouser", "Pull", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Boot"]
10
11 # Creation du model
12 model = ModelClassification([
13     Convolutional(img_size=28, kernel_size=3, nb_kernel=4, lr=0.5,
14     activation=relu, d_activation=d_relu),
15     Flatten(26), # 26 = 28-3+1
16     LayerOptimizer(2704, 10, lr=0.5, gamma=0.5, activation=softmax, d_activation=d_softmax),
17 ],
18 loss_function=cross_entropy,
19 d_loss_function=d_cross_entropy
20 )
21
22 # Entrainement
23 losses = []
24 accs = []
25 epochs = 100
26 for epoch in range(epochs+1):
27     y, loss, acc = model.backpropagation(x_train, y_train)
28     losses.append(loss)
29     accs.append(acc*100)
30
31 # Affichage résultat
32 predicted_output, loss, acc = model.predict_accuracy(x_test, y_test)
33 print(f"Après {epochs} générations : le taux de bonnes réponses est {acc*100}%")
```

# TransfertApprentissage.py

```
1 import tensorflow as tf
2
3 model = tf.keras.models.load_model('model.h5')
4 model.trainable = False
5 model_adapt = tf.keras.Sequential(
6     [tf.keras.layers.Input(shape=(374, 129, 1))]
7     + model.layers[0:-1]
8     + [tf.keras.layers.Dense(12, activation="softmax",
9         name="adaptation")]
10 )
11 print(model_adapt.summary())
```

# Un cas de non convergence de la descente de gradient

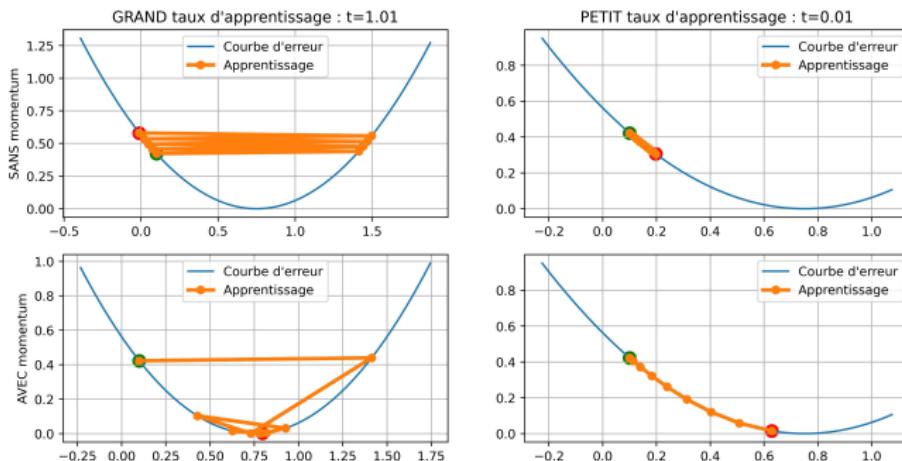


Figure – Descente de gradient SANS/AVEC momentum où  $\gamma = 0.5$

# La terminaison de la descente de gradient

## En pratique

Pour terminer les itérations de la rétropropagation sur un réseau de neurones trois conditions de terminaison existent :

- Un nombre maximal d'itération de la rétropropagation
- Un seuil minimal pour l'erreur
- Un seuil minimal pour la norme infinie du gradient

Ainsi, en pratique, si une de ces trois conditions est atteinte, l'algorithme se termine. Le variant de boucle du nombre d'itérations assure alors la terminaison de l'algorithme

# Les hypothèses pour la terminaison

## En théorie

Une fonction de  $R^n$  dans  $R$  de classe  $C^2$ , strictement convexe et coercive ( $\lim_{\|x\| \rightarrow +\infty} f(x) = +\infty$ ) admet un unique minimum. On peut donc trouver une suite  $t_k$  de taux d'apprentissage de sorte que la descente de gradient converge.

## Exemple avec les hypothèses

La relation de récurrence s'écrit alors :

$$x_{k+1} = x_k - t_k \nabla f(x_k)$$

avec  $x_k$  tendant vers l'unique minimum de  $f$

# La rétropropagation sur le XOR

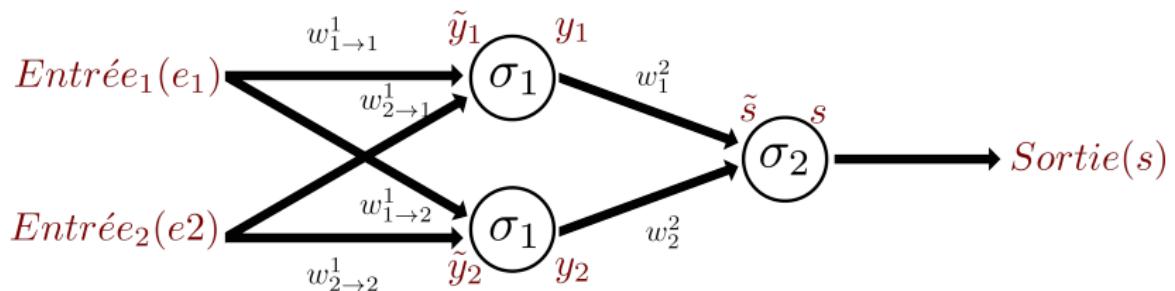


Figure – Schéma du réseau de neurones reproduisant le XOR

$$\begin{cases} \tilde{y}_1 = w_{11}^1 \times e_1 + w_{21}^1 \times e_2 \\ \tilde{y}_2 = w_{12}^1 \times e_1 + w_{22}^1 \times e_2 \\ \tilde{s} = w_1^2 \times y_1 + w_{11}^2 \times y_2 \end{cases} \quad \text{et} \quad \begin{cases} y_1 = \sigma_1(\tilde{y}_1) \\ y_2 = \sigma_1(\tilde{y}_2) \\ s = \sigma_2(\tilde{s}) \end{cases} \quad (1)$$

# Une simplification matricielle

## Convention adoptée

@ Le produit matriciel

\* Le produit d'Hadamard

f La fonction d'erreur  $f : s \rightarrow (s - s_{attendue})^2$

$$\begin{cases} \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \end{pmatrix} = \begin{pmatrix} w_{11}^1 & w_{21}^1 \\ w_{12}^1 & w_{22}^1 \end{pmatrix} @ \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} \\ \tilde{s} = \begin{pmatrix} w_1^2 & w_2^2 \end{pmatrix} @ \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \end{cases} \quad (2)$$

# La rétropropagation matricielle

$$\begin{cases}
 \begin{pmatrix} \frac{\partial f}{\partial w_1^2}(w_1^2) & \frac{\partial f}{\partial w_2^2}(w_1^2) \end{pmatrix} = \left( \frac{\partial f}{\partial s}(s) \right) * \left( \sigma'_2(\tilde{s}) \right) @ \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}^t \\
 \begin{pmatrix} \frac{\partial f}{\partial w_{11}^2}(w_{11}^2) & \frac{\partial f}{\partial w_{21}^2}(w_{21}^2) \\ \frac{\partial f}{\partial w_{12}^2}(w_{12}^2) & \frac{\partial f}{\partial w_{22}^2}(w_{22}^2) \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial y_1}(y_1) \\ \frac{\partial f}{\partial y_2}(y_2) \end{pmatrix} * \begin{pmatrix} \sigma'_1(\tilde{y}_1) \\ \sigma'_1(\tilde{y}_2) \end{pmatrix} @ \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}^t
 \end{cases} \quad (3)$$

avec

$$\begin{pmatrix} \frac{\partial f}{\partial y_1}(y_1) \\ \frac{\partial f}{\partial y_2}(y_2) \end{pmatrix}^t = \left( \frac{\partial f}{\partial s}(s) \right) * \left( \sigma'_2(\tilde{s}) \right) @ \begin{pmatrix} w_1^2 & w_2^2 \end{pmatrix} \quad (4)$$

# Des fonctions d'activation

Fonction	Formule	Dérivée
Sigmoïde	$\frac{1}{1 + e^{-x}}$	$f(x) \times (1 - f(x))$
Tangente Hyperbolique (Tanh)	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$
Unité Linéaire Rectifiée (ReLU)	$\max(0, x)$	$\begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{sinon} \end{cases}$

# La fonction d'activation : Sigmoïde

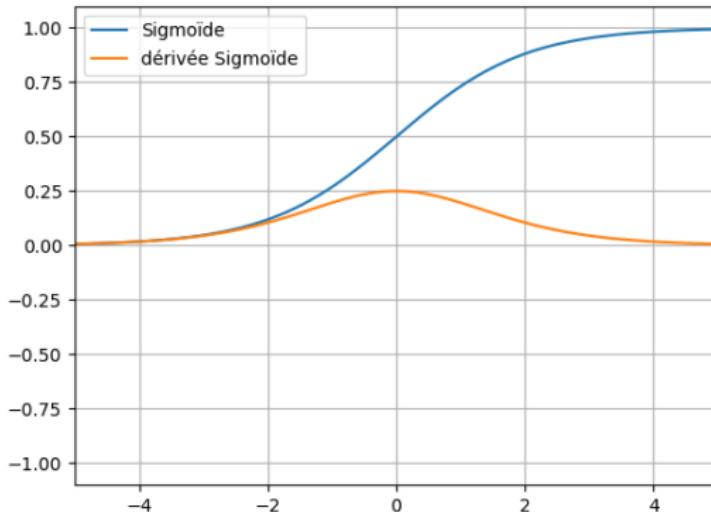


Figure – Sigmoïde

# La fonction d'activation : TANH

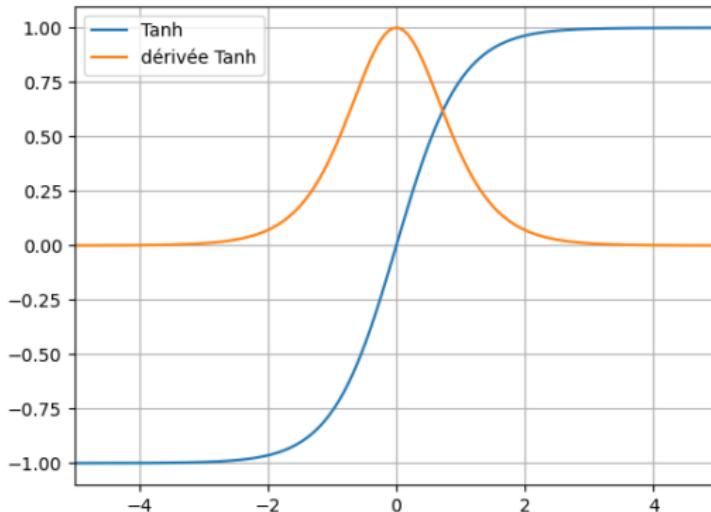


Figure – TANH

# La fonction d'activation : ReLu

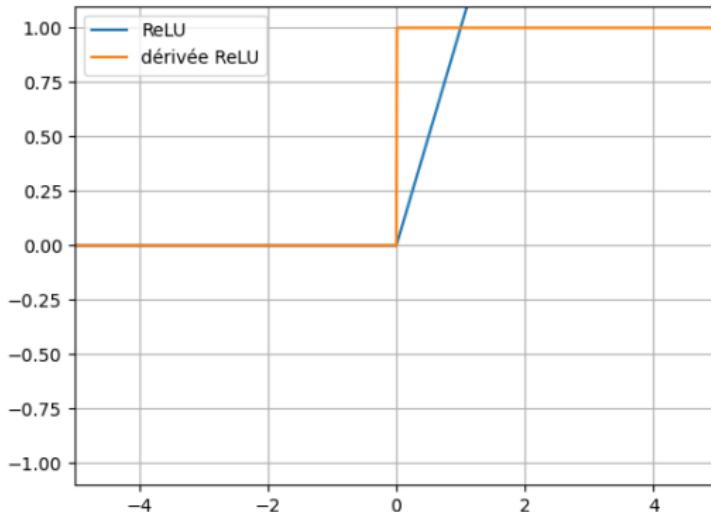


Figure – ReLu

# La rétropropagation pour la classification

## Cross-entropy

La fonction d'erreur des problèmes de classification est Cross-entropy :

- $L = - \sum_{k=1}^n y_i \log(p_i)$  avec  $y_i$  la sortie attendue
- $\frac{\partial L}{\partial a_i} = p_i - y_i$

# Une base de données plus complexe

## Description

Images de taille  $28 \times 28$  pixels en noir et blanc :

- 60 000 images pour l'entraînement.
- 10 000 autres pour la vérification.

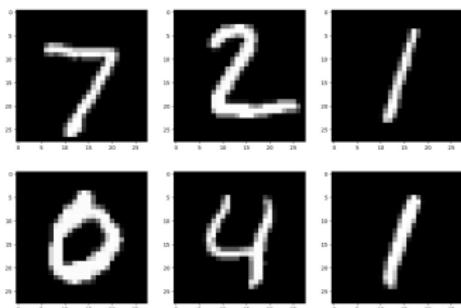


Figure – MNIST

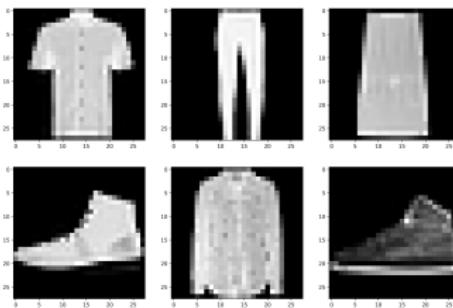


Figure – Fashion MNIST

# Mes résultats



```
1 dico = {  
2     0: 'T-shirt',  
3     1: 'Trouser',  
4     2: 'Pull',  
5     3: 'Dress',  
6     4: 'Coat',  
7     5: 'Sandal',  
8     6: 'Shirt',  
9     7: 'Sneaker',  
10    8: 'Bag',  
11    9: 'Boot',  
12 }
```

Figure – Exemple sur un échantillon de 40 images  
Fashion MNIST

# Réseau de neurones modèle

```
1 Layer (type)          Output Shape         Param #  
2 ======  
3 resizing (Resizing)    (None, 32, 32, 1)      0  
4 conv2d (Conv2D)        (None, 30, 30, 32)     320  
5 conv2d_1 (Conv2D)      (None, 28, 28, 64)     18496  
6 max_pooling2d (MaxPooling2D) (None, 14, 14, 64) 0  
7 dropout (Dropout)      (None, 14, 14, 64)     0  
8 flatten (Flatten)      (None, 12544)        0  
9 dense (Dense)          (None, 128)          1605760  
10 dropout_1 (Dropout)    (None, 128)          0  
11 dense_1 (Dense)        (None, 8)           1032  
12 ======  
13 Total params: 1,625,608  
14 Trainable params: 1,625,608  
15 Non-trainable params: 0
```

# Réseau de neurones adapté

```
1 Layer (type)          Output Shape         Param #  
2 ======  
3 resizing (Resizing)    (None, 32, 32, 1)      0  
4 conv2d (Conv2D)        (None, 30, 30, 32)     320  
5 conv2d_1 (Conv2D)      (None, 28, 28, 64)     18496  
6 max_pooling2d (MaxPooling2D) (None, 14, 14, 64) 0  
7 dropout (Dropout)      (None, 14, 14, 64)     0  
8 flatten (Flatten)      (None, 12544)        0  
9 dense (Dense)          (None, 128)          1605760  
10 dropout_1 (Dropout)    (None, 128)          0  
11 adaptation (Dense)    (None, 12)           1548  
12 ======  
13 Total params: 1,626,124  
14 Trainable params: 1,548  
15 Non-trainable params: 1,624,576
```