Chapter 1

Maintenance Manual

This chapter describes how to download, extract and prepare the project files for testing and execution. It also provides information about the role of the different source code files.

Downloading and Preparing Project Files

All the data required to run and test this project can be downloaded from:

`https://drive.google.com/drive/folders/1L1yn4k5KosWot9qp4G0eUxdV7neYoGGK?usp=sharing`

There are three *.zip* files in the shared Google Drive repository.

- KITTI_dataset.zip - This file contains all the images required for the training and evaluation of networks.
- pretrained_models.zip - This archive includes the saved *'U-NET TF v3'* model.
- source_code.zip - Extracting this file provides all source code Python files and the configuration files.

Download each of them individually (this is faster than requesting all together due to Google Drive's policies) in a directory of your choice and extract the contents. To perform the last step, you will need an extraction tool. I suggest using 7zip as it is free, supports many extensions and has a graphical user interface. To install this software, visit `https://www.7-zip.org/`, download the appropriate version for your hardware and install. Now, to *unzip* a file, navigate to the download directory, double-click on the file and find the 'Extract' action in the opened application window. Repeat these steps for all three files. The final folder structure should look like in Figure 1.

Project Structure

As shown in Figure 1, the project directory contains 7 Python files, 1 text file, 4 subdirectories and a serialised *.pickle* file. The full directory tree is presented in Figure 2. It first lists all Python and text files in the directory and then lists the subdirectories and their contents. The '+' sign indicates a directory, while the '\' symbol indicates that this is the last subdirectory within the parent folder. Indentation within the tree is also introduced to improve clarity. The following description list provides information about each of the project elements.

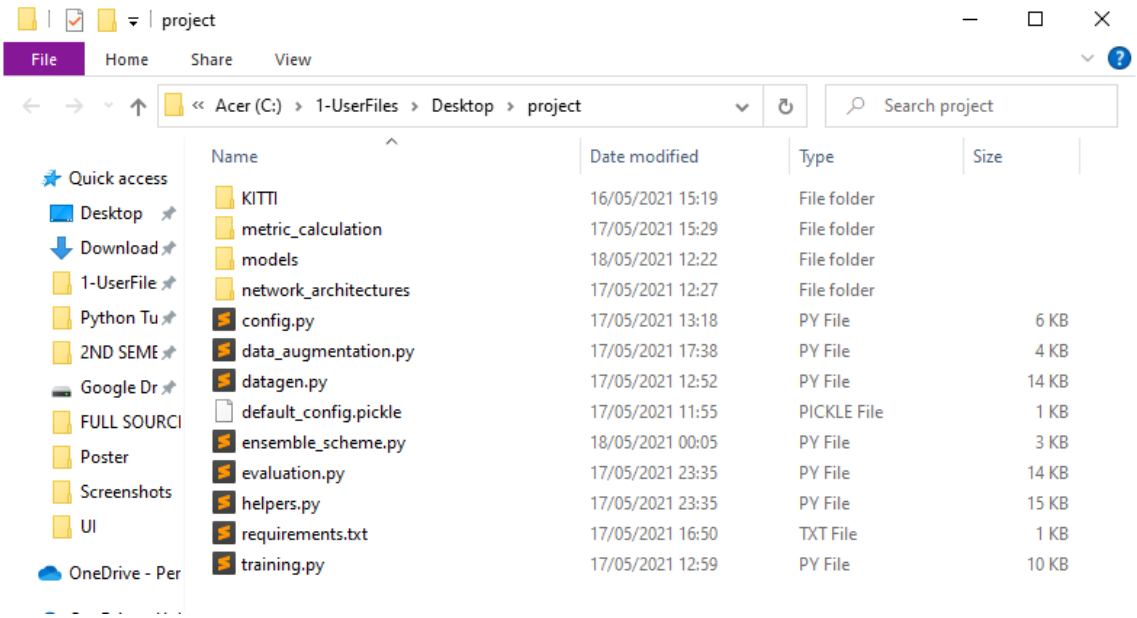

Figure 1: Expected directory structure for the project files.

Subdirectories:

The **KITTI/** directory contains all the visual data required for the training and prediction stages. The images are organised in subfolders within **KITTI/data_road/**. All annotated labels including the augmented ones reside in **enc_gt_images_1/**, while all training and testing raw images are contained in the **training/** directory. They are divided into classes - **um** (urban marked), **umm** (urban multiple marked lanes) and **uu** (urban unmarked). The **new_augs/** folder includes all augmented version of the original KITTI road images.

The **metric_calculation/** folder is used to perform the post-predictive evaluation. That is, after a model generates segmentation masks for some testing images, the files in this directory are used to calculate the performance of this model on these data. There are three scripts and three subdirectories.

- **helpers.py** - This Python file includes helper methods, a class definition and a reference dictionary required by the score calculation operations.
- **score_calc_and_save_to_pickle.py** - We use this file to calculate performance from predicted segmentation masks. It generates tables with scores, statistical results (from related t-test) and pickled/serialised Python dictionary objects with detailed measurements.
- **score_compare.py** - After all the scores have been calculated by the above script, this Python file is executed to generate plots, tables and figures to graphically visualise the performance for the various metrics.
- **ALL PREDICTED IMAGES** - This directory contains the predictions of all models that will be compared. It is required that the folders containing the segmentation masks have the 'pred_imgs' sequence in their names so that the

algorithm can recognise them. The segmentation masks have to be organised in two subdirectories - **ground_truth_masks** and **predicted_masks**.

- **Figures and Graphs** and **OUTPUT FILES** - These two folders are initially empty and they are to store the two Python scripts' outputs. The produced graphs, tables and images are saved to the former, while the latter is used by the score comparison algorithm.

The other two directories in the project repository, namely **models/** and **network_architectures** contain the pretrained models and the scripts for the initialisation of the neural networks, respectively. Each saved model is stored with additional log information - **config.txt**, **used_images.txt** and **loss_and_accuracy.pdf**. The first file shows what parameters were used to train the model. The second one contains the names of all the images the model learned from. The last PDF file is a figure containing two line charts - one for the loss and validation loss and another for accuracy and validation accuracy for all training iterations (epoch).

Python scripts:
A list of all Python source files with their roles and specific details.

- **training.py** - This script is executed when a model needs to be trained. It implements a wrapper method for the Keras `model.fit()` method. This wrapper generates plots of the progression of the loss and accuracy during training. It also saves the current model and its configuration as a text file within the directory where the model is exported. The method returns a `History.history` Keras object that contains details about the training process.

  In the main section of this script (which specifies the code segment executed if the script is initiated), we generate a DataGen instance that will supply data to the model. We also initialise a model from a given architecture - the two network architectures supported are included in the **network_architectures/** directory within the project repository. The model version must be specified manually on line 150 of the script. The two available versions are implemented using the methods `u_net()` and `unet_model()`. The former initialises the traditional U-net model with resize-convolutions as upsampling operations, while the latter creates a modified U-net addressed in the thesis as MobileUNetV2.

- **evaluation.py** - Here, we perform the first stage of the evaluation process. It involves loading a saved model from a given directory and using it to segment the road region from testing images provided by a DataGen instance. The script also allows the option to time the execution speed of the algorithms. Furthermore, users are allowed to view the predictions and decide whether to save them on the disk. The file is universal and works for both U-net and MobileUNetV2, as well as, for the ensemble method (provided the '-e' flag is requested in the script execution command).

- **config.py** - Here, we create a dictionary object containing configuration parameters and export it to the disk as a *.pickle* file. The values for the parameters are obtained either by parsing command-line arguments or by retrieving the default values of the variables. See the User Manual for usage and applications.
- **u_net.py** - Contains two methods that create a U-Net model, compile and return it. The only difference between the two is that the first one uses "resize-convolutions" as part of the expansive path, while the other employs transposed convolutions. The whole architecture is divided into blocks within the code using comments to increase readability.
- **mobile_net_v2.py** - Implements a method that returns the modified U-net architecture prepared for training. The method downloads the pre-trained MobileNetV2 architecture from the Keras package and selects 5 downsampling blocks from it for the encoder of the U-net. The upsampling is performed by applying transposed convolutions available in the **pix2pix.py** script which is originally implemented by TensorFlow[1].
- **metric_calculation/score_compare.py** - This script can be used to generate graphs and figures from calculated metrics in a pickled dictionary format. The first two methods create a scatter chart where the scores of the provided models for two metrics are plotted and shown on the screen. The next method produces an MS Excel table which includes the available models in the pickled dictionary and their mean scores for 5 of the performance metrics - Precision, Recall, F_1-score, IoU and Pixel Accuracy (PA). Finally, the last method retrieves the *p*-value of the difference between two models for a certain metric.
- **metric_calculation/score_calc_and_save_to_pickle.py** - Executing this script requires some predicted segmentation masks to be available in the **ALL PREDICTED IMAGES** directory within the project. Running the file analyses the provided segmentation masks and computes metric scores for all metrics - F_1, IoU, Pixel Accuracy, etc. It produces two Excel tables comparing each model's predictions to the remaining ones in the abovementioned directory. The first table provides percentage differences between every combination of 2 models, while the second one shows the corresponding statistical paired t-test results for the observed differences. It also produces several text files containing the same information, but presented differently.
- **metric_calculation/helpers.py** - This file contains two methods - a helper for visualising dictionaries and another one for loading dictionaries from *.pickle* files. It also contains a class definition for a bidirectional dictionary (Bidict) object. This class manages two dictionary objects - one mapping keys to values while the other associating values with keys (inverse of the first one). Lastly, a constant variable is defined in this file. It is a Bidict object that is used for

[1] It is licensed under the Apache License, Version 2.0 (the "License") which is available at `http://www.apache.org/licenses/LICENSE-2.0.`. The full version of copyright information is included in the script file (lined 1 to 13)

referencing between directory names of predicted segmentation masks and their corresponding model names.

- **helpers.py** - Most of the methods in this file are used for image visualisation or during the evaluation stage. There are methods that prepare images prior to them being inputted to the neural network for inference. Another set of methods interprets predictions and generates RGB segmentation masks and overlaid images. The DisplayCallback() object class responsible for visualising model's progress between epochs is also defined in this script.
- **datagen.py** - The DataGen class that implements the *keras.utils.Sequence* interface is defined here. DataGen is a subscriptable class to generate batches of training data in the form of Numpy arrays. The generated and returned data is a tuple of two tuples - one for the image batch and one for the label batch. This class also creates two generator objects that yield validation samples and testing samples. All important image operations before training is initiated are performed by this object including normalisation of pixel values, train-validation splits, data shuffling, batch assembly and data loading
- **data_augmentation.py** - This script is responsible for data preprocessing. More precisely, it manipulates existing images in order to generate augmented copies of them. During data augmentation transformations (zoom/crop, rotation and mirroring) are applied to every image-label pair. The operations are carried out by the efficient *Augmentor* Python package that implements a **DataPipeline** object which applies the given modifications in order. The essential feature of this class is that it offers identical parallel augmentation for images and their labels. Also, this library automatically zooms the image after rotation which prevents the appearance of black regions in the corners of the rotated image.
- **ensemble_scheme.py** - Only one method is implemented in this Python file. The '`apply_voting_to_ensemble_predictions(predictions, savepath)`' method takes a list of predictions as input and an optional output directory path. It merges the predictions of multiple models by applying a hard majority voting ensemble scheme. If the optional output path is provided, the method also generates and saves intermediate outputs of the vote computation process.

Other files:

The **requirements.txt** file contains the names of the required third-party Python libraries and their versions. It can be used with the **pip**[2] package to install all dependencies. For the purpose, open *Command Prompt*, head to the project directory and type:

```
pip install -r requirements.txt
```

This will download and prepare all packages for you.

[2]pip is a tool for managing packages in Python. It is connected to an online repository of packages and is tasked with downloading and installing libraries at request.

The **default_config.pickle** is a configuration file that contains a serialised Python dictionary object with all default training parameters. Details about the generation of this file and its usage are available in **config.py**.

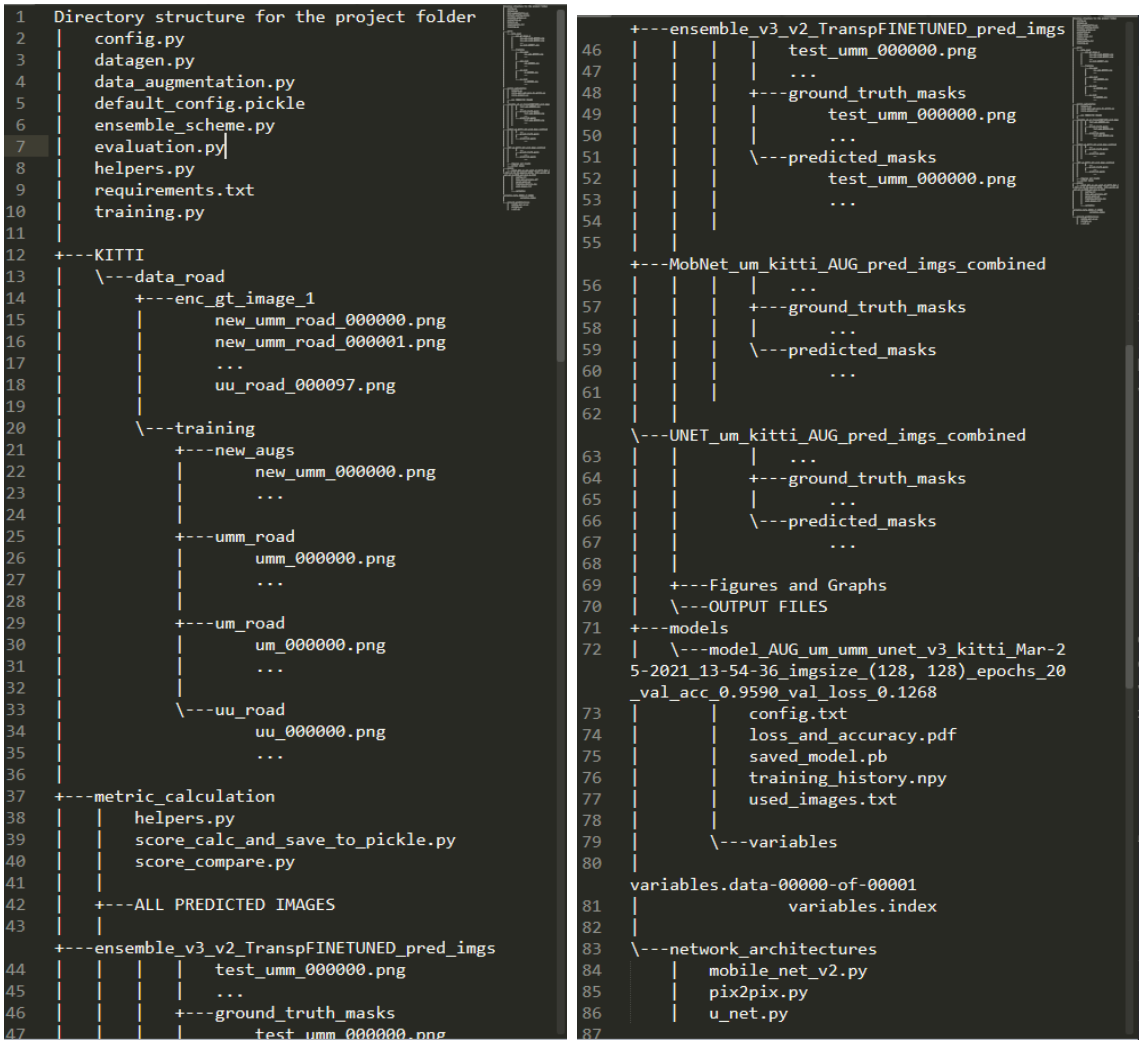

Figure 2: Directory structure of the project folder. The whole folder tree is separated into two screenshots to allow for better readability.

All Python files have extensively documented. Each class and method is described at the beginning of their definition by using docstrings. In-line comments have also been positioned next to most of the lines within the code to increase readability and interpretability further. Figure 3 provides an example of two helper methods and their associated comments. The User Manual shows examples of the usage of all scripts with descriptions and screenshots.

Dependencies and Requirements

Disk Space

The size of the project directory after all files are extracted is around 1 Gigabyte (GB). Saving a trained model usually requires around 750 Megabytes (MB) of free storage space. That is because two sets of weight are saved during the training process:

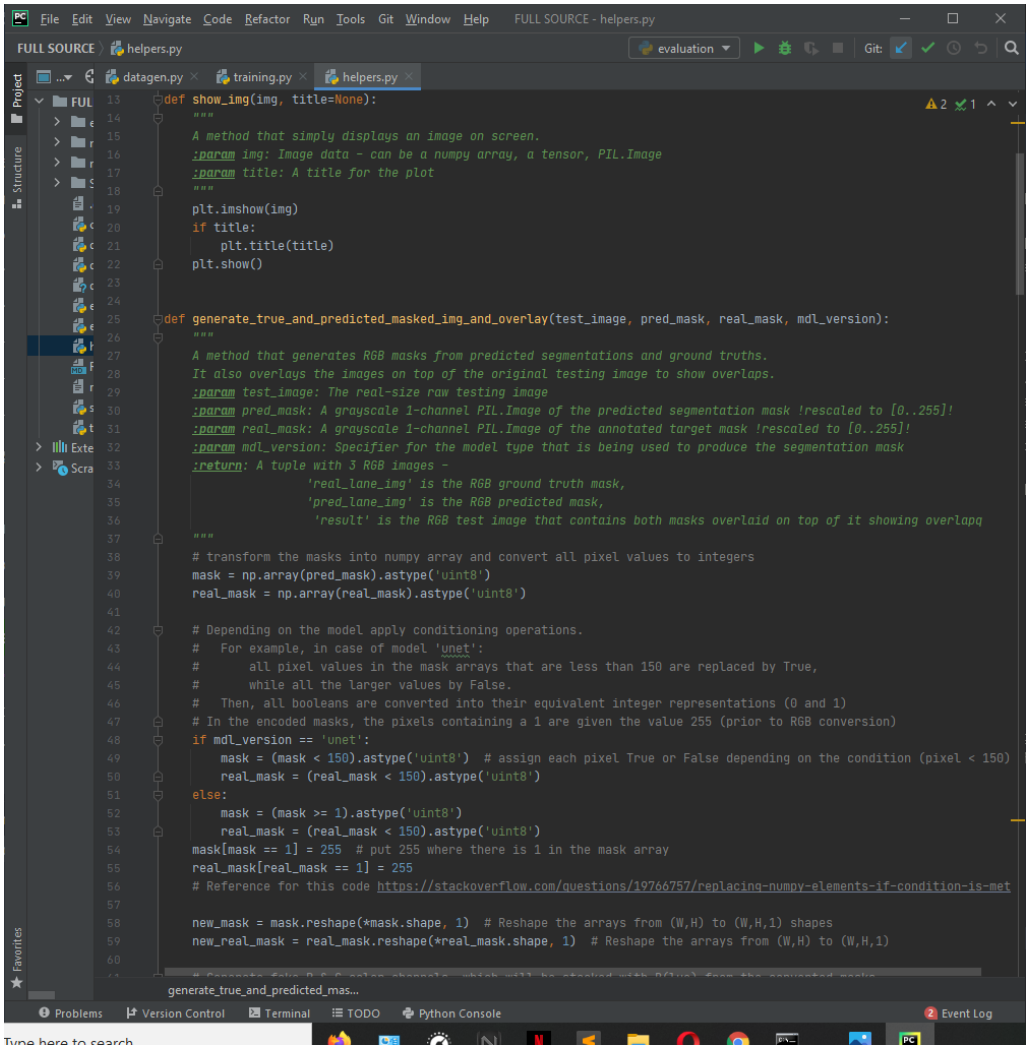

Figure 3: A segment of the **helpers.py** file. The green text below the definition of a method (encapsulated within """ ... """) provides an overview of the whole method. It also gives information about the different parameters (*:param*), as well as, the output of the method (*:return*). The grey text preceded by the hash symbol '#' indicates in-line comments that provide additional details about the adjacent code fragments.

- Only the best set of weights (here 'best' means the ones that produce the lowest.hdf5 loss). They are saved by one of the built-in Keras callback objects - **ModelCheckpoint**. File extension is *.hdf5*. This file does not include the network architecture and therefore it has to be accessible within the project.
- The final state of the model at the end of the training (after the last epoch). This also saves the network architecture and various other hyperparameters associated with the model. The model is stored in a folder that contains all required variables for loading. It does **NOT**, however, include the loss function and the optimiser objects, which means that they have to be installed on the system before a pretrained model is loaded.

System and Hardware

Python 3.9 and later is required to use and experiment with this project. To execute the script files, it is suggested to install the versions of the Python libraries specified in the **requirements.txt** file.

Note: Older versions of these technologies might also work, however, complications and incompatibility issues may arise due to version-specific functionalities.

A middle-class hardware configuration has to be sufficient for most applications. The speed of the processes like training and prediction depends strongly on the availability of GPU, RAM, CPU cores and the efficiency of I/O operations. The examples were tested on a higher-than-middle-class machine with the following specifications:

- Device name: Acer Nitro 5 (2019)
- Processor: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
- GPU: NVIDIA GeForce GTX 1650 4GB GDDR5 VRAM
- Installed RAM: 8.00 GB (7.85 GB usable)
- System type: 64-bit operating system, x64-based processor