```python
# TIPE 2019-2020

# Importations

import numpy as np
import scipy.ndimage as nd
import os
import random
from PIL import Image
import imageio
import matplotlib.pyplot as plt
import pathlib
import glob
import shutil
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow.keras.layers as layers
import sqlite3
import pickle

# Fonctions pratiques, manipulation basique d'images

luminance_image = lambda image : np.dot(image[...,:3], [0.2989, 0.5870,
    ↪ 0.1140])

def distribution_luminance(im) :
    l,c = np.shape(im)
    histo = np.array([0]*256)
    for i in range(l) :
        for j in range(c) :
            histo[ im[i,j] ] += 1
    return histo/(l*c)


def intersection_cercles(x,y,r, x2,y2,r2):
    distance = np.sqrt((x2-x)**2+(y2-y)**2)
    return distance + r <= r2 or (abs(r-r2) <= distance <= r+r2)

def imshowcircles (image, points, ax, ratio, cmap="gray"):
    ax.imshow(image, cmap)
    for i in range(len(points)):
        x,y,h = points[i]
        c = plt.Circle((y,x),1.4*ratio**h,color='blue',fill=False)
        ax.add_artist(c)
        plt.text(y, x, str(i), color="white", fontsize=12)

def extraction(img, minis, sigma, ratio, dir, prefix=""):
    for i in range(len(m)):
        x,y,h = minis[i]
        radius = int(1.6*sigma*(ratio**h))
        try :
            imageio.imwrite(dir+prefix+str(i)+".png", img[x-radius:x+radius
                ↪ +1,y-radius:y+radius+1,:])
        except :
            pass

# Convolution et noyaux gaussiens

def noyau_gauss(sigma, dim2D=True):
    taille = 2*int(np.ceil(sigma))+3
    if dim2D:
```

```
        noyau = np.fromfunction(lambda x, y: (1/(2*np.pi*sigma**2)) * np.exp
            ↪ ((-1*((x-(taille-1)/2)**2+(y-(taille-1)/2)**2)) / (2*sigma**2)
            ↪ ), (taille, taille))
    else:
        noyau = np.fromfunction(lambda x: np.exp((-1*(x-(taille-1)/2)**2) /
            ↪ (2*sigma**2)), (taille,))
    return noyau / np.sum(noyau)

def etendre(image,d):
    etendu = np.c_[ np.flip(image[:,2:2+d],1), image, np.flip(image[:,-2-d
        ↪ +1:-1],1)]
    etendu = np.r_[ np.flip(etendu[2:2+d],1), etendu, np.flip(etendu[-2-d
        ↪ +1:-1],1)]
    for i in range(d-1) :
        etendu[i,i] = etendu[2*d-i,2*d-i]
        etendu[-i-1,-i-1] = etendu[-2*d-i-1,-2*d-i-1]
        etendu[i,-i-1] = etendu[2*d-i,-2*d-i-1]
        etendu[-i-1,i] = etendu[-2*d-i-1,2*d-i]
    return etendu

def convolution(image, noyau ):
    d = (noyau.shape[0]-1)//2
    etendu = etendre(image,d)
    resultat = np.zeros((image.shape[0], image.shape[1]))
    for i in range(d,image.shape[0]):
        for j in range(d,image.shape[1]):
            resultat[i,j] = np.sum((etendu[i-d:i+d+1, j-d:j+d+1]*noyau))
    return resultat

def convolution_separabilite(image, noyauX, noyauY):
    d = (noyauX.shape[0]-1)//2
    etendu = etendre(image,d)
    resX = np.zeros((image.shape[0]+2*d, image.shape[1]))
    for i, v in enumerate(noyauX):
        resX += v * etendu[:, i : image.shape[1] + i]
    resY = np.zeros((image.shape[0], image.shape[1]))
    for i, v in enumerate(noyauY):
        resY += v * resX[i : image.shape[0] + i]
    return resY


# Pyramide d'échelle et recherche des minimums

def minimas(img ,pyramid_height, sigma, ratio, min=0):
    X,Y = np.shape(img)
    pyramid = np.empty((pyramid_height,X,Y))
    temp = img
    minis = []

    for i in range(pyramid_height):
        temp2 = nd.gaussian_filter(img,sigma*ratio**i)
        pyramid[i,:,:] = temp-temp2
        temp = temp2

    for h in range(pyramid_height-2,1+min,-1):
        for x in range(1,X-1):
            for y in range(1,Y-1):

                val = pyramid[h,x,y]

                if ( val < -1
```

```
                and val < pyramid[h,x+1,y]
                and val < pyramid[h,x-1,y]
                and val < pyramid[h,x,y+1]
                and val < pyramid[h,x,y-1]
                and val < pyramid[h,x+1,y+1]
                and val < pyramid[h,x+1,y-1]
                and val < pyramid[h,x-1,y+1]
                and val < pyramid[h,x-1,y+1]
                and val < pyramid[h+1,x,y]
                and val < pyramid[h+1,x+1,y]
                and val < pyramid[h+1,x-1,y]
                and val < pyramid[h+1,x,y+1]
                and val < pyramid[h+1,x,y-1]
                and val < pyramid[h+1,x+1,y+1]
                and val < pyramid[h+1,x+1,y-1]
                and val < pyramid[h+1,x-1,y+1]
                and val < pyramid[h+1,x-1,y+1]
                and val < pyramid[h-1,x,y]
                and val < pyramid[h-1,x+1,y]
                and val < pyramid[h-1,x-1,y]
                and val < pyramid[h-1,x,y+1]
                and val < pyramid[h-1,x,y-1]
                and val < pyramid[h-1,x+1,y+1]
                and val < pyramid[h-1,x+1,y-1]
                and val < pyramid[h-1,x-1,y+1]
                and val < pyramid[h-1,x-1,y+1] ):

                    inside = False
                    for i in range(len(minis)):
                        x2,y2,h2 = minis[i]
                        if intersection(sigma, ratio, x,y,h,x2,y2,h2) :
                            inside = True
                            break
                    if not inside : minis.append((x,y,h))

    return pyramid, minis

# Apprentissage automatique

# Avec Tensorflow
data_dir=pathlib.Path('/content/drive/My Drive/datasets/morvan/sorted')

batch_size = 15
img_height = 50
img_width = 50

train_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)

val_ds = tf.keras.preprocessing.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
AUTOTUNE = tf.data.experimental.AUTOTUNE
train_ds = train_ds.cache().prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

num_classes = 2

data_augmentation = tf.keras.Sequential(
  [
    layers.experimental.preprocessing.RandomFlip("horizontal", input_shape=(
       ↪ img_height, img_width, 3)),
    layers.experimental.preprocessing.RandomRotation(0.1),
    layers.experimental.preprocessing.RandomZoom(0.1),
  ]
)

model = tf.keras.Sequential([
  data_augmentation,
  layers.experimental.preprocessing.Rescaling(1./255),
  layers.Conv2D(32, 3, activation='relu'),
  layers.Conv2D(32, 3, activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(num_classes)
])

model.compile(
  optimizer='adam',
  loss=tf.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=['accuracy'])

epochs = 6
history = model.fit(
  train_ds,
  validation_data=val_ds,
  epochs=epochs
)

probability_model = tf.keras.Sequential([
  model,
  tf.keras.layers.Softmax()
])

def test_sample(dir, randomize=True, max=8):
  fig=plt.figure(figsize=(20,20*height))
  images=glob.glob(dir+"*.png")
  if randomize : random.shuffle(images)
  for i,img_path in enumerate(images):
    if i>=max : break
    try:
      image = tf.keras.preprocessing.image.load_img(img_path, target_size=(
         ↪ img_height, img_width))
      input_arr = tf.keras.preprocessing.image.img_to_array(image)
      input_arr = np.array([input_arr])
      p1,p2 = probability_model.predict(input_arr)[0]
      ax=fig.add_subplot(1,max,i+1)
      ax.imshow(image)
      ax.title.set_text(str(round(p1,3))+"␣/␣"+str(round(p2,3)))
    except:
      pass
```

```
def test_image(dir):
  image = tf.keras.preprocessing.image.load_img(dir, target_size=(img_height, img_width))
  input_arr = tf.keras.preprocessing.image.img_to_array(image)
  input_arr = np.array([input_arr])
  predictions = probability_model.predict(input_arr)

  plt.imshow(image)
  print(predictions)

# Perceptron

def sigmoid (inputs, derivative=False):
    outputs = 1/(1+np.exp(-inputs))
    if derivative:
        outputs = np.exp(-inputs)*(outputs**2)
    return outputs

def ReLU (inputs, derivative=False):
    if derivative:
        return inputs>0
    return np.max(inputs,np.zeros_like(inputs))

def sum_of_square (outputs, expected_outputs, derivative=False ):
    if derivative :
        return np.sum( outputs - expected_outputs )
    return 0.5*np.sum( (outputs - expected_outputs)**2 )


default_settings = {
        "cost_function": sum_of_square,
        "activation_function": sigmoid,
        "size": [3,1], #[size of input layer, ... , size of output layer]
        "init_bias": 0.1,
        "min_weight":-1,
        "max_weight":1
}

class Perceptron:
    def __init__( self, settings=default_settings ):
        self.__dict__.update( settings )
        self.num_neurons = np.sum( self.size )
        self.num_layers = len(self.size)-1

        self.weights = np.array([ np.random.uniform( self.min_weight, self.max_weight, (self.size[k+1], self.size[k]) )  for k in range(self.num_layers) ])
        self.biases  = np.array([ np.array( [self.init_bias]*self.size[k+1] ) for k in range(self.num_layers) ])


    def update_weights( self, gradient, learning_rate ):
        self.weights -= learning_rate*gradient

    def update_biases( self, gradient, learning_rate ):
        self.biases -= learning_rate*gradient


    def feedforward( self, inputs, trace=False):
```

```python
        outputs = inputs
        if trace:
            derivatives = [] #Dérivé de la fonction d'activation
            detailled_outputs = [outputs]

        for k in range( self.num_layers ):
            aggregation = np.dot( self.weights[k], outputs ) + self.biases[k]
            outputs = self.activation_function( aggregation )

            if trace:
                detailled_outputs.append( outputs )
                derivatives.append( self.activation_function( aggregation, derivative=True ) )

        if trace : return derivatives, detailled_outputs
        return outputs


    def gradient( self, inputs, expected_outputs ):

        derivatives, detailled_outputs = self.feedforward( inputs, trace=True )
        cost_derivative = self.cost_function( detailled_outputs[-1], expected_outputs, derivative=True )

        delta = derivatives[-1]*cost_derivative #Delta de la dernière couche
        deltas = []

        for k in range(self.num_layers-2,-1,-1):
            deltas.append( delta )
            delta = derivatives[k]*np.dot( delta, self.weights[k+1] )
        deltas.append( delta )
        deltas = deltas[::-1]

        weights_gradient = np.array([ np.array( [ detailled_outputs[k]*delta for delta in deltas[k] ] ) for k in range(self.num_layers) ])
        biases_gradient = deltas
        return weights_gradient, biases_gradient


    def train( self, X, Y, learning_rate, batch_size=10, epoch=200 ):

        n = len(X)

        for _ in range(epoch) :
            indices = np.arange(n)
            np.random.shuffle(indices)
            X = X[indices]
            Y = Y[indices]

            for k in range( 0, n, batch_size ):
                temp_weights_gradient = np.array([np.zeros_like(k) for k in self.weights])
                temp_biases_gradient = np.array([np.zeros_like(k) for k in self.biases])

                for i in range( k, min( k+batch_size, n ) ):
                    temp_gradient = self.gradient( X[i], Y[i] )
                    temp_weights_gradient += temp_gradient[0]
                    temp_biases_gradient += temp_gradient[1]
```

```python
                temp_weights_gradient *= 1/batch_size
                temp_biases_gradient *= 1/batch_size

                self.update_weights( temp_weights_gradient , learning_rate )
                self.update_biases( temp_biases_gradient , learning_rate )


    def evaluate( self , X, Y ):

        n=len(X)
        X_forwarded = np.array([ self.feedforward(k) for k in X ])
        mean_error = self.cost_function( X_forwarded , Y) / n
        range_of_values = np.max(X_forwarded) - np.min(X_forwarded)
        return 1 - ( mean_error / range_of_forwarded_values )


    def export_to_bs( self , filename ):

       with open( filename , 'wb' ) as f:
           print("\nPickling... ")
           pickle.dump(self.__dict__ , f)
           print("completed!\n")

    def import_from_bs( self , filename ):

        with open( filename , 'rb' ) as f:
            print("\nUnpickling... ")
            self.__dict__.update( pickle.load( f ) )
            print("completed!\n")

# Prolongements

def floodfill ( array , severity ,  outcol , x, y) :
    """
    test
    """
    incol = np.copy(array[x,y])
    shape = np.shape(array)
    margin = [severity]*3

    def aux(x,y):
        col = array[x,y]
        if (np.abs((np.int16(col) - np.int16(incol))) <= margin ).all() :
            array[x,y] = outcol
            x,y = min(max(1,x),shape[1]-1), min(max(1,y),shape[0]-1)
            aux(x,y+1); aux(x,y-1); aux(x+1,y); aux(x-1,y)
        return
    aux(x,y)
```