TIPE : Réseaux de neurones convolutifs pour le diagnostic médical

Thème : Santé et prévention

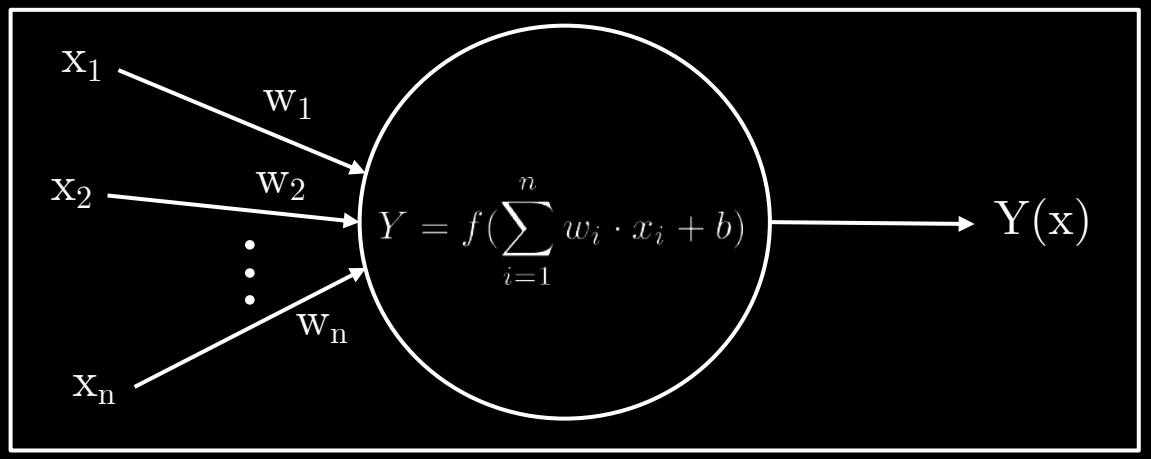
Problématique : Comment construire un réseau de neurone pour établir un diagnostic médical à partir d'images ?

Yassine LARAKI

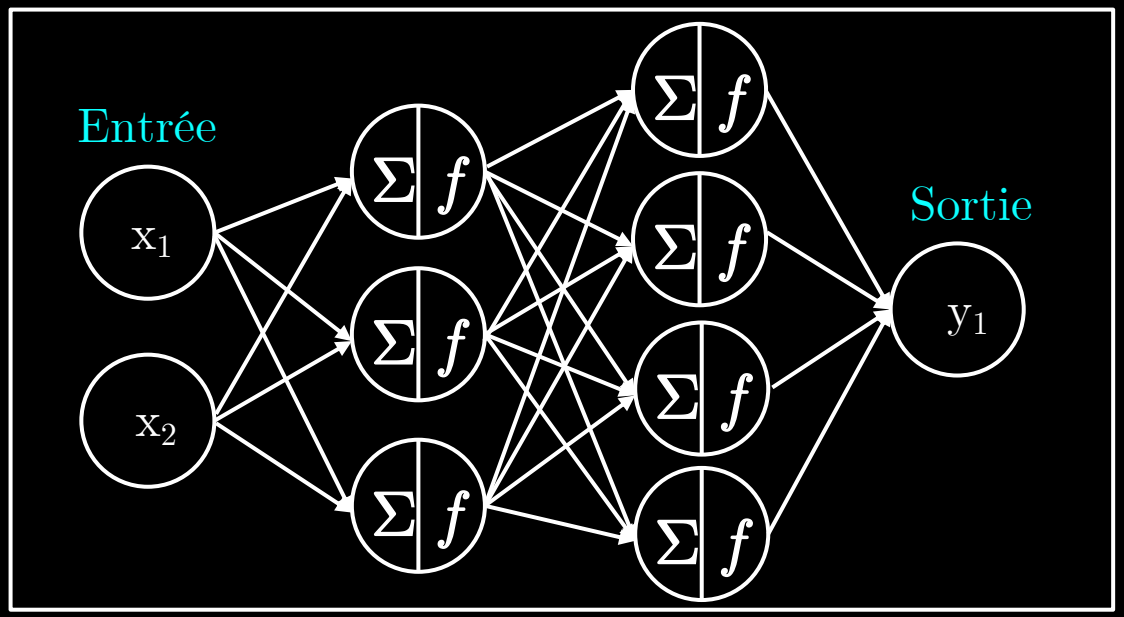
Numéro SCEI : 10104

MP – Option informatique

Neurone formel et perceptron multicouche



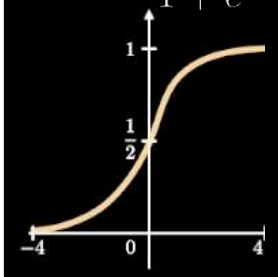
Neurone formel



Perceptron multicouche

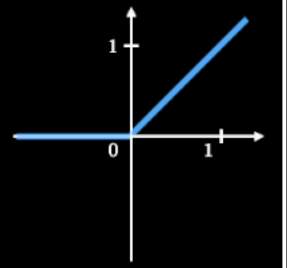
Sigmoïde

$$f(x) = \frac{1}{1 + e^{-x}}$$



Relu

$$f(x) = max(0, x)$$



Softmax

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Fonctions d'activation

Algorithme du perceptron multicouche

1. Initialisation aléatoire des poids

2. Propagation directe

$x = [x_1 , ... , x_{n0}] \in \{\text{entrées}\}$

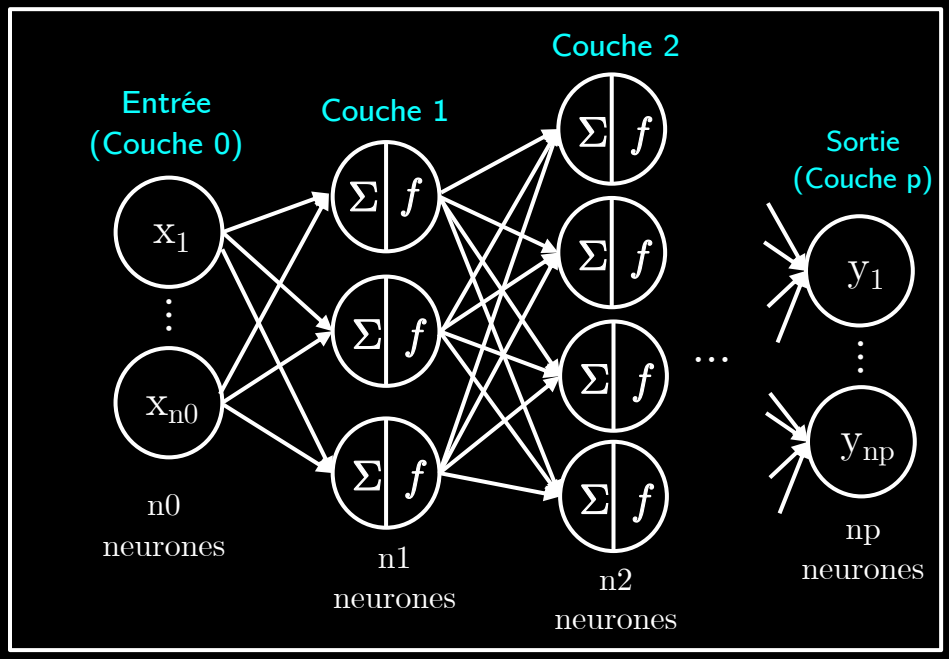
$s = [s_1 , ... , s_{nj}]$: sortie associée

On pose $x(0) = x$

Pour $i = 1$ à p:

$x(i) = (Y_{i,1}(x(i-1)), ... , Y_{i,ni}(x(i-1)))$

On pose $y = x(p)$



Perceptron multicouche

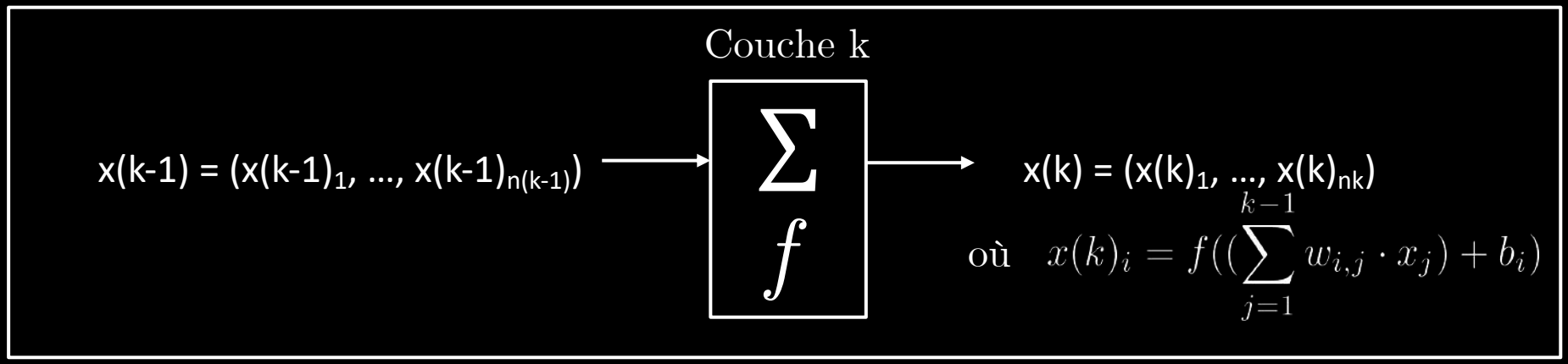
3. Calcul de l'erreur

Fonction d'erreur : $E = -\sum_{i=1}^{n} s_i \cdot log(y_i)$ (categorical cross entropy)

Dérivée de l'erreur : $\frac{\partial E}{\partial y_i} = \frac{\partial E}{\partial x(np)_i} = y_i - s_i$ (en utilisant f = softmax comme fonction d'activation pour la dernière couche)

Algorithme du perceptron multicouche

4. Rétropropagation du gradient



Couche du perceptron

Connus : $\dfrac{\partial E}{\partial x(k)_i}$

(i) $$\frac{\partial E}{\partial w_{i,j}} = \sum_{l=1}^{nk} \frac{\partial E}{\partial x(k)_l} \cdot \frac{\partial x(k)_l}{\partial w_{i,j}} = \frac{\partial E}{\partial x(k)_i} \cdot x(k-1)_j \cdot f'(\sum_{l=1}^{n(k-1)} w_{i,l} \cdot x(k-1)_l + b_i)$$

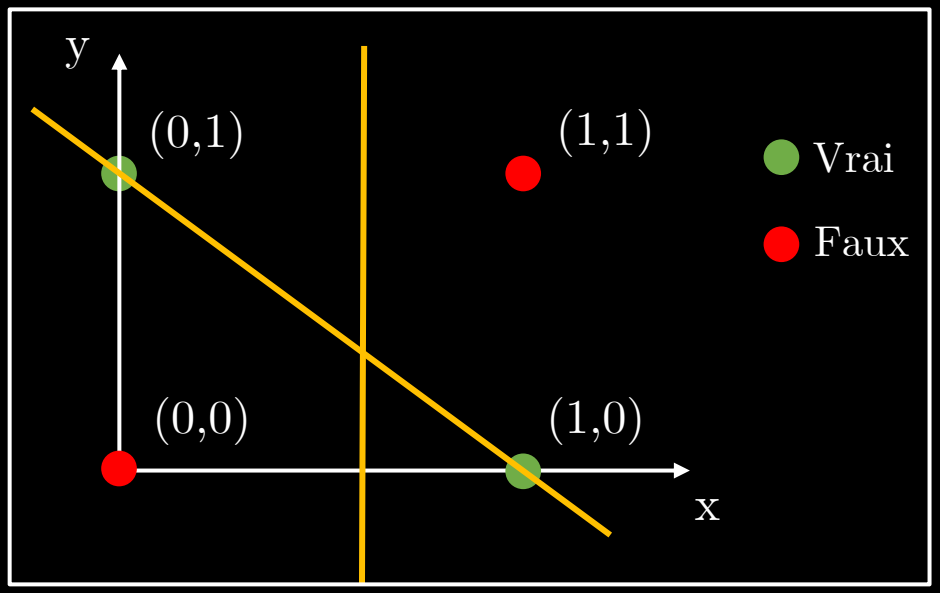
(ii) $$\frac{\partial E}{\partial b_i} = \sum_{l=1}^{nk} \frac{\partial E}{\partial x(k)_l} \cdot \frac{\partial x(k)_l}{\partial b_i} = \frac{\partial E}{\partial x(k)_i} \cdot f'(\sum_{j=1}^{n(k-1)} w_{i,j} \cdot x(k-1)_j + b_i)$$

(iii) $$\frac{\partial E}{\partial x(k-1)_j} = \sum_{i=1}^{nk} \frac{\partial E}{\partial x(k)_i} \cdot \frac{\partial x(k)_i}{\partial x(k-1)_j} = \sum_{i=1}^{nk} \frac{\partial E}{\partial x(k)_i} \cdot w_{i,j} \cdot f'(\sum_{l=1}^{n(k-1)} w_{i,l} \cdot x(k-1)_l + b_i)$$

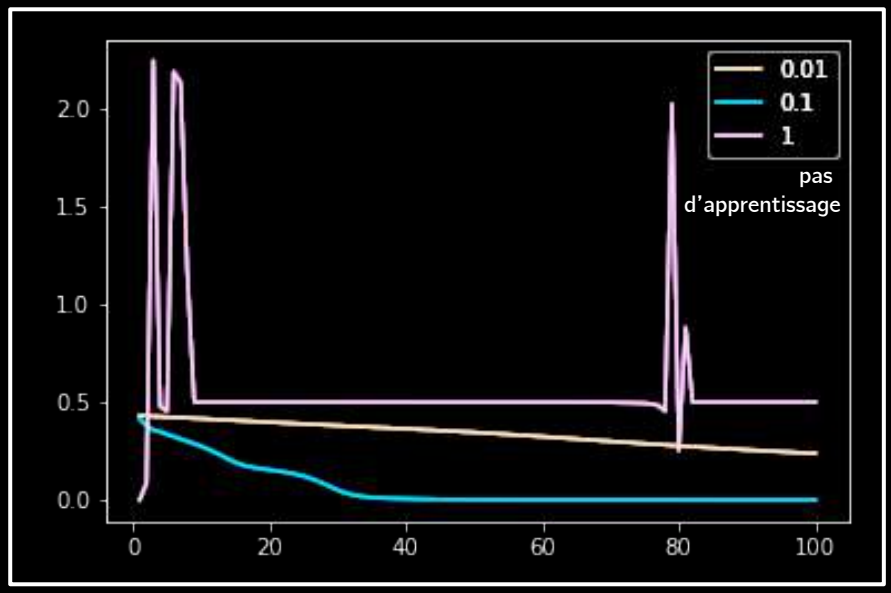
Mise à jour des poids : $w_{i,j} = w_{i,j} - \alpha \dfrac{\partial E}{\partial w_{i,j}}$

Mise à jour des biais : $b_i = b_i - \alpha \dfrac{\partial E}{\partial b_i}$

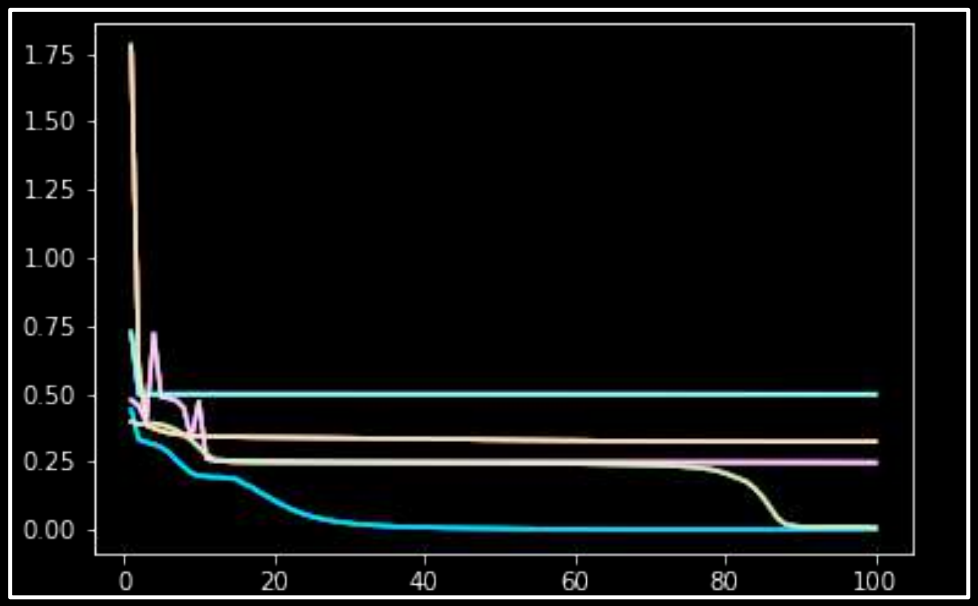
XOR



XOR

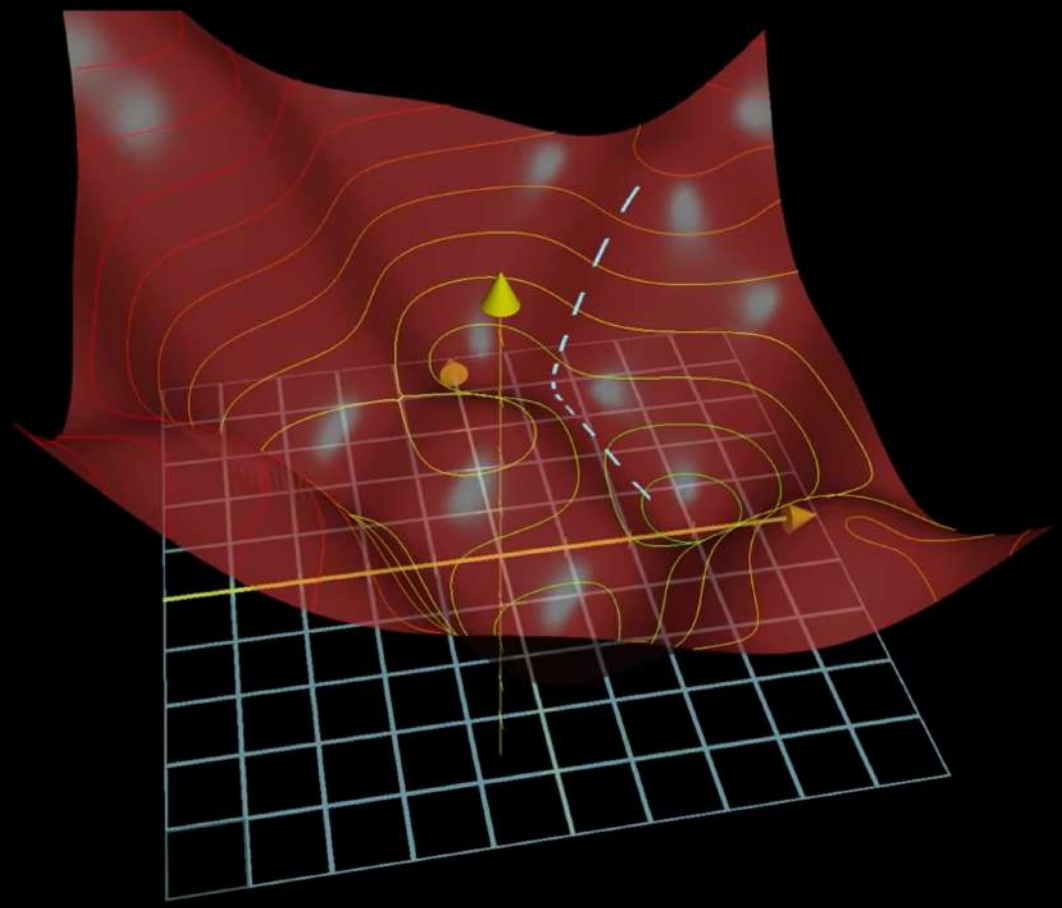


Évolution de l'erreur dans un perceptron
en fonction du pas d'apprentissage
Perceptron de type : [2,5,3,1]



Différents essais avec le même réseau de neurones et pas d'apprentissage mais avec des poids/biais réinitialisés aléatoirement à chaque début d'apprentissage

Visualiser la descente de gradient



Tracé de l'erreur en fonction de deux poids

Source : 3Blue1Brown (Youtube)

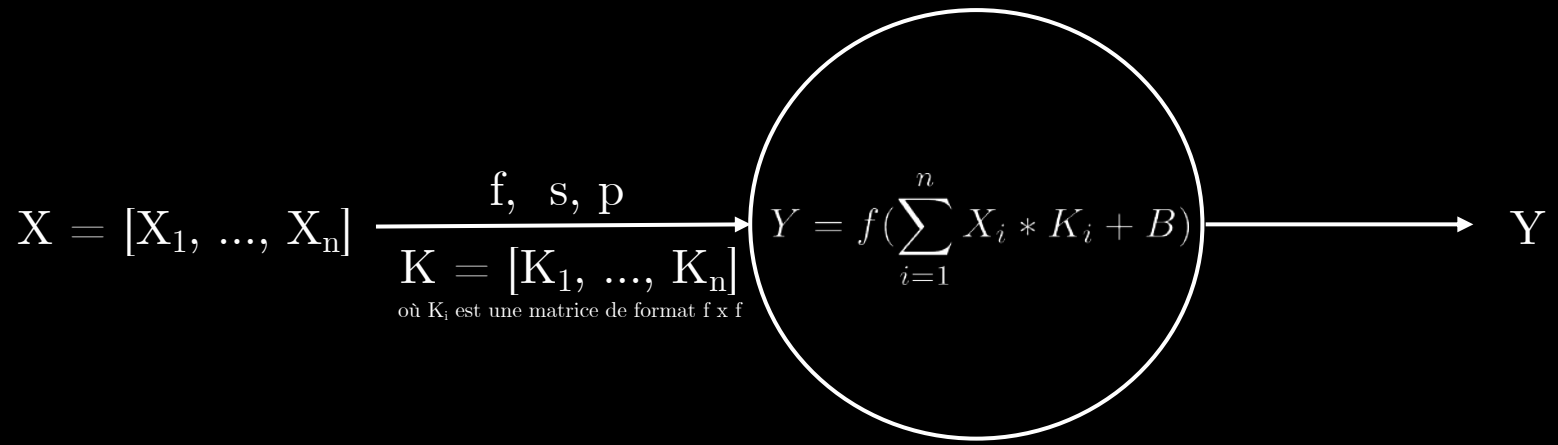
Convolution

Hyperparamètres : stride (noté s), pooling (noté p), taille du filtre (noté f)

Format de la sortie : $\lfloor \frac{n+2p-f}{s} \rfloor + 1$

Ici :
f = (3,3)
p = 0
s = 1

Opération de convolution

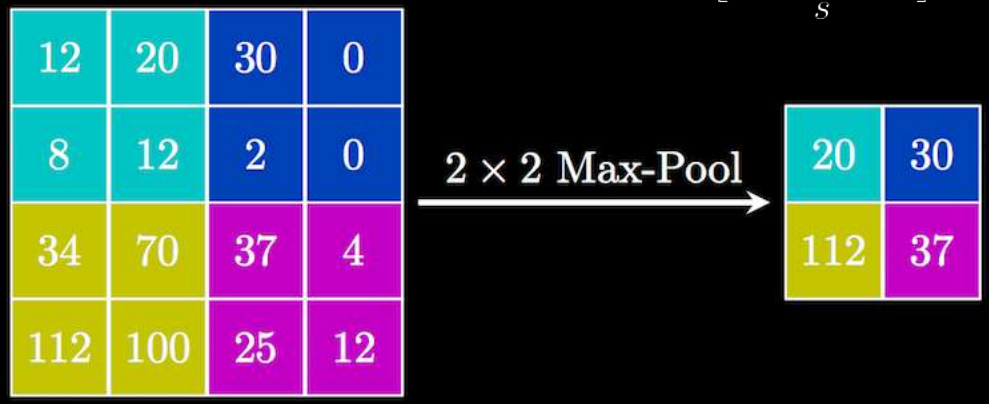


Neurone de convolution

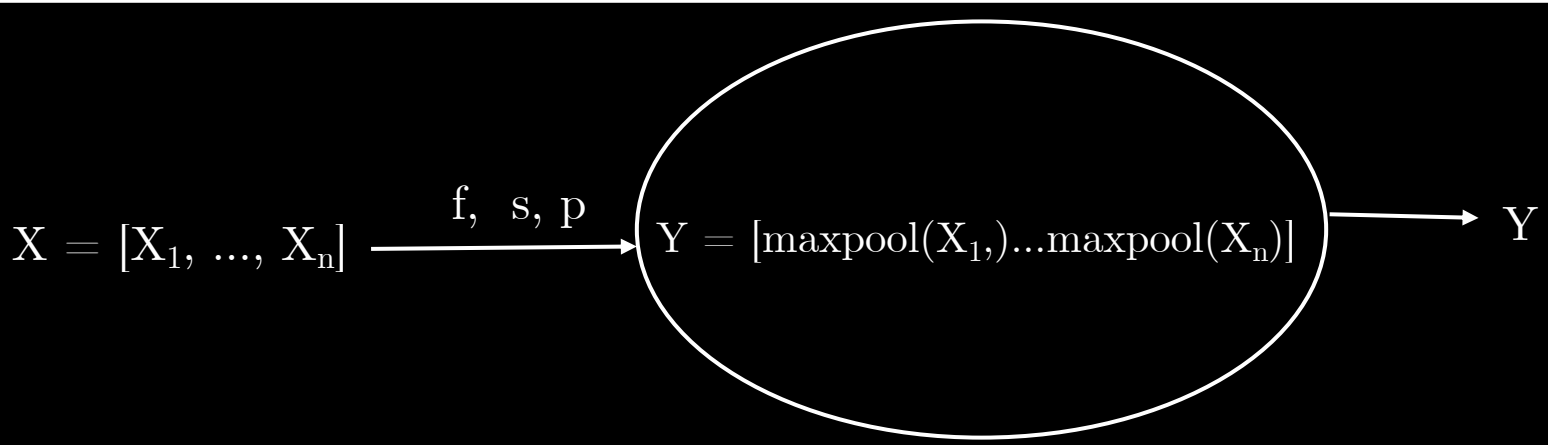
Maxpooling

Hyperparamètres : stride (noté s), taille du filtre (noté f) (s = f souvent), pooling (noté p) (p = 0 souvent)

Format de la sortie : $\lfloor \frac{n+2p-f}{s} \rfloor + 1$



Opération de convolution

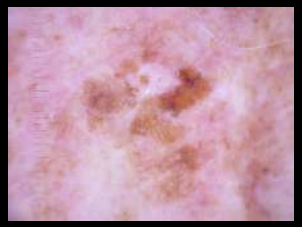


Couche de maxpooling

Base de donnée HAM10000

- Lésions bénignes (noté 'bkl')

0




- Kératose/ Bowen (noté 'akiec')

1


- Carcinome basocellulaire (noté 'bcc')

2


- Dermatofibrome (noté 'df')

3


- Mélanome (noté 'mel')

4


- Nævus mélanocytaire (noté 'nv')

5

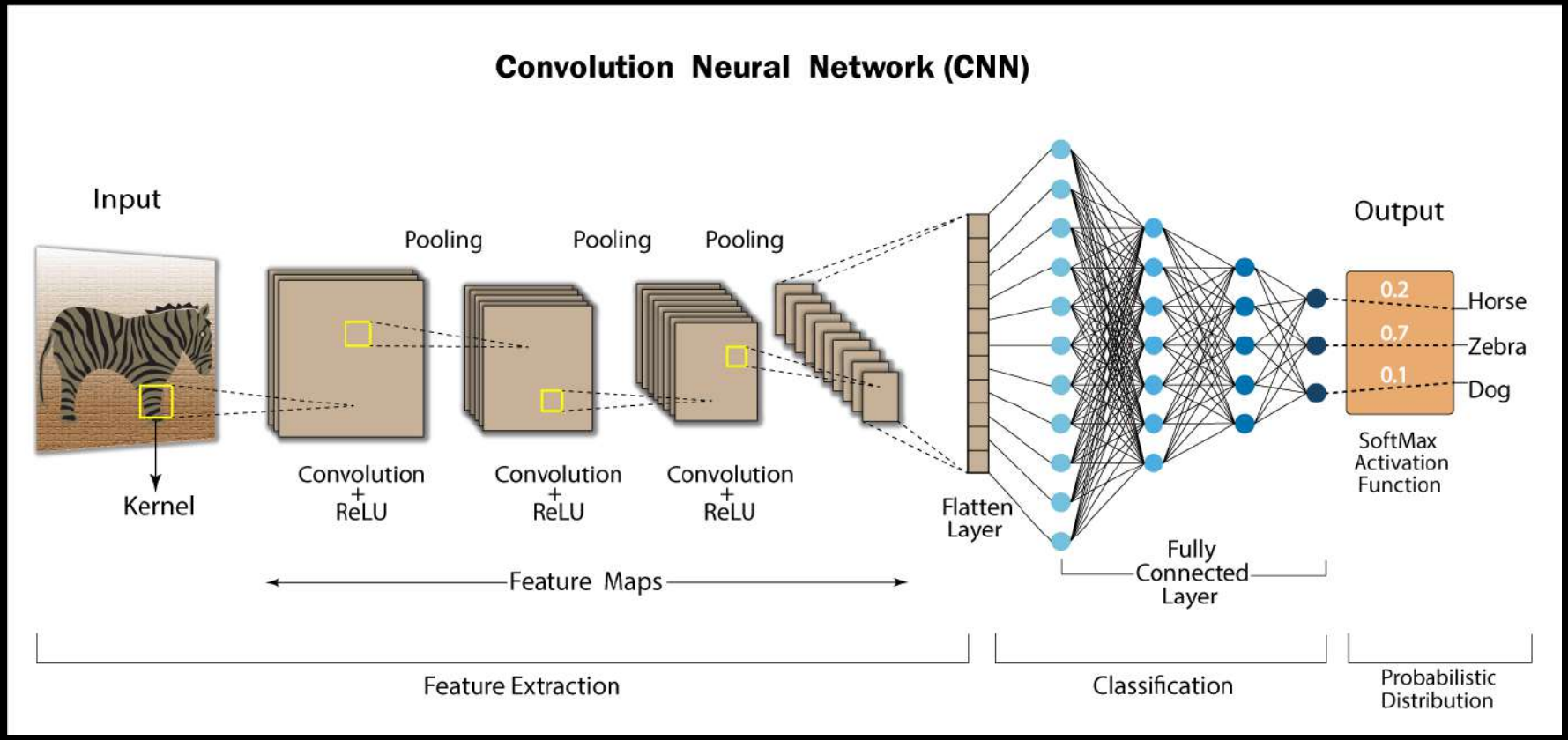

- Lésions vasculaires (noté 'vasc')

6


- Nombre de classes : 7
- Nombre d'images : 10 015
- Inclus : mode d'examen, âge, sexe.

Source : *https://dataverse.harvard.edu/*[9]

Reconnaissance d'image

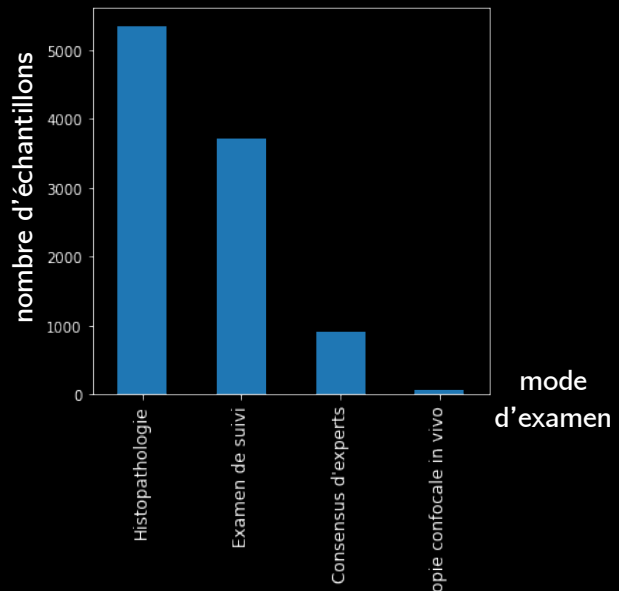


Structure type d'un réseau de classification d'images

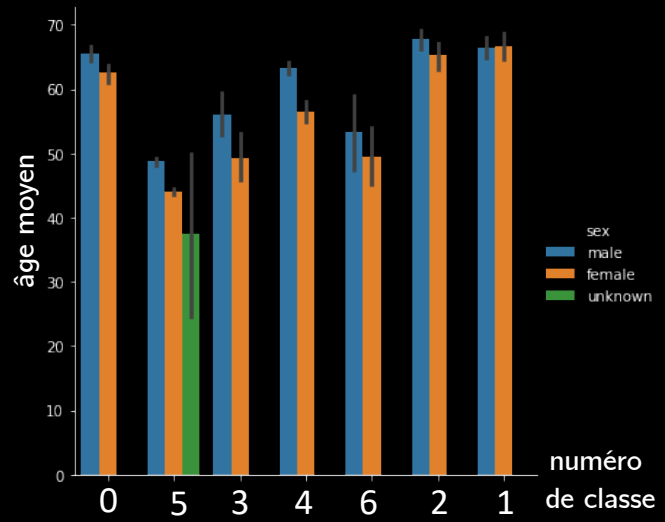
Source : *https://www.analyticsvidhya.com/*

Visualisation de la donnée

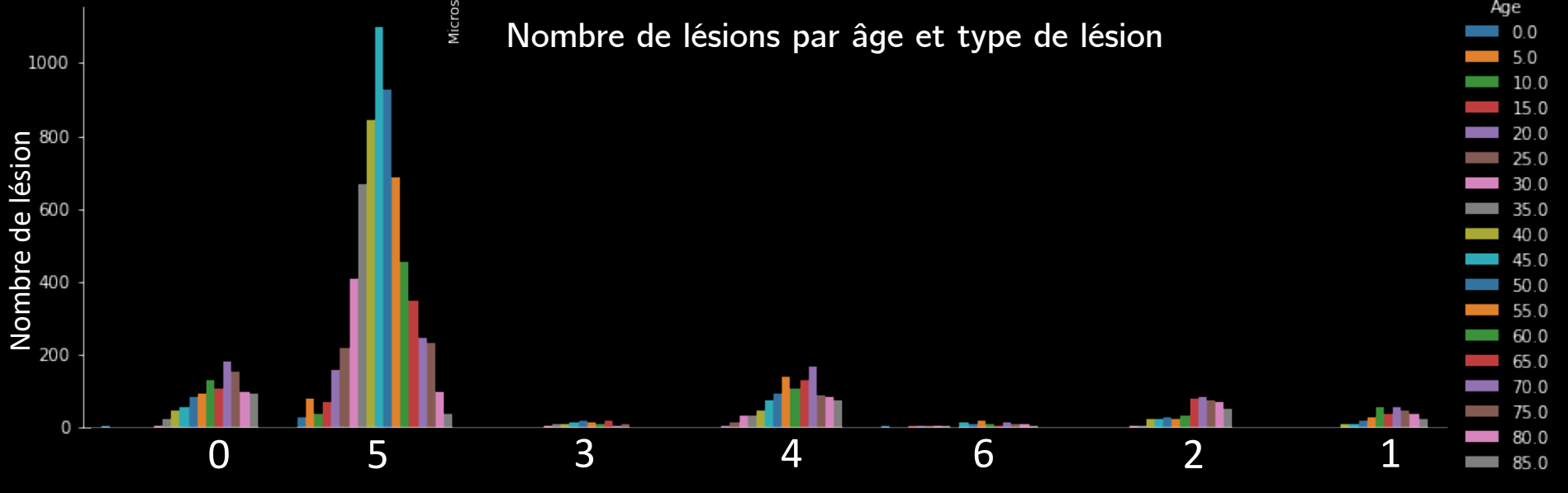
Nombre d'échantillons par mode d'examen



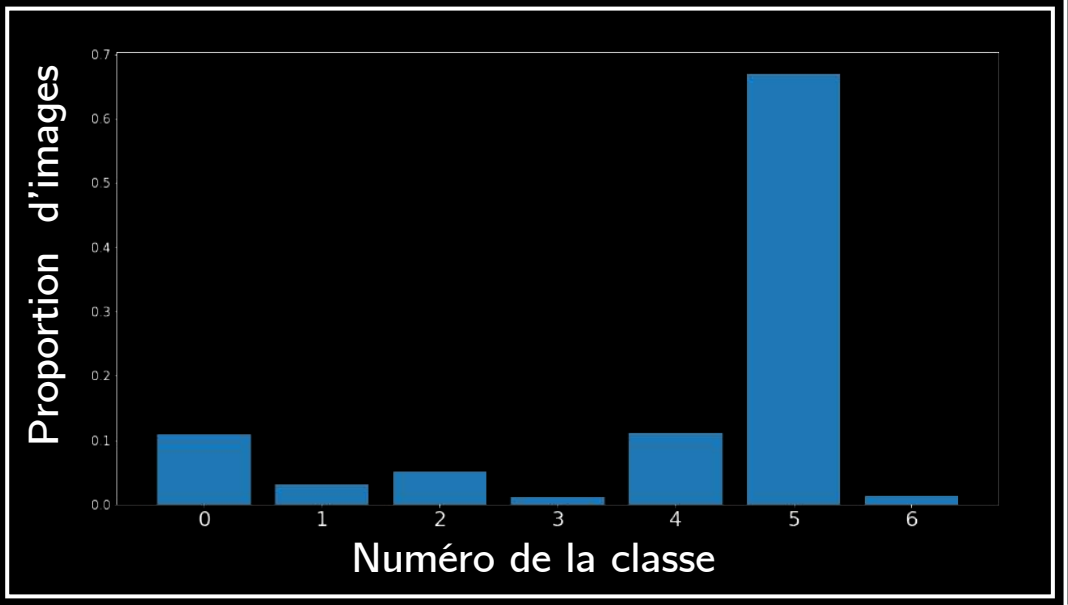
Âge moyen et sexe des malades par classe

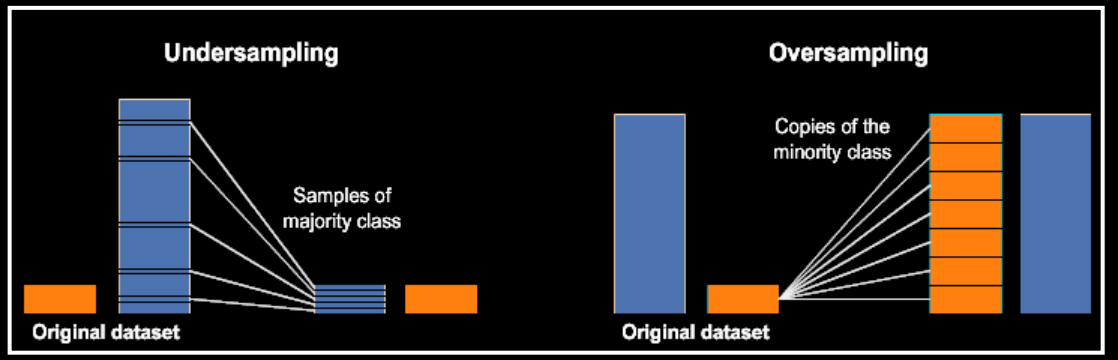


Nombre de lésions par âge et type de lésion



Effets du déséquilibre de classe



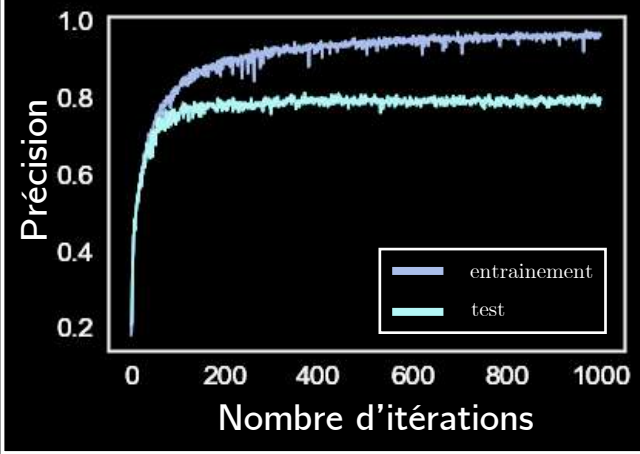


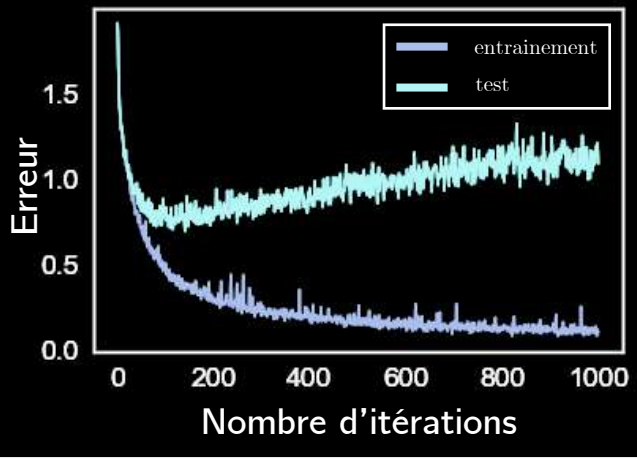
Une solution

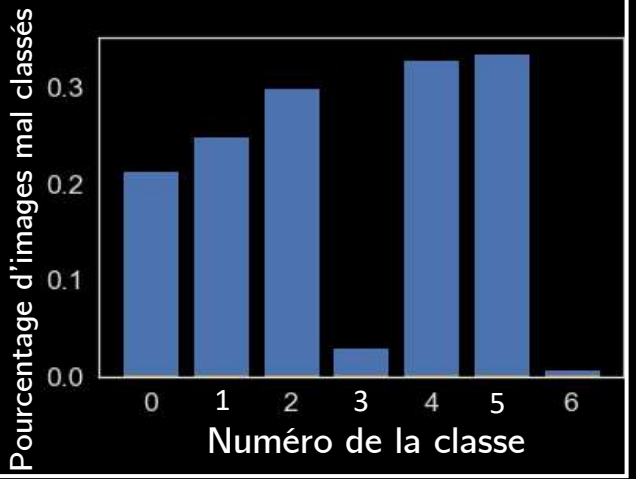
Source : https://medium.com/

Résultats

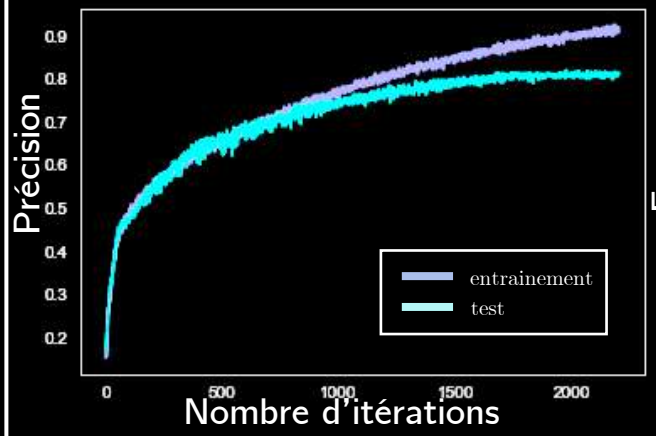
1. Pas : 0.01, réseau : [Conv, Maxpool, Dropout(0.3)]*3 + [Dense, Dense]

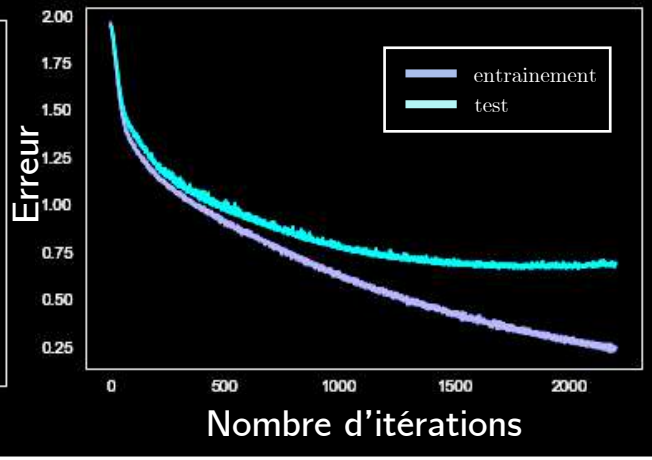


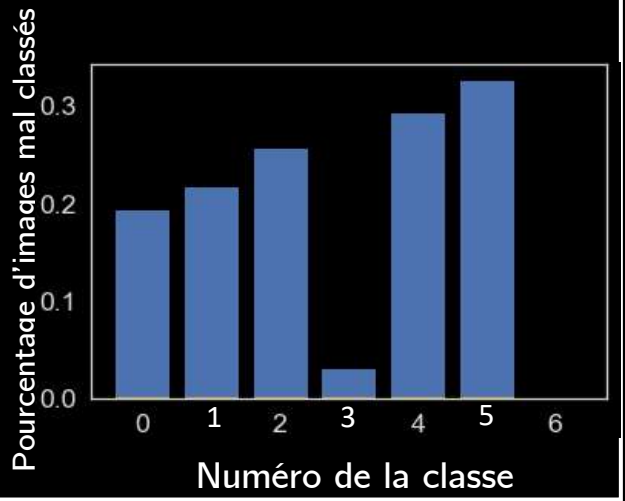




2. Pas : 0.0001, même réseau





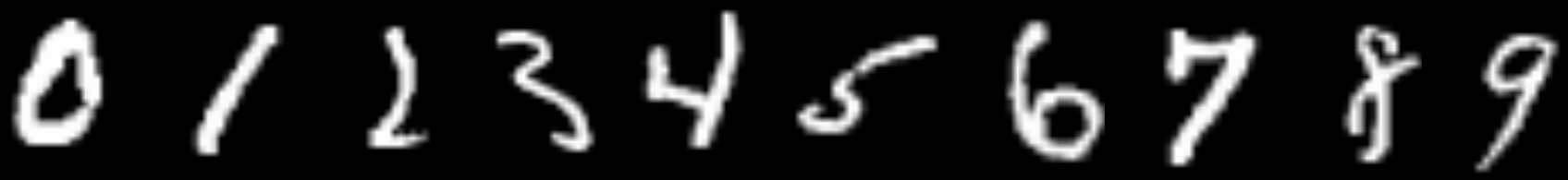


Conclusion : pour répondre à la problématique

- Visualiser les données et équilibrer les classes
- Structure du réseau :
  - Conv + Relu, Maxpooling, Dropout
  - Perceptron multicouche
- Fonctions d'activation : Relu, Softmax
- Fonction d'erreur : Categorical Cross Entropy
- Pas d'apprentissage : 0.01 à 0.0001
- Répétitions : 100 à 10'000 suivant le pas du réseau
- Faire plusieurs modèles et comparer les précisions

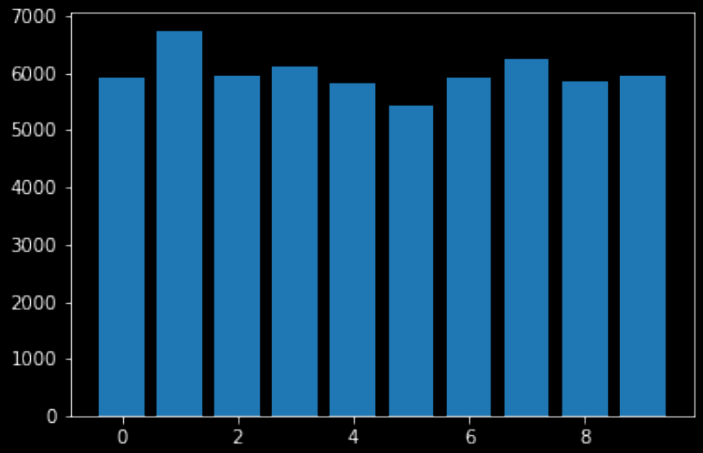
Annexe : Base de donnée MNIST

Quelques exemples de chiffres de la base de donnée

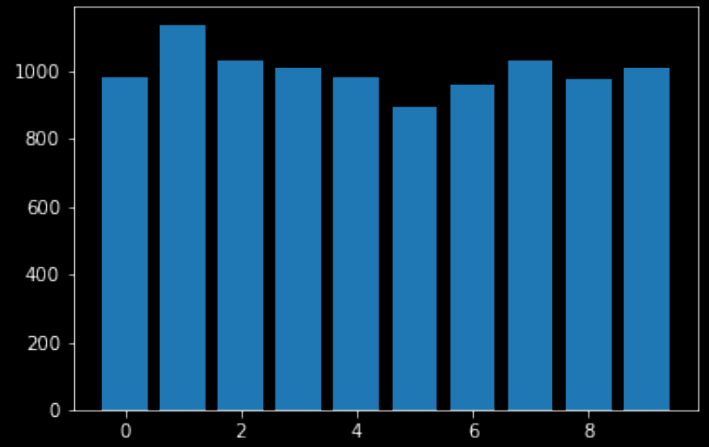


Quelques chiffres

- Nombre de classes : 10
- Nombre d'images de la base d'apprentissage : 60 000
- Nombre d'images de la base de test : 10 000



Nombre d'image représentant chaque lettre dans la base d'apprentissage

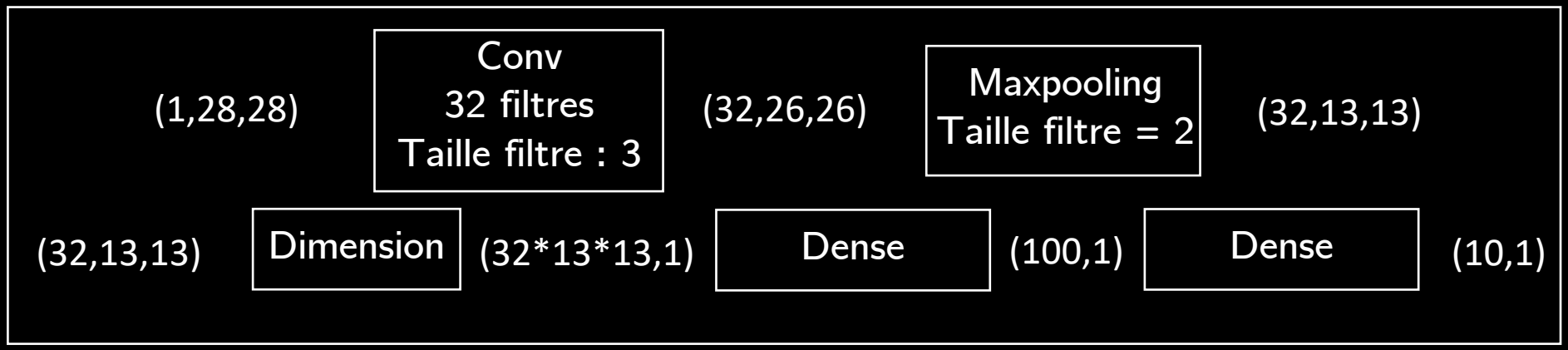


Nombre d'image représentant chaque lettre dans la base de test

Source : *module python Keras (keras.datasets.mnist)*

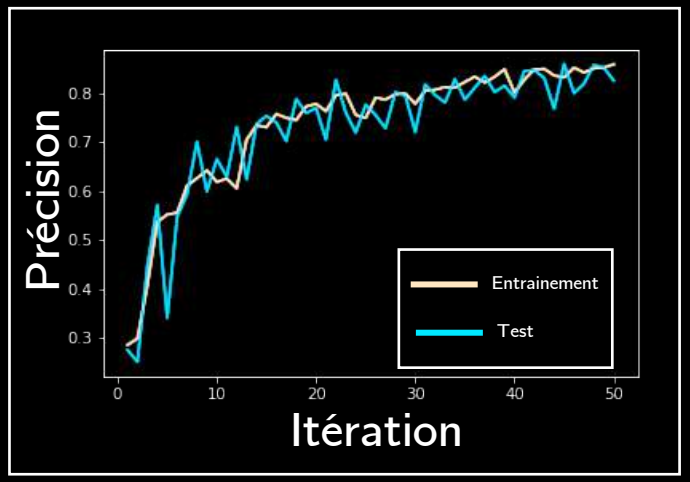
Annexe : Classification MNIST

Réseau convolutif simple :



- Réseau utilisé : Implémenté en Python, 50 répétitions (lent de l'ordre de 1 heure par répétition)
- Fonctions utilisées : Relu comme fonction d'activation pour les couches intermédiaires et Softmax comme dernière couche.
- Pas : 0.01

Précision test finale : 85%



Annexe : couche Dropout, et rétropropagation

Propagation directe : p réel entre 0 et 1

$$Y = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \end{pmatrix} \bullet \begin{pmatrix} 0 & 1/p & 1/p & 1/p \\ 1/p & 1/p & 1/p & 0 \\ 0 & 1/p & 1/p & 1/p \\ 0 & 1/p & 1/p & 0 \end{pmatrix} = p^{-1} \begin{pmatrix} 0 & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & 0 \\ 0 & x_{3,2} & x_{3,3} & x_{3,4} \\ 0 & x_{4,2} & x_{4,3} & 0 \end{pmatrix}$$

$$Y_{i,j} = X_{i,j} \cdot Mi, j \quad \forall i, j$$

Rétropropagation

$$\frac{\partial E}{X_{i,j}} = \frac{\partial E}{Y_{i,j}} \cdot \frac{\partial Y_{i,j}}{X_{i,j}} = M_{i,j} \cdot \frac{\partial E}{Y_{i,j}} \quad \forall (i, j)$$

Annexe : rétropropagation de la couche de convolution

Propagation directe

$$Y_i = \sum_{j=1}^{m} X_j \cdot Ki, j + B_i \qquad \forall i \in \{1, .., n\}$$

$$Y_{i,k,l} = B_{i,k,l} + \sum_{j=1}^{n} X_{j,k+a-1,l+b-1} \cdot K_{i,j,a,b}$$

Rétropropagation

Biais :

$$\frac{\partial E}{\partial B_{i,k,l}} = \frac{\partial E}{\partial Y_{i,k,l}} \cdot \frac{\partial Y_{i,k,l}}{\partial B_{i,k,l}}$$

$$\frac{\partial E}{\partial B_{i,k,l}} = \frac{\partial E}{\partial Y_{i,k,l}}$$

$$\boxed{\frac{\partial E}{\partial B_i} = \frac{\partial E}{\partial Y_i}}$$

Annexe : rétropropagation de la couche de convolution

Propagation directe

$$Y_i = \sum_{j=1}^{m} X_j \cdot Ki, j + B_i \qquad \forall i \in \{1, .., n\}$$

$$Y_{i,k,l} = B_{i,k,l} + \sum_{j=1}^{n} X_{j,k+a-1,l+b-1} \cdot K_{i,j,a,b}$$

Rétropropagation

Poids :

$$\frac{\partial E}{\partial K_{i,j,a,b}} = \sum_{k,l=1}^{s} \frac{\partial E}{\partial Y_{i,k,l}} \frac{\partial Y_{i,k,l}}{K_{i,j,a,b}}$$

$$= \sum_{k,l=1}^{s} X_{j,a+k-1,b+l-1} \cdot \frac{\partial E}{\partial Y_{i,k,l}}$$

$$= (X_j * \frac{\partial E}{\partial Y_i})_{a,b}$$

$$\boxed{\frac{\partial E}{\partial K_{i,j}} = X_j * \frac{\partial E}{\partial Y_i}}$$

Annexe : rétropropagation de la couche de convolution

Propagation directe

$$Y_i = \sum_{j=1}^{m} X_j \cdot Ki, j + B_i \qquad \forall i \in \{1, .., n\}$$

$$Y_{i,k,l} = B_{i,k,l} + \sum_{j=1}^{n} X_{j,k+a-1,l+b-1} \cdot K_{i,j,a,b}$$

Rétropropagation

Entrée : $\frac{\partial E}{\partial X_{j,c,d}} = \sum_{i=1}^{n} \sum_{k,l=1}^{s} \frac{\partial E}{\partial Y_{i,k,l}} \cdot \frac{\partial Y_{i,k,l}}{\partial X_{j,c,d}}$

$$= \sum_{i=1}^{n} \sum_{k,l=1}^{n} K_{i,j,c+1-k,d+1-l} \cdot \frac{\partial E}{\partial Y_{i,k,l}}$$

$$= \sum_{i=1}^{n} ((rot_{180}(\frac{\partial E}{\partial Y_i}) \overset{\text{full}}{*} K_{i,j})_{c,d}$$

$$\boxed{\frac{\partial E}{X_j} = \sum_{i=1}^{n} rot_{180}(\frac{\partial E}{\partial Y_i}) \overset{\text{full}}{*} K_{i,j}}$$

TIPE Yassine Laraki _ Réseaux de neurones convolutifs pour le diagnostic médical

June 7, 2022

[ ]:
```
import numpy as np                                                    # Essentiel␣
↪pour les array, calculs, ...
import matplotlib.pyplot as plt                                       # Pour les␣
↪graphiques/images
from scipy import signal      # Calcul de convolution : le coder soi-même est␣
↪simple mais les boucles for
                              # sont très couteuses et n'optimisent pas les␣
↪calculs (réseau trop lent)

from skimage.measure import block_reduce    # Calcul de maxpooling (même raison␣
↪que convolution, néanmoins pour
                                            # la rétropropagation, il n'y a pas␣
↪de telle fonction donc on utilise
                                            # des boucles etc ce qui rend le␣
↪programme plus lent)
import time                # Chronomètre
import scipy               # Pour la fonction scipy.special.expit qui permet de␣
↪calculer sigmoid sans overflow
```

1 Implémentation du réseau de neurones

1.1 Couches

[ ]:
```
class Dense():

    def __init__(self,taille_entree,taille_sortie):
        self.poids = np.random.randn(taille_sortie, taille_entree)
        self.biais = np.random.randn(taille_sortie,1)

    def propagation_directe(self, entree):
        self.entree = entree
        return (np.dot(self.poids, self.entree) + self.biais)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        grad_poids = np.dot(grad_sortie, self.entree.T)
```

```
        self.poids -= pas_apprentissage * grad_poids
        self.biais -= pas_apprentissage * grad_sortie
        return np.dot(self.poids.T, grad_sortie)
```

[ ]:
```
class Convolution():

    def __init__(self, dimensions_entree, taille_filtre, profondeur_sortie): #␣
↪profondeur_sortie = nombre de filtres
        self.profondeur_sortie = profondeur_sortie
        self.profondeur_entree, self.hauteur_entree, self.largeur_entree =␣
↪dimensions_entree
        self.dimensions_entree = dimensions_entree
        self.dimensions_sortie = (profondeur_sortie, self.hauteur_entree -␣
↪taille_filtre + 1,
                                  self.largeur_entree - taille_filtre + 1)
        self.dimensions_filtres = (profondeur_sortie, self.profondeur_entree,␣
↪taille_filtre, taille_filtre)
        self.filtres = np.random.randn(*self.dimensions_filtres)
        self.biais = np.random.randn(*self.dimensions_sortie)

    def propagation_directe(self, entree):
        self.entree = entree
        self.sortie = np.copy(self.biais)
        for i in range(self.profondeur_sortie) :
            for j in range(self.profondeur_entree):
                self.sortie[i] += signal.correlate2d(self.entree[j], self.
↪filtres[i,j], "valid")
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):
        grad_filtres = np.zeros(self.dimensions_filtres)
        grad_entree = np.zeros(self.dimensions_entree)
        for i in range(self.profondeur_sortie):
            for j in range(self.profondeur_entree):
                grad_filtres[i,j] = signal.correlate2d(self.entree[j],␣
↪grad_sortie[i], "valid")
                grad_entree[j] += signal.correlate2d(grad_sortie[i], self.
↪filtres[i,j], "full")
        grad_biais = grad_sortie

        # Mise à jour
        self.filtres -= pas_apprentissage * grad_filtres
        self.biais -= pas_apprentissage * grad_biais

        return grad_entree
```

[ ]:
```
class Maxpooling():

    def __init__(self, dimensions_entree, taille_filtre, stride):
        self.profondeur_entree, self.hauteur_entree, self.largeur_entree =␣
↪dimensions_entree
        self.dimensions_entree = dimensions_entree
        self.taille_filtre = taille_filtre
        self.filtre_h, self.filtre_l = taille_filtre
        self.stride = stride

        self.hauteur_sortie = int(1 + (self.hauteur_entree - self.filtre_h) /␣
↪stride)
        self.largeur_sortie = int(1 + (self.largeur_entree - self.filtre_l) /␣
↪stride)
        self.profondeur_sortie = self.profondeur_entree

    def propagation_directe(self, entree):

        self.entree = entree
        sortie = np.zeros((self.profondeur_sortie, self.hauteur_sortie, self.
↪largeur_sortie))
        stride = self.stride

        for c in range(self.profondeur_sortie):
            for i in range(self.hauteur_sortie):
                for j in range(self.largeur_sortie):
                    sortie[c, i, j] = np.max(entree[c, i * stride : i * stride␣
↪+ self.filtre_h , j * stride : j * stride + self.filtre_l ])
        return sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):

        grad_entree = np.zeros((self.profondeur_entree, self.hauteur_entree,␣
↪self.largeur_entree))
        entree = self.entree
        stride = self.stride
        for c in range(self.profondeur_sortie):
            for i in range(self.hauteur_sortie):
                for j in range(self.largeur_sortie):
                    intermediaire = entree[c, i * stride : i * stride + self.
↪filtre_h , j * stride : j * stride + self.filtre_l]
                    i_max, j_max = np.where(np.max(intermediaire) ==␣
↪intermediaire)
                    i_max, j_max = i_max[0], j_max[0]
                    grad_entree[c, i * stride : i * stride + self.filtre_h, j *␣
↪stride : j * stride + self.filtre_l][i_max, j_max] = grad_sortie[c, i, j]
```

```
        return grad_entree
```

[ ]:
```
class Dropout():

    def __init__(self, q_bernouilli):
        self.p_bernouilli = 1 - q_bernouilli

    def propagation_directe(self, entree):
        self.entree = entree

        self.masque_binaire = np.random.binomial(1, self.p_bernouilli, size =␣
↪entree.shape)/ self.p_bernouilli
        self.sortie = entree * self.masque_binaire
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return grad_sortie * self.masque_binaire
```

[ ]:
```
class Dimension():

    def __init__(self,dimension_entree, dimension_sortie):
        self.dimension_entree = dimension_entree
        self.dimension_sortie = dimension_sortie

    def propagation_directe(self, entree):
        return np.reshape(entree, self.dimension_sortie)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.reshape(grad_sortie, self.dimension_entree)
```

1.2 Couches d'Activation

[ ]:
```
# Tangente hyperbolique
class Tanh():
    def __init__(self):
        tanh = lambda x : np.tanh(x)
        tanh_p = lambda x : 1 - np.tanh(x)**2
        self.activation = tanh
        self.derivee_activation = tanh_p

    def propagation_directe(self, entree):
        self.entree = entree
        return self.activation(self.entree)

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.multiply(grad_sortie, self.derivee_activation(self.entree))
```

[ ]:
```
# Sigmoide

class Sigmoide():

    def __init__(self):

        def sigmoide(x):
            return scipy.special.expit(x)

        def sigmoide_p(sig):
            return sig * (1- sig)

        self.activation = sigmoide
        self.derivee_activation = sigmoide_p

    def propagation_directe(self, entree):
        self.entree = entree
        self.sig = self.activation(self.entree)
        return self.sig

    def retropropagation(self, grad_sortie, pas_apprentissage):
        return np.multiply(grad_sortie, self.derivee_activation(self.sig))
```

[ ]:
```
# Softmax

class Softmax():

    def propagation_directe(self, entree):
        maxi = np.max(entree)
        entree = entree - maxi
        expo = np.exp(entree)
        self.sortie = expo/np.sum(expo)
```

```
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage): # L'erreur␣
↪utilisée étant toujours cce
        # en complément de softmax, et comme on a déjà calculé la dérivée de␣
↪l'erreur par rapport à
        # l'entrée du softmax, on la transmet (cf cce)

        return grad_sortie
```

[ ]:
```
# ReLU

class Relu():

    def propagation_directe(self, entree):
        self.entree = entree
        self.sortie = np.maximum(0, entree)
        return self.sortie

    def retropropagation(self, grad_sortie, pas_apprentissage):
        inter = grad_sortie.copy()
        inter[inter <= 0] = 0
        return inter
```

1.3 Erreur

[ ]:
```
# Erreur quadratique moyenne (eqm)
```

[ ]:
```
def eqm(sortie_voulue, sortie):
    return np.mean(np.power(sortie_voulue - sortie, 2))

def eqm_derivee(sortie_voulue, sortie):
    return 2 * (sortie - sortie_voulue) / np.size(sortie)
```

[ ]:
```
# Erreur croisée binaire (binary crossentropy notée bce)

def bce(sortie_voulue, sortie):
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7) # Pour ne pas avoir␣
↪de division par zero/ log(0)
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7) # Pour ne pas avoir de division␣
↪par zero/ log(0)
    return -np.mean(sortie_voulue * np.log(sortie) + (1 - sortie_voulue) * np.
↪log(1-sortie))

def bce_derivee(sortie_voulue, sortie):
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
```

```
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    return ((1 - sortie_voulue) / (1 - sortie) - sortie_voulue/ (sortie)) / np.
↪size(sortie)
```

[ ]:
```
# Erreur croisée (categorical crossentropy noté cce)

def cce(sortie_voulue, sortie):
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
    return -np.sum(np.log(sortie) * sortie_voulue)

def cce_derivee(sortie_voulue, sortie): # Ici, pour un soucis de rapidité de␣
↪calcul, on calcule directe
                                        # la dérivée de l'erreur par rapport à␣
↪l'entrée du softmax en utilisant
                                        # cce comme erreur à la dernière couche.
↪ En effet, la formule est bien
                                        # plus simple comme ceci, et on␣
↪utilisera toujours Softmax comme fonction
                                        # d'activation en dernière couche avec␣
↪cce (cf. Softmax)
    sortie_voulue = np.clip(sortie_voulue, 1e-7, 1 - 1e-7)
    sortie = np.clip(sortie, 1e-7, 1 - 1e-7)
    return sortie  - sortie_voulue
```

1.4 Réseau de neurones

[ ]:
```
def precision_erreur(res, entree_t, sortie_t):

    succes = 0
    total = 0
    e = 0
    for i in range(len(entree_t)):
        s = res.prediction(entree_t[i])
        e += res.erreur(sortie_t[i],s)
        maxi = np.argmax(s)
        if maxi == np.argmax(sortie_t[i]):
            succes += 1
        total += 1

    return (succes/total,e/total)
```

[ ]:
```
class Reseau():

    def __init__(self, couches, erreur, erreur_derivee):
        self.couches = couches
```

```
        self.erreur = erreur
        self.erreur_derivee = erreur_derivee

    def prediction(self, entree):
        sortie = entree
        for couche in self.couches:
            sortie = couche.propagation_directe(sortie)
        return sortie

    def entrainement(self,entree_e , sortie_e , entree_t, sortie_t, iterations,␣
↪pas_apprentissage): #entree_e et sortie_e : entrée et sortie entrainement
        nb_entrainements = len(entree_e)
        liste_erreur = []
        liste_erreur_t = []
        precision_e = []
        precision_t = []
        tini = time.time()
        for itera in range(iterations):
            print("Itération numéro ", itera+1)
            erreur = 0
            titer = time.time()
            succes_e = 0
            tot_e = 0
            for i in range(nb_entrainements):
                # Propagation directe
                sortie = entree_e[i]

                for couche in self.couches:
                    sortie = couche.propagation_directe(sortie)
                if np.argmax(sortie) == np.argmax(sortie_e[i]):
                    succes_e += 1
                tot_e += 1
                # Ajout de l'erreur

                erreur += self.erreur(sortie_e[i],sortie)

                # Rétropropagation

                sortie_retro = self.erreur_derivee(sortie_e[i], sortie)

                for couche in reversed(self.couches):
                    sortie_retro = couche.retropropagation(sortie_retro,␣
↪pas_apprentissage)
                # Traitement de l'erreur

            erreur /= nb_entrainements
```

```
            liste_erreur.append(erreur)
            print("Erreur : ", erreur)

            prec_test, err_test = precision_erreur(self, entree_t, sortie_t)
            liste_erreur_t.append(err_test)
            print("Erreur sur la base de test :", liste_erreur_t[-1])

            precision_e.append(succes_e/tot_e)
            precision_t.append(prec_test)
            print("Précision sur la base entrainement :", precision_e[-1])
            print("Précision sur la base test :", precision_t[-1])
            print("Durée de l'itération :", round(time.time() - titer,2) , "s")
            print()
        print()
        print("Fin de l'apprentissage")
        print("Durée de l'apprentissage : ", round(time.time() - tini,2) , "s")
        return (liste_erreur, liste_erreur_t, precision_e, precision_t)

    def test(self, entree_t, sortie_t):
        nb_entrainements = len(entree_t)
        erreur = 0
        for i in range(nb_entrainements):

            # Propagation directe
            sortie = entree_t[i]
            for couche in self.couches:
                sortie = couche.propagation_directe(sortie)
            # Ajout de l'erreur

            erreur += self.erreur(sortie_t[i],sortie)
        erreur /= nb_entrainements
        return erreur
```

Voici trois applications de ce code ci-dessous :

2 XOR (ou exclusif)

[ ]:
```
# Données (entree_e et sortie_e : liste des entrées/sorties voulues)

entree_e = [np.reshape([0,0],(2,1)), np.reshape([0,1],(2,1)), np.
↪reshape([1,0],(2,1)), np.reshape([1,1],(2,1))]
sortie_e = [np.array([0]),np.array([1]),np.array([1]),np.array([0])]


# Structure du réseau (Reseau(couches, erreur, erreur_derivee))
```

```
couches = [Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1), Tanh()]
reseau1 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau2 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau3 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)

# Entrainement (reseau.entrainement(entree_e,sortie_e, iterations,␣
↪pas_apprentissage))

err_pas1, _, _,_,_ = reseau1.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.01)
err_pas2, _, _,_,_ = reseau2.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
err_pas3, _, _,_,_ = reseau3.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 1)
```

[ ]:
```
x = [x for x in range(1,101)]
plt.plot(x, err_pas1)
plt.plot(x, err_pas2)
plt.plot(x, err_pas3)
plt.legend(["0.01","0.1", "1"])
plt.savefig("XOR : pas d'apprentissage 2")
plt.show()
```

[ ]:
```
couches = [Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1), Tanh()]

reseau_al1 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau_al2 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau_al3 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau_al4 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)
reseau_al5 = Reseau([Dense(2,5), Tanh(), Dense(5,3), Tanh(),Dense(3,1),␣
↪Tanh()],eqm,eqm_derivee)

err_alea1, _, _, _ = reseau_al1.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
err_alea2, _, _, _ = reseau_al2.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
err_alea3, _, _, _ = reseau_al3.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
```

```
err_alea4, _, _, _ = reseau_al4.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
err_alea5, _, _, _ = reseau_al5.entrainement(entree_e, sortie_e, entree_e,␣
↪sortie_e, 100, 0.1)
```

[ ]:
```
x = [x for x in range(1,101)]
plt.plot(x, err_alea1)
plt.plot(x, err_alea2)
plt.plot(x, err_alea3)
plt.plot(x, err_alea4)
plt.plot(x, err_alea5)
plt.savefig("XOR : initialisation des poids et biais 2")
plt.show()
```

3 MNIST (classification de chiffres écrits à la main)

3.1 Pour que cela soit plus maniable, dans un premier temps, commencons par classer deux chiffres quelconques

Pour ce faire, nous allons importer un module de Keras, qui ne sert qu'à importer les données du MNIST

[ ]:
```
from keras.datasets import mnist  # Données du MNIST
```

[ ]:
```
# Chargement de la donnée

(entree_e, sortie_e), (entree_t, sortie_t) = mnist.load_data()  # entree_t et␣
↪sortie_t : Entrée et sortie test

# Quelques données

print(len(entree_e))
print(len(entree_t))
compte_e = [0]*10
compte_t = [0]*10
abcs = [x for x in range(10)]

for x in sortie_e:
    for i in range(10):
        if x == i:
            compte_e[i] += 1

for x in sortie_t:
    for i in range(10):
        if x == i:
            compte_t[i] += 1
# Traitement de la donnée
```

```
def nettoyage_mnist(chiffre1, chiffre2, entree, sortie, limite):

    indices_1 = np.where(sortie == chiffre1)[0][:limite] # Indices du chiffre␣
↪1, avec au maximum limite indices
    indices_2 = np.where(sortie == chiffre2)[0][:limite]
    indices = np.concatenate((indices_1, indices_2))
    indices = np.random.permutation(indices)

    x = entree[indices]
    y = sortie[indices]
    x = x.reshape(len(x), 1, 28, 28) # Les images ont pour format (28,28) mais␣
↪notre réseau prend en entrée une image
                                     # de format (profondeur, hauteur, largeur)␣
↪à 3 dimensions
    x = x.astype("float32")/255      # x est de type uint8 et contient des␣
↪entiers de 0 à 255

    vecteur1 = np.reshape(np.array([1,0]), (2,1))
    vecteur2 = np.reshape(np.array([0,1]), (2,1))

    l = []
    for i in y:
        if i == chiffre1:
            l.append(vecteur1)
        else:
            l.append(vecteur2)

    y = np.array(l)
    return (x,y)

(entree_e, sortie_e) = nettoyage_mnist(0,1,entree_e,sortie_e,1000)
(entree_t, sortie_t) = nettoyage_mnist(0,1,entree_t,sortie_t,-1)

print(entree_e.shape)
print(sortie_e.shape)
print(entree_t.shape)
print(sortie_t.shape)
```

[ ]:
```
plt.bar(abcs, compte_e)
plt.savefig('mnist_compte_e', transparent = True)
plt.show()
plt.bar(abcs, compte_t)
plt.savefig('mnist_compte_t', transparent = True)
plt.show()
```

[ ]:
```
# Réseau de neurones
# Rappelons le format des couches utilisées:
"""Dense(taille_entree,taille_sortie)
    Convolution(dimensions_entree, taille_filtre, profondeur_sortie
    Dimension(dimension_entree, dimension_sortie)"""


couches = [
    Convolution((1,28,28),3,5),
    Tanh(),
    Dimension((5,26,26),(5*26*26,1)),
    Dense(5*26*26,100),
    Tanh(),
    Dense(100,2),
    Sigmoide()
]

reseau = Reseau(couches, bce, bce_derivee)
(erreur_e, erreur_t, precision_e, precision_t) = reseau.entrainement(entree_e ,␣
 ↪sortie_e , entree_t, sortie_t, 10, 0.1)
```

[ ]:
```
abcs = [x for x in range(1,11)]
plt.figure()
plt.plot(abcs, erreur_e)
plt.plot(abcs, erreur_t)
plt.savefig("Mnist 2 chiffres : erreurr", transparent = True)
plt.show()

plt.figure()
plt.plot(abcs, precision_e)
plt.plot(abcs, precision_t)
plt.savefig("Mnist 2 chiffres : précisionn")
plt.show()
```

3.2 Version complète avec les 10 chiffres

[ ]:
```
# Chargement de la donnée

(entree_e, sortie_e), (entree_t, sortie_t) = mnist.load_data()  # entree_t et␣
 ↪sortie_t : Entrée et sortie test

# Traitement de la donnée

def nettoyage_mnist(entree, sortie, limite): # limite = nombre d'exemples au␣
 ↪maximum par chiffre
```

```
    indices = []
    for i in range(10):
        indices.extend(np.where(sortie == i)[0][:limite])

    indices = np.random.permutation(indices)

    x = entree[indices]
    y = sortie[indices]
    x = x.reshape(len(x), 1, 28, 28) # Les images ont pour format (28,28) mais␣
↪notre réseau prend en entrée une image
                                     # de format (profondeur, hauteur, largeur)␣
↪à 3 dimensions
    x = x.astype("float32")/255      # x est de type uint8 et contient des␣
↪entiers de 0 à 255

    identite = np.eye(10) # Matrice identité de taille 10

    l = []

    for i in y:
        l.append(identite[i])

    y = np.reshape(np.array(l),(len(y),10,1))
    return (x,y)

(entree_e, sortie_e) = nettoyage_mnist(entree_e,sortie_e,1000)
(entree_t, sortie_t) = nettoyage_mnist(entree_t,sortie_t,-1)

print(entree_e.shape)
print(entree_t.shape)
```

[ ]:
```
# Réseau de neurones
# Rappelons le format des couches utilisées:
"""Dense(taille_entree,taille_sortie)
    Convolution(dimensions_entree, taille_filtre, profondeur_sortie
    Dimension(dimension_entree, dimension_sortie)"""


couches = [
    Convolution((1,28,28),3,32),
    Relu(),
    Maxpooling((32,26,26),(2,2),2),
    Dimension((32,13,13),(32*13*13,1)),
    Dense(32*13*13,100),
    Relu(),
    Dense(100,10),
```

```
    Softmax()
         ]

reseau = Reseau(couches, cce, cce_derivee)

erreur, erreur_t, precision_e, precision_t = reseau.entrainement(entree_e ,␣
↪sortie_e , entree_t, sortie_t, 20, 0.1)
```

[ ]:
```
abcs = [x for x in range(1,51)]
plt.figure()
plt.plot(abcs, erreur_e)
plt.plot(abcs, erreur_t)
plt.savefig('/Users/yassinelaraki/Desktop/TIPE/Mnist 10 chiffres : erreur')
plt.show()

plt.figure()
plt.plot(abcs, precision_e)
plt.plot(abcs, precision_t)
plt.savefig('/Users/yassinelaraki/Desktop/TIPE/Mnist 10 chiffres : précision')
plt.show()
```

4 HAM10000 (skin lesions)

[ ]:
```
import pandas as pd
from PIL import Image
```

[ ]:
```
ham = pd.read_csv("/Users/yassinelaraki/Desktop/TIPE RE/dataverse_files/
↪HAM10000_metadata")
ham
```

4.1 Exploration de la base de donnée

[ ]:
```
from collections import Counter

types = ['bkl', 'akiec', 'bcc', 'df', 'mel', 'nv', 'vasc']
types_explicite = ['Lésions bénignes comme la kératose',
                   'Kératose actinique/ carcinome intraépithélial ou maladie de␣
↪Bowen',
                    'Carcinome basocellulaire',
                    'Dermatofibrome',
                    'Mélanome',
                    'Naevus mélanocytaire',
                    'Lésions vasculaires'
                   ]
types_reduits = ['Lésions bénignes',
```

```
                  'Kératose/ carcinome/ Bowen',
                   'Carcinome basocellulaire',
                   'Dermatofibrome',
                   'Mélanome',
                   'Naevus mélanocytaire',
                   'Lésions vasculaires'

]
indices = []

for x in types:
    indices.append(np.where(ham['dx'] == x)[0])

compte = []
for x in indices:
    compte.append((len(x))/10015)

plt.figure(figsize = (19,10))
plt.xticks(fontsize=25)
plt.yticks(fontsize=15)

plt.bar([x for x in range(7)], compte)

plt.savefig('Répartition des lésions dans la base de donnnée pourcentage',␣
 ↪transparent = True)
```

[ ]:
```
plt.figure(figsize = (5,5))

loca_lesions = []
for x in ham['dx_type']:
    if x == 'histo':
        loca_lesions.append('Histopathologie')
    elif x == 'follow_up':
        loca_lesions.append('Examen de suivi')
    elif x == 'consensus':
        loca_lesions.append("Consensus d'experts")
    elif x == 'confocal':
        loca_lesions.append("Microscopie confocale in vivo")


loca_lesions = pd.Series(loca_lesions)
plt.xticks(fontsize = 12)
loca_lesions.value_counts().plot(kind='bar')
plt.savefig('Type de détermination du type des lésions', transparent = True,␣
 ↪bbox_inches="tight")
```

```
[ ]: plt.figure(figsize = (10,5))
     plt.title("Localisation de la lésion", fontsize = 17)
     ham['localization'].value_counts().plot(kind='bar')
```

```
[ ]: plt.figure(figsize = (10,5))
     ham['age'].hist(bins=50)

     plt.savefig('Nombre de lésions par âge', transparent = True,␣
      ↪bbox_inches="tight")
```

```
[ ]: ham['sex'].value_counts().plot(kind='bar')
     plt.savefig('Lésions par sexe', transparent = True, bbox_inches="tight")
```

```
[ ]: import seaborn as sns

     sns.catplot(x= "dx", y="age", hue="sex", kind="bar", data=ham)
     plt.savefig('Lésions par âge et sexe', transparent = True, bbox_inches="tight")
```

```
[ ]: g = sns.catplot(x="dx", kind="count", hue="age", palette='tab10', data=ham,␣
      ↪height = 4, aspect = 3)

     plt.setp(g._legend.get_texts(), fontsize=10)
     g.set_xlabels('Type de lésion', fontsize=10)
     g.set_ylabels("Nombre d'occurences", fontsize=10)
     g._legend.set_title('Âge')
     plt.savefig('Occurence du nombre de lésions par sexe et type de lésion',␣
      ↪transparent = True, bbox_inches="tight")
```

```
[ ]: g = sns.catplot(x= "localization", kind = "count", hue="dx", data=ham,␣
      ↪palette="tab10", height = 5, aspect = 3)


     g.set_xlabels('Localisation', fontsize=10)
     g.set_ylabels("Nombre d'occurence", fontsize=10)
     g._legend.set_title('Type de lésions')
```

4.2 Importation des images et implémentation du réseau

Malheureusement, l'utilisation des techniques précédentes est trop lent pour aboutir avec ce jeu de donnée. En effet, une itération prend à peu près 1 jour. C'est pourquoi nous allons faire le modèle sur Keras, en n'utilisant que des couches, fonctions d'erreurs et structures déjà codées auparavant. Le résultat est alors le même, mais beaucoup plus rapide, ce qui permet de faire plus d'itéations, et de comparer différents modèles.

Néanmoins, l'importation des données diffère sur 2 points entre Keras et le réseau codé auparavant. En effet, Keras prend des images de la forme (hauteur, largeur, profondeur) au lieu de (profondeur, hauteur, largeur). L'importation d'une image sur python se faisant dans le format (hauteur, largeur, profondeur), il faut appliquer une fonction convertir (ci-dessous) à chaque image de la base de

donnée pour le mettre en entrée du réseau codé auparavant. De plus, la sortie est au format (nombre_de_classe,) sur Keras au lieu (nombre_de_classe,1). Il suffit alors, au début du code, d'écrire identite = np.reshape(identite, (7,7,1)), à la suite de sa définition, si l'on souhaite utiliser cette base de donnée au réseau de neurone codé précedemment.

Tout le reste étant similaire, implémentons le réseau sur Keras

4.2.1 Une première implémentation naïve : déséquilibre de classe

[ ]:
```
# Fonction utile

def convertir(im): # Convertir (x,y,3) en (3,x,y)
    a = im[:,:,0]
    b = im[:,:,1]
    c = im[:,:,2]
    res = np.array([a,b,c])
    return res
```

[ ]:
```
# Traitement de la donnée

types = ['bkl', 'akiec', 'bcc', 'df', 'mel', 'nv', 'vasc']

# Matrice identité 7 (utile pour la sortie sous forme [1,0,0,0,0,0,0],[0,1,...]␣
 ↪etc)
identite = np.eye(7)


# Indices pour chaque type de lésion

indices = []

for x in types:
    indices.append(np.where(ham['dx'] == x)[0].tolist())

# Séparation de la base de donnée en test et entrainement

indices_test = []
indices_entrainement = []

for ind in indices:
    k = 0
    seuil = seuil = 4* len(ind)/5
    indices_test.append([])
    indices_entrainement.append([])
    for i in ind:
        if k <= seuil:
            indices_entrainement[-1].append(i)
            k += 1
```

```
        else:
            indices_test[-1].append(i)
            k += 1

# On applatit les listes, et on mélange aléatoirement les indices

def applatir(li):
    res = []
    for x in li:
        res.extend(x)
    return res

indices_test = np.random.permutation(applatir(indices_test))
indices_entrainement = np.random.permutation(applatir(indices_entrainement))

# Enfin, on crée les listes entree_e, sortie_e, ...

entree_e, sortie_e, entree_t, sortie_t = [],[],[],[]
k = 0
for i in indices_test:
    x= "/Users/yassinelaraki/Desktop/TIPE RE/dataverse_files/HAM10000/" +␣
 ↪ham['image_id'][i] + '.jpg'
    entree_t.append(np.asarray(Image.open(x).resize((64,64)))/255.)
    sortie_t.append(ham['dx'][i])

for i in indices_entrainement:
    x= "/Users/yassinelaraki/Desktop/TIPE RE/dataverse_files/HAM10000/" +␣
 ↪ham['image_id'][i] + '.jpg'
    entree_e.append(np.asarray(Image.open(x).resize((64,64)))/255.)
    sortie_e.append(ham['dx'][i])

# On convertit les sorties sous formes de chaînes de caractères en␣
 ↪[1,0,0,0,0,0,0] ou [0,1,...] etc
s_e, s_t = [],[]

for x in sortie_e:
    for i in range(7):
        if x == types[i]:
            s_e.append(identite[i])

for x in sortie_t:
    for i in range(7):
        if x == types[i]:
            s_t.append(identite[i])

# On convertit les listes en array
```

```
sortie_e = np.asarray(s_e)
entree_e = np.asarray(entree_e)

sortie_t = np.asarray(s_t)
entree_t = np.asarray(entree_t)
```

[ ]:
```
# Importations de fonctions et couches déjà codées, grâce à Keras
```

[ ]:
```
from tensorflow.keras.models import Sequential # Fonctionnement analogue à␣
↪celui de Reseau
from tensorflow.keras.layers import Conv2D # Fonctionnement analogue à celui de␣
↪Convolution
from tensorflow.keras.layers import MaxPool2D # Fonctionnement analogue à celui␣
↪de Maxpooling
from tensorflow.keras.layers import Dense # Fonctionnement analogue à celui de␣
↪Dense
from tensorflow.keras.layers import Flatten # Fonctionnement analogue à celui␣
↪de Dimension (ou
                                            # de l'usage qui en est fait en␣
↪tout cas)
from tensorflow.keras.layers import Dropout #  Fonctionnement analogue à celui␣
↪de Dropout
from tensorflow.keras.optimizers import SGD # (Rétropropagation)
```

[ ]:
```
modele_naif = Sequential()

modele_naif.add(Conv2D(256, (3,3), activation = "relu", input_shape =␣
↪(64,64,3)))
modele_naif.add(MaxPool2D(pool_size = (2,2)))
modele_naif.add(Dropout(0.3))

modele_naif.add(Conv2D(128, (3,3), activation = "relu"))
modele_naif.add(MaxPool2D(pool_size = (2,2)))
modele_naif.add(Dropout(0.3))

modele_naif.add(Conv2D(64, (3,3), activation = "relu"))
modele_naif.add(MaxPool2D(pool_size = (2,2)))
modele_naif.add(Dropout(0.3))
modele_naif.add(Flatten())

modele_naif.add(Dense(32, activation = "relu"))
modele_naif.add(Dense(7, activation = "softmax"))
modele_naif.summary()
```

[ ]:
```
descente_gradient = SGD(learning_rate=0.01)
```

```
modele_naif.compile(optimizer = descente_gradient , loss =␣
↪"categorical_crossentropy", metrics=["accuracy"])

repetitions = 50
historique = modele_naif.fit(entree_e, sortie_e, epochs = repetitions,
                             validation_data = (entree_t, sortie_t), verbose =␣
↪1 )
```

[ ]:
```
# Pour information, voici le code équivalent sur la classe Réseau (testé mais␣
↪trop lent)

couches = [
    Convolution((3,32,32),3,256),
    Relu(),
    Maxpooling((256,30,30),(2,2),2),
    Relu(),
    Dropout(0.3),

    Convolution((256,15,15),3,128),
    Relu(),
    Maxpooling((128,13,13),(2,2),2),
    Dropout(0.3),

    Dimension((128,6,6),(128*6*6,1)),
    Dense(128*6*6,32),
    Relu(),
    Dense(32,7),
    Softmax()
         ]

reseau_naif_2 = Reseau(couches, cce, cce_derivee)
erreur, precision_e, precision_t = reseau_ham.entrainement(entree_e, sortie_e,
    entree_t, sortie_t, repetitions, 0.01)
```

4.2.2 Une solution : même nombre d'images par classe, en détruisant ou en répliquant des images

[ ]:
```
# Traitement de la donnée

types = ['bkl', 'akiec', 'bcc', 'df', 'mel', 'nv', 'vasc']

# Matrice identité 7 (utile pour la sortie sous forme [1,0,0,0,0,0,0],[0,1,...]␣
↪etc)
identite = np.eye(7)

# Indices pour chaque type de lésion
```

```
indices = []

for x in types:
    indices.append(np.where(ham['dx'] == x)[0].tolist())

# Séparation de la base de donnée en test et entrainement

indices_test = []
indices_entrainement = []

for ind in indices:
    k = 0
    seuil = seuil = 4* len(ind)/5
    indices_test.append([])
    indices_entrainement.append([])
    for i in ind:
        if k <= seuil:
            indices_entrainement[-1].append(i)
            k += 1
        else:
            indices_test[-1].append(i)
            k += 1


# NOUVEAUTE : Compte du nombre de donnée de test et d'entraînement pour chaque␣
 ↪classe

compte_test = []
compte_entrainement = []

for x in indices_test:
    compte_test.append(len(x))

for x in indices_entrainement:
    compte_entrainement.append(len(x))

# NOUVEAUTE : On augmente/diminue le nombre d'indices par classe

courant_t = [0] * 7
courant_e = [0] * 7

for i in range(7):
    while len(indices_test[i]) >= 101:
        (indices_test[i]).pop()
    while len(indices_test[i]) <= 99:
        (indices_test[i]).append(indices_test[i][courant_t[i]])
```

```
        courant_t[i] = (courant_t[i] +1) % compte_test[i]

for i in range(7):
    while len(indices_entrainement[i]) >= 501:
        (indices_entrainement[i]).pop()
    while len(indices_entrainement[i]) <= 499:
        (indices_entrainement[i]).append(indices_entrainement[i][courant_e[i]])
        courant_e[i] = (courant_e[i] +1) % compte_entrainement[i]

# On applatit les listes, et on mélange aléatoirement les indices

def applatir(li):
    res = []
    for x in li:
        res.extend(x)
    return res

indices_test = np.random.permutation(applatir(indices_test))
indices_entrainement = np.random.permutation(applatir(indices_entrainement))

# Enfin, on crée les listes entree_e, sortie_e, ...

entree_e, sortie_e, entree_t, sortie_t = [],[],[],[]
k = 0
for i in indices_test:
    x= "/Users/yassinelaraki/Desktop/TIPE RE/dataverse_files/HAM10000/" +␣
 ↪ham['image_id'][i] + '.jpg'
    entree_t.append(np.asarray(Image.open(x).resize((32,32)))/255.)
    sortie_t.append(ham['dx'][i])

for i in indices_entrainement:
    x= "/Users/yassinelaraki/Desktop/TIPE RE/dataverse_files/HAM10000/" +␣
 ↪ham['image_id'][i] + '.jpg'
    entree_e.append(np.asarray(Image.open(x).resize((32,32)))/255.)
    sortie_e.append(ham['dx'][i])

# On convertit les sorties sous formes de chaînes de caractères en␣
 ↪[1,0,0,0,0,0,0] ou [0,1,...] etc
s_e, s_t = [],[]

for x in sortie_e:
    for i in range(7):
        if x == types[i]:
            s_e.append(identite[i])

for x in sortie_t:
    for i in range(7):
```

```
        if x == types[i]:
            s_t.append(identite[i])

# On convertit les listes en array

sortie_e = np.asarray(s_e)
entree_e = np.asarray(entree_e)

sortie_t = np.asarray(s_t)
entree_t = np.asarray(entree_t)
```

[ ]:
```
modele = Sequential()

modele.add(Conv2D(256, (3,3), activation = "relu", input_shape = (64,64,3)))
modele.add(MaxPool2D(pool_size = (2,2)))
modele.add(Dropout(0.3))

modele.add(Conv2D(128, (3,3), activation = "relu"))
modele.add(MaxPool2D(pool_size = (2,2)))
modele.add(Dropout(0.3))

modele.add(Conv2D(64, (3,3), activation = "relu"))
modele.add(MaxPool2D(pool_size = (2,2)))
modele.add(Dropout(0.3))
modele.add(Flatten())

modele.add(Dense(32, activation = "relu"))
modele.add(Dense(7, activation = "softmax"))
modele.summary()
```

[ ]:
```
descente_gradient = SGD(learning_rate=0.01)
modele_naif.compile(optimizer = descente_gradient , loss =␣
↪"categorical_crossentropy", metrics=["accuracy"])

repetitions = 50
historique = modele_naif.fit(entree_e, sortie_e, epochs = repetitions,
                             validation_data = (entree_t, sortie_t), verbose =␣
↪1)
```

4.2.3 Test du modèle

[ ]:
```
sns.set_style("white")

# Erreur (loss = erreur, val_loss = erreur_test)

erreur = historique.history['loss']
erreur_test = historique.history['val_loss']
```

```
repetitions = range(1, len(erreur) + 1)
plt.plot(repetitions, erreur, 'y', label='Erreur entrainement')
plt.plot(repetitions, erreur_test, 'r', label='Erreur test')
plt.xlabel('Répétitions')
plt.ylabel('Erreur')
plt.savefig('Erreur du HAM10000 essai 2')
plt.legend()
plt.show()

# Précision

precision = historique.history['acc']
precision_test = historique.history['val_acc']
plt.plot(repetitions, precision, 'y', label='Précision entrainement')
plt.plot(repetitions, precision_test, 'r', label='Précision test')
plt.xlabel('Répétitions')
plt.ylabel('Précision')
plt.savefig('Précision du HAM10000 essai 2')
plt.legend()
plt.show()

# Prédictions faites sur chaque échantillon
prediction_test = model.predict(entree_t)
classes_test = np.argmax(prediction_test, axis = 1)
vraies = np.argmax(sortie_t, axis = 1)

# Matrice de confusion
cm = confusion_matrix(vraies, classes_test)

fig, ax = plt.subplots(figsize=(6,6))
sns.set(font_scale=1.6)
sns.heatmap(cm, annot=True, linewidths=.5, ax=ax)
plt.savefig('Matrice de confusion essai 2')
plt.show()

sns.set_style("white")

# Pourcentage de prédictions erronées
faux_pourcentage = 1 - np.diag(cm) / np.sum(cm, axis=1)
plt.bar(np.arange(7), faux_pourcentage)

plt.xlabel('Vraie classe')
plt.ylabel('Pourcentage de prédictions incorrectes')
plt.savefig('Graph precisions du HAM essai 2')
plt.show()
```