

# Tutoriel DeepForest

## Détection d'arbres

### Sommaire

1. Présentation .....	1
2. Installation .....	2
3. Utilisation.....	2
3.1. Tuilage.....	3
3.2. Labélisation.....	7
3.1. Entraînement et détection .....	13

## 1. Présentation

DeepForest est un package Python de détection d'objets à partir d'images aériennes. Il comprend des modèles pour la détection des houppiers d'arbres et des oiseaux. Utilisant des réseaux de détection d'objets en apprentissage profond de machine learning, DeepForest prédit des boîtes englobantes autour des arbres dans des images RGB. Basé sur le module de détection d'objets de Torchvision (partie de la bibliothèque PyTorch - framework de machine learning open source), il simplifie l'entraînement des modèles pour la détection.

Il s'agit d'un traitement de détection d'objets et non de segmentation, aussi qu'un traitement exécuté presque en totalité à partir des codes sur une interface de codage python. C'est un type de traitement basé sur un modèle pré-entraîné avec des images de différents types d'arbres. Toutefois, il est conseillé d'effectuer un entraînement personnalisé pour améliorer et spécifier les caractéristiques du type d'arbre d'intérêt.

Son modèle de base utilise une méthode semi-supervisée où des millions d'annotations de qualité modérée sont générées à partir d'un algorithme de détection d'arbres non supervisé LiDAR. Ensuite, des annotations manuelles sont ajoutées aux images RGB sélectionnées de différents types d'arbres (Weinstein et al., 2019).

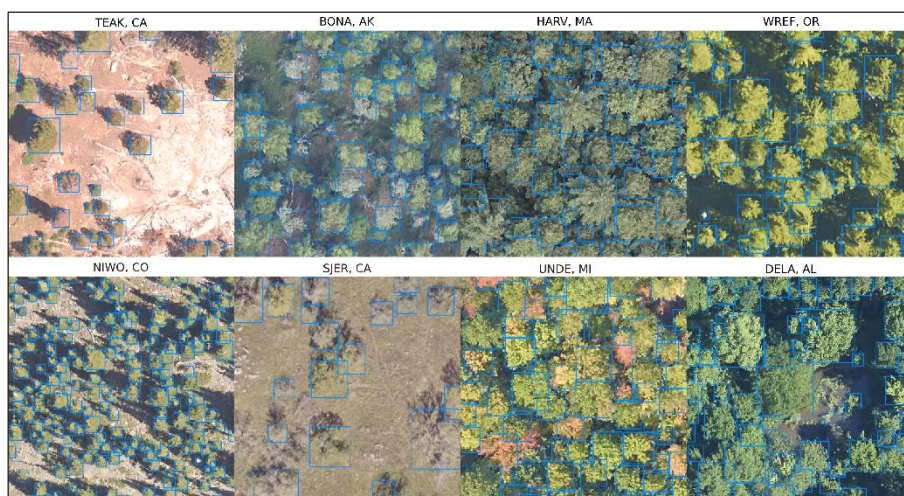


Figure 1 - Les images utilisées pour l'entraînement du modèle de base (Weinstein et al., 2019).

## 2. Installation

Presque tous les traitements sont faits en Python, sauf la labellisation, réalisée avec Label Studio, qui est aussi lancée par une commande Python et fonctionne sur un serveur local.



Python : <https://www.python.org/downloads/>



Label Studio : <https://labelstud.io/>



## 3. Utilisation

Pour cette méthode d'identification des arbres par détection, réalisée par code Python et l'interface Label Studio, trois étapes principales sont considérées :

1. **Tuilage** : L'image contenant les arbres qui serviront à entraîner le modèle est découpée en petits morceaux à l'aide de code Python.
2. **Labélisation** : Les polygones sont indiqués et nommés à l'aide de l'outil "Label Studio" pour marquer les zones où les arbres sont présents.
3. **Entraînement et application du modèle** : Le modèle de détection est entraîné, validé et appliqué à l'aide de code Python, fournissant en résultat des données sauvegardées au format vectoriel, représentant les points et les polygones des arbres détectés.

- Liens utiles :

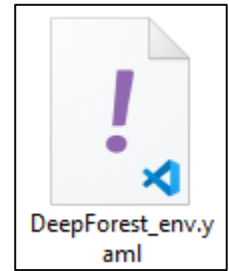
Documentation package DeepForest : <https://deepforest.readthedocs.io/en/latest/index.html>

Mode de test du modèle DeepForest en ligne : <https://huggingface.co/spaces/weecology/deepforest-demo>

Il est conseillé de l'utilisation d'outil Spyder (dans Anaconda) pour lancer les codes python.

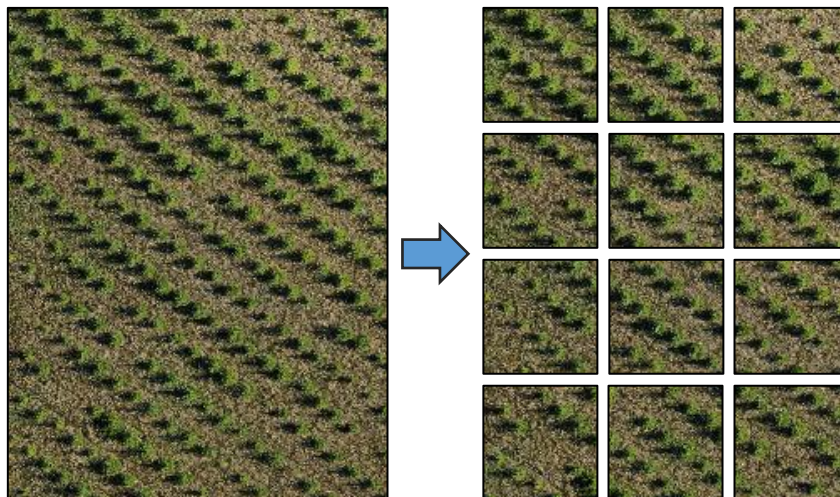
Il est possible d'ajouter directement l'environnement fournis avec toutes les dépendences des bibliothèques python nécessaires (DeepForest\_env.yaml) sur Anaconda.

- Utiliser Anaconda > charger environnement > Lancer Spyder avec le bon environnement



### 3.1. Tuillage

Dans cette première étape, le processus de découpe de l'image .tif en petites tuiles utilisables dans le processus de labellisation et entraînement sera réalisé. Il est important de définir principalement la résolution souhaitée par la valeur de taille en pixels (Partie : # Split the GeoTIFF into tiles; valeurs 1000, 1000), car cela aura une incidence directe sur la quantité de données à prétraiter.



- Instructions :

→ 1. Premièrement, toutes les bibliothèques nécessaires à cette étape sont importées. Il est nécessaire de les installer si elles ne sont pas déjà présentes dans l'environnement Python de travail. De plus, le dossier où seront trouvés et sauvegardés les fichiers d'intérêt pour ce projet (**folder**) est également défini :

```
""" ##### Cropping to labeling ##### """

import rasterio
from rasterio.windows import Window
import os
import glob
import random
from PIL import Image

Image.MAX_IMAGE_PIXELS = 1000000000

# Define folder paths
folder = r'C:\Users\DOSSIER PROPRE À L'UTILISATEUR'
input_tiff_path = rf'{folder}/Ortho_Model.tif'
output_tiff_path = rf'{folder}/Ortho_3b.tif'
```

→ 2. Dans cette deuxième partie du code, l'image initiale choisie pour l'identification et la labellisation des objets est convertie de 4 bandes à 3 bandes. Les images issues du traitement photogrammétrique par drone contiennent généralement 4 bandes, la quatrième étant une bande alpha transparente.

```
#----- Convert_4band_to_3band -----
-

def convert_4band_to_3band(input_path, output_path):
    """
    Converts a 4-band GeoTIFF to a 3-band GeoTIFF (Red, Green, Blue).
    """
    with rasterio.open(input_path) as src:
        # Read the Red, Green, and Blue bands (bands 1, 2, and 3)
        red = src.read(1)
        green = src.read(2)
        blue = src.read(3)

        # Write a new 3-band TIFF
        with rasterio.open(output_path, 'w', driver='GTiff', width=src.width,
height=src.height, count=3, dtype=src.dtypes[0]) as dst:
            dst.write(red, 1) # Write Red band
            dst.write(green, 2) # Write Green band
            dst.write(blue, 3) # Write Blue band

# Convert 4-band to 3-band GeoTIFF
convert_4band_to_3band(input_tiff_path, output_tiff_path)
```

→ 3. Ensuite, l'image est découpée en petites tuiles de la taille définie en pixels :

```
#----- Split_geotiff -----

def split_geotiff(input_file, output_folder, tile_size):
    """
    Splits a GeoTIFF into smaller tiles.
    """
    with rasterio.open(input_file) as src:
        width = src.width
        height = src.height
        tile_width, tile_height = tile_size
        num_tiles_x = width // tile_width
        num_tiles_y = height // tile_height

        if not os.path.exists(output_folder):
            os.makedirs(output_folder)

        for i in range(num_tiles_x):
            for j in range(num_tiles_y):
                window = Window(i * tile_width, j * tile_height, tile_width,
tile_height)
                tile_data = src.read(window=window)

                istr = str(i).zfill(2)
```

```

        jstr = str(j).zfill(2)
        output_filename = os.path.join(output_folder,
f"tile_{istr}_{jstr}.tif")
        create_geotiff(output_filename, tile_data, src.profile, window)

def create_geotiff(output_file, data, profile, window):
    """
    Creates a GeoTIFF file from given data.
    """
    profile['width'] = window.width
    profile['height'] = window.height
    profile['transform'] = rasterio.windows.transform(window,
profile['transform'])

    with rasterio.open(output_file, 'w', **profile) as dst:
        dst.write(data)

# Split the GeoTIFF into tiles
split_geotiff(output_tiff_path, rf'{folder}/Tuiles_crop', (1000, 1000))

```

→ 4. Ensuite, les tuiles sont converties de .tif en .png, car ce format de fichier est nécessaire pour poursuivre la labellisation avec l'outil Label Studio :

```

#----- Convert TIFF to PNG -----

def convert_tiff_list_to_png(file_list, output_folder):
    """
    Converts a list of TIFF files to PNG format and splits them into training
    and validation sets.
    """
    try:
        if not os.path.exists(output_folder):
            os.makedirs(output_folder)

        train_png_folder = os.path.join(output_folder, "Train_png")
        if not os.path.exists(train_png_folder):
            os.makedirs(train_png_folder)

        val_png_folder = os.path.join(output_folder, "Valid_png")
        if not os.path.exists(val_png_folder):
            os.makedirs(val_png_folder)

        tiff_files = glob.glob(os.path.join(file_list, "*.tif"))
        random.shuffle(tiff_files)

        num_files = len(tiff_files)
        train_split = int(0.7 * num_files)
        train_files = tiff_files[:train_split]
        val_files = tiff_files[train_split:]

        for tiff_file in train_files:
            png_file = os.path.splitext(os.path.basename(tiff_file))[0] + ".png"

```



```

train_png_path = os.path.join(train_png_folder, png_file)
with Image.open(tiff_file) as img:
    img.save(train_png_path, format='PNG')
    print(f"Converted {tiff_file} to {train_png_path}")

for tiff_file in val_files:
    png_file = os.path.splitext(os.path.basename(tiff_file))[0] + ".png"
    val_png_path = os.path.join(val_png_folder, png_file)
    with Image.open(tiff_file) as img:
        img.save(val_png_path, format='PNG')
        print(f"Converted {tiff_file} to {val_png_path}")

except Exception as e:
    print(f"Error during conversion: {e}")

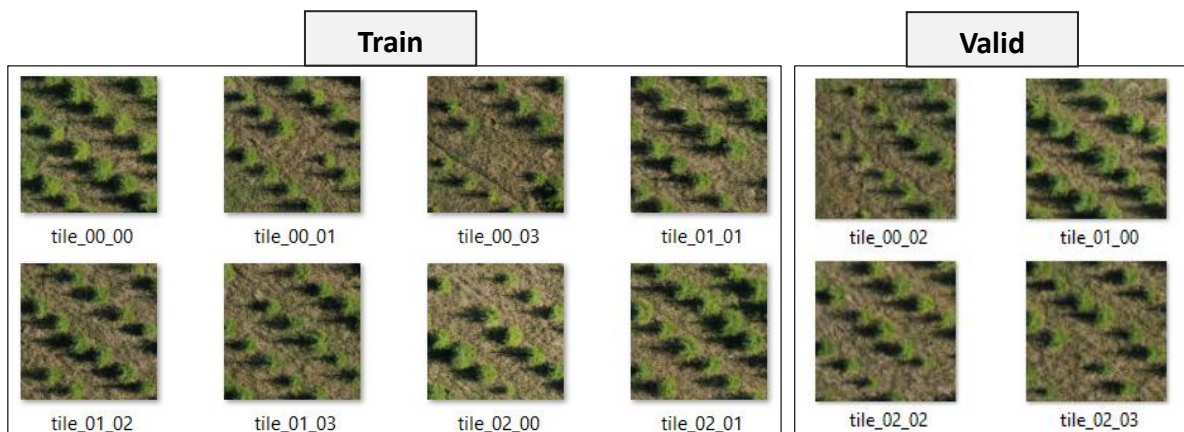
# Convert TIFF files to PNG and split into train/valid sets
tiff_files_to_convert = rf'{folder}/Tuiles_crop'
output_folder_path = folder

convert_tiff_list_to_png(tiff_files_to_convert, output_folder_path)

# After cropping the parts of the image, open the Label Studio application
(https://labelstud.io/)
# to label the trees. Use the polygon detection tool and export in JSON-MIN
format.

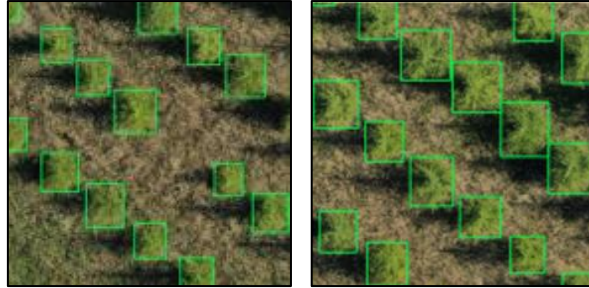
# On Anaconda Prompt
# Install the package
# into python virtual environment
!pip install -U label-studio
# Launch it!
!label-studio
  
```

À la fin, les tuiles seront exportées dans les dossiers "Valid" et "Train" avec une répartition de 70/30% (Figure 4).



### 3.2. Labélisation

Cette étape consiste à indiquer sur les images la nature de chaque objet, ce qui permettra au modèle d'être entraîné selon nos instructions. Avec l'outil Label Studio, qui permet de délimiter la région contenant l'objet et d'attribuer un nom à cette délimitation.



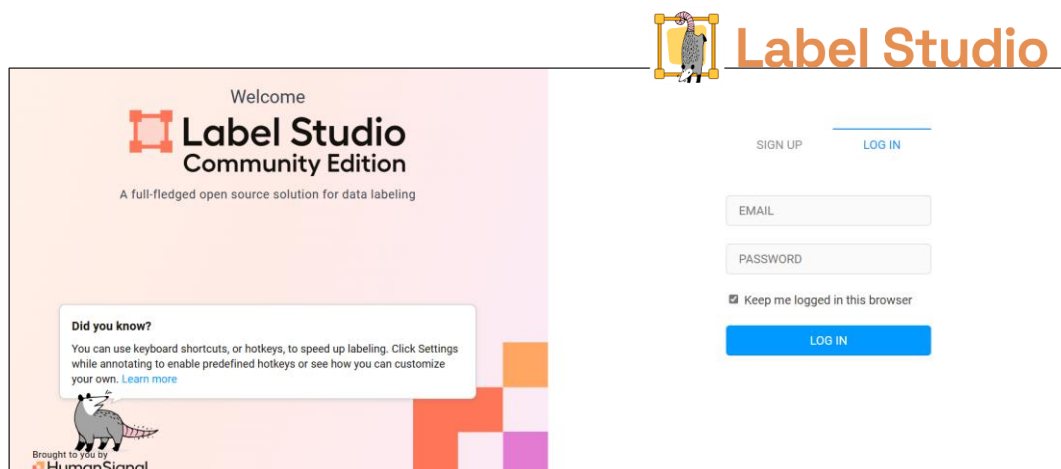
## Label Studio

- Instructions :

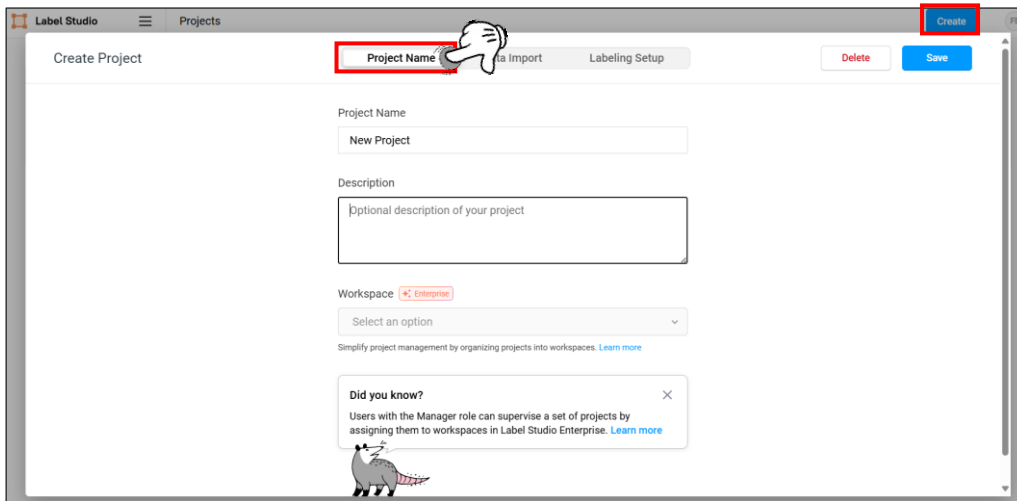
→ 5. Lancez l'outil via un code Python pour l'ouvrir sur un serveur local (<http://localhost:8080/user/login/>) :



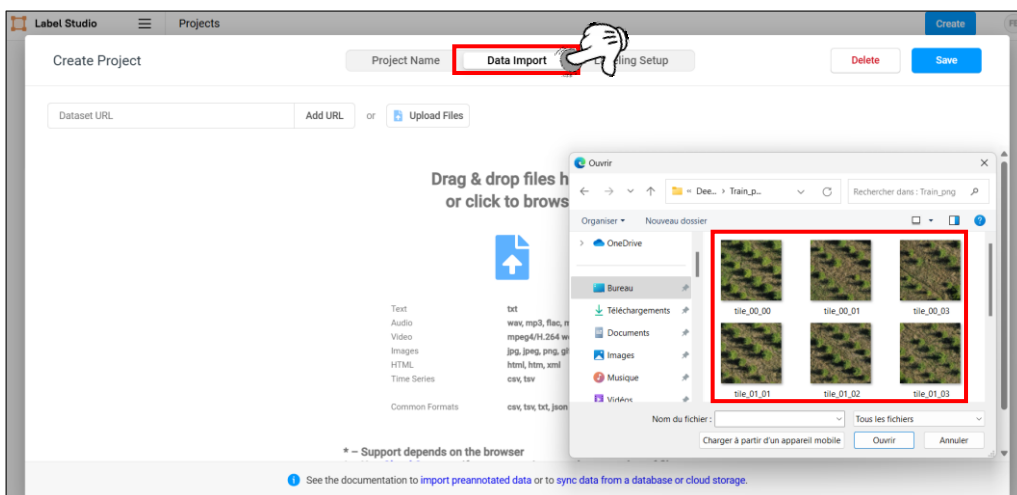
→ 6. Créez un compte sur Label Studio : <https://labelstud.io/>.



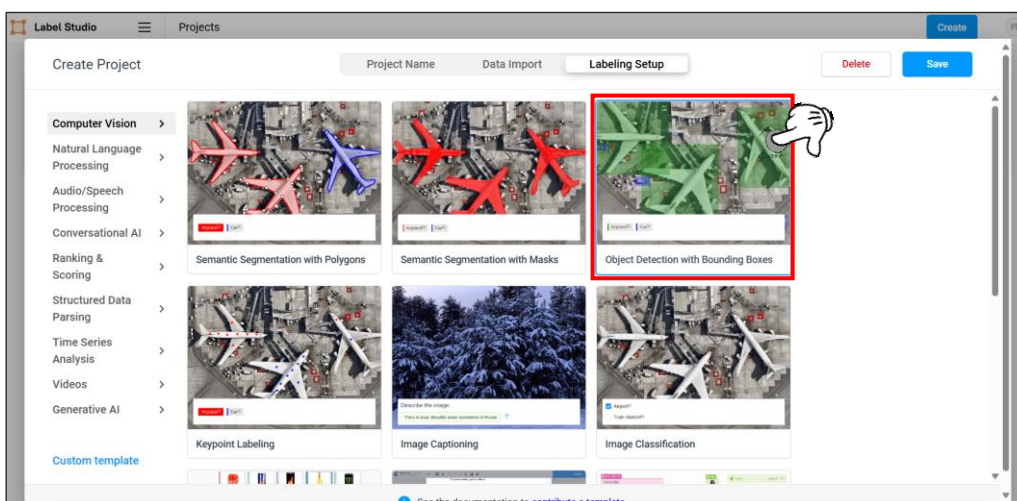
→ 7. Créez un nouveau projet. Faites les étapes suivantes en créant des projets distincts pour Entraînement (Train) et Validation (Valid).



→ 8. Allez sur "Data Import" et téléchargez vos tuiles. Vous voudrez peut-être les examiner et supprimer celles qui ne montrent pas votre objet.

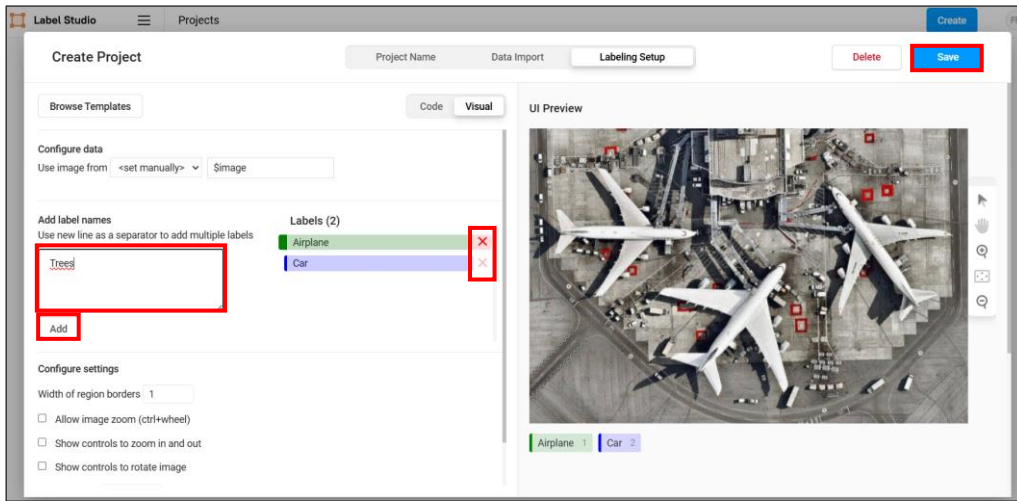


→ 9. Appuyez sur le type de labelisation par "Object Detection with Boxes".











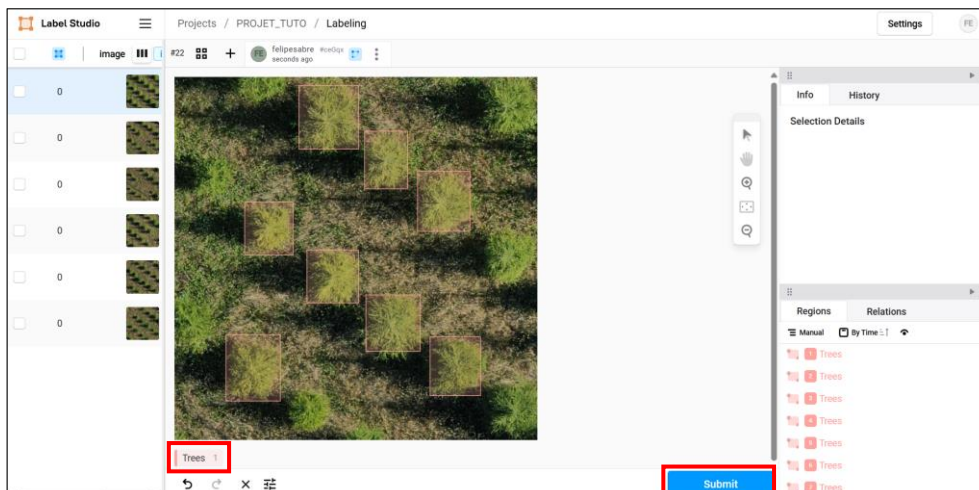
→ 10. Supprimez les légendes de labellisation déjà proposées et créez vos propres légendes, puis enregistrez le projet en cliquant sur "Save".



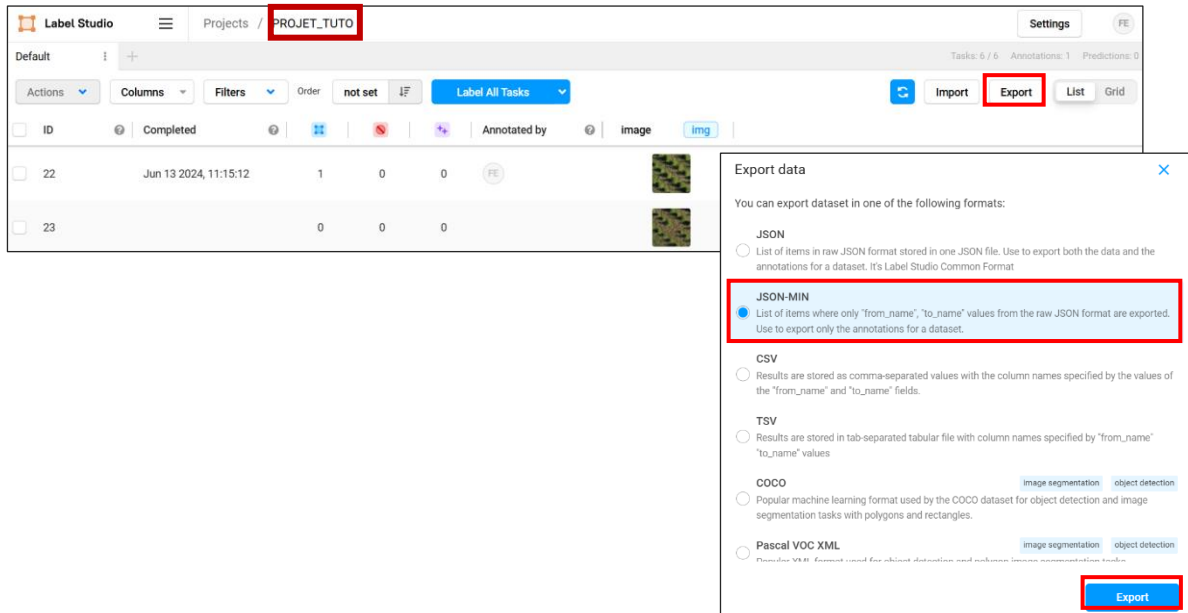
→ 11. Cliquez sur les images une par une pour commencer la labellisation.

ID	Completed				Annotated by	image
22	0	0	0			
23	0	0	0			
24	0	0	0			
25	0	0	0			
26	0	0	0			
27	0	0	0			

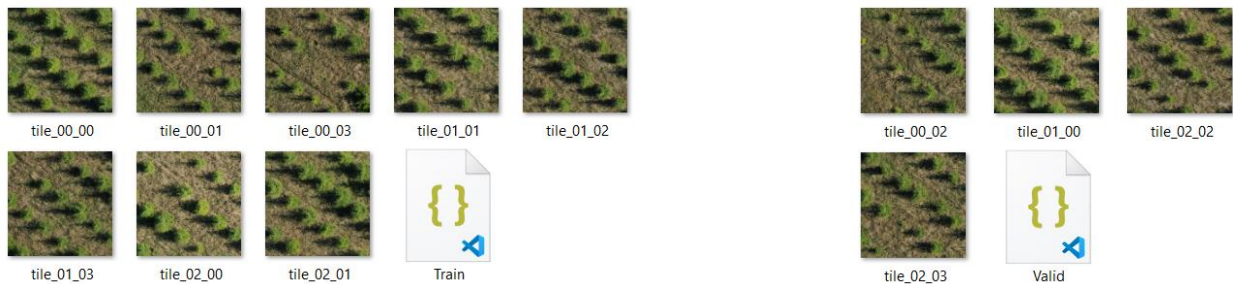
→ 12. Pour commencer la labellisation, cliquez sur la légende portant le nom attribué à la classe d'identification et créez un polygone pour identifier l'objet dans l'image. Vous pouvez également simplement appuyer avec le clavier le numéro de la légende indiqué (1, 2, 3...), pour accélérer le processus. Lorsque vous avez terminé l'annotation, appuyez sur "Submit" et passez à la prochaine image.



→ 13. Après avoir terminé la labellisation de toutes les images, cliquez sur le nom du projet pour revenir à page du projet et pouvoir exporter "Export". Sélectionnez le mode "JSON-MIN" pour l'exportation.



→ 14. Après avoir effectué la labellisation et exporté les fichiers JSON-MIN pour les données Train et Valid, placez les fichiers dans leurs dossiers respectifs en les renommant en "Train.json" et "Valid.json"



→ 15. Importez toutes les bibliothèques nécessaires à cette étape. Assurez-vous de les installer si elles ne sont pas déjà présentes dans l'environnement Python de travail. Les dossiers et fichiers sont déjà prédéterminés avec leurs noms et chemins respectifs dans le code précédent, vous n'avez donc pas besoin de les spécifier à nouveau :

```
""" ##### After Labeling ##### """

#----- JSON-MIN to CSV -----

import csv
import json
import re
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from PIL import Image

# Paths for training and validation JSON and CSV files
TrainJSON = os.path.join(folder, 'Train_png', 'Train.json')
TrainCSV = os.path.join(folder, 'Train_png', 'Train.csv')
ValidJSON = os.path.join(folder, 'Valid_png', 'Valid.json')
ValidCSV = os.path.join(folder, 'Valid_png', 'Valid.csv')
```

→ 16. Dans cette partie, le fichier de labellisation est converti de .JSON à .CSV afin qu'il puisse être lu pour le processus d'entraînement du modèle à l'aide de la bibliothèque DeepForest.

```
# Function to convert JSON to CSV
def json_to_csv(json_path, csv_path):
    # Load JSON data
    with open(json_path) as json_file:
        data = json.load(json_file)

    # Write data to CSV format
    with open(csv_path, 'w', newline='') as csvfile:
        # Create CSV writer object
        csv_writer = csv.writer(csvfile)

        # Write CSV header
        header = ["image_path", "xmin", "ymin", "xmax", "ymax", "label"]
        csv_writer.writerow(header)

        # Process each item in data
        for item in data:
            # Check if the 'image' key exists
            if "image" in item:
                # Extract the image filename using regular expression
                match = re.search(r'tile_(.*).png', item["image"])

                # Check if there is a match, otherwise ignore the item
                if match:
                    # Access the matched group (filename) using .group(1)
                    image_file = "tile_" + match.group(1) + ".png"

                    # Iterate over each label in the item
                    for label in item["label"]:
                        # Get the original image size
                        x_size = label["original_width"]
                        y_size = label["original_height"]

                        # Calculate bounding box coordinates
                        xmin = round(label["x"] * x_size / 100)
                        ymin = round(label["y"] * y_size / 100)
                        xmax = xmin + round(label["width"] * x_size / 100)
                        ymax = ymin + round(label["height"] * y_size / 100)

                        # Set class name (assuming all are "Tree")
                        label_name = "Tree"

                        # Write a CSV row for each label
                        csv_writer.writerow([image_file, xmin, ymin, xmax, ymax,
label_name])
                    else:
                        # Handle the case where there is no 'image' key
                        print(f"Warning: Item {item} does not have an 'image' key")
                print(f"Conversion successful. Data saved to {csv_path}.")

# Convert training and validation JSON files to CSV
json_to_csv(TrainJSON, TrainCSV)
json_to_csv(ValidJSON, ValidCSV)
```

→ 17. Un plot de quatre tuiles aléatoires est réalisé, avec les polygones indiqués par la labellisation, pour une visualisation rapide afin de vérifier que tout est correct avec l'étape de labellisation.

```

#----- Plot with shapes -----

# Function to plot images with bounding boxes from CSV
def plot_with_shapes(csv_path, image_folder):
    # Initialize the previous image name
    previous_image_name = None

    # Read bounding box coordinates from CSV file
    with open(csv_path, 'r') as csvfile:
        reader = csv.DictReader(csvfile)

        for row in reader:
            # Get the image path from the 'image_path' column
            image_path = os.path.join(image_folder, row['image_path'])

            # Check if the image name has changed
            if image_path != previous_image_name:
                # Display plots on the previous figure
                if previous_image_name is not None:
                    plt.show()

                # Open the new image
                image = Image.open(image_path)
                # Create a figure and axis
                fig, ax = plt.subplots()
                # Display the new image
                ax.imshow(image)
                # Hide the axes
                ax.axis('off')
                # Update the previous image name
                previous_image_name = image_path

            # Get bounding box coordinates
            rectangles = [(int(row['xmin']), int(row['ymin']), int(row['xmax']),
int(row['ymax']))]

            # Add rectangles as overlays
            for rect_coords in rectangles:
                xmin, ymin, xmax, ymax = rect_coords
                rectangle = patches.Rectangle((xmin, ymin), xmax - xmin, ymax -
ymin, linewidth=2, edgecolor='r', facecolor='none')
                ax.add_patch(rectangle)

            # Force display on the last entry
            plt.show()

# Plot training images with bounding boxes
plot_with_shapes(TrainCSV, os.path.join(folder, 'Train_png'))

# Plot validation images with bounding boxes
plot_with_shapes(ValidCSV, os.path.join(folder, 'Valid_png'))
  
```

### 3.1. Entraînement et détection

Cette étape consiste à entraîner et appliquer le modèle pour procéder à la détection de l'image souhaitée. Il est intéressant d'appliquer la détection sur une zone différente de celle utilisée pour l'entraînement, mais présentant des caractéristiques similaires, car l'identification repose sur des cibles avec des caractéristiques préalablement définies. Cette étape se déroule toujours en Python.

- Instructions :

→ 18. Importez toutes les bibliothèques nécessaires à cette étape. Assurez-vous de les installer si elles ne sont pas déjà présentes dans l'environnement Python de travail. Les dossiers et fichiers sont déjà prédéterminés avec leurs noms et chemins respectifs dans le code précédent, vous n'avez donc pas besoin de les spécifier à nouveau :

```

""" ##### Create model and apply ##### """
#----- Libraries + Deepforest Base Model -----
-----
# n cas de problème avec l'installation de
la bibliothèque DeepForest → pip install deepforest pydantic (version 2.20)

import os
from shapely.geometry import Polygon, Point
from geopandas import GeoDataFrame, GeoSeries
from deepforest import main

```

→ 19. Ici, le modèle de base pré-entraîné de DeepForest est chargé à des fins purement illustratives pour montrer le résultat potentiel sans entraîner le modèle avec des données personnalisées. À la fin de cette partie sont affichés les résultats de détection effectuée par ce modèle.

```

# Initialize the DeepForest model
model = main.deepforest()
model.use_release()

#----- Test Predictions -----

# Get a list of image files in the training folder
train_images = [f for f in os.listdir(os.path.join(folder, 'Train_png')) if
f.endswith('.png')]

# Select four random images
random_images = random.sample(train_images, 4)

# Perform predictions on the selected images before training
test_images_before = [model.predict_image(path=os.path.join(folder, 'Train_png',
img), return_plot=True, color=(0, 0, 255), thickness=3) for img in random_images]

# Plot the predictions before training
plt.subplots(figsize=(12, 3))
plt.axis('off')
for i, img in enumerate(test_images_before):
    plt.subplot(1, 4, i + 1)
    plt.imshow(img[:, :, ::-1])
plt.show()

```



→ 20. Ensuite, les tuiles et le fichier .csv de labellisation sont importés pour entraîner le modèle. À ce stade, il est possible de paramétrer les configurations d'entraînement ainsi que les paramètres spécifiques au système utilisé, comme l'utilisation du GPU ou du CPU pour le traitement.

```

#----- Training Model -----

# Define paths for training and validation CSV files
train_file = os.path.join(folder, 'Train_png', 'Train.csv')
eval_file = os.path.join(folder, 'Valid_png', 'Valid.csv')

# Configure the model parameters
model.config["accelerator"] = "cpu"
model.config["save-snapshot"] = False
model.config["train"]["epochs"] = 5
model.config["train"]["csv_file"] = train_file
model.config["train"]["root_dir"] = os.path.dirname(train_file) + "/"
model.config["batch_size"] = 1
model.config["validation"]["csv_file"] = eval_file
model.config["validation"]["root_dir"] = os.path.dirname(eval_file) + "/"
model.config["train"]["fast_dev_run"] = False

# Create and train the model
model.create_trainer()
model_fname = os.path.join(folder, 'Model')

if os.path.exists(model_fname):
    model = main.deepforest.load_from_checkpoint(model_fname)
else:
    model.trainer.fit(model)
    # Save the trained model
    model.trainer.save_checkpoint(model_fname)
    
```

→ 21. La prédiction de détection est réalisée sur le nouveau modèle créé, uniquement pour les mêmes quatre tuiles précédemment choisies aléatoirement à des fins de démonstration. Les résultats des quatre tuiles précédentes ainsi que des quatre nouvelles sont également affichées, pour comparer la détection des deux modèles.

```

#----- Post-training Predictions -----

# Evaluate the model
res_val = model.trainer.validate(model)
results = model.evaluate(eval_file, os.path.dirname(eval_file), iou_threshold=0.4)
print(f"Recall: {results['box_recall']}, Precision: {results['box_precision']}")

# Perform predictions on the same images after training
test_images_after = [model.predict_image(path=os.path.join(folder, 'Train_png',
img), return_plot=True, color=(77, 255, 0), thickness=3) for img in random_images]

# Plot the predictions before and after training
fig, axarr = plt.subplots(2, 4, figsize=(12, 7)) # Create a figure and a set of
subplots
fig.suptitle('Predictions Before and After Training', fontsize=16)

for i, img in enumerate(test_images_before):
    axarr[0, i].imshow(img[:, :, :-1])
    axarr[0, i].axis('off')
    
```

```

axarr[0, i].set_title(f'Before {i+1}')

for i, img in enumerate(test_images_after):
    axarr[1, i].imshow(img[:, :, :-1])
    axarr[1, i].axis('off')
    axarr[1, i].set_title(f'After {i+1}')

# Save the plot to a file
plt.savefig(os.path.join(folder, 'Plot_before_after.png'))
plt.show()
plt.close()

```

→ 22. Le nouveau modèle est ensuite appliqué à l'image complète, et les résultats sous forme de vecteurs de polygones de détection et de points de centroides des polygones sont enregistrés dans le dossier du projet.

```

#----- Predict on Entire Image -----

raster_path = os.path.join(folder, 'Ortho_3b.tif')
predicted_raster = model.predict_tile(raster_path, return_plot=True,
patch_size=1000, color=(255, 77, 0), thickness=3, patch_overlap=0.25)
prw = model.predict_tile(raster_path, return_plot=False, patch_size=1000,
patch_overlap=0.25)

#----- Save Results -----

import os
import matplotlib.pyplot as plt

#----- Load Image -----

# Open the initial image
with rasterio.open(input_tiff_path) as src:
    # Get the coordinate reference system (CRS)
    crs = src.crs
    # Get the transformation of the image
    transform = src.transform
    # Other relevant information can be obtained, such as width, height, etc.

#----- Process Results -----

# Convert each point to geographical coordinates
geopositions_points = [] # For points
geopositions_polygons = [] # For polygons

for index, row in prw.iterrows():
    # Get pixel coordinates of the result
    xmin, ymin, xmax, ymax = row['xmin'], row['ymin'], row['xmax'], row['ymax']
    # Calculate the geographical coordinates of the center of the bounding box
    x_center = (xmin + xmax) / 2
    y_center = (ymin + ymax) / 2
    lon, lat = transform * (x_center, y_center)
    # Add the point to the list of points
    geopositions_points.append(Point(lon, lat))
    # Calculate the geographical coordinates of the corners of the bounding box
    lon1, lat1 = transform * (xmin, ymin)
    lon2, lat2 = transform * (xmin, ymax)
    lon3, lat3 = transform * (xmax, ymax)

```

```

lon4, lat4 = transform * (xmax, ymin)
# Create a polygon with these coordinates
polygon = Polygon([(lon1, lat1), (lon2, lat2), (lon3, lat3), (lon4, lat4)])
# Add the polygon to the list of polygons
geopositions_polygons.append(polygon)

# Add geographical coordinates to the results
prw['lon'] = [point.x for point in geopositions_points]
prw['lat'] = [point.y for point in geopositions_points]

# Create a GeoDataFrame with the geopositioned results (points)
prw_geopositioned_points = GeoDataFrame(prw, geometry=geopositions_points, crs=crs)

# Save the geopositioned results (points) in a geospatial format
prw_geopositioned_points.to_file(os.path.join(folder,
'arbres_geopositioned_points.shp'))

# Create a GeoDataFrame with the geopositioned results (polygons)
prw_geopositioned_polygons = GeoDataFrame(prw, geometry=geopositions_polygons,
crs=crs)

# Save the geopositioned results (polygons) in a geospatial format
prw_geopositioned_polygons.to_file(os.path.join(folder,
'arbres_geopositioned_polygons.shp'))

```

→ 23. Ensuite, les données statistiques des résultats sont calculées et affichées dans la console, ainsi qu'un histogramme tracé pour visualisation.

```

#----- Calculate Characteristics -----

# Calculate characteristics of the bounding boxes
prw['largeur'] = prw['xmax'] - prw['xmin']
prw['hauteur'] = prw['ymax'] - prw['ymin']
prw['superficie'] = prw['hauteur'] * prw['largeur']
resolution = 1.3 * 1.3 # Resolution of the drone is 1.3cm / pixel
prw['taille'] = [(round(row['superficie'] / resolution / 1000) * 1000 / 100) for
index, row in prw.iterrows()]
#----- Visualize Data -----

# Visualize the histogram of tree areas
plt.hist(prw['taille'], bins=20, edgecolor='black', alpha=0.7)
plt.title('Distribution of leaf area')
plt.xlabel('Values in dm2')
plt.ylabel("Number of trees")
plt.grid(True)
plt.show()

#----- Calculate Geographical Coordinates -----

# Open the dataset for geographical coordinate calculation
dataset = rasterio.open(raster_path)

# Calculate geographical coordinates
x_center = (prw.iloc[0]["xmax"] + prw.iloc[0]["xmin"]) // 2
y_center = (prw.iloc[0]["ymax"] + prw.iloc[0]["ymin"]) // 2
x_diam = (prw.iloc[0]["largeur"])
y_diam = (prw.iloc[0]["hauteur"])

```

```

diam = (x_diam + y_diam) // 2
pt_center = (x_center, y_center)
pt_geo = dataset.transform * pt_center
pt_gps = GeoSeries([Point(pt_geo[0], pt_geo[1])],
crs='EPSG:32631').to_crs('epsg:4326')[0].coords[0]

# Display information
print(f"Coordinate system: {dataset.crs}")
print(f"Center in pixel value (x,y): {pt_center}")
print(f"Diameter in pixels (x,y) and average: {x_diam}, {y_diam}, {diam}")
print(f"Position in UTM coordinates: {pt_geo}")
print(f"Position in GPS coordinates (lon, lat): {pt_gps}")

```

→ 24. Il est également possible d'exporter les résultats des polygones au format .csv et .xlsx.

```

#----- Optionnel : resultats en mode csv et xlsx -----
# Save results to Excel and CSV
prw.to_excel(os.path.join(folder, 'resultats.xlsx'))
prw.to_csv(os.path.join(folder, 'resultats.csv'), index=False)

```

#### Résultat :

Les résultats sous forme de vecteurs de polygones et de points de centroides peuvent être visualisés dans le logiciel QGIS, en ajoutant la couche vecteur et le raster "Orthomosaique.tif".



- **Conclusion :**

Ce tutoriel présente une méthode simple pour entraîner et appliquer le traitement de detection basé sur l'objet. Il est important de rappeler que la fiabilité et les performances sont meilleures lorsque les données utilisées pour l'entraînement du modèle sont plus nombreuses et de meilleure qualité. Cependant, cela montre qu'il peut y avoir encore des erreurs. Il est donc nécessaire d'étudier la cause de ces erreurs, de faire des tests et d'améliorer l'entraînement des modèles.