Instruction Code Manual for Building Footprint Extraction with Deep Learning

Elective course Supervisor

Prof W.H. Bakker

Kushanav Bhuyan

Email: kushanavb@gmail.com

Table of Contents

List of figures

1. Introduction

The document is a brief instruction manual so as to run the code of two notebooks ***Building footprint extraction*** and ***Prediction buildings***. The steps mentioned below will provide a step-by-step routine and understanding of the logic of the code.

Please contact kushanavb@gmail.com for any queries and doubts related to the code. The GitHub repository includes all of the codes, data, and the documented report. Please follow the link for the data and if you want to replicate the entire process of training and predicting the model.

Motivation for the code

The detection of buildings is crucial in recognizing them as an essential component of elements-at-risk (EaR) for disaster risk assessment. Implementing approaches for a hazard impact assessment on EaR is important but cannot be executed without proper data sets. Which is why it is necessary to have databases with updated information about elements exposed to hazards for response activities and support crisis preparedness.

Pre-requisites

1. Google account for Google Drive
2. Google Colaboratory

Steps to remember before starting

1. Make sure to have downloaded all the files.

2. Make sure to structure the folders in a logical way so that there is no confusion of which step of the routine the user was in. (See the example below)

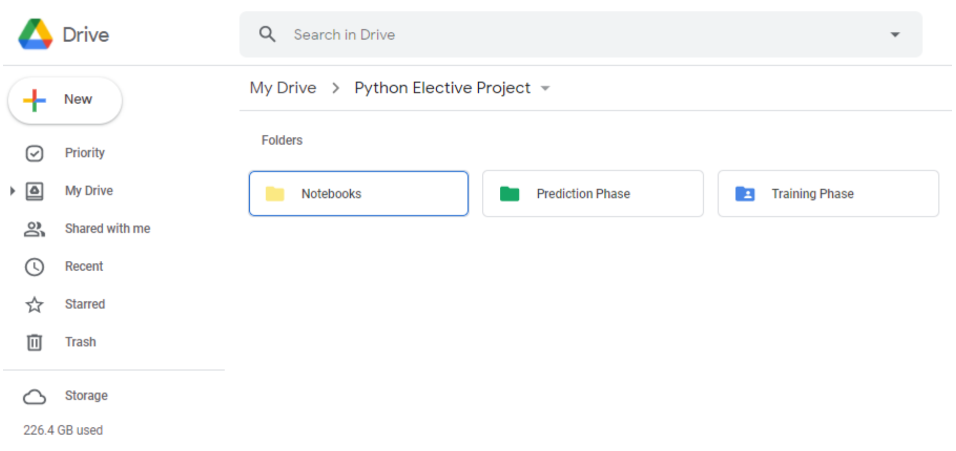


Figure 1: Folder structure example

3. The notebooks are divided into certain *sections* that will guide the user as to what the code does and the purpose of the executing the code.

4. Remember to name the satellite images and the corresponding label images in the same manner. For example, if a satellite image is called "image_1", name the respective label image as "image_1" as well. Repeat this convention as it makes it easy to match the satellite image and the label image.

2. Steps for data preparation and running the code

Step 1: Setting and structuring the folder in Google Drive

One of the most important things to remember is that a good and concise way of structing your folders can make it a really easy to run codes along with saving the necessary results. Otherwise, it can get really messy and confusing.

As a measure to maintain the consistency, go to your **Google Drive** account and make the following folders and sub-folders where the associated data, codes and results will be stored:

1. ***Training Phase*** – Consisting of the training data and the model weights as checkpoints.
    a. Model training data
    b. Checkpoints

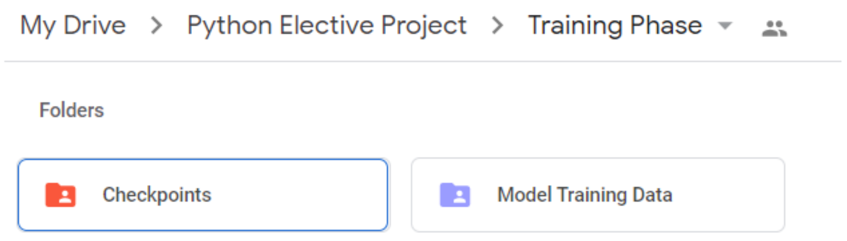


Figure 2: Training Phase folder structure.

2. ***Prediction Phase*** – Consisting of the images to classify, the predicted images and saved arrays.
    a. Images to classify
    b. Images classified
    c. Arrays

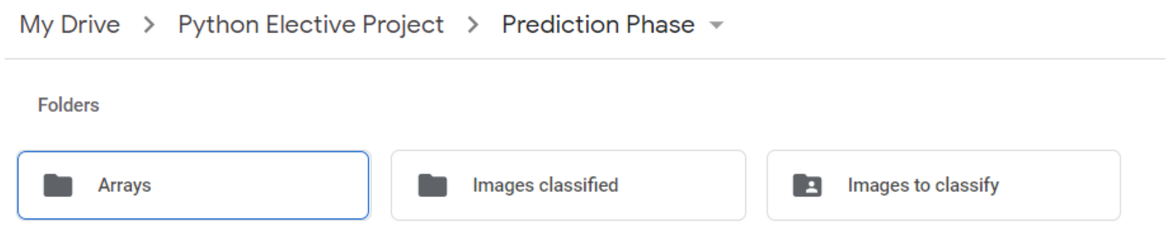


Figure 3: Prediction Phase folder structure.

3. ***Notebooks*** – Consisting of the codes required to perform the building footprint extraction.

Step 2: Downloading the data

The data can be found in Google Drive shared link. The folder **Data** in the GitHub consists of the link to the drive which is open for anyone to access.

Go to **Data** > **Link for the Training Data (For Training)**

Go to **Data** > **Link for Satellite Image for Prediction (For classification after training is performed)**

Following the links will take you to the shared Google Drive space where you can download the data. Download and then upload the data in your respective Google Drive account.

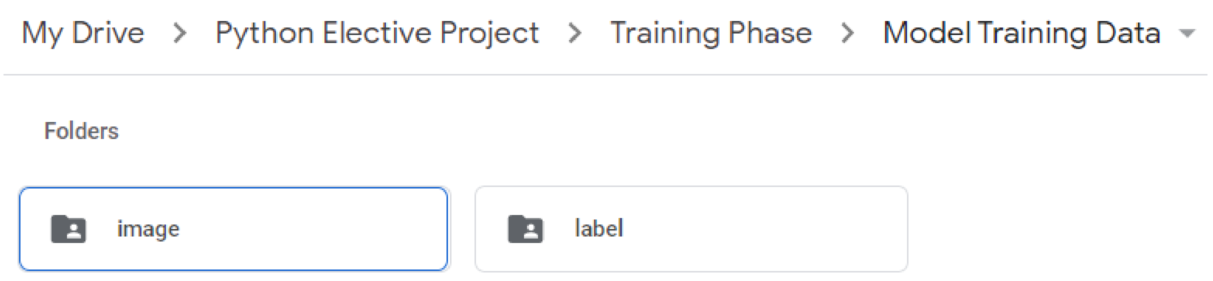

My Drive > Python Elective Project > Training Phase > Model Training Data
Folders
image
label

Figure 4: Directory of the satellite and label images.

The training data will be sorted into the folders *image* and *label* that will contain the satellite and label images, respectively (like figure 4).

Step 3: Downloading the notebook

Effort was given to write the entire code in one Jupyter notebook so as to automate and streamline the entire process of building detection and prediction with the U-Net deep learning model.

1. Follow the GitHub link and find the repository called "Building-Footprint-Extraction". This repository will contain all the necessary data (links) and codes required to:
   (1) Load necessary libraries,
   (2) Load satellite and mask images,
   (3) Generate patches of desired sizes for model training,
   (4) Setup the Deep Learning model,
   (5) Visualisation of training and validation metric curves,
   (6) Model accuracy testing on the test data set, and
   (7) Prediction of buildings on desired study area.

   Please read the project report to better understand the background of the study area, the complex building-to-non-building relationship, and how the data is prepared.

2. Download the codes from the folder *Codes* and load it in Google Colab (**Step 4**).

(The notebook can also be run on local machines using Jupyter Notebook but the installation of the required libraries and packages with the right dependencies can be difficult and cumbersome)

Step 4: Loading the notebook

Go to Google Colab and log-in using your Google account. (Remember: Use the account where the codes and data are saved, in reference to **Step 2**)

Go to **File** > **Open notebook** > **Upload** > **Choose file** > Choose the notebook ***Building footprint extraction***.

Do the same for the second notebook.

Remark: In order to run the notebook properly, it is better to opt for the higher RAM of 25GB as the satellite images are patched over 512x512 pixel sizes, which consumes a lot of RAM. Thus, with the pre-allocated 12GB RAM the runtime sessions will crash due to lack of memory. To avoid this situation, the following steps can be followed:

1. Click on **Runtime** on the top-left tab section.
2. Select **Change runtime type.**
3. Under **Runtime shape**, choose **High RAM**.

This should enable higher RAM (25GB) allocation to the runtime session.

Step 5: Loading necessary modules

The notebook requires utilities and model definitions which are written in python files (.py) and are crucial for initiating and training the model. There are two modules for,

1. Model network
2. Loss function and metrics

The modules are found in the *Codes* folder named "unet_model" and "loss_metrics".

The first module consists of the deep learning model network. The second module consists of the loss functions and metrics required to monitor the training loss and evaluating the training progress, respectively.

As we see in the figure on the right, click on the icon and that will prompt a pop-up screen and then guide to the folder where the two modules are present in your local drive and load them.

After uploading the two modules, simply go back to the code and the run the corresponding cells to load the modules into the session (as seen in the figure on the right).

Remark: After you stop the session or restart the session, you will need to load them again.

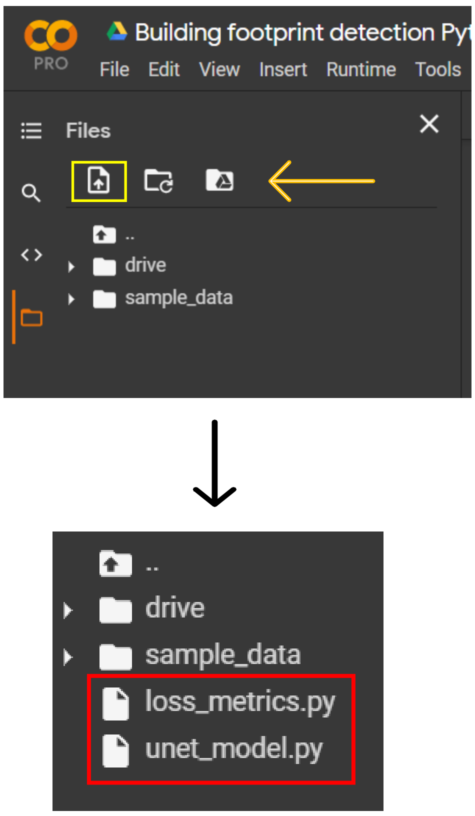


Figure 5: Uploading the modules.

Step 6: Running the notebooks (explained section-by-section)

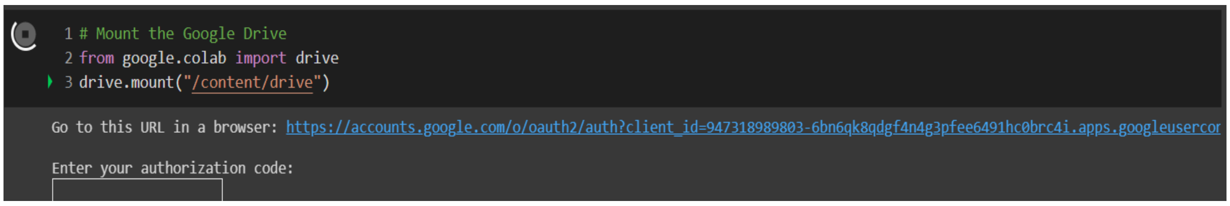
As you move from one section to another, please run the cells one-by-one as you go along.

Part 1 – First Notebook

The next steps are pretty straightforward and easy to implement. Based on the proceedings of the sections of the **first** notebook, let us see what the code contains:

1. The *first* section is the model introduction giving insight on the structure of the model network.

2. The *second* section loads the libraries and the modules that are required to load and train the model with the satellite and label images. Run the cells for the second section.



```
# Mount the Google Drive
from google.colab import drive
drive.mount("/content/drive")

Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercor

Enter your authorization code:
```

Figure 6: Google account log in.

Log in using your Google account to use the Google Drive where your data is saved (figure 6).

3. The *third* section loads the satellite imagery data and the label image data that are annotations of the buildings. This section contains a class called "BuildingDataset" that allow reading the satellite and label image as NumPy arrays[1] and generates a list of the satellite images with their corresponding label images. This step is followed by a visualisation code that allow to visually display the data (figure 7). This section also contains global variables that allow the user to set important variables that would dictate how the data is read (*NBANDS*) and also how the data is patched for training (*PATCHSIZE*).

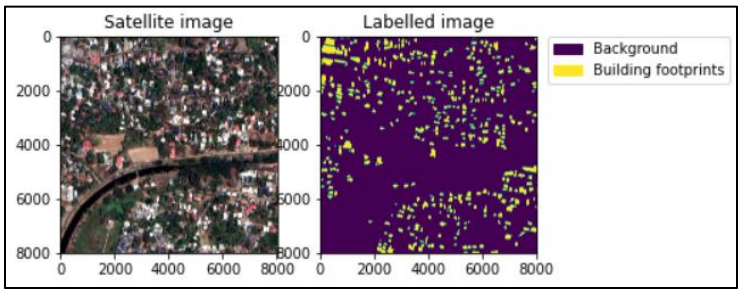


Figure 7: Visualisation example of the satellite image with the corresponding label image.

Remark: In the global variable called *directory*, add the path to the folder where the satellite and label images are stored in. For example, referring to figure 4 and 8, the path would be "/content/drive/MyDrive/Python Elective Project/Training Phase/Model Training Data".

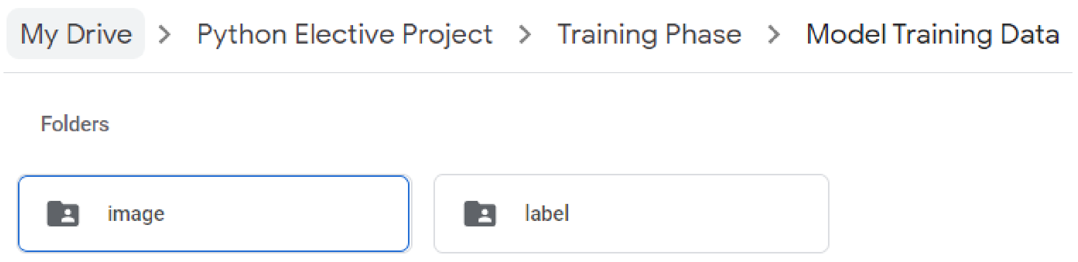

My Drive > Python Elective Project > Training Phase > Model Training Data
Folders
image
label

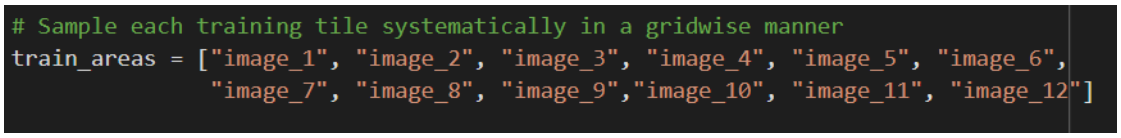
Figure 8: Folder name of the satellite and label images.

As you use the *BuildingDataset* class, enter the names of the folders for the *folder_*name argument where the satellite and label images are located in, which are "image" and "label" (figure 8).

4. The *fourth* section generates patches of the data by slicing the images into the required number of patch size (section three) using a function called the "gridwise_sample". Accordingly, the data is patched according to the patch size for both the training and testing sets.

[1] A NumPy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension (Johnson, 2018).

Remark: The training and testing sites are manually chosen. As you would see in the code, the test data is selected manually and hence, careful consideration is required to properly choose a site that would best test the accuracy of the model (check figure 9).



```
# Sample each training tile systematically in a gridwise manner
train_areas = ["image_1", "image_2", "image_3", "image_4", "image_5", "image_6",
               "image_7", "image_8", "image_9","image_10", "image_11", "image_12"]
```

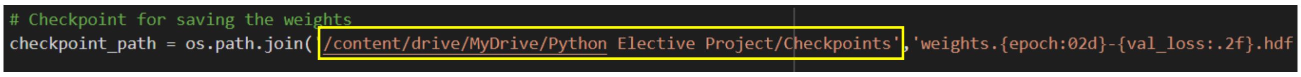
Figure 9: Training site selection.

The training data is stored in variables *X_train* (satellite image) and *Y_train* (label image) and the testing data is stored in variables *X_test* (satellite image) and *Y_test* (label image).

5. The *fifth* section sets up required hyper-parameters and general settings that allow the model to train in an efficient way. Some of the important ones to notice are:
    a. Number of epochs (Set according to your testing needs. 20 should be a good start.)
    b. Batch size (Try 12 first)
    c. Optimizers (Choose between the available ones, preferably use Adam)

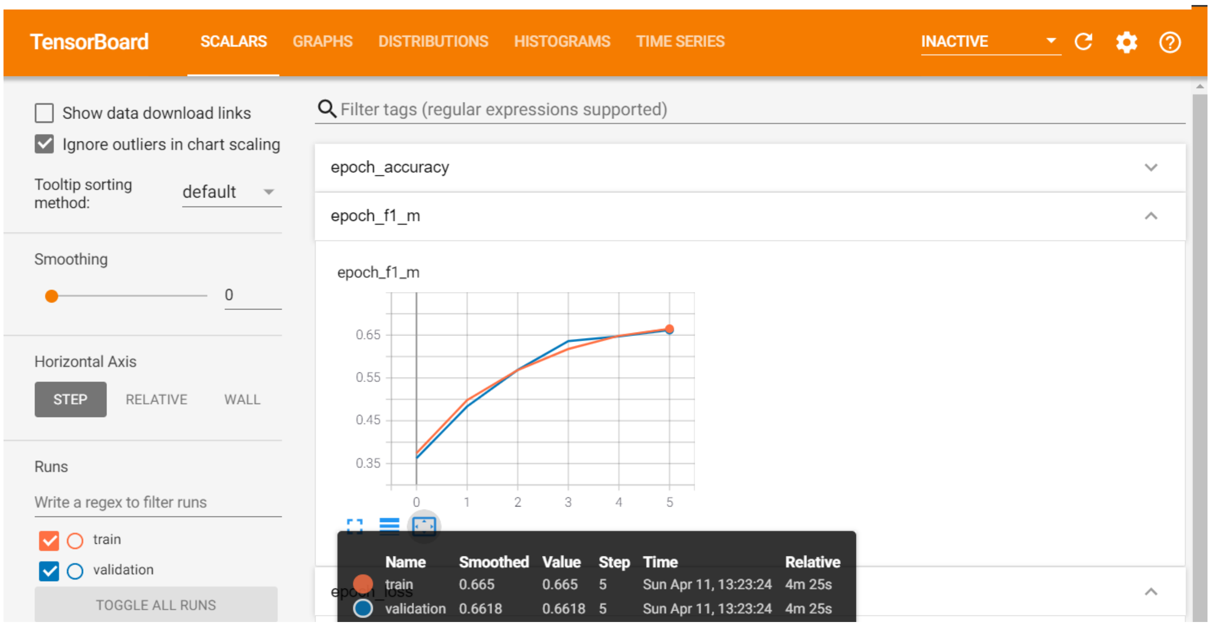
Remark: The modules "unet_model" and "loss_metrics" are utilised in this section. The former module consists of the deep learning model and the latter contains the loss function and the required metrics for model accuracy evaluation.

Accordingly, *general settings* and *callbacks* are set to save the models at each run as *checkpoints* so that if the notebook stops[2], then the learned weights from the modelling are saved in a folder called "Checkpoints". Set the directory of the checkpoint to the Checkpoints folder (figure 10).



```
# Checkpoint for saving the weights
checkpoint_path = os.path.join('/content/drive/MyDrive/Python Elective Project/Checkpoints','weights.{epoch:02d}-{val_loss:.2f}.hdf
```

Figure 11: Checkpoint path



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing: 0
Horizontal Axis: STEP RELATIVE WALL
Runs
Write a regex to filter runs
☑ train
☑ validation
TOGGLE ALL RUNS
Filter tags (regular expressions supported)

	Name	Smoothed	Value	Step	Time	Relative
●	train	0.665	0.665	5	Sun Apr 11, 13:23:24	4m 25s
○	validation	0.6618	0.6618	5	Sun Apr 11, 13:23:24	4m 25s

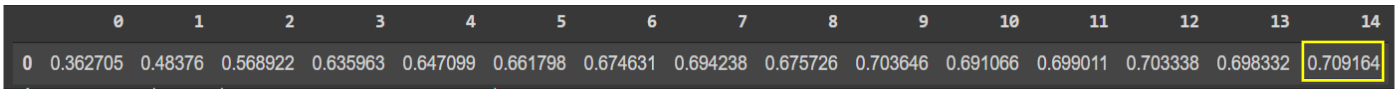
Figure 10: A snippet of Tensor Board

[2] The notebook can stop for OOM issues, internet connection cut-off and other such issues.

Furthermore, real-time model performance can be monitored using the Tensor Board callback (see figure 11).

6. The *sixth* section initiates the model from the module and then the training starts using the *model.fit* API of TensorFlow. This section also contains the code to *visually plot* the training and validation curves for loss and error rate and *logging the metrics* at epoch.

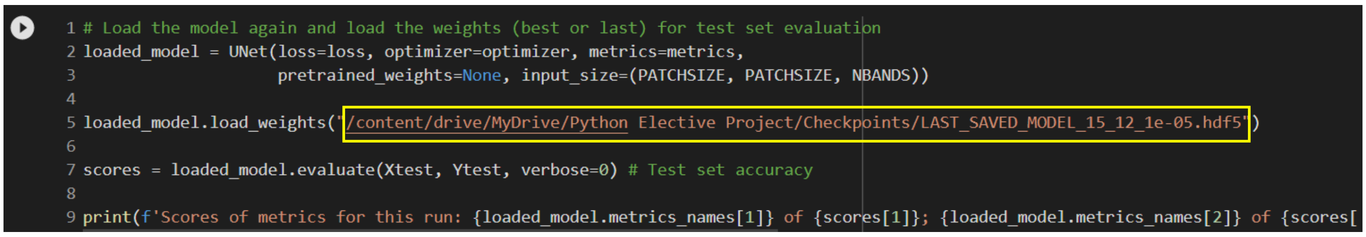
   After the training completes, due to the *checkpoints* being saved at every run (or epoch), the user can now choose the best weights learned from training the model and then implement the same



```
          0        1         2         3         4         5         6         7         8         9        10        11        12        13        14
0  0.362705  0.48376  0.568922  0.635963  0.647099  0.661798  0.674631  0.694238  0.675726  0.703646  0.691066  0.699011  0.703338  0.698332  0.709164
```

Figure 12: Log of F1-score metrics.

   for testing purposes with the test data set. This can be checked in the log of the metrics discussed briefly ago. Choose the model with the highest F1-score like 70.9% in figure 12 by entering the path of the highest model as seen in figure 13.



```python
# Load the model again and load the weights (best or last) for test set evaluation
loaded_model = UNet(loss=loss, optimizer=optimizer, metrics=metrics,
                    pretrained_weights=None, input_size=(PATCHSIZE, PATCHSIZE, NBANDS))

loaded_model.load_weights('/content/drive/MyDrive/Python Elective Project/Checkpoints/LAST_SAVED_MODEL_15_12_1e-05.hdf5')

scores = loaded_model.evaluate(Xtest, Ytest, verbose=0) # Test set accuracy

print(f'Scores of metrics for this run: {loaded_model.metrics_names[1]} of {scores[1]}; {loaded_model.metrics_names[2]} of {scores[
```

Figure 13: Path selection of the best model.

   Remarks: Within this section, at line 13 in the first cell, the user can select a desired validation split rate that would split the training data into the decided ratio between training and validation data set. For example, a value of 0.1 would split the training data into 90% used for training and 10% used for real-time validation. Depending on the availability of data, the values would vary proportionately.

7. The final *seventh* section includes the testing of the model with respect to the *model weights* obtained after training the model. Initially, the *model* and *model weights* are loaded and then testing is evaluated on the test data set using the *model.evaluate* API. Based on the results obtained, the user would either re-train the model if satisfiable results are not met or would move on to prediction if the results are satisfiable enough.

   The next part of the section contains the code for visualising the predictions on the test image.

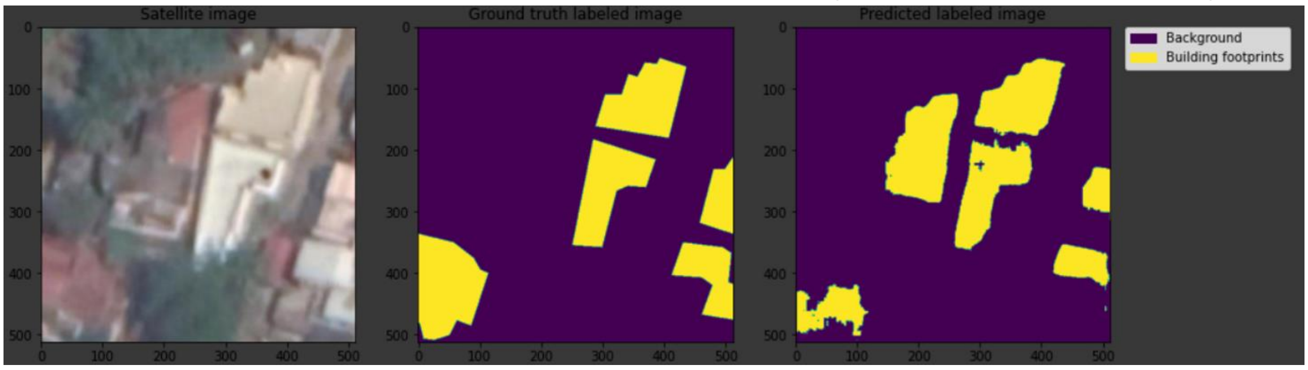


Figure 14: An example of prediction after the model is trained.

The visualisation code is a function “visualize_prediction” which is followed by the *model.predict* API that makes the predictions over the test data and then visualises it using the Matplot library. The figure below shows a prediction over the test data which the model did not see at all during the training of the model.

The result above is a snippet of what is possible in terms of detection with just a few thousand samples of building data in complex regions like Palakkad, Kerala where buildings are often multi-shaped with different roof colours.

Part 2 - Second Notebook

The next step is the classification of entire satellite images for prediction and product generation. This will be followed in the **second** notebook.

1. The sections in the second notebook also begin with the same steps as mounting the Google Drive to read the data from the user’s drive, loading of the required libraries and the necessary modules, and the model weights.

2. Accordingly, the data and the libraries are imported into the session and then similarly to step 4 in **Part 1**, the satellite image is patched into the same size as what the model was trained on to predict the buildings.

3. The predictions are run on the satellite image using the *model.predict* API, similar to **Part 1** step 7.

   Remark: A common issue that the user might face with Google Colab is the OOM[3] errors that stops the current working session and restarts it altogether. This is because if the satellite image is huge (over ~3.5 GB), the predictions over the image are of huge disk size as well and crashes the session due to OOM. A solution to this issue is to save the results as a NumPy array (.npy) file. This allows the user to save the data without losing it if the session restarts due to OOM issues.

   IMPORTANT

   If the user’s satellite image is over ~3.5GB (like the provided images), the user is suggested to save the prediction as a **.npy** array file in the folder *Arrays* and then restart the session by clicking on **Runtime** and choosing **Restart Runtime** on the top of the notebook panel. Generally advised to do this every time as the purpose of this notebook is to predict on big areas for product generation of building footprints.

4. After restarting the session, load the saved **.npy** array file by directly going to the section “Save and Load as Arrays” and continue ahead.

5. The next step is crucial as it helps stitch the predicted patches into one whole (original) image. The predictions are performed at the patch level that we previously set in the training phase as well as in the prediction phase (step 3 **Part 2**), but they now need to be stitched together to join them and get the output at the same size as the input test image.

[3] A memory error means that your program has ran out of memory. This means that your program somehow creates too many objects.

6. After this step, *rasterio* is used to fuse the coordinate system meta data from the satellite image into the prediction image as the prediction image does not contain any geo-reference information.

7. Finally, the predicted georeferenced image is exported as a GeoTIFF using an LZW compression technique to reduce the size (disk size) of the image. The final output of the image can be seen in the project report. A snippet of that can be seen here in figure 15.

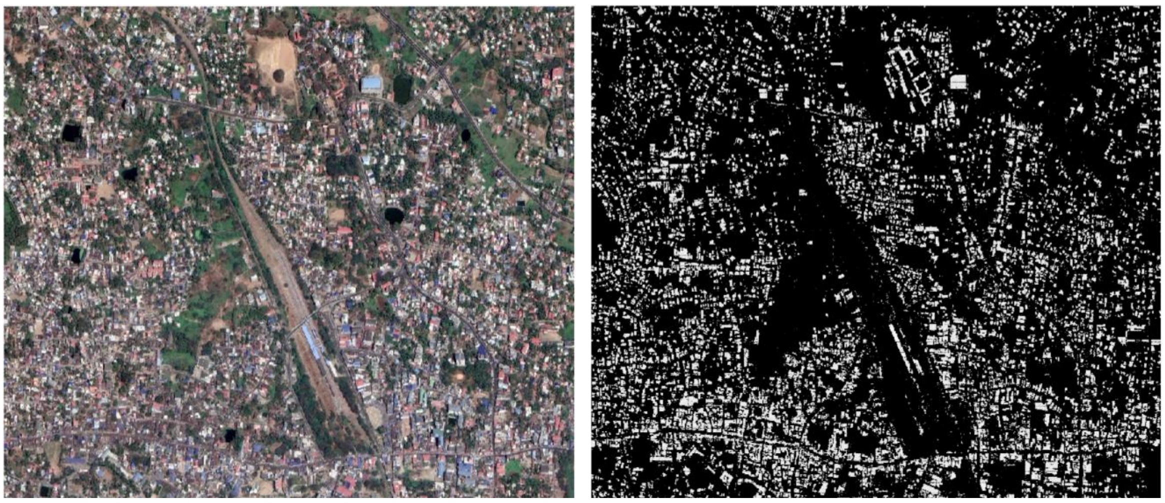


Figure 15: Prediction over a quarter of the study area.

Congratulations!! You have helped in designing a database of buildings in data-scarce regions where now stakeholders can use your output for disaster management and risk mitigation purposes. This will be one of your first steps in creating a building database and slowly with more in-depth knowledge of the code and the concepts behind deep learning, you can generate better quality databases.

If you liked this tutorial, please share this with your colleagues and friends for future use and reference.

3. References

Instance-aware Semantic Segmentation via Multi-task Network Cascades. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016-December, 3150–3158. Retrieved from http://arxiv.org/abs/1512.04412

Johnson, J. (2018). Python Numpy Tutorial (with Jupyter and Colab). Retrieved April 16, 2021, from https://cs231n.github.io/python-numpy-tutorial/

U-net: Convolutional networks for biomedical image segmentation. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9351, 234–241. https://doi.org/10.1007/978-3-319-24574-4_28

Landslide Detection Using Residual Networks and the Fusion of Spectral and Topographic Information. IEEE Access, 7, 114363–114373. https://doi.org/10.1109/access.2019.2935761

Building Footprint Extraction from Very-High-Resolution Satellite Image Using Object-Based Image Analysis (OBIA) Technique. In Lecture Notes in Civil Engineering (Vol. 33, pp. 517–529). https://doi.org/10.1007/978-981-13-7067-0_41

Automatic building segmentation of aerial imagery using multi-constraint fully convolutional networks. Remote Sensing, 10(3). https://doi.org/10.3390/rs10030407