



# Reconnaissance d'image

*Travaux d'initiative personnelle encadrés*

2021-2022

CADET Hugo

# Table des matières

- 1) Preamble
- 2) Aspect théorique
- 3) Mon experience
- 4 ) Annexe

**Problématique :** Comment créer un algorithme permettant à un robot de se repérer dans un espace aquatique en cartographiant l'espace autour dans le but de détecter et reconnaître les déchets qui l'entourent ?

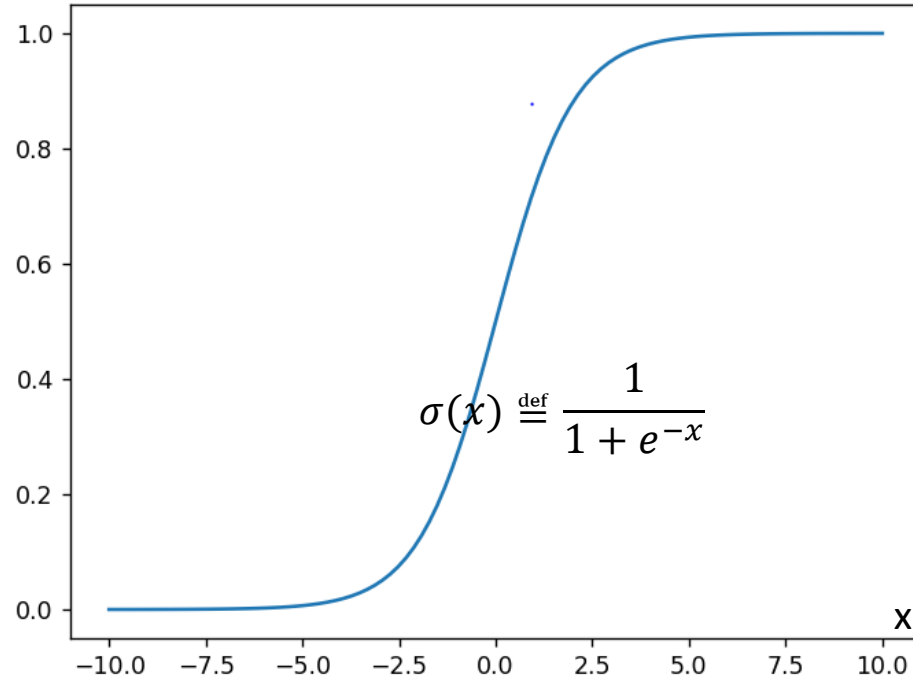
**Objectif du TIPE:** A partir de la lecture du flux camera de son environnement, élaborer un programme afin que le robot soit capable de prédire si l'objet qui se trouve devant lui est un déchet ou non.

Préambule :

**Objectif du RN :** *Le but principal d'un Réseau de Neurones est d'apprendre, de s'entraîner à résoudre une tâche. Dans le cas présent, de prédire correctement le label d'une image.*

Fonction Sigmoid :

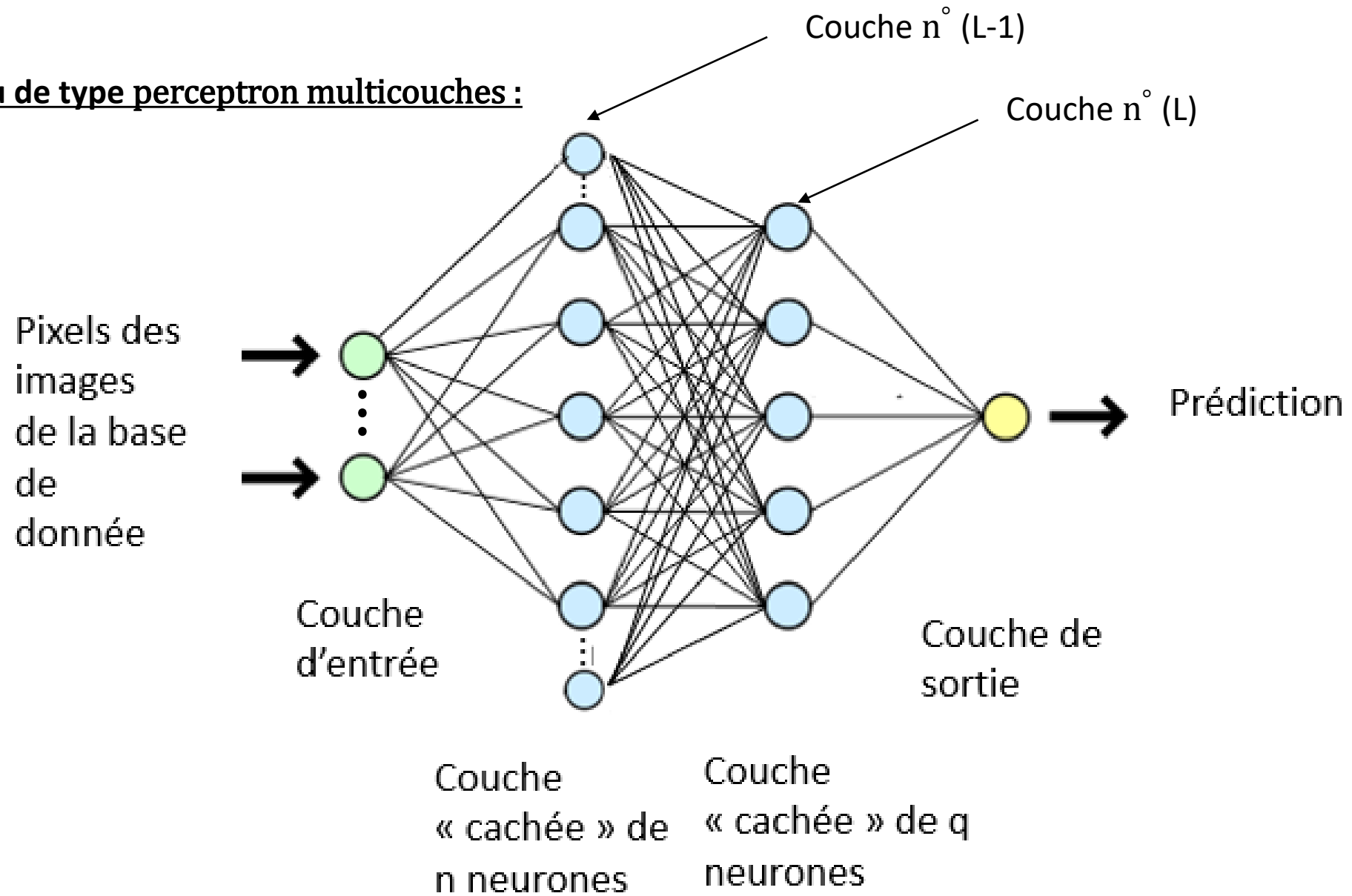
$$\sigma(x) = \begin{cases} 1 & \text{si } x \rightarrow +\infty \\ 0 & \text{si } x \rightarrow -\infty \end{cases}$$



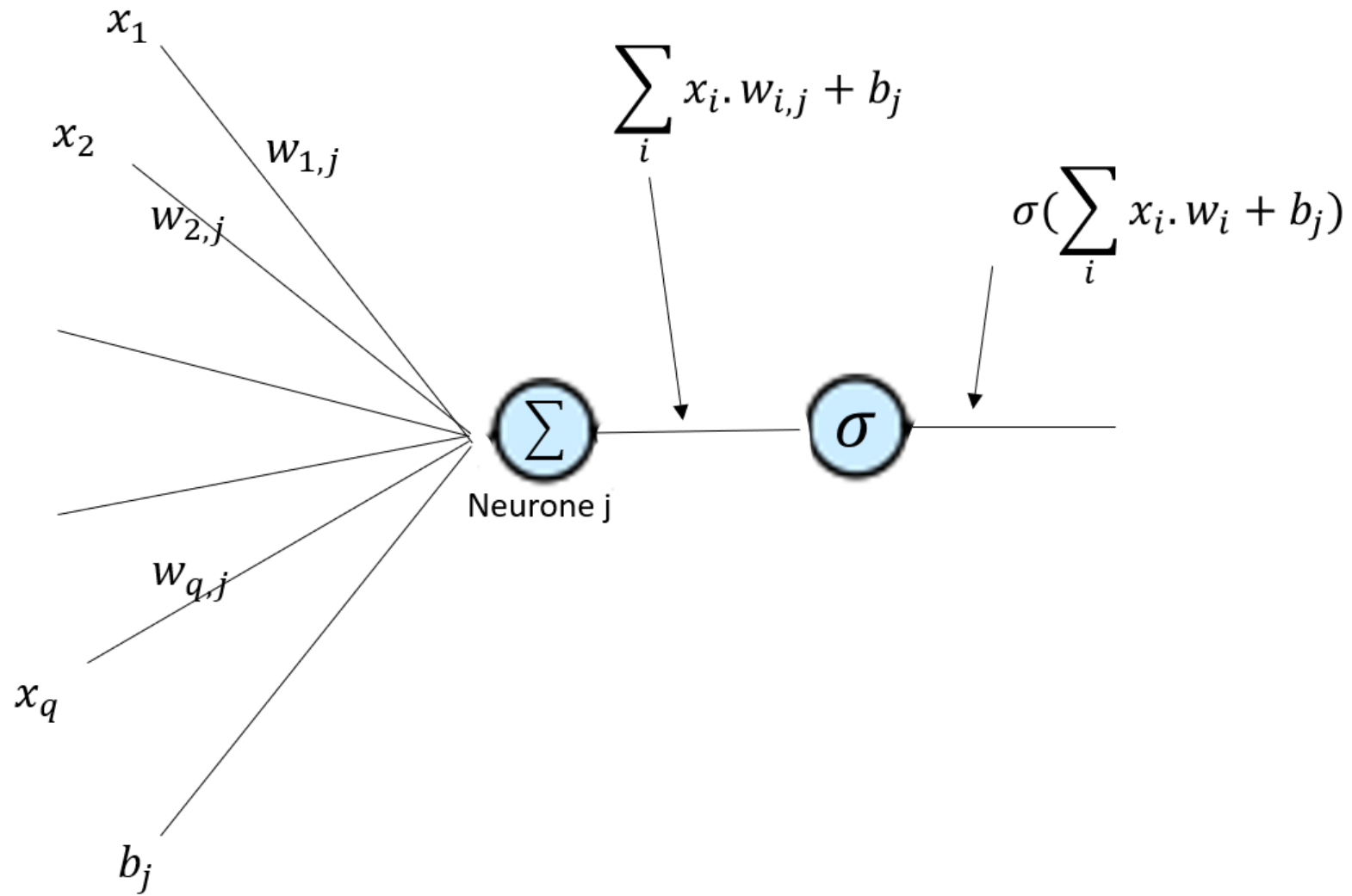
$$S = \left\{ f \in F(\mathbb{R}^p, \mathbb{R}^q) \left| \exists \begin{cases} \sigma, \text{sigmoïdale} \\ x, w_j \in \mathbb{R}^n \\ \alpha_j, b_j \in \mathbb{R}, N \in \mathbb{N} \end{cases}, f(x) = \sum_{j=1}^N \alpha_j \sigma\left(\sum_{i=1}^n w_{j,i} x_i + b_j\right) \right. \right\}$$

*Il est admis que  $S$  est dense dans  $C \subseteq \mathcal{C}^0(\mathbb{R}^p, \mathbb{R}^q)$ , où  $C$  est un compact munit de la norme infinie*

Réseau de type perceptron multicouches :

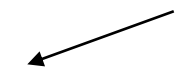


Aspect théorique :



### Généralisation :

$$B^{(l)} = [b_1^{(l)} \dots b_q^{(l)}]$$

couche n°(l) 

Conformément au schéma diapositive n° 6, la matrice de poids entre les deux couches cachées est de la forme :

$$W = \begin{bmatrix} w_{1,1}^{(l)} & \dots & w_{1,q}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{n,1}^{(l)} & \dots & w_{n,q}^{(l)} \end{bmatrix}$$

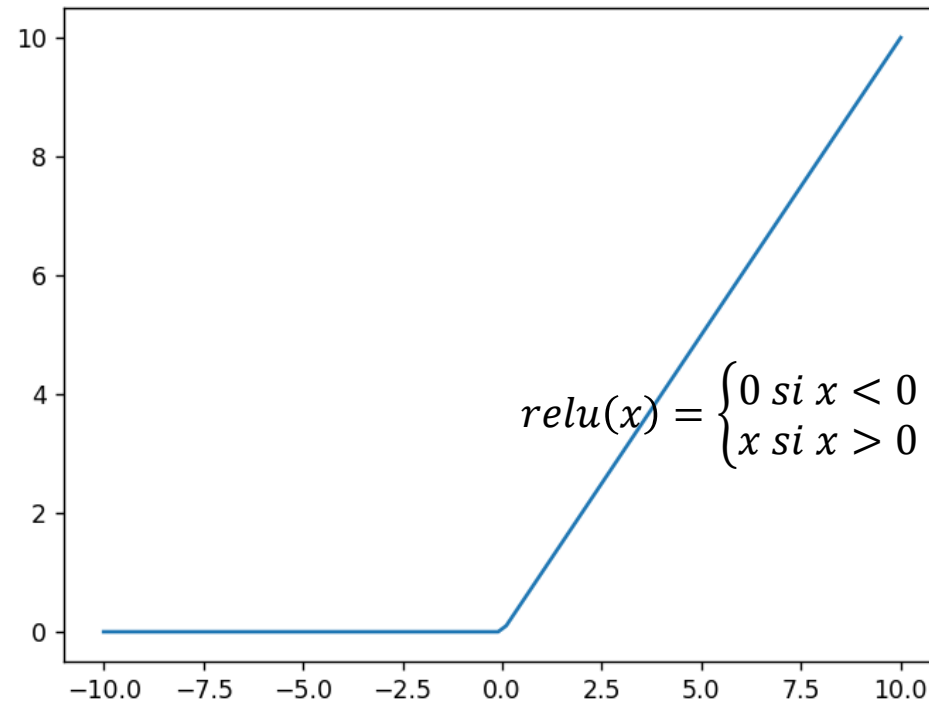
$$Y^{(l)} = [y_1^{(l)} \dots y_q^{(l)}]$$

$$Y^{(l)} = A^{(l-1)} \cdot W^{(l)} + B^{(l)} \xrightarrow{\text{Fonction d'activation}} A^{(l)} = F(A^{(l-1)} \cdot W^{(l)} + B^{(l)}) = F(Y^{(l)}) \quad \text{Où par abus, } A^{(l)} = F(X^{(l)}) = \begin{bmatrix} F(x_1^{(l)}) \\ \vdots \\ F(x_n^{(l)}) \end{bmatrix}^T$$

$$\text{avec } F(x): \begin{cases} \mathbb{R}^n \rightarrow \mathbb{R}^q \\ X \rightarrow F(X) \end{cases}, \text{ une fonction d'activation}$$



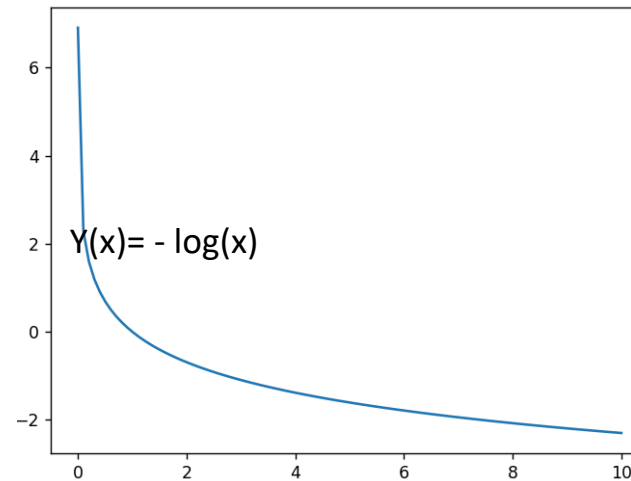
## Grappe de la fonction Relu



## Retropropagation :

$$y \in \{0,1\}$$

$$\text{Le log Perte : } \mathcal{L}_{\log}(p, y) = -y \cdot \log(p) - (1-y) \cdot \log(1-p) = -y \cdot \log\left(\frac{1}{1+e^{-wx-b}}\right) - (1-y) \cdot \log\left(1 - \frac{1}{1+e^{-wx-b}}\right)$$



On définit le gradient de la perte par rapport aux poids de la  $k^{ième}$  couche comme :  $\nabla_k \mathcal{L}(W) = \left( \frac{d\mathcal{L}}{dw_{i,j}^{(k)}} \right)_{\substack{1 \leq i \leq n \\ 1 \leq j \leq q}}$

$$B^{(l)} \leftarrow B^{(l)} - \eta \nabla_k \mathcal{L}$$

$$W^{(l)} \leftarrow W^{(l)} - \eta \nabla_k \mathcal{L}$$

Illustration de de l'entraînement en se limitant au cinq premiers imla diminution de  $\mathcal{L}$  au fur et à mesure des itérations :

```
[[0.69147414]
 [0.69017637]
 [0.6874136 ]
 [0.69130707]
 [0.69864523]]]
Vecteur perte
itération : 0, precision: 0.5135644310474755, perte: 0.6939195990562439, learning rate: 0.3
[[0.6981409]
 [0.7245783]
 [0.7098616]
 [0.6983102]
 [0.6569205]]
itération : 1, precision: 0.46382818387339864, perte: 0.6882178783416748, learning rate: 0.3
[[0.7191336]
 [0.8127722]
 [0.7661722]
 [0.7194454]
 [0.5650101]]
itération : 2, precision: 0.46269781461944237, perte: 0.6789221167564392, learning rate: 0.297029702970297
[[0.7656563 ]
 [1.0624086 ]
 [0.9176998 ]
 [0.76553965]
 [0.37772083]]
itération : 3, precision: 0.46269781461944237, perte: 0.6711932420730591, learning rate: 0.29120559114735
[[0.68463063]
 [1.0385733 ]
 [0.88778716]
 [0.68330926]
 [0.3840827 ]]
itération : 4, precision: 0.5422004521477016, perte: 0.6489043235778809, learning rate: 0.2827238749003398
```

# Expérience :

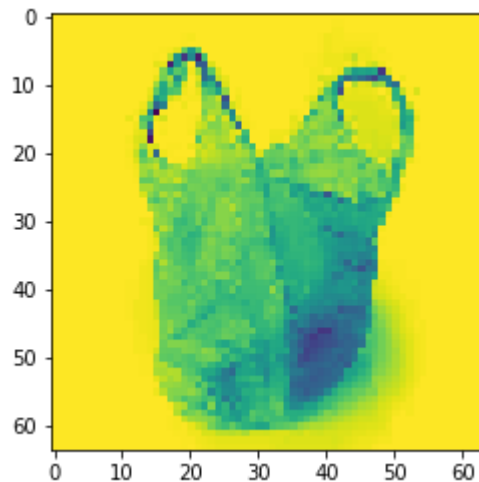
**Rappel du but :** A partir de la lecture du flux camera de son environnement, le robot doit être capable de prédire si l'objet qui se trouve devant lui est un déchet ou pas.

Moyen pour y parvenir : Faire des captures de manière rapprochée du flux vidéo pour ensuite les faire analyser par l'intelligence artificielle.

Sac plastique provenant de la base de donnée 1



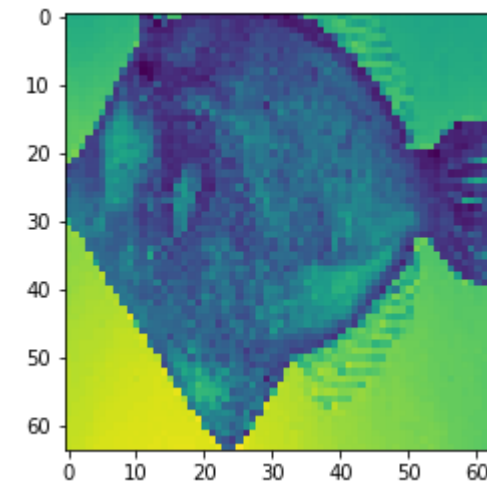
Même image lue par le réseau provenant de la base de donnée 1



Sac plastique provenant de la base de donnée 0



Même image lue par le réseau provenant du data set 0

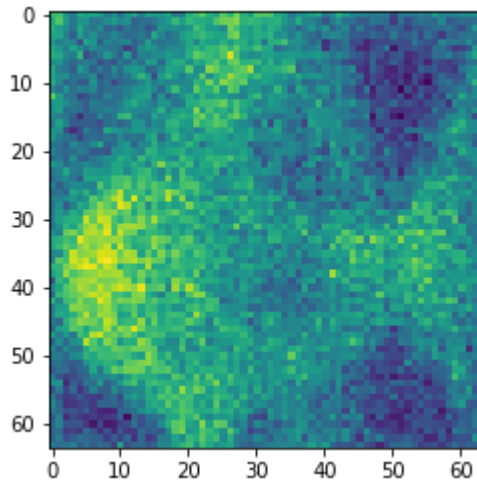


Moyenne des poids associée à chaque pixel, de la première couche dense (à gauche) et à de la deuxième couche dense (à droite) :

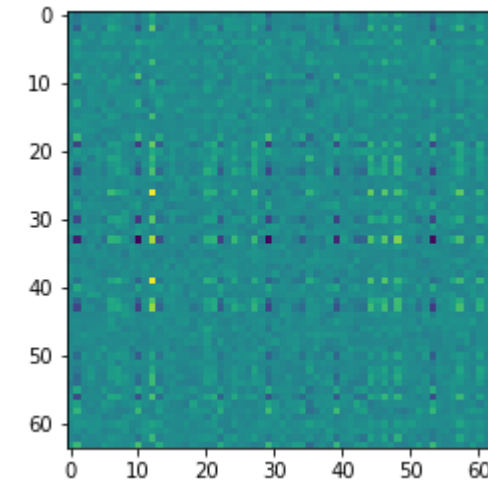
Pixel jaune = pixel associé à un poids fort

Pixel bleu = pixel associé à un poids faible

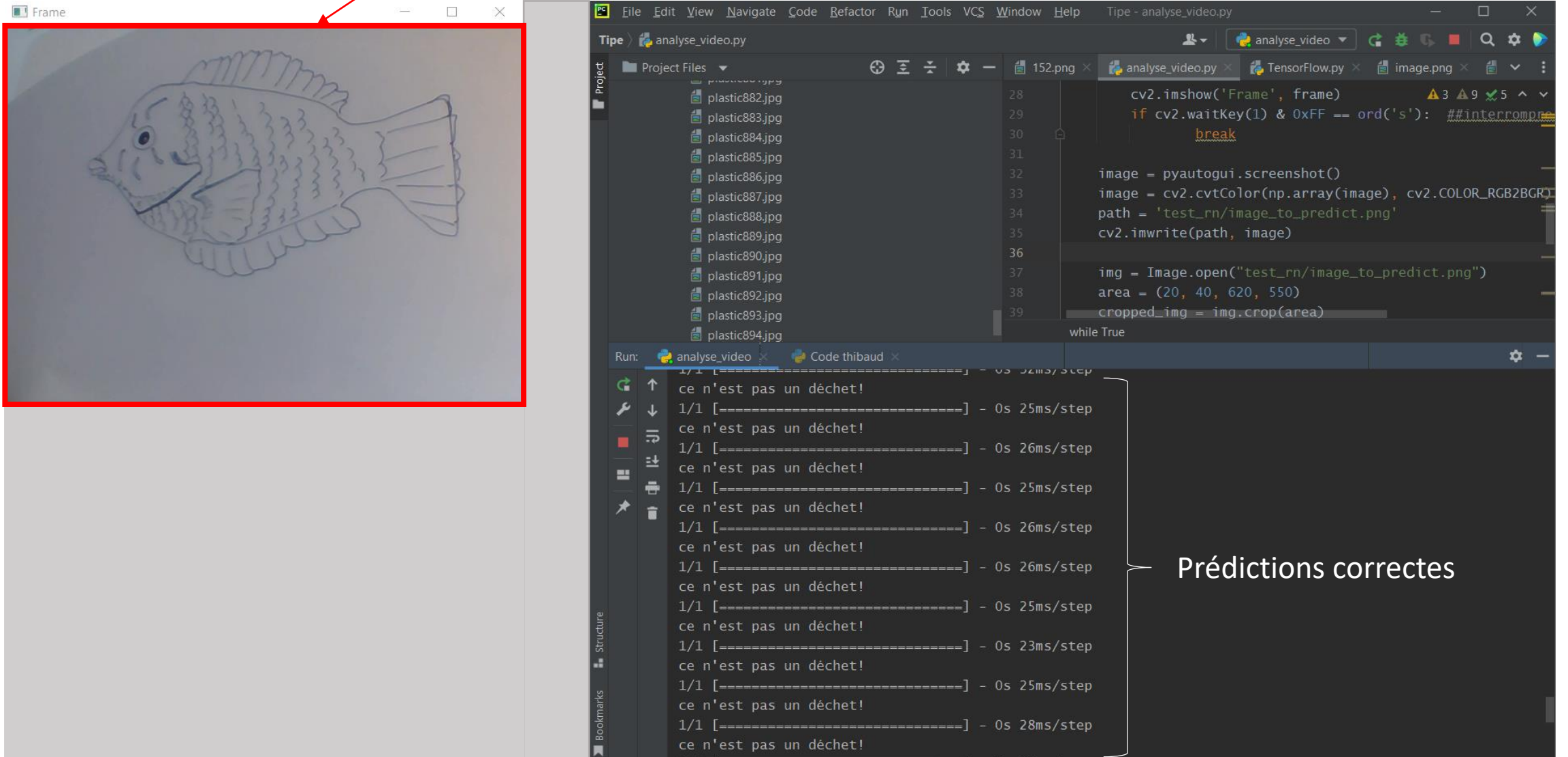
```
plt.imshow(model.stack[0].weights.mean(axis=1).reshape(64, 64))  
plt.show()
```



```
plt.imshow(model.stack[2].weights)  
plt.show()
```



Zone d'analyse vidéo de la webcam de l'ordinateur



The image displays a video analysis application interface. On the left, a window titled 'Frame' shows a video feed of a hand-drawn fish on a piece of paper. A red rectangle highlights this area, with a red arrow pointing to it from the text 'Zone d'analyse vidéo de la webcam de l'ordinateur'. On the right, a code editor window titled 'Tipe - analyse\_video.py' shows the following Python code:

```
28 cv2.imshow('Frame', frame)
29 if cv2.waitKey(1) & 0xFF == ord('s'): ##interrompre
30     break
31
32 image = pyautogui.screenshot()
33 image = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)
34 path = 'test_rn/image_to_predict.png'
35 cv2.imwrite(path, image)
36
37 img = Image.open("test_rn/image_to_predict.png")
38 area = (20, 40, 620, 550)
39 cropped_img = img.crop(area)
40
41 while True
```

Below the code editor, a 'Run' window shows the output of the program, which consists of a series of predictions: 'ce n'est pas un déchet!'. Each prediction is preceded by a progress bar and a timestamp, indicating the time taken for each step. A bracket on the right side of the 'Run' window points to the text 'Prédictions correctes'.





```
Tipe > analyse_video.py
Project Files
152.png x analyse_video.py x application_tf.py x cz.py x
Run: analyse_video x
1/1 [=====] - 0s 20ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 26ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 24ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 38ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 26ms/step
c'est du plastique !
1/1 [=====] - 0s 24ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 24ms/step
c'est du plastique !
1/1 [=====] - 0s 25ms/step
c'est du plastique !
1/1 [=====] - 0s 26ms/step
c'est du plastique !
```

## Critique :

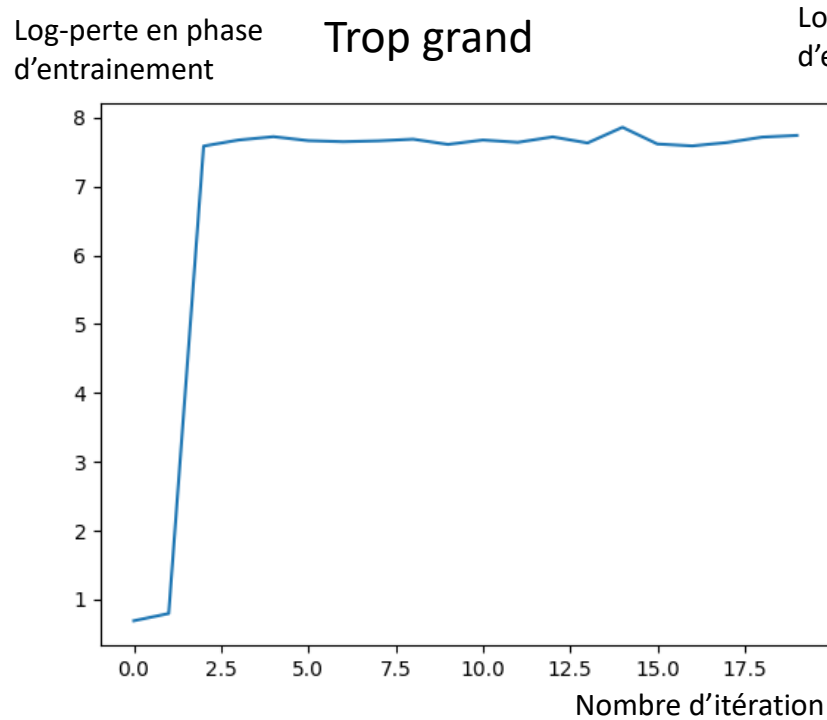
- 1) Cf diapositive n° 17
- 2) La vidéo est saccadée en utilisant le réseau de neurones confectionné, ce qui rend délicat son utilisation réelle pour le robot en mouvement et soumis aux courants.

### Solutions possibles :

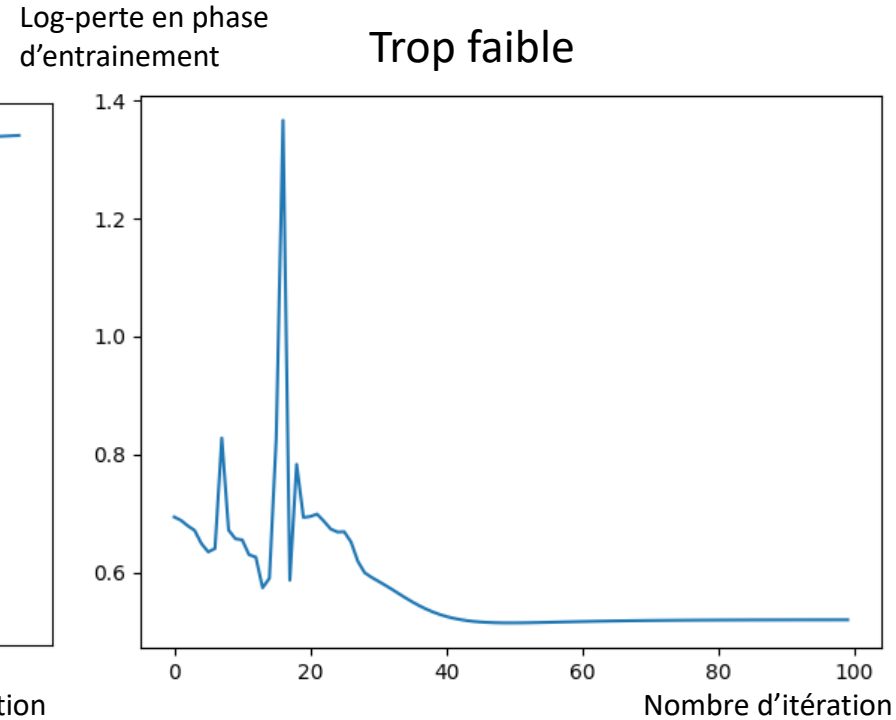
- Utiliser un réseau plus efficace avec notamment un réseau de convolution
- Utiliser un autre langage que python, du type C++, plus rapide.

## Influence sur les performances des différentes hyper-paramètres :

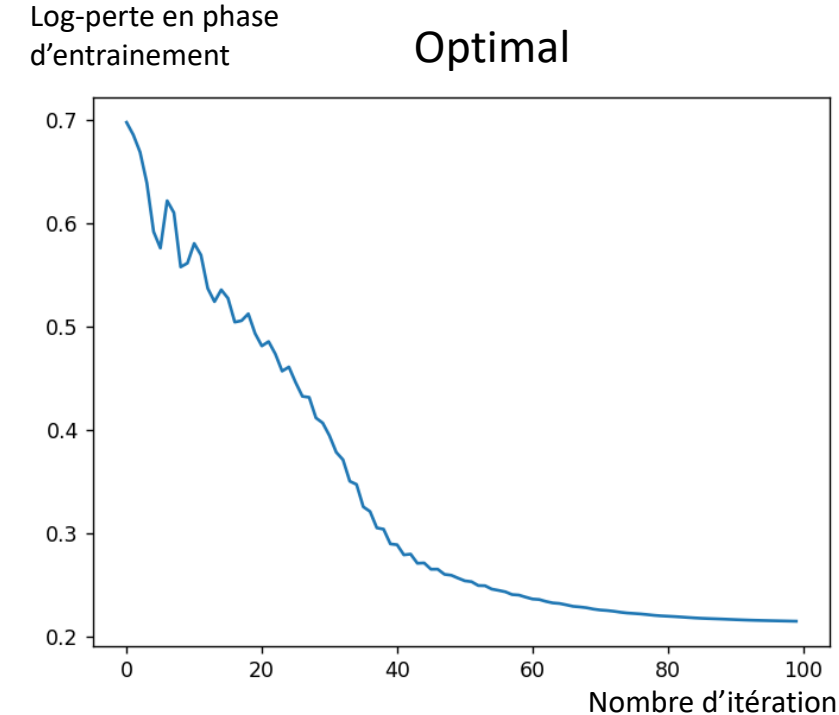
Log perte en fonction du taux d'apprentissage sans «lot » :



La fonction de perte augmente et ne converge pas vers le minimum.



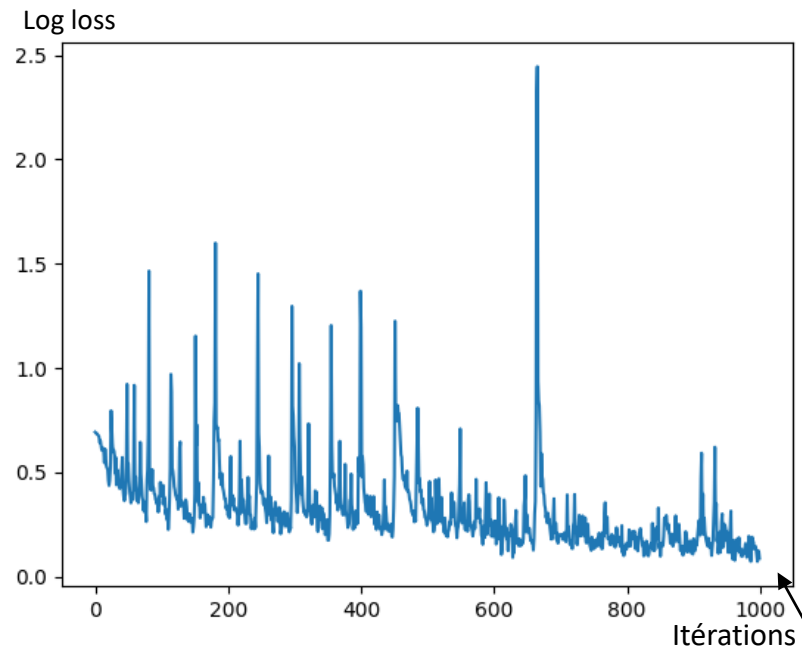
La fonction de perte converge trop rapidement sans avoir pu atteindre la valeur optimale.



La fonction de perte converge de manière optimale.

## Utilisation de lots :

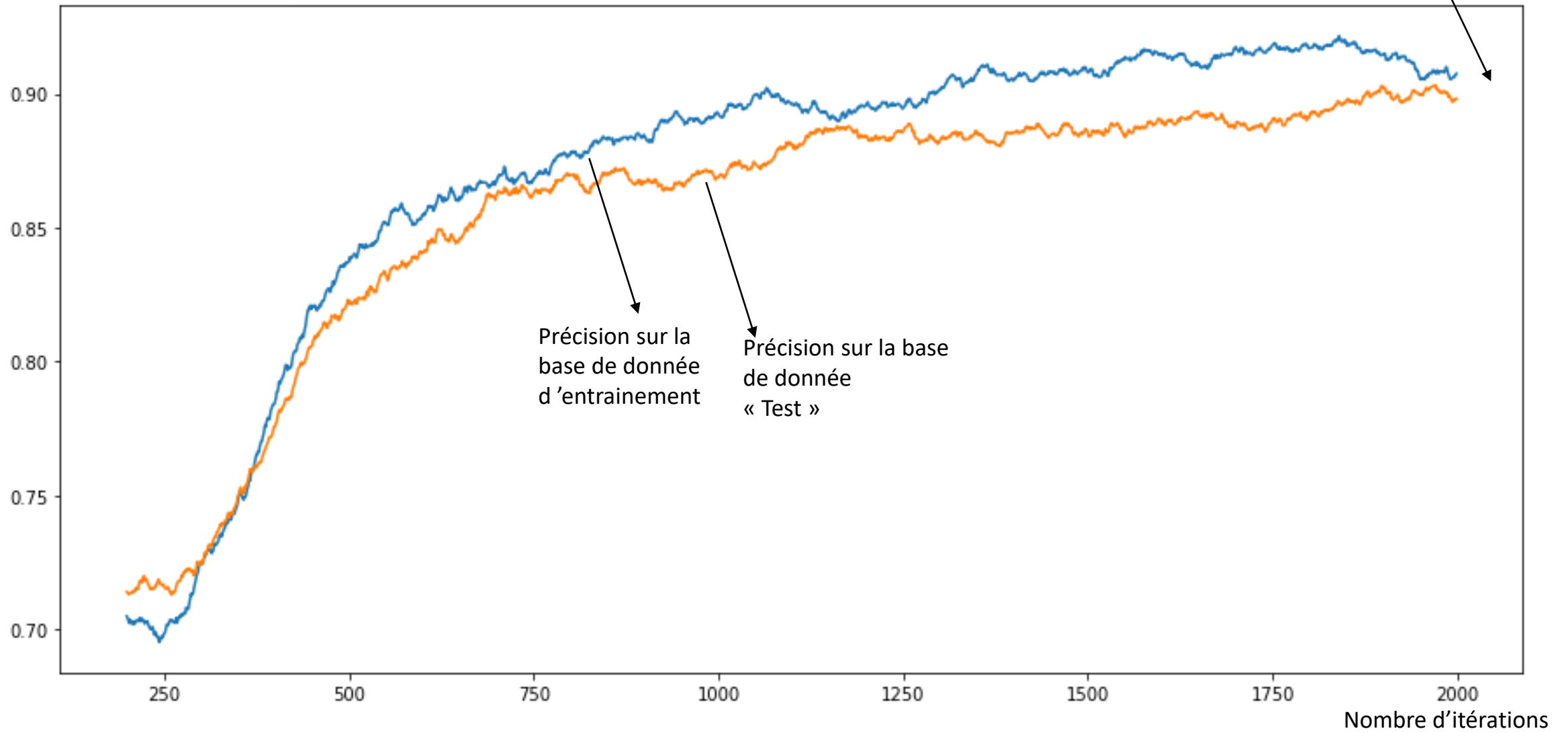
Lot de 32, avec un taux d'apprentissage  
de 0,1



Log perte de 0,08

## Comparaison

Précision



## Annexe

### Les équations fondamentales de la Rétropropagation :

La propagation avant étant sensée être effectuée, la donnée  $\frac{\partial \mathcal{L}}{\partial Y}$  est connue et est largement utilisée.

$$\frac{\partial \mathcal{L}}{\partial W} = X^T \cdot \frac{\partial \mathcal{L}}{\partial Y}, \text{ avec } \frac{\partial \mathcal{L}}{\partial Y} \stackrel{\text{def}}{=} \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial y_1} & \dots & \frac{\partial \mathcal{L}}{\partial y_q} \end{bmatrix} \text{ et } \frac{\partial \mathcal{L}}{\partial W} \stackrel{\text{def}}{=} \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{1,1}} & \dots & \frac{\partial \mathcal{L}}{\partial w_{1,q}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{n,1}} & \dots & \frac{\partial \mathcal{L}}{\partial w_{n,q}} \end{bmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial B} = \frac{\partial \mathcal{L}}{\partial Y}, \text{ avec les même notations que précédemment}$$

$$\frac{\partial \mathcal{L}}{\partial X} = \frac{\partial \mathcal{L}}{\partial Y} \cdot W^T$$

$$\text{Erreur vis à vis du neurone de la couche de sortie } \delta^{(l)} = \frac{\partial \mathcal{L}}{\partial a^{(l)}} \odot \sigma'(y^{(l)})$$

$$\text{Avec } \sigma'(y^{(L_{\text{sortie}})}) = \sigma(y^{(L_{\text{sortie}})}) \cdot (1 - \sigma(y^{(L_{\text{sortie}})}))$$

### Démonstration formules fondamentales :

Diminuer la fonction de perte :

Dans le cas général d'une fonction vectorielle différentiable à valeur dans  $\mathbb{R}$  et convexe,  $f$ , on cherche le pas  $h$  tq  $f(X + h) \leq f(X)$ .

*f étant différentiable* on a  $f(X + h) = f(X) + \langle \nabla f | h \rangle + o(h)$ . Ainsi il suffit que  $\langle \nabla f(X) | h \rangle \leq 0$ . En posant alors  $h = -\nabla f(X)$  il est immédiat que  $f(X+h)$  va évoluer de telle sorte à diminuer de valeur, et ce qu'importe la valeur de  $X$ . On peut enfin contrôler l'importance de cette diminution en appliquant un coefficient variable sur le pas  $h$ , c'est le taux d'apprentissage  $\eta$ . *In fine*, en itérant ce processus et au vu des hypothèses, il est certain que  $f$  se « dirige » vers son minimum global.

## Réseau de neurones de type, perceptrons Multicouches

```
class relu_activation:
    # propagation_avt pass

    def propagation_avt(self, entrée):
        self.entrée = entrée # enregistrement pour les dérivées
        # partielles
        self.sortie = np.maximum(0, entrée) # relu

    def retro_propagation(self, dy):
        ##dy dérivé de la fonction de perte par rapport à la sortie
        # de la fonction d'activation des neurones de la couche suivante (pour
        # bien "propager")
        self.dx = dy.copy()
        acti = self.entrée.copy()
        for i in range(np.shape(acti)[0]): ##regle de la chaine
            for j in range(np.shape(acti)[1]):
                if acti[i, j] <= 0:
                    acti[i, j] = 0
                else:
                    acti[i, j] = 1
        self.dx *= acti
```

**class** couche\_Dense:

```
    def __init__(self, n_entrée, n_neurones):
        self.poids = 0.01 * np.random.randn(n_entrée, n_neurones)
        self.biais = np.zeros((1, n_neurones))

    # propagation_avt pass
    def propagation_avt(self, entrée):
        self.entrée = entrée
        self.sortie = np.dot(entrée, self.poids) + self.biais

    # retro_propagation pass
    def retro_propagation(self, dy):
        self.der_poids = np.dot(self.entrée.T, dy)
        self.der_biais = np.sum(dy, axis=0, keepdims=True)
        self.dx = np.dot(dy, self.poids.T)
```

**class** Sigmoid:

```
    def propagation_avt(self, entrée):
        self.entrée = entrée
        n, p = np.shape(entrée)
        sigmo = entrée.copy()
        for i in range(n):
            for j in range(p):
                sigmo[i, j] = (1 + math.exp(-sigmo[i, j])) **
                (-1)
        self.sortie = sigmo

    def retropropagation(self, dy):
        self.dx = dy * (1 - self.sortie) * self.sortie
```



```
class perte_crois e_d_entropie_binaire:
```

```
def propagation_avt(self, prediction, y):
    x_1 = np.clip(prediction, 1e-7, 1 - 1e-7)
    x_2 = -y * np.log(x_1) - (1 - y) * np.log(1 - x_1)
    log_loss_binaire = np.mean(x_2, axis=-1)
    data_perte = np.mean(log_loss_binaire)

    return data_perte

def retro_propagation(self, a, y):
    taille_data = len(a)
    sortie = len(a[0])
    a2 = np.clip(a, 1e-7, 1 - 1e-7)
    self.dx = -(y / a2 - (1 - y) / (1 - a2)) / sortie
    self.dx = self.dx / taille_data
```

```
class descente_de_gradient: # ajustement des para
```

```
def __init__(self, lr, gamma, moment):
    # hyper_parametres
    self.lr = lr
    self.gamma = gamma
    self.moment = moment

def para_aams(self, couche):

    if self.moment:
        if hasattr(couche,
                    'momentum') == False: couche.momentum =
np.zeros_like(couche.poids)
        couche.bias_moments = np.zeros_like(couche.biais)
        poids_aa = self.moment * couche.momentum - self.lr *
couche.der_poids
        couche.momentum = poids_aa
        biais_aa = self.moment * couche.bias_moments - self.lr *
couche.der_biais
        couche.bias_moments = biais_aa

    else:
        poids_aa = -self.lr * couche.der_poids
        biais_aa = -self.lr * couche.der_biais

    couche.poids += poids_aa
    couche.biais += biais_aa
```

## Import des données et de leur post-traitement :

```
X=[]
Y=[]
largeur_image=64

files= os.listdir('data_set/0')

for image in files:

    path = 'data_set/0/{}'.format(image)
    image_tp=cv2.imread(path,cv2.IMREAD_GRAYSCALE)
    res = cv2.resize(image_tp, dsize=(64, 64),
interpolation=cv2.INTER_CUBIC)
    X.append(res) #data
    Y.append(0) #label_entrainement

files2= os.listdir('data_set/1')

for image in files2:

    path2 = 'data_set/1/{}'.format(image)
    image_tp2 = cv2.imread(path2, cv2.IMREAD_GRAYSCALE)
    res = cv2.resize(image_tp2, dsize=(64, 64),
interpolation=cv2.INTER_CUBIC)
    X.append(res)
    Y.append(1) # label

X,y=np.array(X), np.array(Y).astype('uint8')
X_test,y_test=np.array(X), np.array(Y).astype('uint8')
```

```
#normalisation contre l'overfitting + flatten Ainsi X en de dimension
nombre_image_to_predict*dimension_image**2
```

```
X = X.reshape(X.shape[0],X.shape[1]*X.shape[2]).astype(np.float32)/255
X_test =
X_test.reshape(X_test.shape[0],X_test.shape[1]*X_test.shape[2]).astype
(np.float32)/255
y = y.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)
```

```
##shuffle de manière uniforme
```

```
def shuffle(data,label):
    n = np.shape(data)[0]
    for i in range(n):
        k = np.random.randint(i, n)
        data[i], data[k] = data[k], data[i]
        label[i],label[k]=label[k],label[i]
```

```
shuffle(X,y)
```

## Définition du model

```
dense1 = couche_Dense(largeur_image**2, largeur_image**2)
activation1 = relu_activation()
dense2 = couche_Dense(largeur_image**2, 64)
activation2 = relu_activation()
dense3 = couche_Dense(64, 1)
activation3 = Sigmoid()
fonction_perte = perte_croisée_d_entropie_binaire()
update_para = descente_de_gradient(taux_apprentissage, gamma, momentum)
#si présence de lot
#taille_des_lots= taille_des_lots
```

```
for i in range(nombre_itération_voulue):
    #si présence de lot
    # lot=tuple([np.random.choice(np.arange(len(X)),taille_des_lots)])
    dense1.propagation_avt(X)
    #dense1.propagation_avt(X[lot])
    activation1.propagation_avt(dense1.sortie)
    dense2.propagation_avt(activation1.sortie)
    activation2.propagation_avt(dense2.sortie)
    dense3.propagation_avt(activation2.sortie)
    activation3.propagation_avt(dense3.sortie)
    perte= fonction_perte.propagation_avt(activation3.sortie, y)
    #perte= fonction_perte.propagation_avt(activation3.sortie, y[lot])
    PERTE.append(data_loss)
    EPOCH.append(i)
    predictions = (activation3.sortie > 0.5) * 1
    precision = np.mean(predictions == y)
    #precision = np.mean(predictions == y[lot])

    print('itération : {}, precision: {},perte: {}, learning rate: {}'.format(i,
precision,perte,para_aa.lr))
```

##### retro\_propagation #####

```
fonction_perte.retro_propagation(activation3.sortie, y)
#fonction_perte.retro_propagation(activation3.sortie, y[lot])
activation3.retropropagation(fonction_perte.dx)
dense3.retro_propagation(activation3.dx)
activation2.retro_propagation(dense3.dx)
dense2.retro_propagation(activation2.dx)
activation1.retro_propagation(dense2.dx)
dense1.retro_propagation(activation1.dx)
```

#####ajustement#####

```
if i>=2:
    if PERTE[i-1]<=data_loss:
        para_aa.lr*=para_aa.gamma
para_aa.para_aams(dense1)
```

```
para_aa.para_aams(dense1)
para_aa.para_aams(dense2)
para_aa.para_aams(dense3)
```

## Analyse vidéo avec le réseau de neurones entraîné

```
video = cv2.VideoCapture(0)

frame_width = int(video.get(3))
frame_height = int(video.get(4))

size = (frame_width, frame_height)

result = cv2.VideoWriter('filename.avi',
                        cv2.VideoWriter_fourcc(*'MJPG'), 10, size)

poids1= np.load('hyper_para/poids_1.npy')
poids2= np.load('hyper_para/poids_2.npy')
poids3= np.load('hyper_para/poids_3.npy')
biais1= np.load('hyper_para/biais_1.npy')
biais2= np.load('hyper_para/biais_2.npy')
biais3= np.load('hyper_para/biais_3.npy')

class Layer_Dense:
    def __init__(self, weights,biaises):
        self.weights = weights
        self.biases = biaises

    def forward(self, inputs):
        self.inputs = inputs
        self.output = np.dot(inputs, self.weights) + self.biases

class Activation_ReLU:...           #cf programme
class Activation_Sigmoid:...       #cf programme

dense1 = Layer_Dense(poids1, biais1)
dense2 = Layer_Dense(poids2, biais2)
dense3 = Layer_Dense(poids3, biais3)
activation1 = Activation_ReLU()
activation2 = Activation_ReLU()
activation3 = Activation_Sigmoid()
```

```
while True:
    # lecture flux video
    ret, frame = video.read()
    if ret == True:
        result.write(frame)
        cv2.imshow('Frame', frame)
        if cv2.waitKey(1) & 0xFF == ord('s'): ##interrompre l'analyse
            break

    image = pyautogui.screenshot()
    image = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)
    path = 'test_rn/image.png'
    cv2.imwrite(path, image)

    img = Image.open("test_rn/image.png")
    area = (20, 40, 620, 520)
    cropped_img = img.crop(area)
    cropped_img.save('test_rn/image_to_predict.png')
    t_début=time.precess_time()
    X = []

    path = 'test_rn/image_to_predict.png'
    image_tp = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    res = cv2.resize(image_tp, dsize=(64, 64), interpolation=cv2.INTER_CUBIC)
    X.append(res)
    X = np.array(X)
    X = X.reshape(X.shape[0], X.shape[1] * X.shape[2]).astype(np.float32) /

255

    dense1.forward(X)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    activation2.forward(dense2.output)
    dense3.forward(activation2.output)
    activation3.forward(dense3.output)
    t_fin=time.precess_time()
    if activation3.output > 0.5:
        print("c'est du plastique !« , 'En {} secondes '.format(t_fin-t_début)
    else:
        print(« ce n'est pas du plastique ! », 'En {} secondes'.format(t_fin-
t_début))
```

## Analyse vidéo TensorFlow : (Solution efficace avec cnn)

```
import cv2
import pyautogui
import time
from time import sleep
import numpy as np
import os
Import tensorflow as tf

video = cv2.VideoCapture(0)

frame_width = int(video.get(3))
frame_height = int(video.get(4))

size = (frame_width, frame_height)

result =
cv2.VideoWriter('filename.avi', cv2.VideoWriter_fourcc(*'MJPG'),
10, size)
model = tf.keras.models.load_model('saved_model/my_model')
```

```
while True:
    #lecture flux video
    ret, frame = video.read()
    if ret == True:
        result.write(frame)
        cv2.imshow('Frame', frame)
        if cv2.waitKey(1) & 0xFF == ord('s'): ##interrompre l'analyse
            break

    image = pyautogui.screenshot()
    image = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)
    path = 'test_rn/image_to_predict.png'
    cv2.imwrite(path, image)

    img = Image.open("test_rn/image_to_predict.png")
    area = (20, 40, 620, 550)
    cropped_img = img.crop(area)
    cropped_img.save('test_rn/image.png')

    image_to_predict = cv2.imread('test_rn/image.png', cv2.IMREAD_COLOR)
    img_to_predict = np.expand_dims(cv2.resize(image_to_predict, (64, 64)),
axis=0)
    resultat = model.predict(img_to_predict)
    if resultat - 0.5 > 0:
        print("c'est du plastique !")
    else:
        print("ce n'est pas un déchet!")
```

## Programme TensorFlow :

```
img_height = 64
img_width = 64
batch_size = 30
## import data
ds_train =
tf.keras.preprocessing.image_dataset_from_directory(
    'data_set',
    validation_split=0.2,
    shuffle=True,
    subset="training",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=batch_size,
)
##
ds_validation =
tf.keras.preprocessing.image_dataset_from_directory(
    'data_set',
    validation_split=0.2,
    shuffle=True,
    subset="validation",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=batch_size)
##
```

```
class_names = ds_validation.class_names
##definition réseau
model = tf.keras.Sequential([
    layers.experimental.preprocessing.Rescaling(1./255),
    layers.Conv2D(64,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(16,4, activation='relu'),
    layers.MaxPooling2D(),
    layers.Flatten(),
    layers.Dense(64,activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
##Définition paramètres
class_names = ds_train.class_names
##
model.compile(optimizer='adam',loss=tf.losses.BinaryCrossentropy(from_logits=True),metrics=['BinaryAccuracy'],)
logdir="logs"
tensorboard_callback =
keras.callbacks.TensorBoard(log_dir=logdir,histogram_freq=1,
write_images=logdir, embeddings_data=ds_train)
## entraînement
model.fit(
    ds_train,
    validation_data=ds_validation,
    epochs=2,
    callbacks=[tensorboard_callback]
)
## Sauvegarde
model.save('saved_model/my_model')
```