A prototype for automated crop boundary delineation with a focus on rice paddy fields

Training Guide
Output 1

Issue 1.0
July 2024

Prepared by:

University of Twente

Changelog

Issue	Changes	Delivered
1.0	Initial version	23.07.2024

Contents

1 Introduction

This document provides a step-by-step training guide for operating the prototype designed for crop boundary delineation (prototype accessible here). The following sections describe all the steps required to generate the output product, starting with the initial downloading of the satellite data used to delineate the crop boundaries.

This documentation begins with an overview of the prototype's architecture in Section 2. Section 3 offers a step-by-step user guide for executing the prototype code. Section 4 covers the installation process, while Section 5 provides an overview of frequently asked questions. The appendix lists all the libraries used in the prototype, along with their versions.

2 Overall Architecture of Prototype

To generate the crop boundaries delineation results, the Sentinel-2 images for the 62 sample plots distributed in both Cambodia (32 sample plots) and Viet Nam (30 sample plots) were downloaded from Google Earth Engine (GEE). To train and test the proposed U-Net deep learning segmentation model, the dataset was divided into train, validation, and test sets for both locations (see [1] for more details). A modified version of U-Net architecture called Satellite U-Net [2] was implemented using the TensorFlow library to delineate the field boundaries, which were then post-processed in two steps to improve the segmentation results. In the first post-processing step, a morphological segmentation approach was used to close the field boundaries. In the second post-processing step, the raster images were vectorized to create the final geospatial database. Figure 1 shows the overall workflow of the developed prototype. The details of the individual steps are given in the following sections of this document.

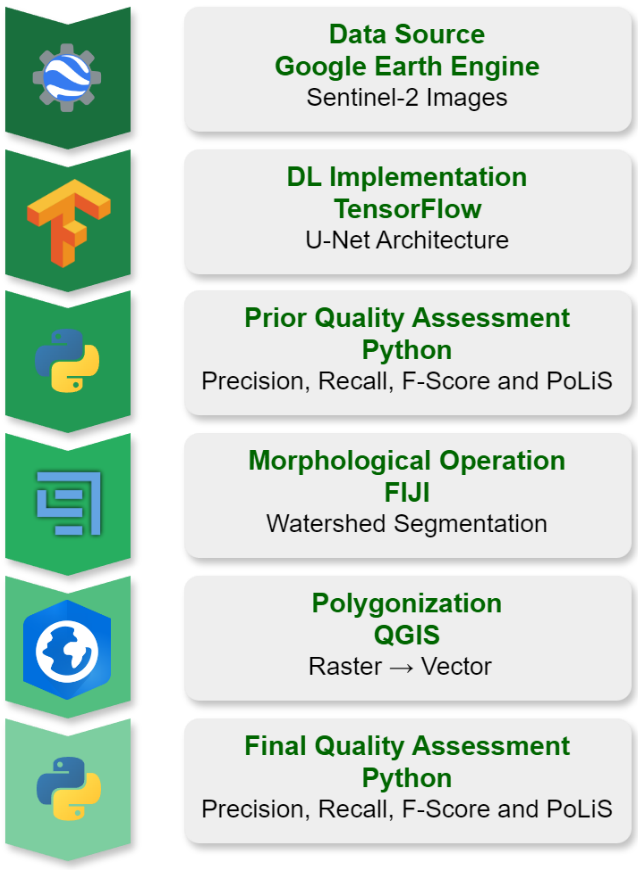


Figure 1. The overall workflow of the prototype. For each stage, the platform used is reported.

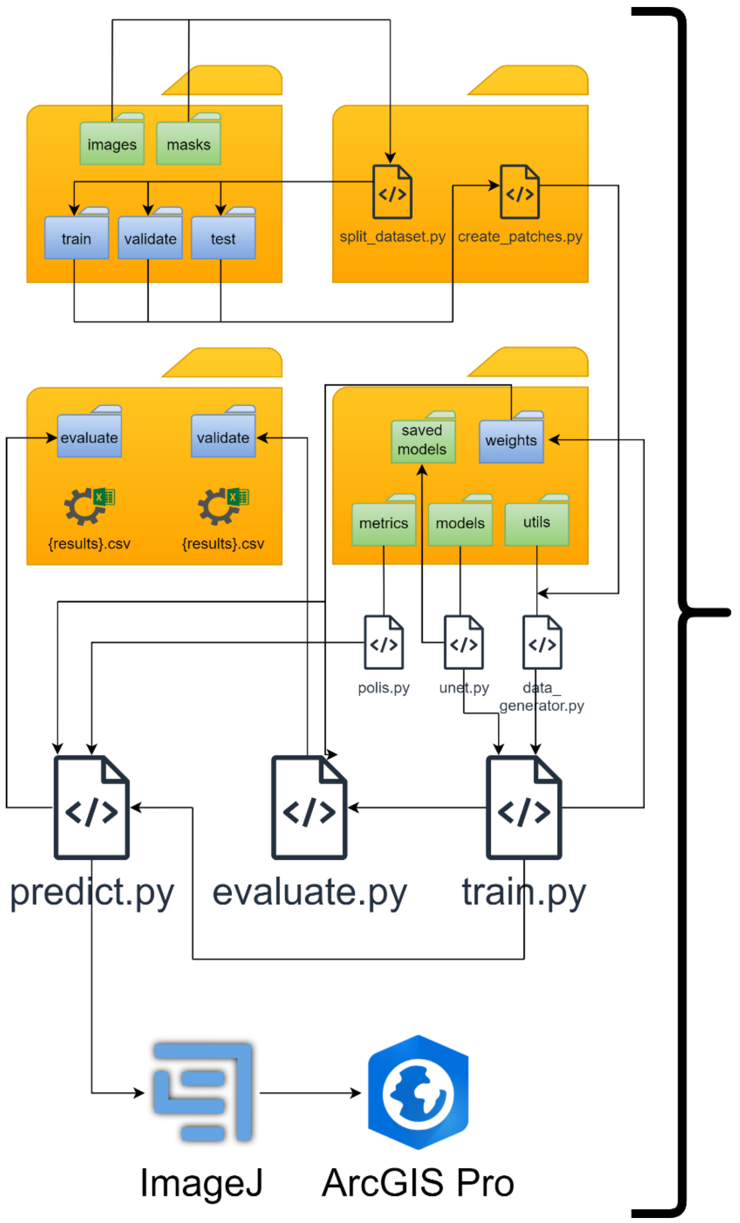


Figure 2. Visual representation of the arrangement of folders and files within the prototype structure.

A visual representation of the arrangement of folders and files within the prototype structure is reported in Figure 2. In the main folder, there are 3 subfolders (**input**, **deep_learning_models,** and **preprocess)** and the following 5 scripts:

1. **download_sentinel2_data.ipynb:** used to download Sentinel-2 images from GEE;
2. **run_dl_pypeline.py**: used to instantiate the U-Net model architecture;
3. **base.py**: which includes the configuration parameters;
4. **train.py**: used to train the deep learning model;
5. **evaluate.py**: used to compute the accuracy metrics;
6. **predict.py**: used to generate the delineation results in all the sample plots.

In the **input** folder, the **tileAsia.gpkg** file reports the location of the 62 sample plots in Cambodia and Viet Nam and their split into training, test and validation sets. In the **preprocess** folder, there are the python codes used to prepare the Sentinel-2 data:

7. **split dataset.py**: to split the dataset into training, test, and validation sets.

8. **create patches.py**: to divide large images into smaller patches.

In the **deep_learning_models** folder, the prototype stores the untrained models in the **saved_models** subfolder with the name of specified version of the U-Net model and channel number and trained model is stored in the **weights** subfolder and leverages the following scripts present the subfolders **utils**, **metrics** and **models**:

9. **data_generator.py:** used to generate normalised patches and reference data;
10. **polis.py**: used to calculate the accuracy metric for vector data;
11. **unet.py**: includes both vanilla and modified version of the U-Net architecture for satellite images.

3 Step-by-step User Guide

3.1 Satellite data download and preprocess

The first step aims to download the Sentinel-2 images used to generate the crop boundary results. To this end, we used the GEE platform, which requires a user registration considering the following link https://code.earthengine.google.com/register.

Initialize and authenticate GEE with the activation code provided by GEE at the beginning of the **download_sentinel2_data.ipynb** script to download Sentinel-2 images. For each Sentinel-2 image, only the RGB (B2, B3, B4) and NIR (B8) bands along with the scene classification map (SCL) were downloaded. This study utilises Sentinel-2 images with less than 40% cloud coverage (a variable name given as 'cloud_th' in the **download_sentinel2_data.ipynb** script). The SCL band of each Sentinel-2 image was used to filter the cloud, cloud shadows, and cirrus from the scene. To create cloud-gap-free Sentinel-2 data, for each sample plot, a monthly composite was generated computing the median spectral value of all the downloaded Sentinel-2 images for that specific month. The variable 'geometry' specifies the region of interest in the form of the geographic coordinates of corner points. The variables' start_date' and 'stop_date' define the temporal coverage of the image collection to be used for aggregation. A preprocessing step to eliminate potential Not a Number (NaN) values and spectral outliers is recommended when downloading Sentinel-2 data for different study areas or time ranges. We suggest using the standard procedure of saturating pixel values below the 1st percentile and above the 99th percentile of the spectral distribution computed for each band.

3.2 Code Implementation Details

The main file in the prototype is the base.py which controls the deep learning model and its parameters with classes and functions. Before running the prototype, please ensure that the directories and parameters are correctly specified.

****

```
#%% Configure the model parameters
config = base.Base()
```

There are four classes under 'base.py' which are Area, Source, SelectedModel and Base. Only the parameters within this class Base can be modified. To run the prototype, do not change any definitions. The class Base contains hyperparameters that can be changed by the user to test different hyperparameters. Changeable parameters: IMAGE_SIZE, NUM_CHANNELS, MODEL, FROM_SCRATCH, BATCH_SIZE, LEARNING_RATE, EPOCHS, PATIENCE and BIN_THRESHOLD.

```
class Base:
    INPUT_DIR = r'' (path of the working directory)
    AREA = Area.ASIA (name of the study area)
    SOURCE = Source.S2 (image source that used to train the model)
    IMAGE_SIZE = 256 (image size for creating patches)
    NUM_CHANNELS = 4 (band number of the images)

    MODEL = SelectedModel.SATELLITE_UNET (u-net model for satellite images)
    FROM_SCRATCH = True (in case of the usage pretrained model set it False)
    BATCH_SIZE = 8 (number of training samples used in one iteration to update model
parameters)
    LEARNING_RATE = 1e-4 (controls the step size at each iteration during training)
    EPOCHS = 1000 (the maximum number of iteration)
    PATIENCE = 10 (controls the model training and stops training if there is no
significant change after 10 epochs)
    BIN_THRESHOLD = .5 (threshold for determining a pixel is a boundary or not)
```

****

```
#%% Split dataset as train-test-validate
split_dataset_obj = SplitDataset()
split_dataset_obj.split_dataset()
```

To run the prototype, the images and masks must be divided into training, testing, and validation sets. To facilitate this, a script named 'split_dataset.py' has been created. This script uses a user-defined GeoPandas file to split the images and masks accordingly. The split_dataset.py script looks for a '.gpkg' file in a subfolder named 'input'. The file path must be specified in the split_dataset function.

```
    def split_dataset(self):
        tiles_gdf = gp.read_file(r'input/tilesAsia.gpkg') ()
        tiles_gdf = tiles_gdf[['id', 'split', 'country']]
```

The '.gpkg' file should include an ID for each tile, a column indicating whether the tile is used for training, testing, or validation, and a column specifying the country name as the study area. Once the script has been run for a dataset, it does not need to be run again, as the training, testing, and validation datasets are created and saved in their respective subfolders.

****

```
#%% Create image patches
create_patches_obj = CreatePatches()
create_patches_obj.create_patch()
```

The images used for training and evaluating the model can have varying heights and widths. To manage this variability, the create_patches.py script was developed. This script generates subimages from the original images, which are then utilized in data_generator.py. The size of these patches is controlled by the image_size parameter specified in base.py. It is important to note that if the specified image_size parameter is larger than the dimensions of the input image, the script will raise an error.

****

```
#%% Data generator
train_generator = DataGenerator(config.train_patches_dir)
validate_generator = DataGenerator(config.validate_patches_dir)
test_generator = DataGenerator(config.test_patches_dir)
```

The create_patches.py script prepares the input for the data_generator.py script. The data_generator script, controlled by the batch_size parameter in base.py, generates small batches of images. This approach facilitates computations within the model by avoiding the need to read entire images at once, thus reducing computational burden. The data generator must be run train, test and validation separately.

****

```
#%% Create DL model
create_obj = Unet().create_model()
```

The Satellite U-Net [2] is created according to the specifications in unet.py, with hardcoded parameters including a depth level of 5, the sigmoid activation function, the Adam optimizer, and the binary focal cross-entropy loss function. To experiment with different hyperparameters, one can modify these parameters directly in unet.py.

****

```
#%% Train DL model
create_obj = Unet().create_model()
train_obj = Train(train_generator, validate_generator)
train_obj.train()
```

The train.py script invokes the Satellite U-Net model and begins the training process using parameters defined in base.py. While the default number of epochs is set to 1000, the training process may conclude much earlier if the early stopping criteria are met.

****

```
#%% Evaluate DL model
evaluate_obj = Evaluate(validate_generator)
evaluate_obj.evaluate_models()
```

The evaluate.py script uses data from the validation generator to assess the model's performance on the validation set. It calculates four accuracy metrics: accuracy, recall, precision, and F1-score, and writes the results to a .csv file in the evaluate subfolder within the output folder.

****

```
#%% Update DL model name
test_generator.update_modelName(train_obj.model_name)
```

The update_modelName function automatically updates the weight file name in base.py. If from_scratch is set to False, this function should be commented out and not executed.

****

```
#%% Create image predictions
predict_obj = Predict(test_generator)
predict_obj.create_predictions()
```

The create_predictions function in predict.py generates probability images for each patch produced by the data generator for the test images.

****

```
#%% Create mosaic from predicted image patches
predict_obj.mosaic_predictions()
```

The mosaic_predictions function in predict.py combines the image patches to produce a single output image for each tile. After this step, post-processing should be performed using ImageJ [3][4] and ArcGIS.

****

```
#%% Georeferencing the images after morphological operations
predict_obj.georeference_watershed()
```

The implementation of morphological operations with ImageJ causes the images to lose their geolocation information. To restore the original geolocation, the images need to be transformed accordingly. The georeferenced_watershed function in predict.py performs this georeferencing procedure. For implementation details about morphological operations, please refer to Section 3.3.

****

```
#%% Evaluate results after morphological operation
predict_obj.evaluate_watershed()
```

The evaluate_watershed function uses the output from ImageJ to evaluate accuracy after morphological operations. It computes four accuracy metrics—accuracy, recall, precision, and F1-score—along with the confusion matrix. The confusion matrix is then saved to a .csv file in the evaluate subfolder within the output folder.

****

```
#%% Evaluate results after polygonization
predict_obj.calculate_polis()
predict_obj.evaluate_polis()
```

As the final step in the prototype, the accuracy assessment known as PoLiS [5] will be carried out using two functions from predict.py: calculate_polis and evaluate_polis. This step will be implemented after the simplification and polygonization processes in ArcGIS. For detailed implementation instructions in ArcGIS, please refer to Section 3.3.

****

3.3 Post-processing Component

After installing (please refer to Section 4.2 for installation instructions) and running the program, navigate to Plugin->MorphoLibJ->Segmentation->Morphological Segmentation to apply morphological segmentation, as illustrated in Figure 3.

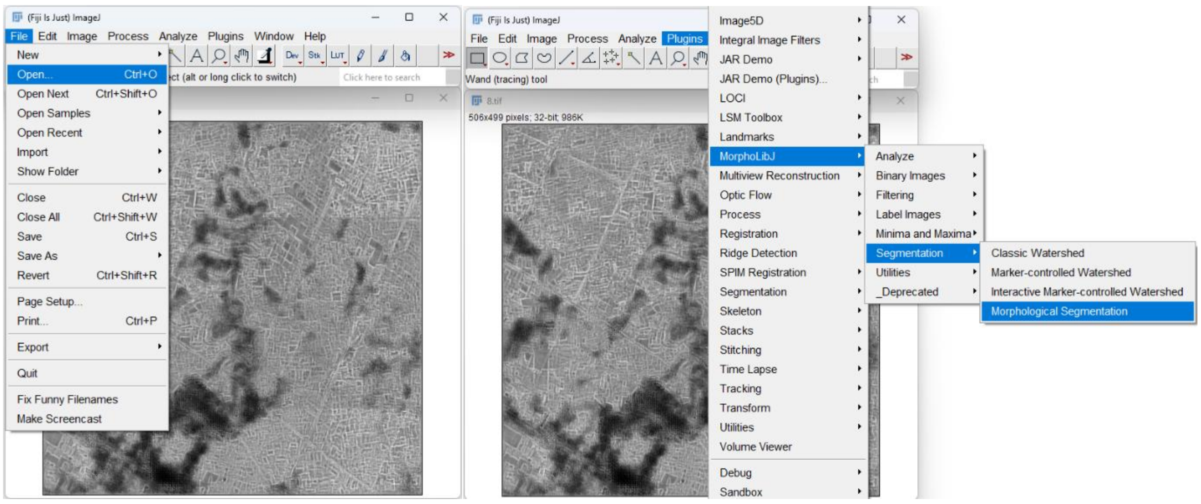
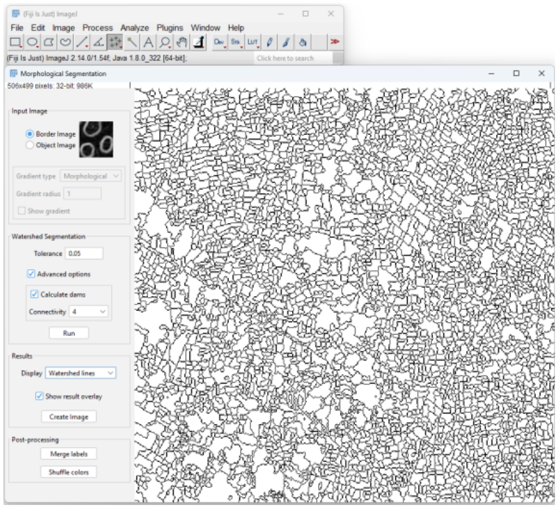

File
Open... Ctrl+O
Plugins
MorphoLibJ
Segmentation
Classic Watershed
Marker-controlled Watershed
Interactive Marker-controlled Watershed
Morphological Segmentation

Figure 3. The implementation steps of morphological segmentation for an image

To implement the morphological segmentation, enter the empirically determined parameters provided in Table 1 into the GUI, as shown in Figure 4. The resulting segmentation image should be generated as 'Watershed lines'. The following figure illustrates scenarios produced with the 'from_scratch' mode, while Table 1 lists the parameters used for the other models. We would like to remark that in case of using the prototype on a new study area, the list of parameters may require some fine-tuning to optimize the performances to the specific properties of the considered study area.



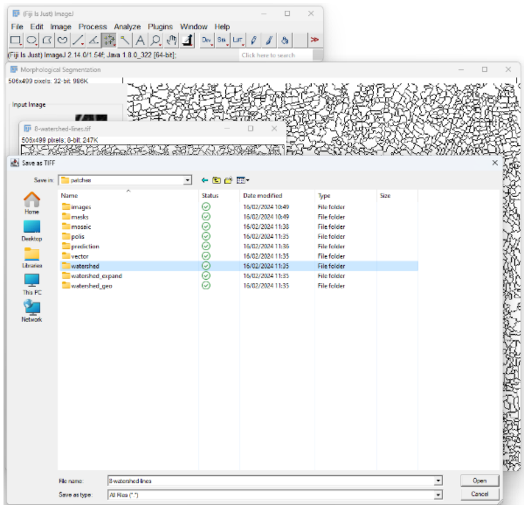
Morphological Segmentation

Input Image: [✓ Border Image] Object Image

Watershed Segmentation: Tolerance 0.05; Advanced options [✓]; Calculate dams [✓]; Connectivity 4

Results: Display Watershed lines; Show result overlay [✓]

Figure 4. Segmentation with determined parameters.



Save as TIFF — Save in: patches

Name	Status	Date modified	Type	Size
images	✓	16/02/2024 10:49	File folder	
masks	✓	16/02/2024 10:49	File folder	
mosaic	✓	16/02/2024 11:38	File folder	
polis	✓	16/02/2024 11:35	File folder	
prediction	✓	16/02/2024 11:36	File folder	
vector	✓	16/02/2024 11:35	File folder	
watershed	✓	16/02/2024 11:35	File folder	
watershed_expand	✓	16/02/2024 11:35	File folder	
watershed_geo	✓	16/02/2024 11:35	File folder	

File name: 8-watershed-lines
Save as type: All Files (*.*)

Figure 5 Saving results into the watershed folder.

Each processed image should be saved in the folder, namely watershed in the test folder as shown in Figure 5, that will be created by the predict.py script within the prototype.

Table 1 The empirically determined paramaters for morphological segmentation.

PARAMETERS	S2_PRETRAINED	S2_SCRATCH
INPUT IMAGE	Border Image	Border Image
TOLERANCE	0.1	0.05
ADVANCED OPTIONS	True	True

CALCULATE DAMS	True	True
CONNECTIVITY	4	4
DISPLAY	Watershed lines	Watershed lines

After the implementation of morphological segmentation, the coordinate information of the TIFF images was lost. The 'georeference_watershed' function in the **predict.py** script uses the coordinates from the original images to georeference the morphological segmentation results and save them in the **watershed_geo** folder. To check the improving accuracy, in **predict.py**, there is the function named {*func: evaluate_watershed*} used to evaluate the morphological segmentation results compared to the original mask and write the results in the **'..\output\evaluation'** folder as a csv file with the accuracy metrics that are precision, recall, and F1-score for each test image.

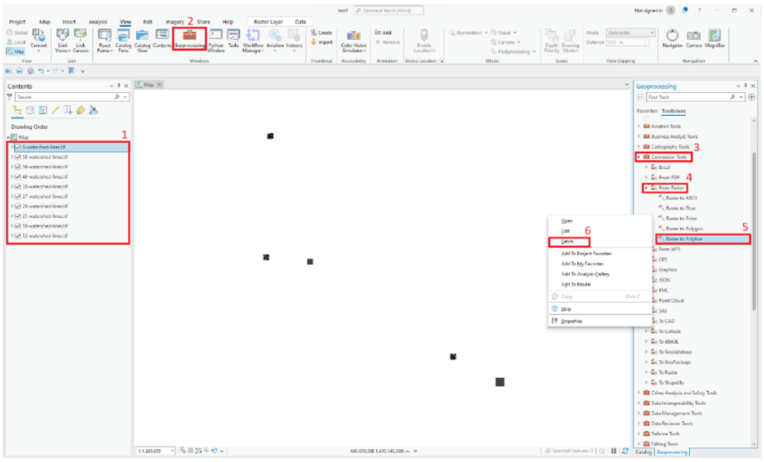
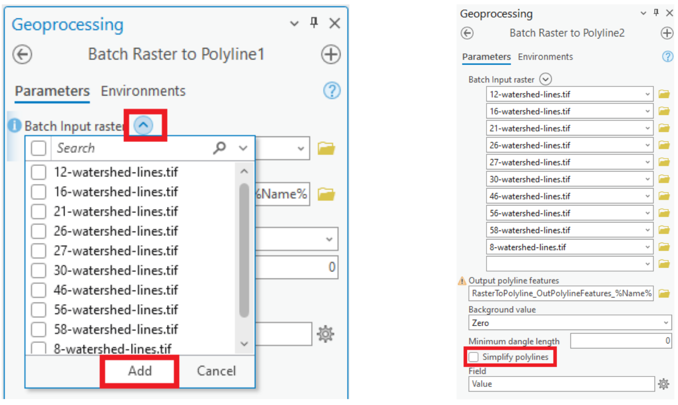


Figure 6 The batch process implementation for raster image conversion to polylines.

The raster images converted polyline by using the 'Raster to Polyline' command in the Conversion Toolbox under Geoprocessing Pane as shown in Figure 6. Batch mode is used by right-clicking on the 'Raster to Polyline' function and selecting the images to be processed (Figure 7), as the same process is applied to each image with the same parameters. It should be noted that the 'Simplify polylines' option unchecked in this conversion as shown in Figure 7.



Batch Input raster: 12-watershed-lines.tif, 16-watershed-lines.tif, 21-watershed-lines.tif, 26-watershed-lines.tif, 27-watershed-lines.tif, 30-watershed-lines.tif, 46-watershed-lines.tif, 56-watershed-lines.tif, 58-watershed-lines.tif, 8-watershed-lines.tif

Output polyline features: RasterToPolyline_OutPolylineFeatures_%Name%

Background value: Zero

Minimum dangle length: 0

☐ Simplify polylines

Field: Value

Figure 7 The selection of images to be converted from raster to polylines.

This step aims to create a new vector file that is identical to the source images and is based on vectors with polylines.

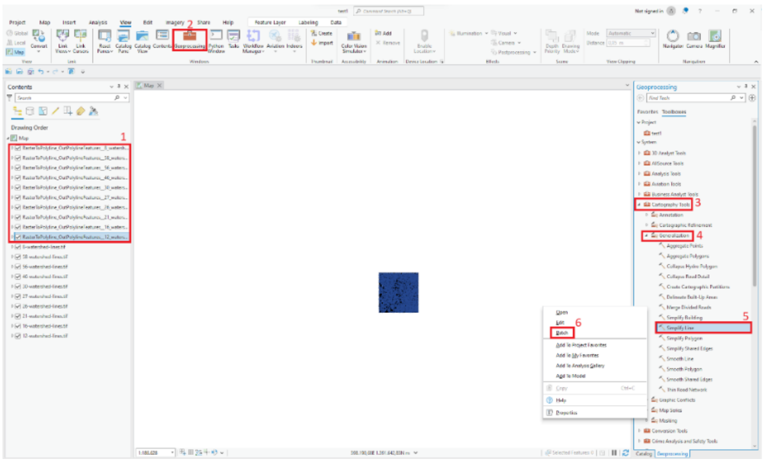
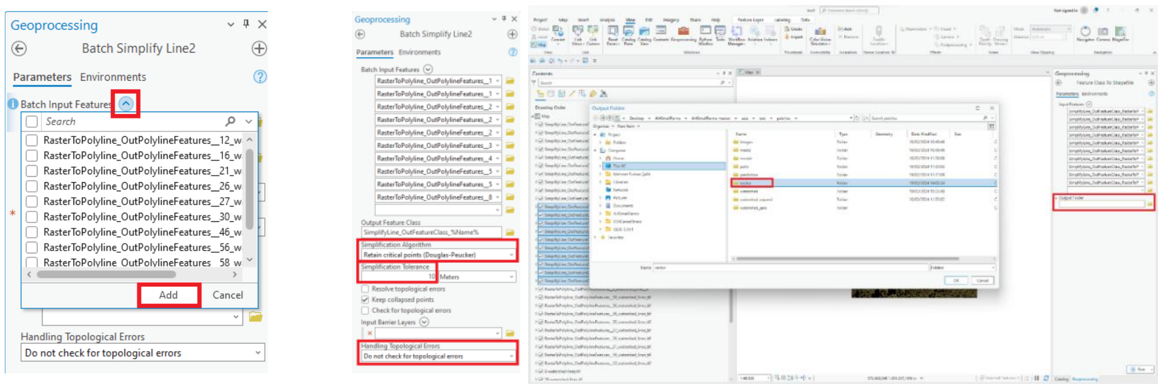


Figure 8 The batch process implementation for line simplification procedure.

The generalization of lines with simplification procedure is done by using the 'Simplify line' command in the Cartography Toolbox under the Geoprocessing Pane, as shown in Figure 8. Batch mode is used by right-clicking on the 'Simplify line' function and selecting the images to be processed (Figure 9), as the same process is applied to each image with the same parameters. It should be noted that the Douglas-Peucker is chosen as a simplification algorithm with a tolerance of 10 meters and the handling topological errors option unchecked, as shown in Figure 9. As a final step, the simplified vectors need to be converted to a shapefile from the conversion toolbox as shown in Figure 9 (left) and exported to the 'vector' folder as shown in Figure 9 (right), where further analysis can be performed.



Geoprocessing

Batch Simplify Line2

Parameters Environments

Batch Input Features

RasterToPolyline_OutPolylineFeatures__12_w
RasterToPolyline_OutPolylineFeatures__16_w
RasterToPolyline_OutPolylineFeatures__21_w
RasterToPolyline_OutPolylineFeatures__26_w
RasterToPolyline_OutPolylineFeatures__27_w
RasterToPolyline_OutPolylineFeatures__30_w
RasterToPolyline_OutPolylineFeatures__46_w
RasterToPolyline_OutPolylineFeatures__56_w
RasterToPolyline_OutPolylineFeatures__58_w

Add Cancel

Handling Topological Errors: Do not check for topological errors

Simplification Algorithm: Retain critical points (Douglas-Peucker)

Simplification Tolerance: 10 Meters

Figure 9 The selection of vector files to be simplified (left) and exporting final product into the vector folder.

3.4 Integration of geospatial attributes

This section reports and example on how to populate the geospatial database with freely available geospatial attributed (e.g., rice data and maps). A freely available rice map was utilized to illustrate how to add the rice attributes to the crop field present in the geospatial dataset. To achieve this, a Python script was developed that requires only two inputs: a vector file and a reference raster file. The resulting vector file includes an attribute for each field, indicating whether it is classified as rice or not rice.

```
#%% Combining Vector and Raster
vector_dir = r'…\asia\test\patches\vector'
raster_dir = r'…'
output_dir = r'…asia\test\patches\vector_rice'

vector = gpd.read_file(os.path.join(vector_dir,'8_watershed_lines_tif.shp'))
vector_org = vector.copy()
vector_crs = vector.crs
raster_crs = 'EPSG:4326'

transformer = Transformer.from_crs(vector_crs, raster_crs, always_xy=True)
vector['geometry'] = vector['geometry'].apply(lambda geom: transform(transformer.transform, geom))

shapefile_bounds = vector.total_bounds
shapefile_bbox = box(*shapefile_bounds)

hit_index = []
with rasterio.open(os.path.join(raster_dir,'MSEAsia2019.tif')) as src:
    for index_window, window in tqdm(src.block_windows(1)):
        window_bounds = src.window_bounds(window)
        window_polygon = box(*window_bounds)

        # Code is too slow without this control!!!
        if not window_polygon.intersects(shapefile_bbox):
            continue
        raster_data = src.read(1, window=window)
        transform = src.window_transform(window)

        for index_polygon, polygon in vector.iterrows():
            if polygon.geometry.intersects(window_polygon):
                mask = geometry_mask([polygon.geometry], transform=transform, invert=True, out_shape=raster_data.shape)
                if (raster_data[mask] == 255).any():
                    hit_index.append(index_polygon)

vector_org["rice"] = 0
vector_org.loc[hit_index, "rice"] = 1
vector_org.to_file(os.path.join(output_dir,'8_watershed_lines_tif_RICE.shp'))
```

4 Installation Guide

4.1 Setting environment for prototype

Before running the prototype, all the Python packages used within the workflow should be installed. It can be done in three ways;

- Missing Python packages can be installed in any existing local environment of the end-user, which is NOT recommended. Different versions of the different libraries can conflict with each other and can cause a crash.
- A new environment can be created as given below, and all packages can be installed in specified versions in the appendix. Replace the name of the environment with <new-env>. In the case of GPU usage, the Python and TensorFlow versions should be exactly the version given in the appendix.

```
conda create --name <new-env>
conda activate <new-env>
```

- The environment provided in GitHub can be installed by running the following commands in the project directory, which is the recommended way to use the prototype.

```
conda env create -f environment.yml
conda activate ai4small
```

4.2 ImageJ and Plugin Installation

Please refer to the https://imagej.net/downloads to download the program that is required to run the program. It is recommended that you download the 'Fiji' distribution of the program to run it. After downloading the software, unzip the folder and run the 'ImageJ-win64.exe' file within the folder to launch the software. A specific plugin needs to be installed in the software so that morphological segmentation can be implemented. The software should be updated to facilitate this process, as shown in the figure below.

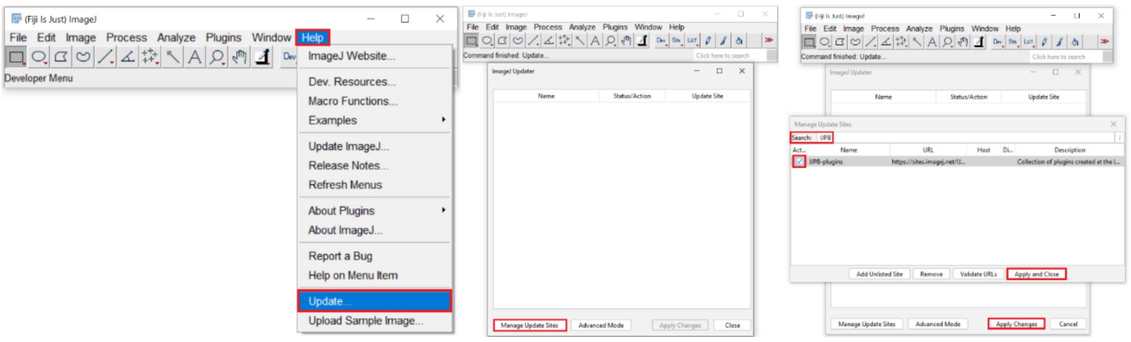


Figure 10 Plugin installation in the Fiji program for implementing morphological segmentation

Install the necessary plugin by searching for the IJPB plugin in the search bar and then checking the box to install it as shown in Figure 10. For the plugin to work properly, it is necessary to restart the computer after the plugin is installed.

5 Frequently asked questions

•Where can I download the dataset to run the prototype?

The benchmark dataset is publicly available [6]. The crop boundary results are reported in [1].

•What types of prompts are supported?

The prototype conducts field boundary delineation from the provided Sentinel-2 images and mask data.

•What is the structure of the prototype?

The prototype features an object-oriented programming structure with defined classes for input data and hyperparameter manipulation. The preprocessing of input images is conducted in two steps: dividing the images and then creating patches. A deep learning model then receives the prepared images.

•What platforms or libraries does the model use?

Spyder, Jupyter Lab, Jupyter Notebook, and other platforms can run the Python-written model. To use a GPU, one can either create a new environment, use the provided environment file, or use Google Colab. Appendix A provides the necessary libraries.

•What is the architecture of the deep learning model?

The prototype uses an U-Net architecture modified for satellite images. The U-Net architecture is a type of convolutional neural network (CNN) designed primarily for biomedical image segmentation, but it is also effective for satellite image analysis.

•How does the U-Net architecture work for this task?

The network has encoder and decoder parts with a depth of 5 levels, which means the network performs five downsampling operations in the encoder path and five upsampling operations in the decoder path, ensuring a balance between feature extraction and spatial resolution recovery.

•What data is used to train the model?

The model was trained on four bands (RGB-NIR) of Sentinel-2 images with a crop boundary mask. This approach leverages the high-resolution spectral data provided by Sentinel-2 satellites, which capture information in the visible and near-infrared spectrums.

•How long does it take to train the model?

The duration of model training depends on the available computational resources. Typically, it takes around 30-35 minutes on a computer equipped with a GPU of moderate capabilities.

•What output does the model generate?

The model generates a probability map that indicates both the likelihood and strength that each pixel is a boundary.

•Does the model work with other satellite images?

The configuration (base.py) specifies that the prototype can accommodate any type of image, with larger image sizes (height and width). Depending on whether the images have a lower or higher band number, adjustments to the number of channel parameters are necessary.

•How can I use this prototype?

The configuration (base.py) is the prototype's core component. Make sure base.py contains all the necessary paths and parameters before running the code.

•Is any preprocessing required before inputting images?

For the code to run, no specific preprocessing is required. If the provided image is free from as much noise, cloud, and shadow as possible and has sufficient spatial resolution, the results will be better.

•Is any postprocessing required after model output?

Yes, to achieve the final product, morphological operations and polygonization steps are required, as detailed in the technical documentation.

6 References

[1] C. Persello et al., "AI4SmallFarms: A Dataset for Crop Field Delineation in Southeast Asian Smallholder Farms," in IEEE Geoscience and Remote Sensing Letters, vol. 20, pp. 1-5, 2023, Art no. 2505705, doi: 10.1109/LGRS.2023.3323095 https://ieeexplore.ieee.org/document/10278130

[2] Modified version of U-Net: Deep learning for satellite imagery via image segmentation - deepsense.ai

[3] ImageJ: https://imagej.net/downloads

[4] Morphological segmentation tutorial: https://imagej.net/plugins/morphological-segmentation

[5] Polis metric definition is available here: https://ieeexplore.ieee.org/document/6849454

[6] The related benchmark dataset is available here: https://easy.dans.knaw.nl/ui/datasets/id/easy-dataset:321745

7 Appendix

In the case of GPU usage on local computer, the TensorFlow version must be V2.10, and Python version must be V3.9-V3.10. There is no need for any modifications in the case of using Google Colab API.

LIBRARY	VERSION
TENSORFLOW	2.10.1
KERAS_UNET	0.1.2
RASTERIO	1.2.10
SEGMENTATION_MODELS	1.0.1
SCIKIT-LEARN	1.4.0
SCIKIT-IMAGE	0.22.0
GEOPANDAS	0.12.2
PANDAS	2.2.0
SHAPELY	2.0.2
RTREE	1.0.1
STRENUM	0.4.15