

# Présentation

TRAN-THUONG Tien-Thinh

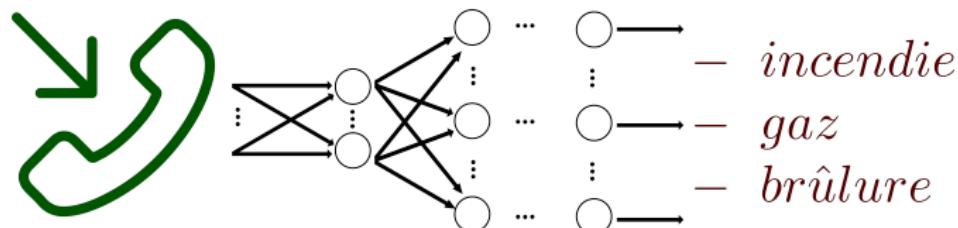
2021-2022

# Problématique

D'après le ministère de la Santé : Il y a eu plus de **31 millions** d'appels d'urgence en 2018. Seuls **69%** des appels étaient décrochés dans la minute.

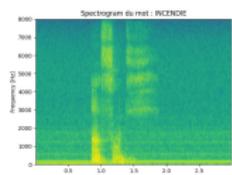
## Objectif

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

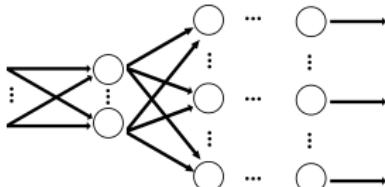


# Les étapes de la reconnaissance automatique de la parole

1 Le traitement acoustique



2 L'apprentissage automatique



3 Le décodage

[ɛ̃] [s] [ã] [d̃] [ĩ]  
↓  
[ɛ̃ s ã d̃ ĩ]  
**INCENDIE**

## Présentation du modèle du perceptron

C'est en 1943, que McCulloch et Pitts introduisent le modèle du perceptron.  
Ce modèle est basé sur le fonctionnement du neurone humain.

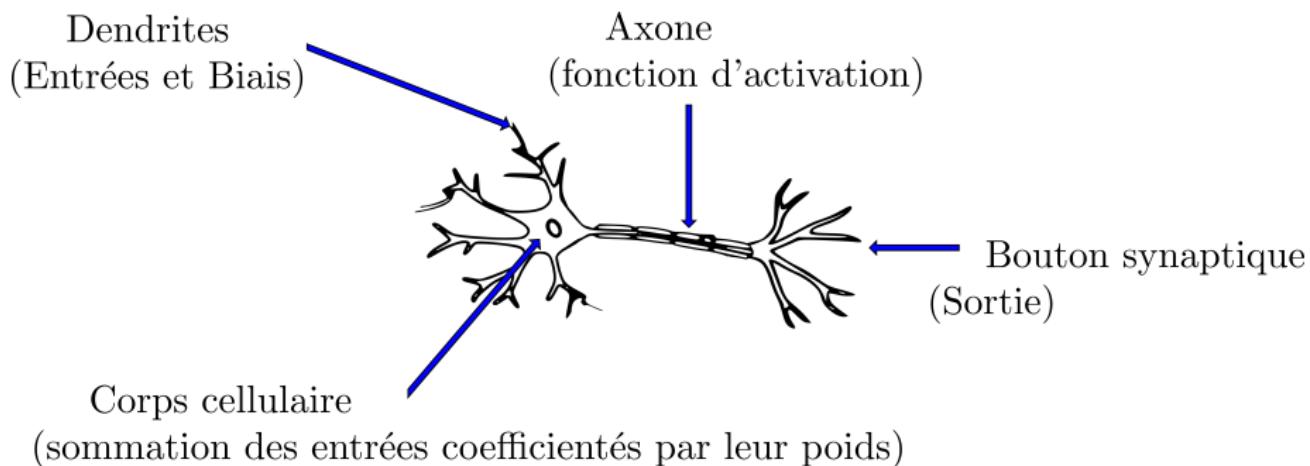


Figure – Schéma d'un neurone humain

# I - Le perceptron

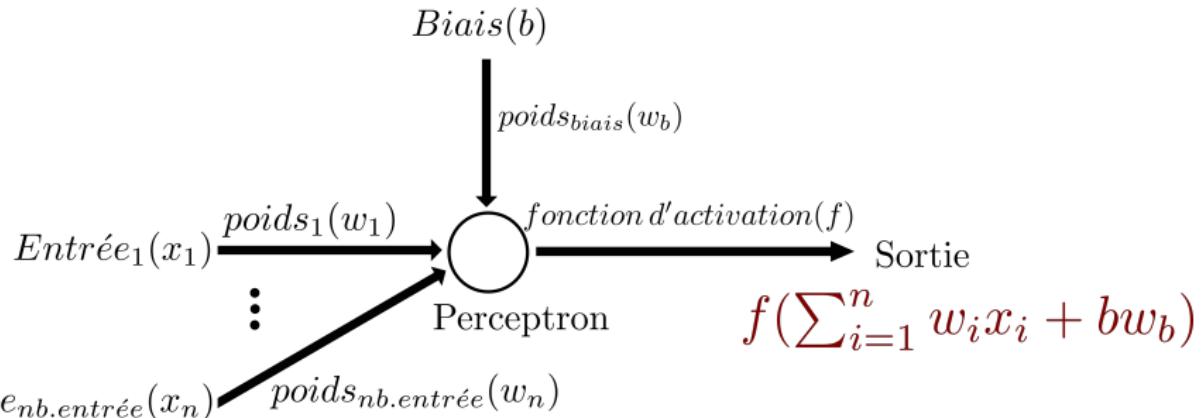


Figure – Schéma d'un perceptron

# I - Représentation informatique

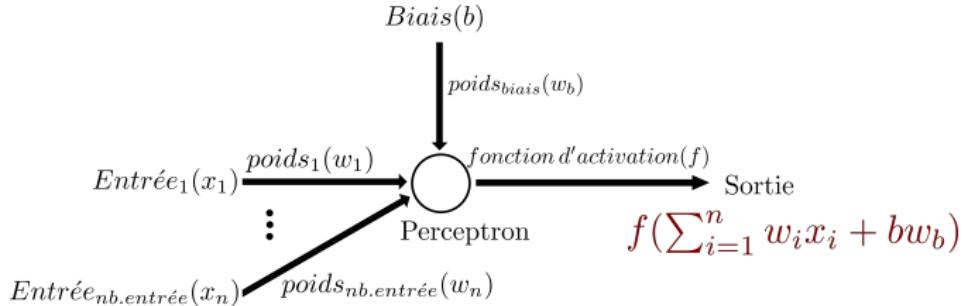


Figure – Schéma d'un perceptron

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

La complexité est en  $O(n)$

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

## II - Fonction d'activation

### Fonction d'activation

Sans fonction d'activation, le neurone sans fonction d'activation est multilinéaire par rapport à ses entrées.

Les fonctions d'activation permettent une classification non linéaire.

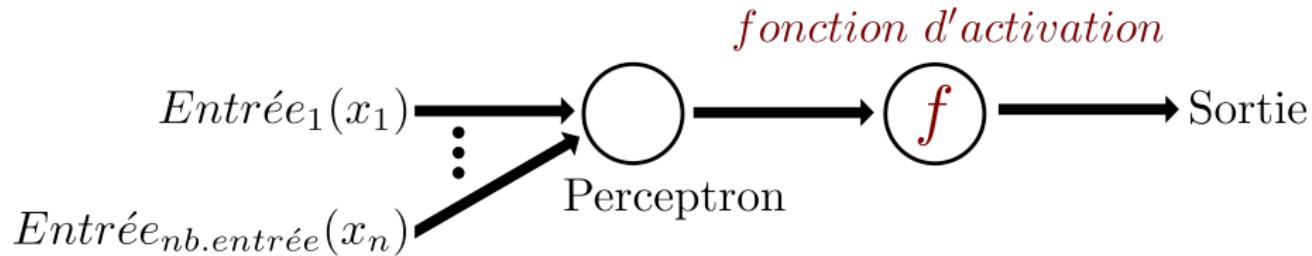
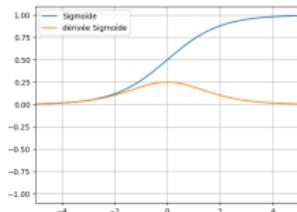
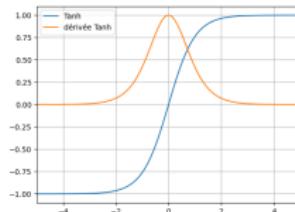


Figure – Utilisation de la fonction d'activation

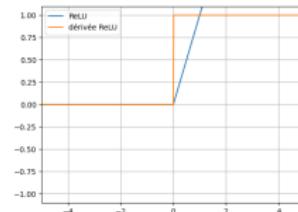
## II - Fonction d'activation



(a) Sigmoïde



(b) Tanh



(c) ReLU

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

## II - Fonction d'activation : Sigmoïde

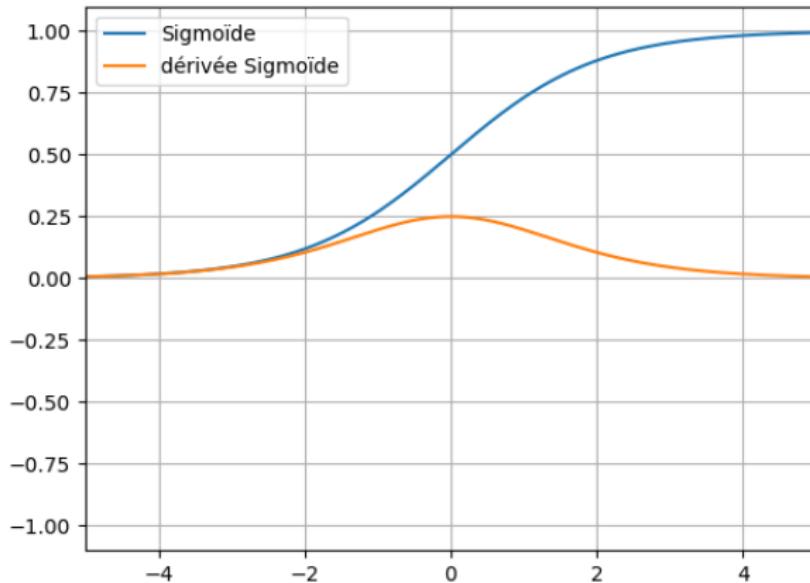


Figure – Sigmoïde

## II - Fonction d'activation : TANH

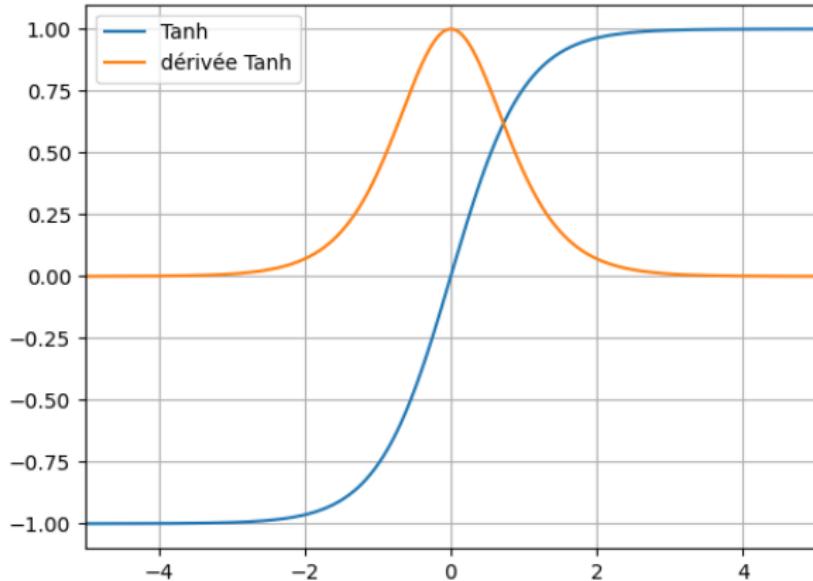


Figure – TANH

## II - Fonction d'activation : ReLu

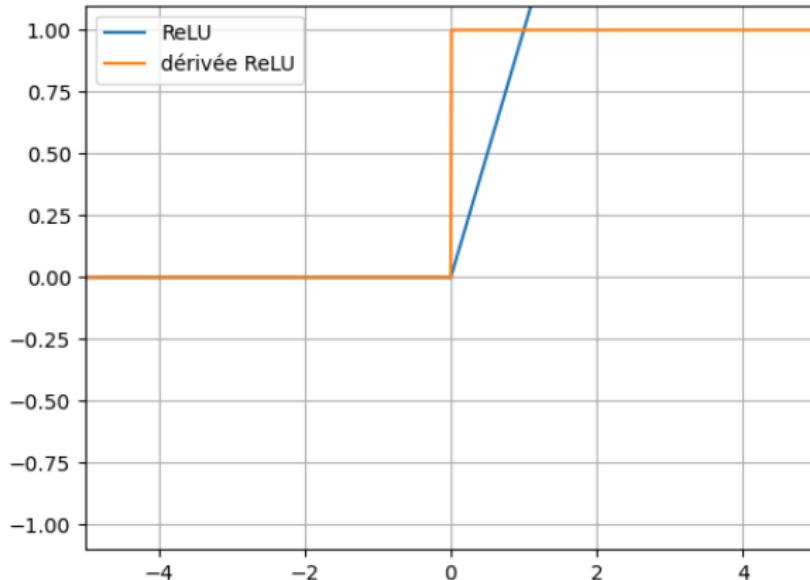


Figure – ReLu

### III - Descente de gradient

#### Descente de gradient

La Descente de Gradient est un algorithme permettant de trouver un minimum local d'une fonction en convergeant.

Elle est utilisée pour trouver le minimum d'une fonction "coût", 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 modèle
- $s_{\text{attendue}}$  la sortie cible.

En effet, trouver des paramètres (poids, architecture du réseau, fonction d'activation) permettant d'avoir une erreur nulle revient à résoudre le problème qu'évalue cette fonction coût par rapport aux données d'entrée.

### III - Descente de gradient

#### Algorithme du gradient

Soit  $n \in \mathbb{N}, \varepsilon > 0$ . On munit  $\mathbb{R}^n$  de son produit scalaire canonique.

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

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

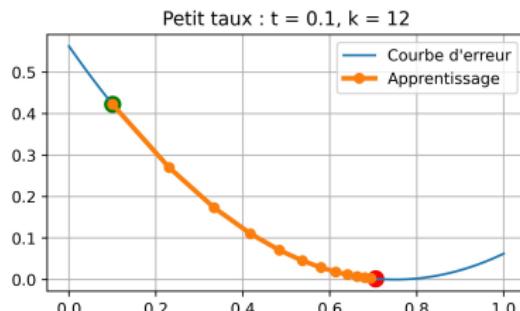
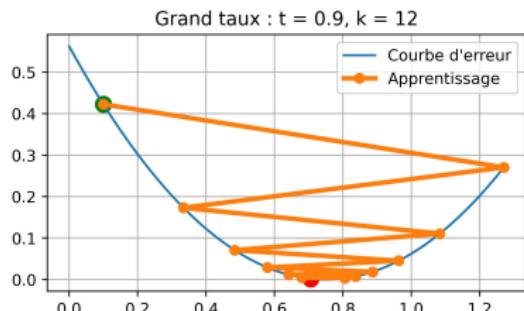


Figure – Descente de Gradient pour  $f(x) = (x - 0.75)^2$ ;  $x_0 = 0.1$  et  $\varepsilon = 0.1$

### III - Importance du choix du taux d'apprentissage

Pour la suite :  $f(x) = (x - 0.75)^2$  et  $x_0 = 0.1$ .

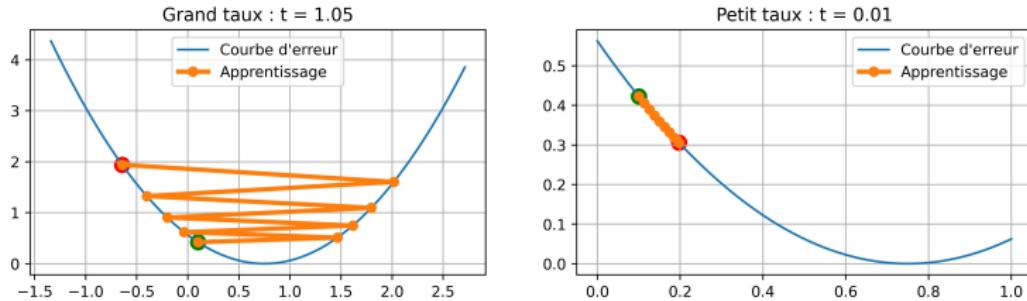


Figure – Descente de Gradient on force l'arrêt à  $k = 8$

Les problèmes rencontrés en fonction du choix du taux d'apprentissage :

- trop grand, la descente de gradient diverge
- trop petit , la descente de gradient converge lentement

# III - Utilisation du momentum

## III - Descente de gradient avec moment

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

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

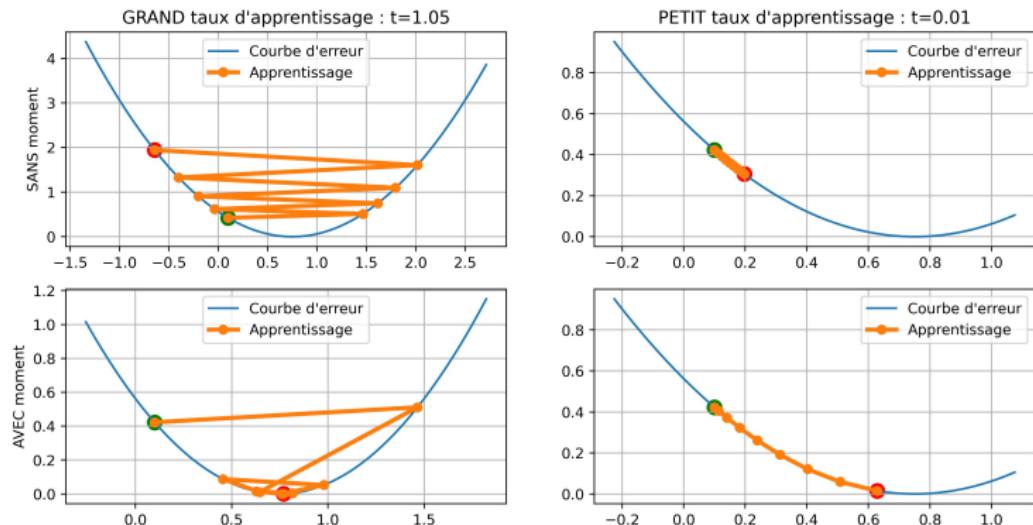


Figure – Descente de gradient SANS/AVEC momentum où  $\gamma = 0.5$ , arrêt à  $k = 4$

### III - Utilisation du momentum pour sortir de minima locaux

#### Autre avantage

Le momentum permet également de s'échapper de minima locaux.

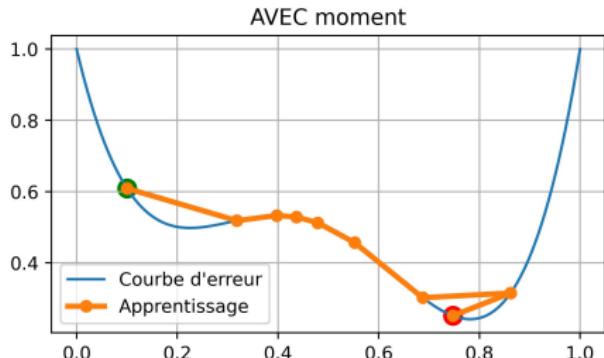
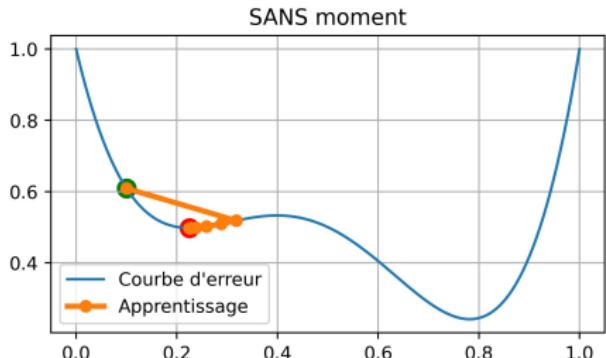
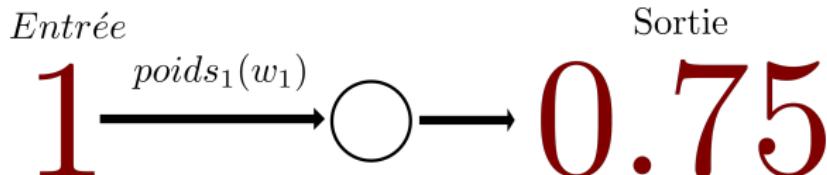


Figure – Descente de gradient SANS/AVEC moment où  $\gamma = 0.5$ , arrêt à  $k = 8$

III - Apprentissage stochastique ou par paquet (Batch)

## La rétropropagation

Ce sont les poids du réseau de neurones qui subissent la descente de gradient.



## Figure – Schéma du perceptron linéaire

```

1 # Petit learning rate
2 reseau = NeuronneLineaire(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 reseau.validation()
8 reseau.plot(
9     ax=plt, title=f"Petit taux : t = 0.1", mini=0, maxi=1
10 )

```

### III - Apprentissage stochastique ou par paquet (Batch)

#### Problème d'imprécision des données

Il faut prendre en compte le fait que les données d'apprentissage ne sont pas toujours exactes.

J'ajoute à mes données une marge d'erreur.

Il est alors préférable d'entraîner notre réseau sur des paquets de données plutôt que donnée par donnée.

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

### III - Apprentissage stochastique ou par paquet (Batch)

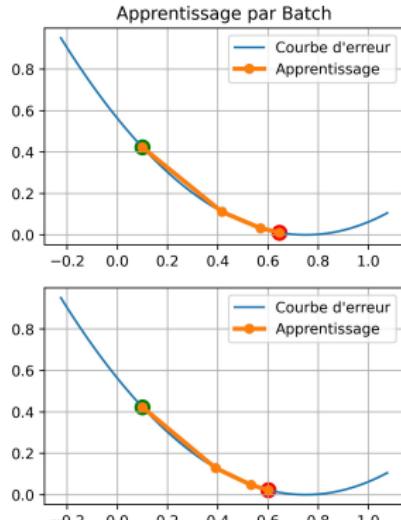
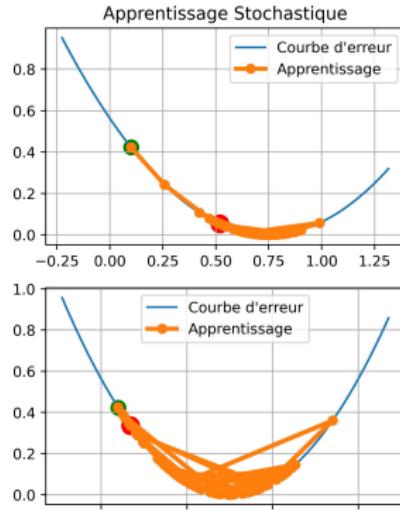
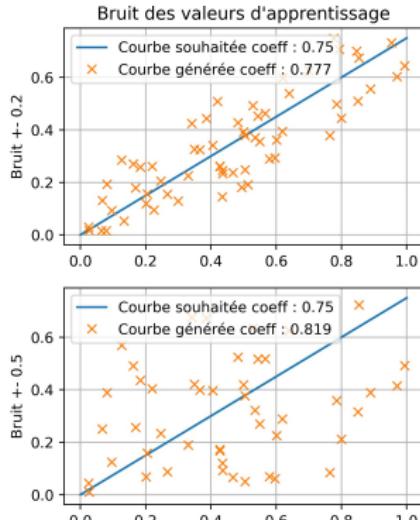


Figure – Comparaison apprentissage stochastique et par paquet,  $f(x) = (x - 0.75)^2$

## IV - Problème de reproduction de l'opérateur XOR

### Problème non linéairement séparables

Une couche de perceptron ne peut pas reproduire les opérateurs non linéairement séparables.

Il faut mettre des couches de perceptron en série (couches cachées) pour pouvoir reproduire ces opérateurs.

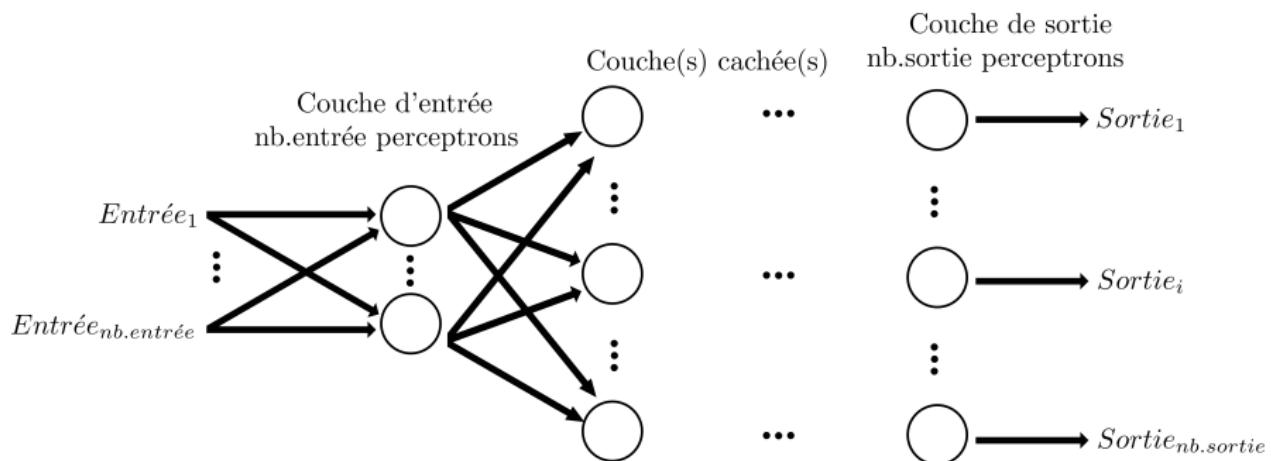


Figure – Schéma d'un réseau de neurones

## IV - Problème de reproduction de l'opérateur XOR

Le XOR nécessite un réseau

Le XOR, « *ou exclusif* », est un opérateur non linéairement séparable.

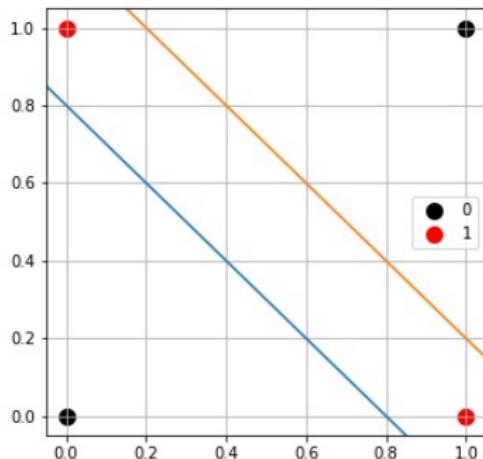


Figure – Schéma de l'opérateur XOR

## IV - Problème de reproduction de l'opérateur XOR

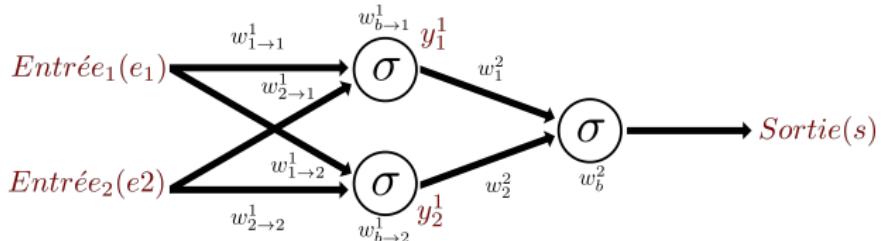


Figure – Schéma du réseau de neurone reproduisant le XOR

### Descente de gradient

$w \leftarrow w - t \frac{\partial f}{\partial w}$  où  $t$  est le taux d'apprentissage et  $f$  la fonction de coût

### Exemple

- $\frac{\partial f}{\partial w_1^2} = 2(s - s_{attendue})\sigma'_2 y_1^1$
- $\frac{\partial f}{\partial w_{1 \rightarrow 1}^1} = 2(s - s_{attendue})\sigma'_2 w_1^2 \sigma'_1 e_1$

## IV - Code

```
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,
8         activation=sigmoid, d_activation=d_sigmoid
9     ),
10    LayerOptimizer(
11        2, 1, lr=10, gamma=0.5,
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
```

## IV - Apprentissage de la reproduction du XOR

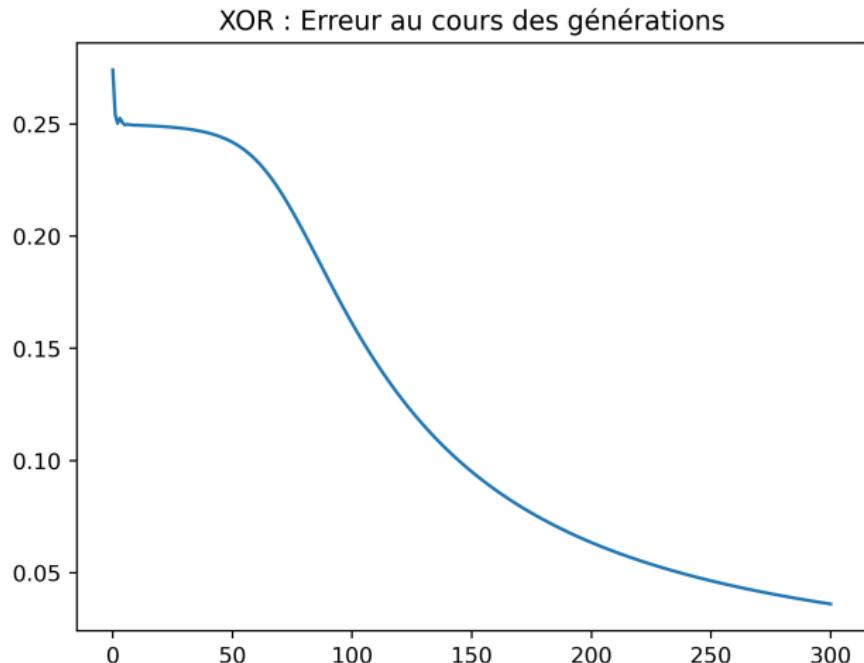


Figure – Erreur sur la reproduction du XOR au cours de l'apprentissage

## IV - Mes résultat

### Données

- 4 données
- 300 générations
- Erreur minimale atteinte 0.036

Entrée :  $\begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \rightarrow \sigma_{couche1} \left( \cdot \times \begin{pmatrix} 0.85 & 5.42 \\ 0.85 & 5.40 \\ 0.14 & 0.44 \end{pmatrix} \right)$

$\rightarrow \sigma_{couche2} \left( \cdot \times \begin{pmatrix} -18.39 \\ 14.42 \\ 0.02 \end{pmatrix} \right)$

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

# V - Reconnaissance d'image

Problématique : Reconnaissance des chiffres écrits à la main

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

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

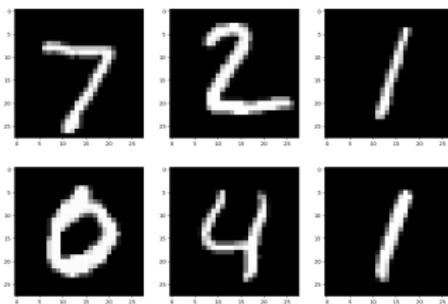


Figure – Exemple d'images

# V - Apprentissage d'un problème de classification

On prend un réseau de neurones avec  $28 \times 28 = 784$  entrées, et 10 sorties.

## Softmax

La fonction d'activation softmax permet de normaliser en probabilités les sorties :

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

## Cross-entropy

La fonction coût des problèmes de classification est Cross-entropy :

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

# V - Softmax

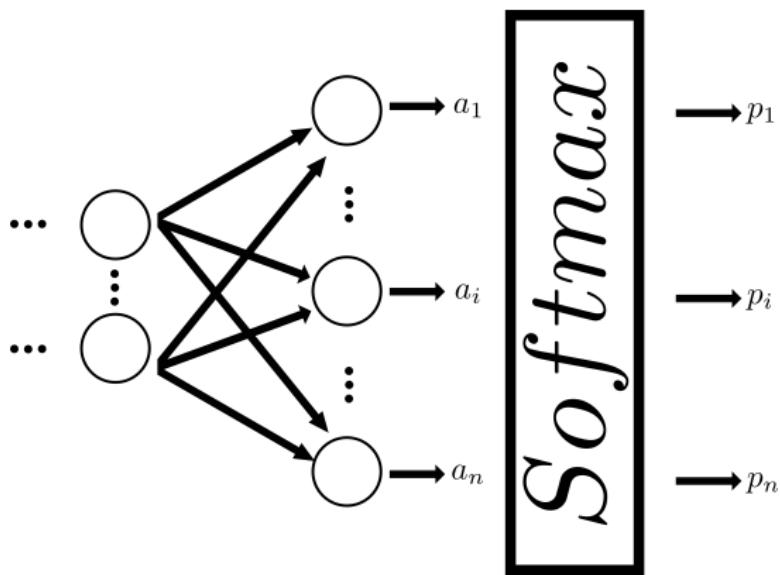
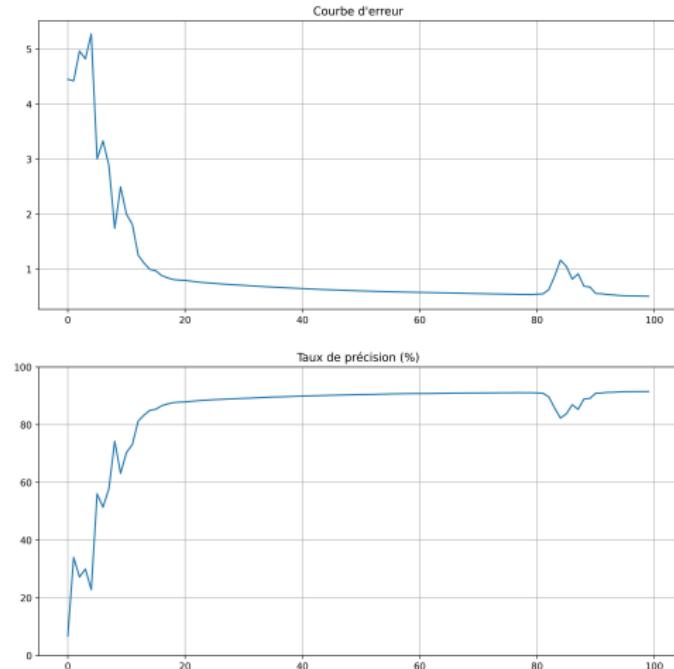


Figure – Schéma d'utilisation du Softmax

## V - Mes résultats



Taux de bonne réponse après 100 générations d'entraînement :

- 91.5% sur les données d'entraînement
- 91.3% sur les données de validation.

Figure – Courbes d'apprentissage

# V - Mes résultats



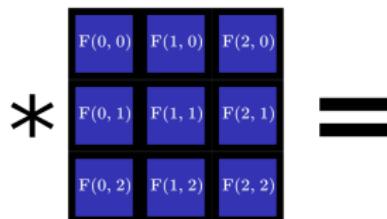
Figure – Exemple sur un échantillon de 40 images de validation

# V - 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

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

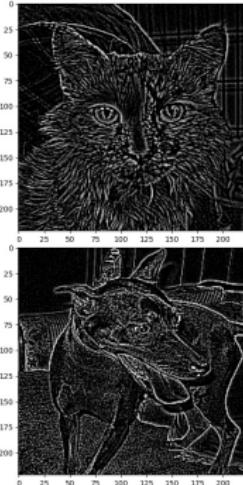


Filtre (Kernel)

O(0, 0)	O(1, 0)	O(2, 0)	O(3, 0)	O(4, 0)
O(0, 1)	O(1, 1)	O(2, 1)	O(3, 1)	O(4, 1)
O(0, 2)	O(1, 2)	O(2, 2)	O(3, 2)	O(4, 2)
O(0, 3)	O(1, 3)	O(2, 3)	O(3, 3)	O(4, 3)
O(0, 4)	O(1, 4)	O(2, 4)	O(3, 4)	O(4, 4)

Image de sortie

# V - Convolution d'image



## Détection de contour

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

```
1 kernel = np.array([
2     [-1, -1, -1],
3     [-1, 8, -1],
4     [-1, -1, -1]
5 ])
6 # convolve
7 def convolve(img):
8     ks = kernel.shape[0]
9     ts = img.shape[0]-ks+1
10    # Initialisation de l'image
11    c_img = np.zeros(shape=(ts,ts))
12    for i in range(ts):
13        for j in range(ts):
14            # sous matrice
15            m = img[i:i+ks, j:j+ks]
16            # produit d'Hadamard
17            c_img[i, j] = np.sum(
18                m*kernel
19            )
20    return c_img
```

# V - Utilisation de réseau de neurones convolutif



```
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

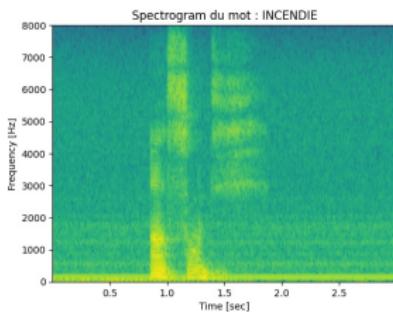
## VI - Reconnaissance de mot

Le site du gouvernement recense ces mots-clés pour les numéro d'appels d'urgences :

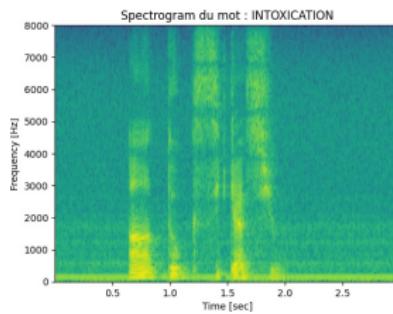
n°15 malaise, hémorragie, brûlure, intoxication

n°17 violences, agression, vol, cambriolage

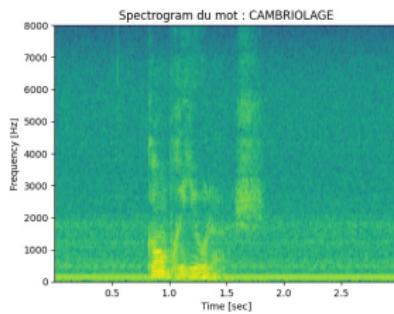
n°18 incendie, gaz, effondrement, électrocution



(a) INCENDIE



(b) INTOXICATION



(c) CAMBRIOLAGE

## VI - Transfert d'apprentissage

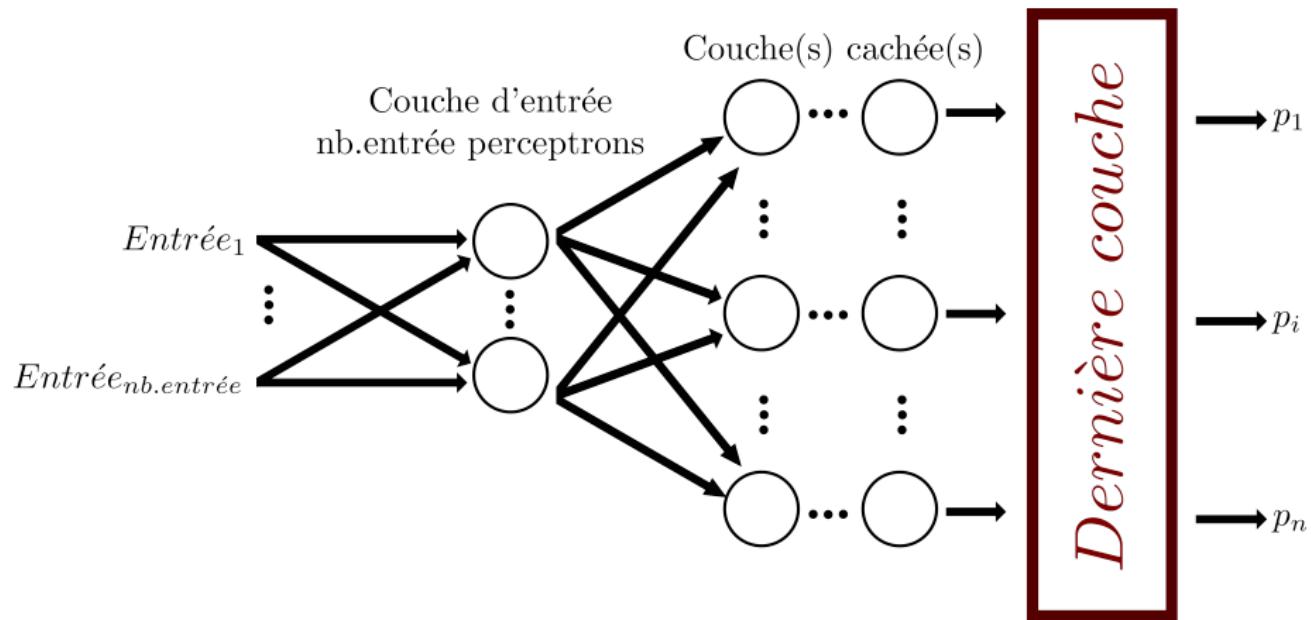


Figure – Schéma de fonctionnement du transfert d'apprentissage

## VI - Réseau de neurones modèle

```
1 Layer (type)          Output Shape         Param #
2 =====
3 resizing (Resizing)    (None, 32, 32, 1)      0
4
5 conv2d (Conv2D)        (None, 30, 30, 32)     320
6
7 conv2d_1 (Conv2D)       (None, 28, 28, 64)    18496
8
9 max_pooling2d (MaxPooling2D) (None, 14, 14, 64) 0
10
11 dropout (Dropout)     (None, 14, 14, 64)    0
12
13 flatten (Flatten)     (None, 12544)          0
14
15 dense (Dense)         (None, 128)            1605760
16
17 dropout_1 (Dropout)   (None, 128)            0
18
19 dense_1 (Dense)        (None, 8)              1032
20 =====
21 Total params: 1,625,608
22 Trainable params: 1,625,608
23 Non-trainable params: 0
```

# VI - Entrainement du réseau modèle

## Modèle

Le réseau modèle est entraîné sur 8 000 données pour reconnaître les 8 mots :

[*"no"*, *"stop"*, *"up"*, *"right"*, *"yes"*, *"left"*, *"go"*, *"down"*]

## Adaptation

Nous voulons que notre réseau de neurones reconnaîsse les 12 mots :

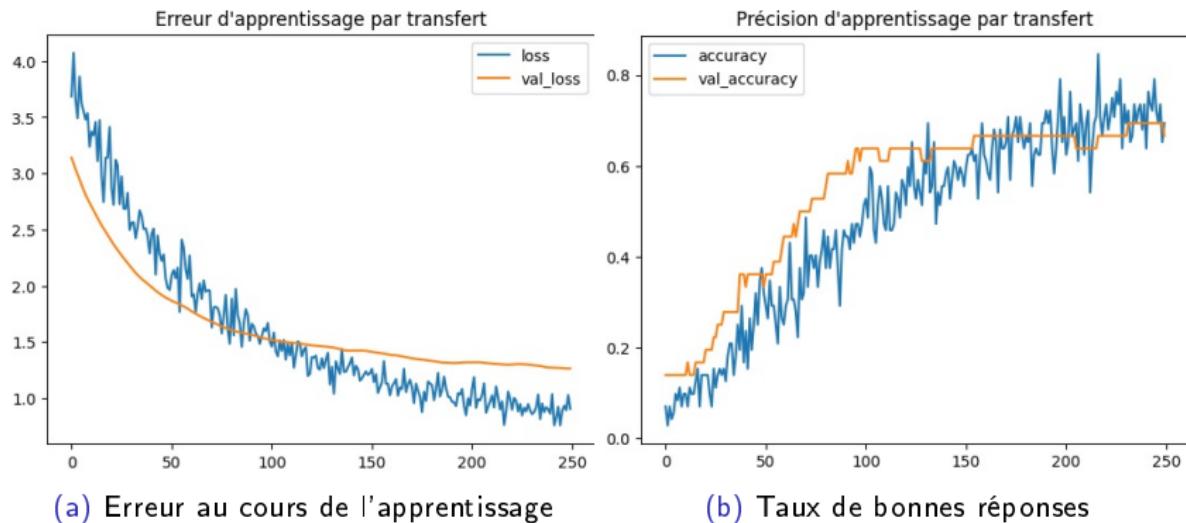
[*"malaise"*, *"hémorragie"*, *"brûlure"*, *"intoxication"*,  
*"violences"*, *"agression"*, *"vol"*, *"cambriolage"*,  
*"incendie"*, *"gaz"*, *"effondrement"*, *"électrocution"*]

```
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 )
```

## VI - Réseau de neurones modèle

```
1 Layer (type)          Output Shape         Param #
2 =====
3 resizing (Resizing)    (None, 32, 32, 1)      0
4
5 conv2d (Conv2D)        (None, 30, 30, 32)     320
6
7 conv2d_1 (Conv2D)       (None, 28, 28, 64)    18496
8
9 max_pooling2d (MaxPooling2D) (None, 14, 14, 64) 0
10
11 dropout (Dropout)     (None, 14, 14, 64)    0
12
13 flatten (Flatten)     (None, 12544)        0
14
15 dense (Dense)         (None, 128)          1605760
16
17 dropout_1 (Dropout)   (None, 128)          0
18
19 adaptation (Dense)    (None, 12)           1548
20 =====
21 Total params: 1,626,124
22 Trainable params: 1,548
23 Non-trainable params: 1,624,576
```

## VI - Mes résultats



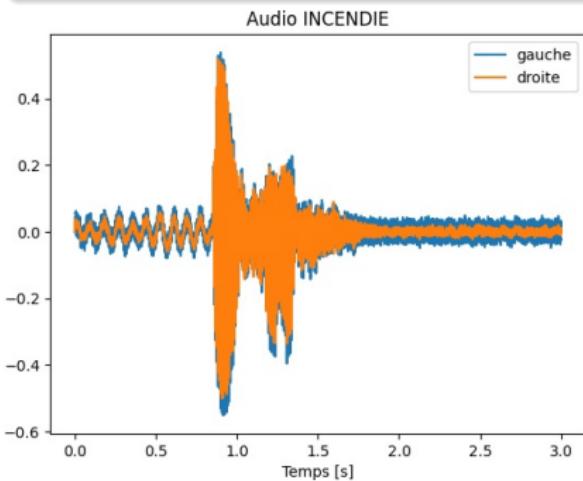
## VI - Analyse de mes résultats

### Données d'apprentissage

Agathe  $1 \times 12$  données

Florent  $1 \times 12$  données

Tien-Thinh  $4 \times 12$  données



### Données d'évaluation

Agathe  $1 \times 12$  données

Florent  $1 \times 12$  données

Tien-Thinh  $1 \times 12$  données

### Précision finale

Agathe 50% accuracy

Florent 75% accuracy

Tien-Thinh 75% accuracy

# VI - Analyse découpage en formant

## Formant - Définition Larousse

Fréquence de résonance du conduit vocal.

Les voyelles se définissent acoustiquement par leurs formants ; seules les consonnes dites vocaliques présentent une structure formantique similaire à celle des voyelles.

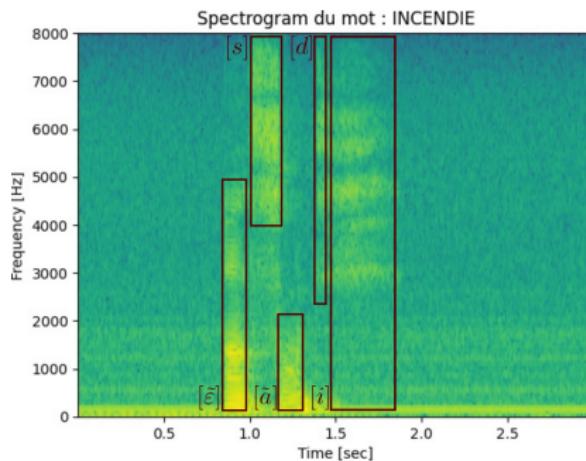


Figure – Analyse de formant du mot "INCENDIE"  
avec l'aide de Mme.Voisin en formation d'orthophonie à la Sorbonne

# VI - Mon réseau de neurones reconnaissant les mots-clés

*Couches issues du transfert d'apprentissage*

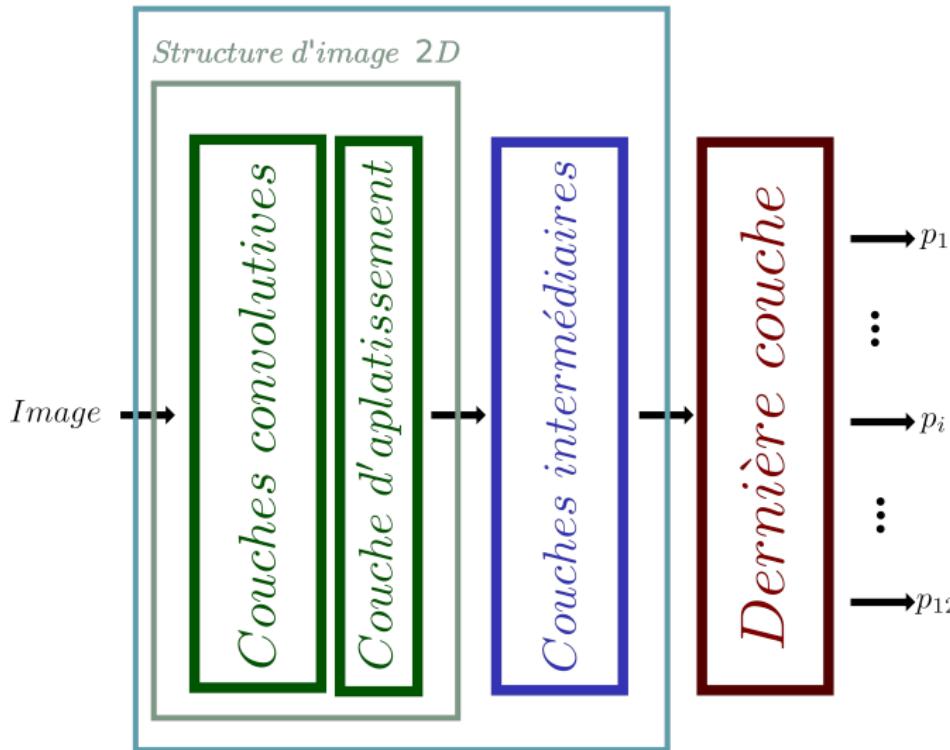


Figure – Structure de mon réseau de neurones

# Annexe

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

# 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 # prédition du résultat
12
13
14    @staticmethod
15    def erreur(y_, y): # sortie calculé, sortie souhaitée
16        return (y - y_) ** 2
17
18
19    @staticmethod
20    def d_erreur(x, y_, y): # sortie calculé, sortie souhaitée
21        return (2*(y_-y) * 1 * x).mean()
```

# NeuroneLineaire.py ||

```
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)
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
```

# NeuroneLineaire.py III

```
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
8 # Sigmoid
9 def sigmoid(x):
10    return 1 / (1 + np.exp(-np.clip(x, -10, 10)))
11 def d_sigmoid(x):
12    f_x = sigmoid(x)
13    return f_x * (1-f_x)
14
15
16 # Tanh
17 def tanh(x):
18    return np.tanh(x)
19 def d_tanh(x):
20    return 1-tanh(x)**2
21
22
```

# Activation.py ||

```
23 # ReLU
24 def relu(x):
25     return np.clip(x, 0, np.inf)
26 def d_relu(x):
27     return 1*(x>0)
28
29
30 # Softmax
31 def softmax(x):
32     exp = np.exp(x-x.max(axis=-1, keepdims=True))
33     return exp / exp.sum(axis=-1, keepdims=True)
34 def d_softmax(x):
35     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, maxi:
4      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
19        self.predicted_output_ = 0 # sortie avant activation
20        self.predicted_output = 0 # sortie
21        self.input_data = 0
```

# Layer.py ||

```
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(
39             (input_data, np.zeros((len(input_data), 1))),
40             axis=1
41     )
42     y1 = self.input_data@self.weight
43     z1 = self.activation(y1)
```

# Layer.py |||

```
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
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
```

# Layer.py |V

```
65     Sauf l'apprentissage
66         qui invoque un taux d'apprentissage variable
67         on utilise la variable gamma
68 """
69
70 def __init__(self, input_n:int, output_n:int, lr:float,
activation, d_activation, bias:bool=True, mini:float=0, maxi:
float=1, gamma:float=0.5):
71     # classe héritée
72     super().__init__(
73         input_n, output_n,
74         lr, activation, d_activation, bias, mini, maxi
75     )
76     self.gamma = gamma
77     self.dw_moment = np.zeros((input_n+1, output_n))
78
79
80 def learn(self, e_2:np.ndarray):
81 """
82     Permet de mettre à jour les poids weight
83     en prenant en compte le moment
84 """
```

# Layer.py V

```
85     e1 = e_2 / (self.input_n+1)
86     e1 = e1 * self.d_activation(self.predicted_output)
87     # e_0 est pour l'entraînement de la couche précédente
88     e_0 = np.dot(e1, self.weight.T)[:, :-1]
89     dw1 = np.dot(e1.T, self.input_data)
90
91     # La différence se trouve est ci-dessous
92     self.dw_moment = dw1.T * self.lr
93     self.dw_moment += self.gamma * self.dw_moment
94     self.weight -= self.dw_moment
95     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:bool=True,
13                 mini:float=0, maxi:float=1):
14         # classe héritée
15         input_n = kernel_size**2
16         output_n = nb_kernel
17         super().__init__(
18             input_n, output_n,
19             lr, activation, d_activation, bias, mini, maxi
20         )
21         self.kernel_size = kernel_size
22         self.nb_kernel = nb_kernel # nombre total de filtre
```

# Convolutional.py II

```
21     self.img_size = img_size
22     self.output_size = max(0, img_size-kernel_size+1)
23
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
31         n = self.output_size
32         k = self.kernel_size
33         t_imgs = np.array([
34             img[lig:lig+k, col:col+k].flatten()
35             for img in imgs
36             for lig in range(n)
37             for col in range(n)
38         ])
39
40         return t_imgs
41
42     def transform_k(self, imgs:np.ndarray):
43         """
```

# Convolutional.py III

```
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):
58     """
59     Transforme la sortie
60     en une liste d'image
61     """
62     n = self.output_size
63     nk = self.nb_kernel
```

# Convolutional.py IV

```
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 carré
76     """
77     Calcule la sortie
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     ])
```

# Convolutional.py V

```
85
86     if self.bias:
87         self.input_data = np.concatenate(
88             (input_data_k, np.ones((len(input_data_k), 1))), 
89             axis=1
90         )
91     else:
92         self.input_data = np.concatenate(
93             (input_data_k, np.zeros((len(input_data_k), 1))), 
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.dettransform_k(y1, nb_donnee, nb_images)
101    z1 = self.dettransform_k(z1, nb_donnee, nb_images)
102    return (y1, z1)
103
104
105    def learn(self, e_2:np.ndarray):
106        """
```

# Convolutional.py VI

```
107     Permet de mettre à jour les poids weight
108 """
109
110     shape = e_2.shape
111     e_2 = self.transform_k(e_2)
112     e1 = e_2 / (self.input_n+1)
113     e1 = e1 * self.d_activation(self.predicted_output)
114     # e_0 est pour l'entraînement de la couche précédente
115     e_0 = np.dot(e1, self.weight.T)[:, :-1]
116     dw1 = np.dot(e1.T, self.input_data)
117     self.weight -= dw1.T * self.lr
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
127     def __init__(self, img_size):
128         self.img_size = img_size
129         self.output_n = img_size**2
```

# Convolutional.py VII

```
129
130
131     def calculate(self, imgs):
132         nb_donne = len(imgs)
133         nb_img = len(imgs[0])
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
20         for layer in self.layers: # Hérité
21             predicted_output_,predicted_output = layer.calculate(
22                 predicted_output)
```

# Model.py II

```
22
23     )
24
25
26     def predict_loss(self, input_data:np.ndarray, target_output:
27         np.ndarray):
28         # Sortie et Erreur
29         predicted_output = self.predict(input_data)
30         loss = self.loss_function(
31             predicted_output, target_output
32         )
33
34
35     def backpropagation(self, input_data:np.ndarray,
36         target_output:np.ndarray):
37         # Entrainement par récurrence
38         predicted_output, loss = self.predict_loss(
39             input_data, target_output
40         )
41         # dérivée
42         d_loss = self.d_loss_function(
```

# Model.py III

```
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,
55                  d_loss_function):
56         # classe héritée
57         super().__init__(layers, loss_function, d_loss_function)
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
```

# Model.py IV

```
62
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
71
72     def backpropagation(self, input_data:np.ndarray,
73     target_output:np.ndarray):
74         # Entrainement par récurrence
75         predicted_output, loss, acc = self.predict_loss(
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)
```

# Model.py V

```
83         self.loss.append(loss)
84     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),
3                                         sampling_rate, pitch_factor)
4
5 def speed_manipulate(data, speed_factor):
6     return librosa.effects.time_stretch(data.astype(np.float32),
7                                         speed_factor)
8
9 def noise_manipulation(data, noise_factor=5_000):
10    noise = np.random.randn(len(data))
11    augmented_data = data + noise_factor * noise
12    augmented_data = augmented_data.astype(type(data[0]))
13    return augmented_data
14
15
16 def shift_manipulation(data, sampling_rate, shift_max):
17    shift = np.random.randint(max(sampling_rate * shift_max, 1))
18    augmented_data = np.roll(data, shift)
19    return augmented_data
```