

Détection et identification d'arbre à partir d'imagerie satellite

Augustin Albert

10 mai 2021

Table des matières

1 Détection des houppiers	2
1.1 L'approche multi échelle	2
1.2 Convolution et séparabilité du filtre gaussien	3
1.3 Mise en place de l'algorithme	4
1.4 Application au site d'étude	5
2 Identification des espèces	6
2.1 Méthodologie de construction du modèle	6
2.2 Évaluation des résultats	6
3 Prolongements envisagables	6
Références	6
A listing	7
B Résultats	18

Introduction

L'étude et le suivi de la répartition des espèces au sein de larges zones forestières est un problème complexe aux applications nombreuses : gestion des ressources naturelles, protection de la biodiversité, prévention des incendies, etc... Les études de terrain peuvent se révéler longues, coûteuses et imprécises du à la nécessité d'interpoler les données recueillies. Parvenir à automatiser ce processus est donc un enjeu critique.

Les techniques existantes reposent sur l'utilisation de données satellite ou aériennes : L'utilisation d'images est une méthode peu coûteuse nécessitant peu de matériel et pas d'intervention sur le terrain lorsque des images satellites récentes de résolution suffisantes, qui deviennent de plus en plus accessible.

UAV LiDar -> méthodes plus simples qui ne requièrent que des images st HR. Différentes méthodes existent, détection de blob ? LoG

Objectifs du TIPE

1. implémenter un algorithme de détection et de délimitation de houppiers (Cime d'arbres) basé sur des images satellites de résolution "moyenne" à l'aide de la théorie de l'Espace d'échelle (« Scale-space »)
2. entraîner un réseau neurone à l'aide de la bibliothèque Tensorflow afin d'identifier des espèces à partir d'images de houppiers de basse résolution
3. appliquer ces derniers au parc ... présentant une variété d'espèces et des patrons plus ou moins réguliers pour confronter les résultats obtenus aux données de terrain et aux techniques existantes.

-limitation à des images aériennes : pourquoi(moins coûteux, accessible sur internet, différents modes d'acquisitions, enjeux/difficultés -d'une part à concevoir ... pour détecter les a -d'autre part à l'utiliser pour construire une base de données permettant identification ultérieure sur la base du machine learning -application au site du parc régional...)

1 Détection des houppiers

1.1 L'approche multi échelle

On modélise un houppier par la figure hum. Il s'agit alors de repérer les amas circulaires de pixels plus lumineux que leur voisins : les blobs. Le diamètre des houppiers peut varier considérablement selon l'espèce et au sein d'une même image et la luminosité au centre des blobs n'est pas constante. Un seuillage de la luminosité est donc à exclure et une simple détection de contours se révèle peu précise lorsque la couverture forestière est dense. Une approche multi-échelle est donc nécessaire. Nous utiliserons la théorie échelle-temps développée par (??indbergh).

L'idée est de générer à partir d'un signal d'origine une famille de signaux dont les détails fins disparaissent progressivement. Aucune opération ne doit donc faire apparaître d'artefacts supplémentaires.

L'image originale est lissée de manière répétée au moyen d'un filtre gaussien aux propriétés de lissage exceptionnelles pour générer la pyramide d'échelle qui sera exploitée ultérieurement. (Voir 2)



FIGURE 1 –
Modélisation
d'un houppier

Les fonctions gaussiennes utilisées sont paramétrées par le paramètre d'échelle σ :

$$G_\sigma(x, y) := \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$



FIGURE 2 – Exemple de
pyramide d'image, Ori-
ginal, CC BY-SA 1.0

À chaque étape, σ est multiplié par un ratio fixe (2 dans la littérature). Puisque la résolution de l'image est réduite de moitié à chaque étape, chaque niveau est appelé octave en référence à la théorie musicale. Il peut être utile de rajouter des intervalles supplémentaires, ce qui est fait dans la suite. On dispose donc de 3 paramètres : σ , le nombre d'octave o et le nombre d'intervalle pour chaque octave i . La hauteur (nombre de niveau de la pyramide) est alors $o \times i$ et le ratio $2^{\frac{1}{i}}$.

Un opérateur laplacien normalisé est appliqué aux images résultante afin d'obtenir la pyramide d'échelle du laplacien du Gaussien ("LoG").

$$LoG_\sigma(x, y) := -\frac{1}{\pi\sigma^4} \left(1 - \frac{x^2 + y^2}{2\sigma^2}\right) \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

La pyramide d'échelle de l'opérateur LoG permet d'extraire des zones d'intérêt indépendamment de leur taille en exploitant la réponse de l'opérateur LoG appliquée à un signal échelon. (Voir 3). Lorsque le rayon caractéristique du blob r varie, le minimum (maximum en valeur absolue) du LoG est atteint au centre du blob. Lorsque le paramètre σ varie, la réponse au centre est minimale lorsque r est relié à σ par la relation $\sqrt{2}\sigma = r$. La réponse du LoG non normalisé s'atténuant lorsque σ augmente, l'opérateur est multiplié par σ^2 pour que la réponse soit indépendante de l'échelle.

La détection des blobs se ramène ainsi à la recherche d'un minimum local relativement à l'espace et global relativement à l'échelle pour identifier à la fois les centres des houppiers et la taille caractéristique de leur rayon.

1.2 Convolution et séparabilité du filtre gaussien

Ces opérations se traduisent par le produit de convolution discrète de l'image par les fonctions G_σ et le laplacien ou directement par la fonction LoG_σ . Les considérations suivantes permettent de réduire le nombre d'opérations élémentaires du programme :

- L'opérateur LoG est bien approximé par la différence des gaussiennes ("DoG") obtenue en réalisant la différence des niveaux successifs de la pyramide d'échelle.
- Une gaussienne prend presque toutes ses valeurs dans un intervalle centré de largeur 3 fois l'écart-type σ . On utilise alors un noyau gaussien de taille $1 + 3 \times E(\sigma)$
- On considère un noyau gaussien de taille $h \times h$. Le produit de convolution pour une image de taille $N \times M$ nécessite $cst \times h^2$ opérations élémentaires. Le filtre de Gauss étant séparable : $G_\sigma(x, y) = G_{1D, \sigma}(x) \times G_{1D, \sigma}(y)$ où $G_{1D, \sigma}(y) := \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right)$, on décompose le calcul en deux étapes. On réalise la convolution de l'image avec $G_{1D, \sigma}(x)$ puis la convolution du résultat avec $G_{1D, \sigma}(y)$, soit $cst \times 2h$ opérations au prix d'espace mémoire supplémentaire.

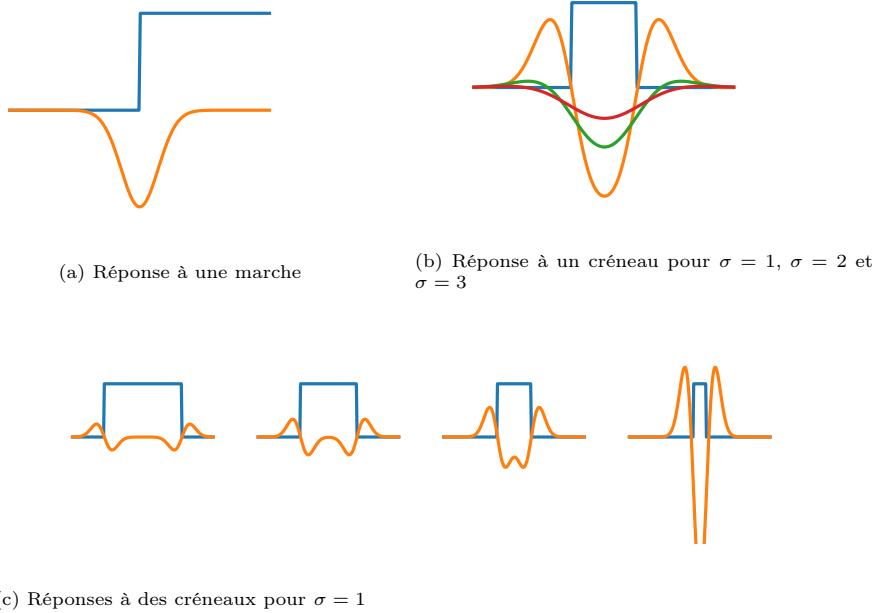


FIGURE 3 – Réponse de l'opérateur LoG à différents signaux. (Convolution des signaux par l'opérateur LoG 1D)

- Lisser équivaut à atténuer l'amplitude des basse fréquence du spectre. D'après le théorème de Shannon, il est donc possible de sous-échantillonner les images lissés sans perdre d'information. En sous-échantillonnant à chaque étape, calculer le produit de convolution d'un niveau donné à partir du niveau précédent permet de réduire le nombre d'opération. (-FFT) si temps

1.3 Mise en place de l'algorithme

L'algorithme envisagé a été implémenté à l'aide du langage Python. Son fonctionnement est le suivant :

1. Conversion en nuance de gris (Inversion éventuelle)
2. Génération de la pyramide d'échelle :
 - convolution par les filtres gaussiens. Pour conserver des images de même taille, un remplissage des bords ("padding") de type miroir est effectué, complétant l'image naturellement.
 - stockage dans un tableau Numpy 3D
3. Détection des minimums : différentes détections peuvent avoir lieu dans la même colonne donc pour conserver uniquement le houppier de rayon maximum lors d'éventuels chevauchements le tableau est parcouru par

échelle décroissante, ie par rayon détectés décroissants (cf formule), chaque case est comparée à ses 26 voisins et on vérifie que les houppiers détectés ne débordent pas sur les précédents. En outre, seul les rayons supérieurs à un certain seuil sont conservés.

4. Extraction des houppiers : le rayon adapté est calculé d'après formule . Aucune délinéation supplémentaire n'est réalisé pour l'instant et une zone carrée correspondant au rayon est extraite.

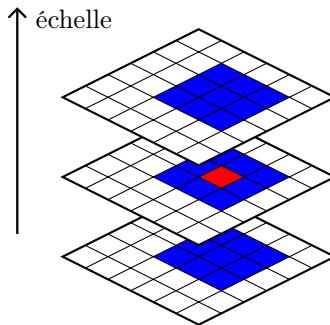


FIGURE 4 – Calcul des minimums dans la pyramide d'échelle de l'opérateur DoG

1.4 Application au site d'étude

Le site retenu est le parc naturel régional du Morvan. La forêt de feuillus est progressivement remplacée par de la monoculture intensive de douglas. Aujourd'hui, 50 % du parc est constitué de conifères. Les deux espèces radicalement différentes par leur forme et répartition fournissent un bon sujet aux applications utiles (suivi de l'évolution du parc par exemple) Les images aériennes proviennent de Géoportail ©IGN, qui fournit également des données sur la couverture forestière (espèces, etc). La sélection des paramètres décrit précédemment nécessite de tâtonner. Ils dépendent des rayons minimum et maximum des arbre et fortement de l'échelle des images utilisées. Cependant, un même jeu de paramètre permet dans l'étude réalisée de détecter deux espèces à la répartition et au rayon moyen différents. Les données de terrain et les paramètres retenus sont les suivants :

image aérienne haute résolution : <0,1m/pixels

échelle : 1 :1000

rayons douglas : 1-5m

rayons feuillus : 5-15m

5 octaves, 5 intervalles et $\sigma = 0.5$

2 Identification des espèces

L'identification d'espèce est un problème de reconnaissance de forme ("pattern recognition") pour lequel les algorithmes d'apprentissage automatique excellent. Dans le cadre de notre problème, toutes les espèces présentes sur le site du parc ... sont connues. Le choix se porte donc sur un apprentissage de type supervisé. Un algorithme de réseau de neurones a été écrit, mais ne permet pas à ce jour le traitement d'image donc la bibliothèque Tensorflow a été utilisée à la place.

2.1 Méthodologie de construction du modèle

L'algorithme précédent a été appliqué à des zones uniformément couvertes par une seule espèce et les images obtenus ont été labellisées. La base de donnée contient 173 images de douglas, 199 images de feuillus et 91 images non labellisées à des fins de vérification. Des images supplémentaires sont générées en appliquant des rotations aux images précédentes. L'architecture utilisé est une version simplifiée du modèle utilisé par , lui même adapté de ResNet, suffisante étant donnée la quantité d'image et le nombre de classes (2).

2.2 Évaluation des résultats

(Si temps)

3 Prolongements envisagables

Différents prolongement sont envisagables : -Sensible aux ... séparer préalablement et éventuellement grossièrement les zones forestières des zones d'habitation ou industrielle. Même une route bétonnée peut éventuellement altérer les résultats. batiement/contruction quo fausserait les resultat. De plus obtient qu'un cercle autour des arbres -une méthode watershed segmentation avec marqueurs que l'on a trouvé pourrait être envisagable pour délimiter plus fidèlement parfaitement les arbres (voir papier) -prendre en compte différentes étape de la croissance dans le modèle.

Références

- [NAV19] Sowmya Natesan, Costas Armenakis, and Udaya Vepakomma. Resnet-based tree species classification using uav images. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W13 :475–481, 06 2019.

A listing

```

# Importations nécessaires

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, manipulations basiques d'images

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

def luminance_distribution ( im ) :

    '''Expect a &#233;w image and return an histogram of
    ↪ luminance'''
    l,c = np.shape(im)
    hist = np.array([0]*256)
    for i in range(l) :
        for j in range(c) :
            hist[ im[i,j] ] += 1
    return hist/(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)
    return distance/2 <= r2 #attention, multiplier les
        ↪ ratios par 2
```

```

def imshowcircles (img , ax , ratio , cmap="gray") :
    ax.imshow(image , cmap)
    for i in range(len(m)) :
        x,y,h = m[ i ]
        c =
             $\hookrightarrow$  plt.Circle((y,x) ,1.4*ratio**h ,color='blue' , fill=False)
        ax.add_artist(c)
        plt.text(y , x , str(i) , color="white" ,
             $\hookrightarrow$  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" ,
                 $\hookrightarrow$  img[x-radius:x+radius+1,y-radius:y+radius+1,:])
        except :
            pass

# Convolution , noyaux de Gauss et FFT (Fast Fourier
 $\hookrightarrow$  Transform)

def kernel_Gauss(sigma , twoD=True) :
    size = 2*int(np.ceil(sigma))+3
    if twoD :
        kernel = np.fromfunction(lambda x , y:
             $\hookrightarrow$  (1/(2*np.pi*sigma**2)) *
             $\hookrightarrow$  np.exp((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))
             $\hookrightarrow$  / (2*sigma**2)) , (size , size))
    else :
        kernel = np.fromfunction(lambda x:
             $\hookrightarrow$  np.exp((-1*(x-(size-1)/2)**2) /
             $\hookrightarrow$  (2*sigma**2)) , (size ,))
    return kernel / np.sum(kernel)

def etendre(image,d):
    etendu = np.c_[ np.flip(image[:,2:2+d],1) , image ,
         $\hookrightarrow$  np.flip(image[:, -2-d+1:-1],1) ]
    etendu = np.r_[ np.flip(etendu[2:2+d],1) , etendu ,
         $\hookrightarrow$  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)) #beaucoup de
                ↪ copies , très mauvais
    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])) )
    print (np . shape (resX))
    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 (temp , sigma*ratio**i)
        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

##complexité bof...

```

```

# 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 # Important to check !

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]) # Convert
                ↪ single image to a batch.
            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]) # Convert single
        ↪ image to a batch.
    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):

    outputs = inputs
    if trace:
        derivatives = [] #Dérivé de la fonction
        ↪ d'activation
        detailed_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:
                detailed_outputs.append( outputs )
                derivatives.append(
                    ↪ self.activation_function(
                        ↪ aggregation , derivative=True ) )

        if trace : return derivatives , detailed_outputs
    return outputs

def gradient( self , inputs , expected_outputs ):

```

```

derivatives, detailed_outputs =
    ↪ self.feedforward( inputs, trace=True )
cost_derivative = self.cost_function(
    ↪ detailed_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([
    ↪ detailed_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]

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_forwarded_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)

[breaklines=true]
```

B Résultats

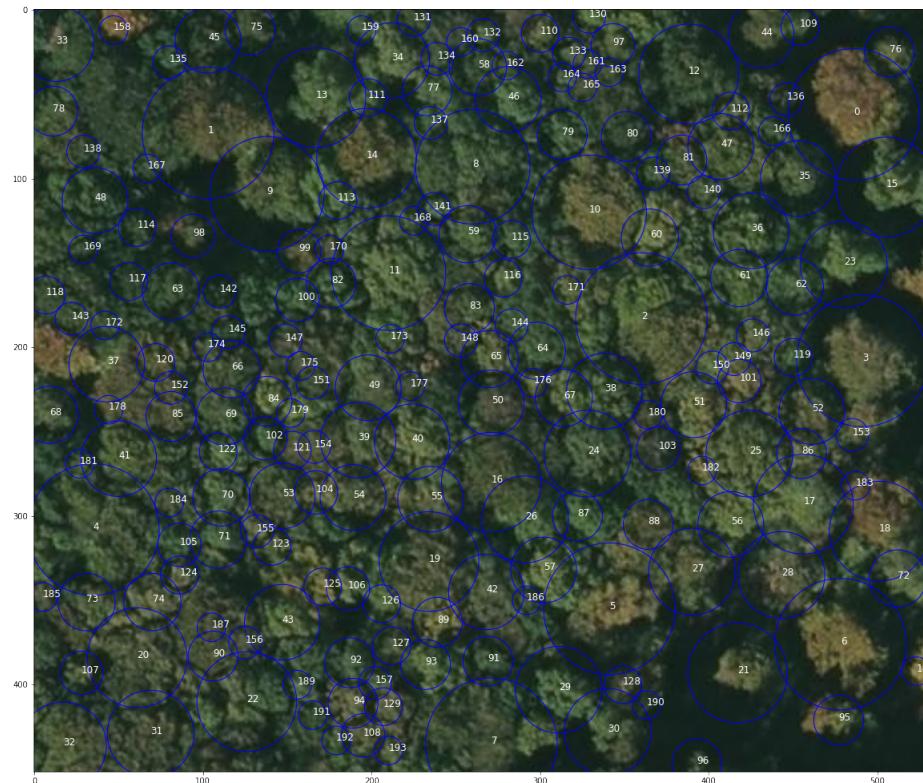


FIGURE 5 – Agencement désordonné de feuillus, ©IGN, 2021

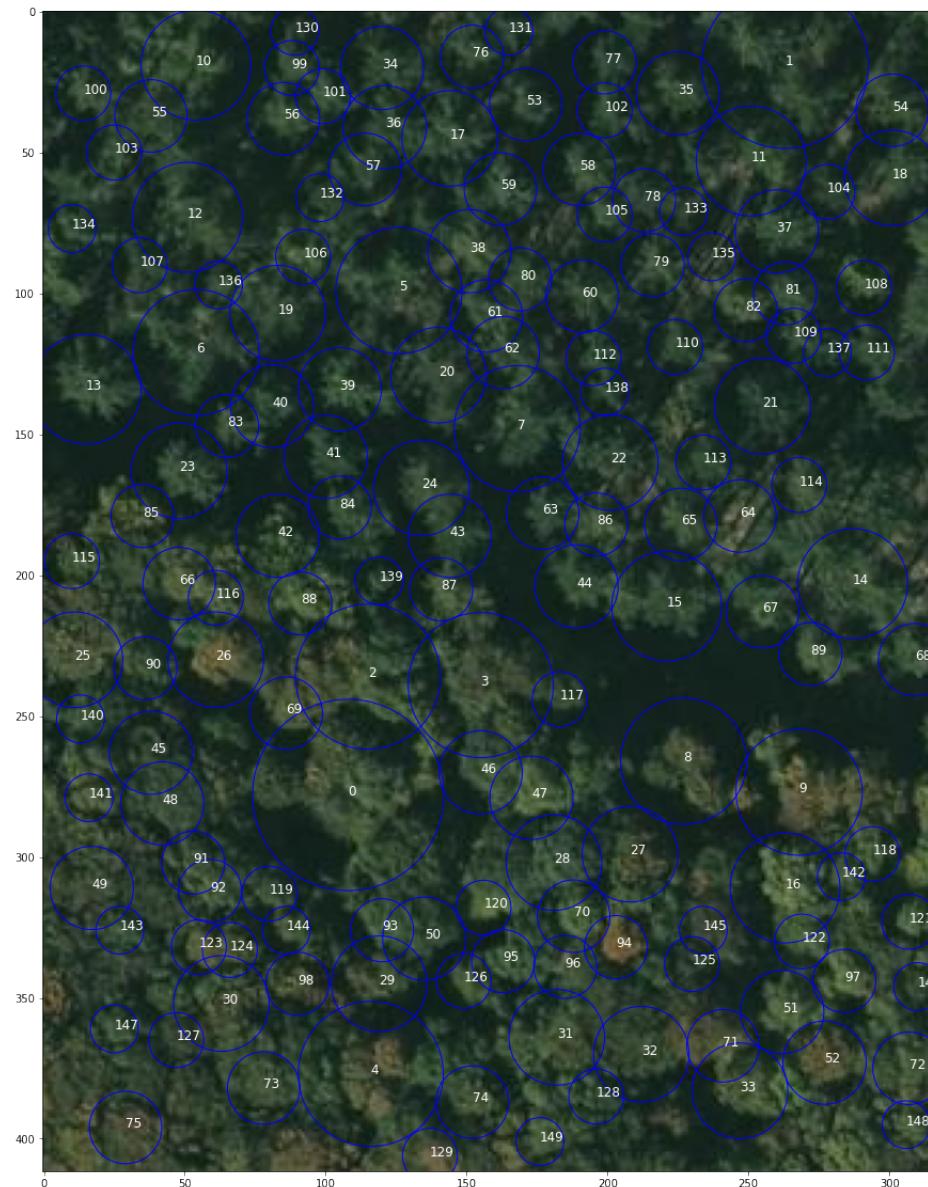


FIGURE 6 – Feuillus désordonnés et douglas semi-ordonné, ©IGN, 2021

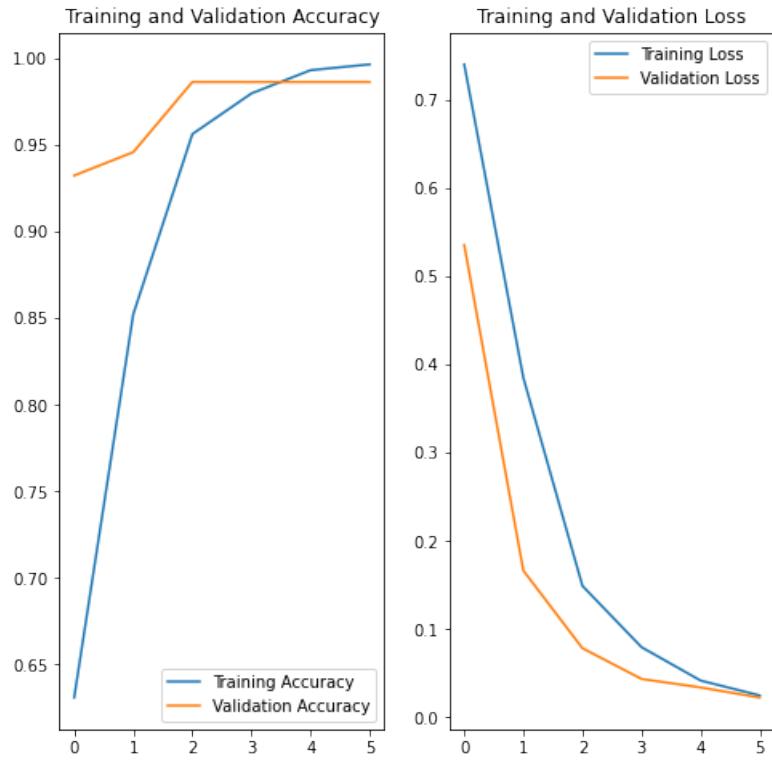


FIGURE 7 – Entraînement du modèle

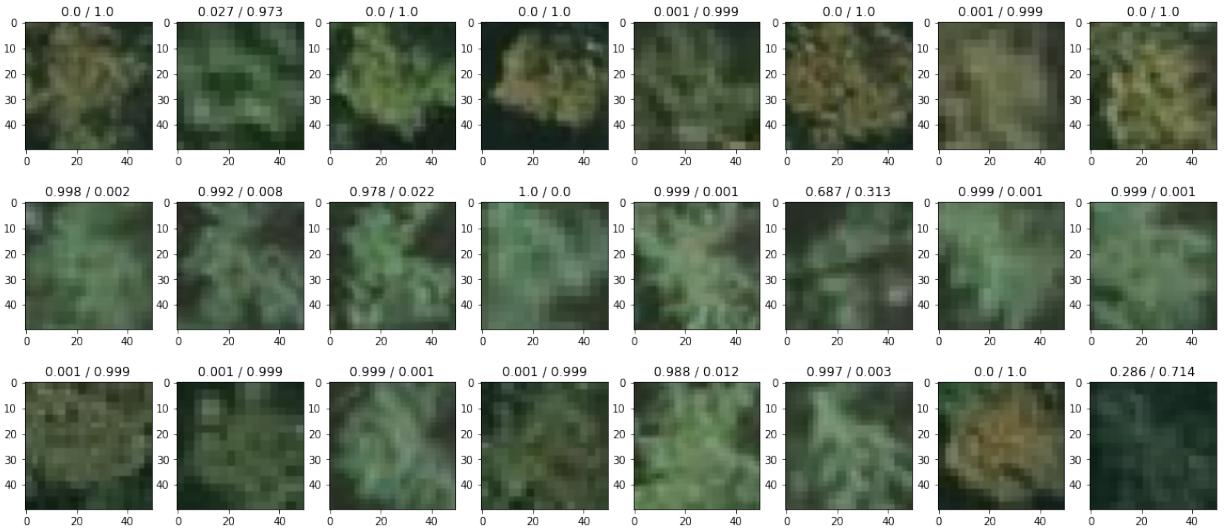


FIGURE 8 – Test du modèle : Feuillus et Douglas étiquettés et arbres non étiquetés, ©IGN, 2021 (Légende : %Douglas/%Feuillus)