Name: Trung Kien Nguyen

Student ID: 104053642

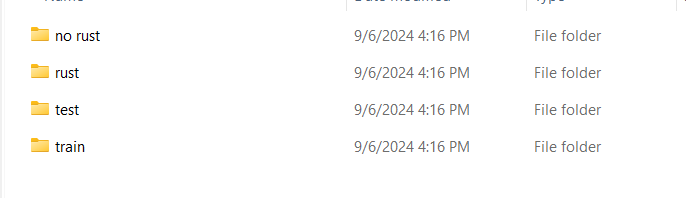
Studio: 1- 3

PORTFOLIO – WEEK 5

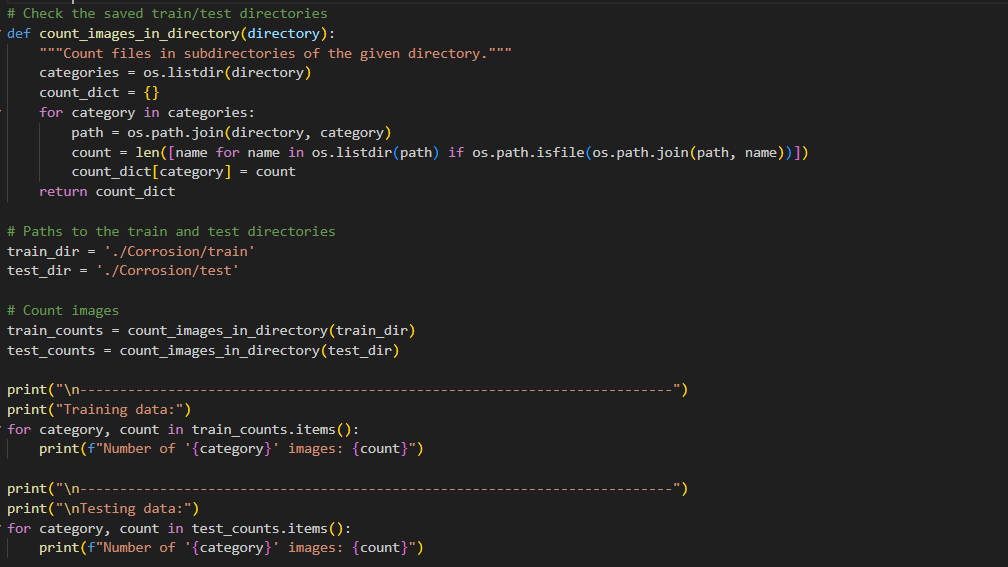
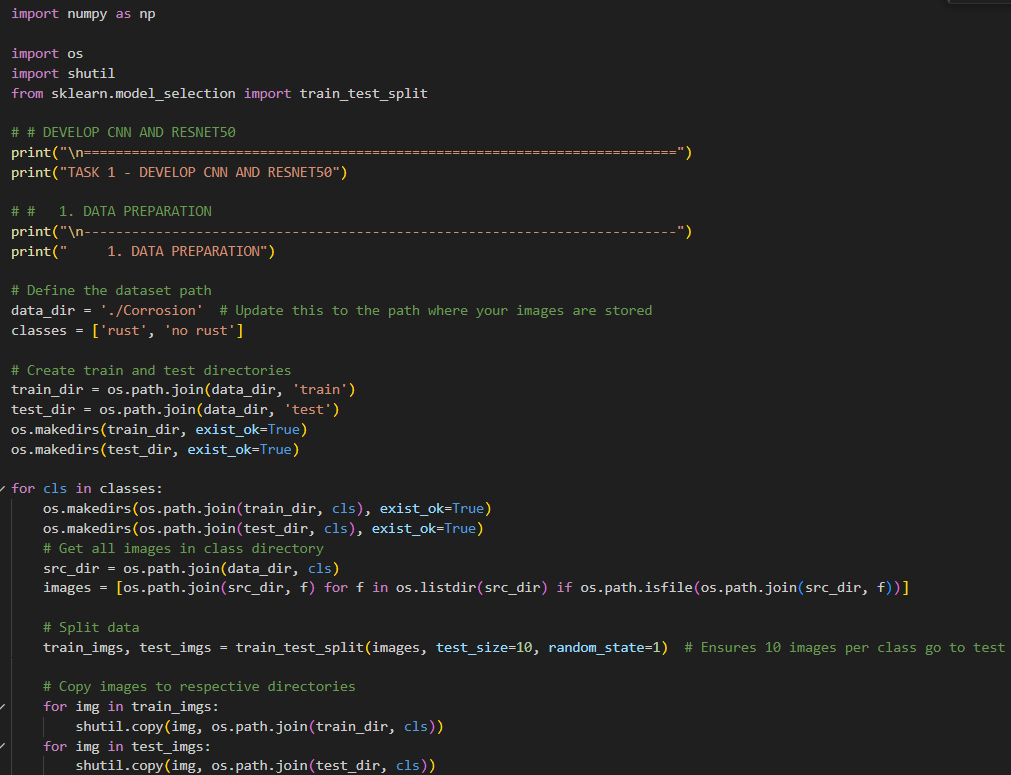
# Step 1: Develop CNN and Resnet50

## Prepare data:

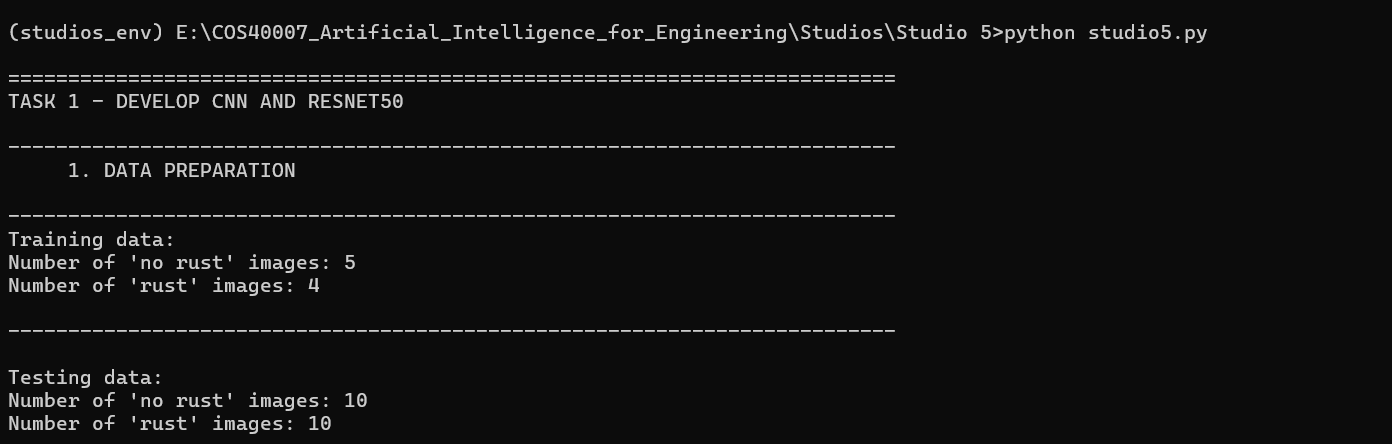
I have taken out 10 rust and 10 no rust images for testing from the directory Corrosion (<https://liveswinburneeduau-my.sharepoint.com/personal/fforkan_swin_edu_au/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Ffforkan%5Fswin%5Fedu%5Fau%2FDocuments%2FCOS40007%2FCorrosion%2D20240820T124029Z%2D001%2FCorrosion&ga=1>). The outputs are saved in two directories named “test” and “train” inside the Corrosion directory:



* “test” folder: This include 10 images from the original “Corrosion/no rust” directory, saved in the “Corrosion/test/no rust” directory, and 10 images from the original “Corrosion/rust” directory, saved in the “Corrosion/test/rust” directory
* “train” folder: This include the other 5 images from the original “Corrosion/no rust” directory, saved in the “Corrosion/train/no rust” directory, and other 4 images from the original “Corrosion/rust” directory, saved in the “Corrosion/train/rust” directory
* Here is the code to implement that, using os (for managing the directories and files), shutil (for copying image files to appropriate destinations), sklearn.model\_selection.train\_test\_split (for splitting dataset to prepare to save):



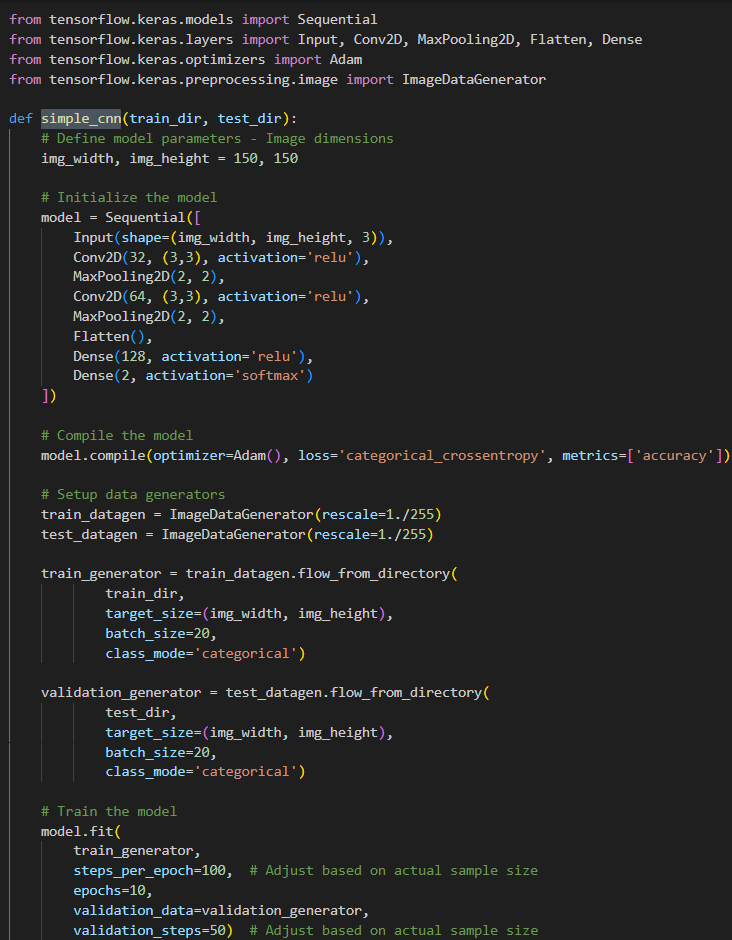
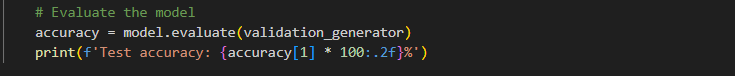
Here are the outputs for this:



I have also committed and pushed the Studio 5 directory, including the output of images splitting, which can be seen in this link:

## Simple CNN model

With the train/test data splitted and prepared from Corrosion folder, I have implemented a CNN model in the method simple\_cnn, with the help from the package tensorflow.keras:

The model has been defined as a Sequential model, meaning that layers are added in sequence:

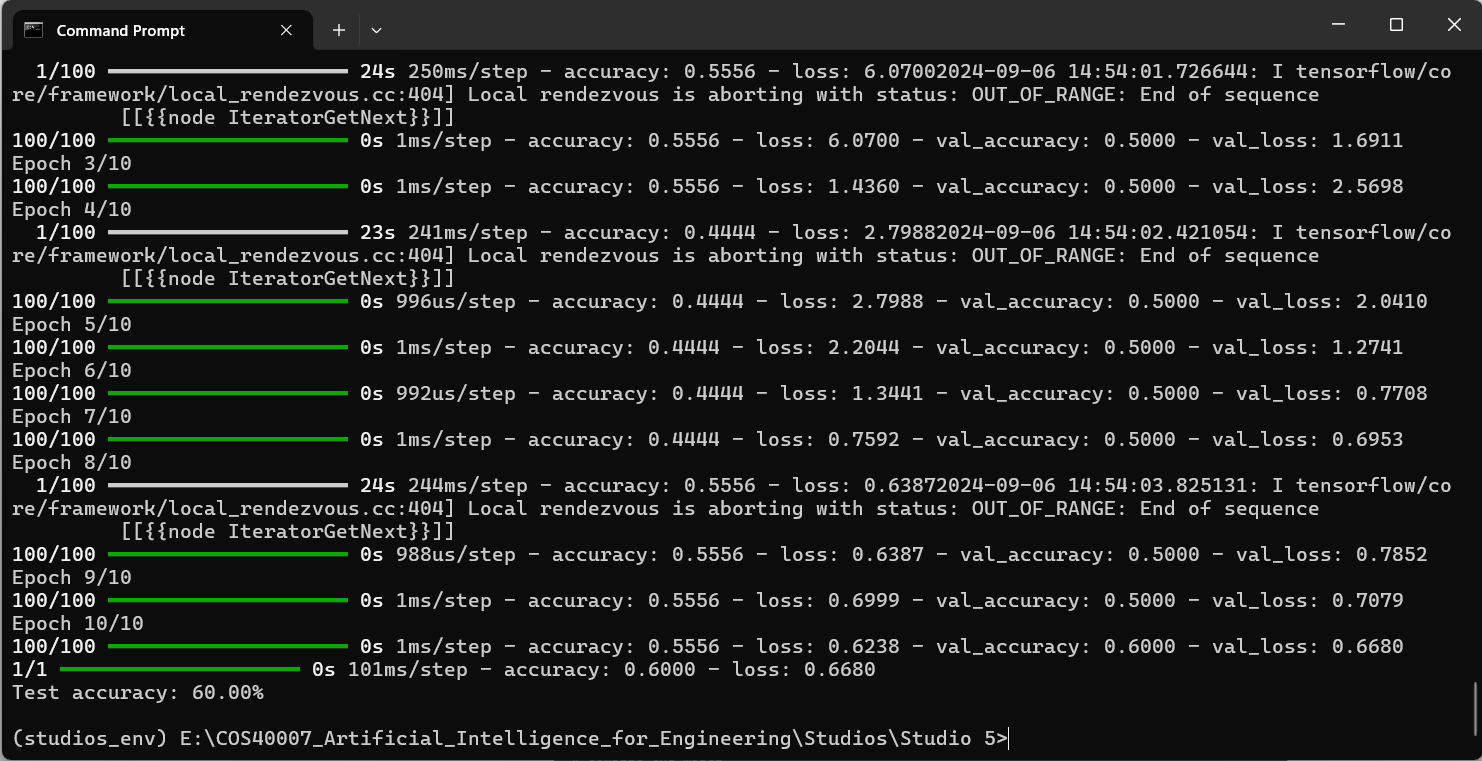
* Input: specify the shape of the input data, which includes image width, height, and 3 color channels (for RGB).
* Conv2D layers are convolutional layers that will help the model to extract features from the input images. There are two of conv2D layers, the first convolutional layer has 32 filters of size 3x3, and the second one has 64 filters of the same size, both using the ReLU activation function.
* MaxPooling2D is used to reduce spartial dimensions after each Conv2D layer
* Flatten convert any 2D features into 1D feature, to fit the next layer – Dense
* Dense layers are fully connected layers. There are two Dense layers used, one is with 128 units (ReLU), and the second is 2-unit (Softmax). So the outout will be the probability distribution between 2 class

The Adam optimizer is used, with a learning objective (loss) set to 'categorical\_crossentropy' suitable for multi-class classification.

Next, ImnageDataGenerator from tensorflow.keras.preprocessing.image is used to scale image pixel values to the range from 0 to 1 for model input. These generators will handle loading and modifying images from the suitable directories to batches.

* train\_generator will load images from train\_dir, which is set to be './Corrosion/train' in the previous step. It resizes the image to a pre-determined size (150x150)
* validation\_generator loads validation data in a similar manner from test\_dir, which is ‘./Corrosion/test’

After setting up CNN model, I train it with 10 epoch (train\_generator), then evaluate it with validation\_generator. Here is the output for these tasks:



Note that the test set is quite small, so the index 60% may not remain the same for all cases. The accuracy may vary between 50% to 60%

## ResNet50

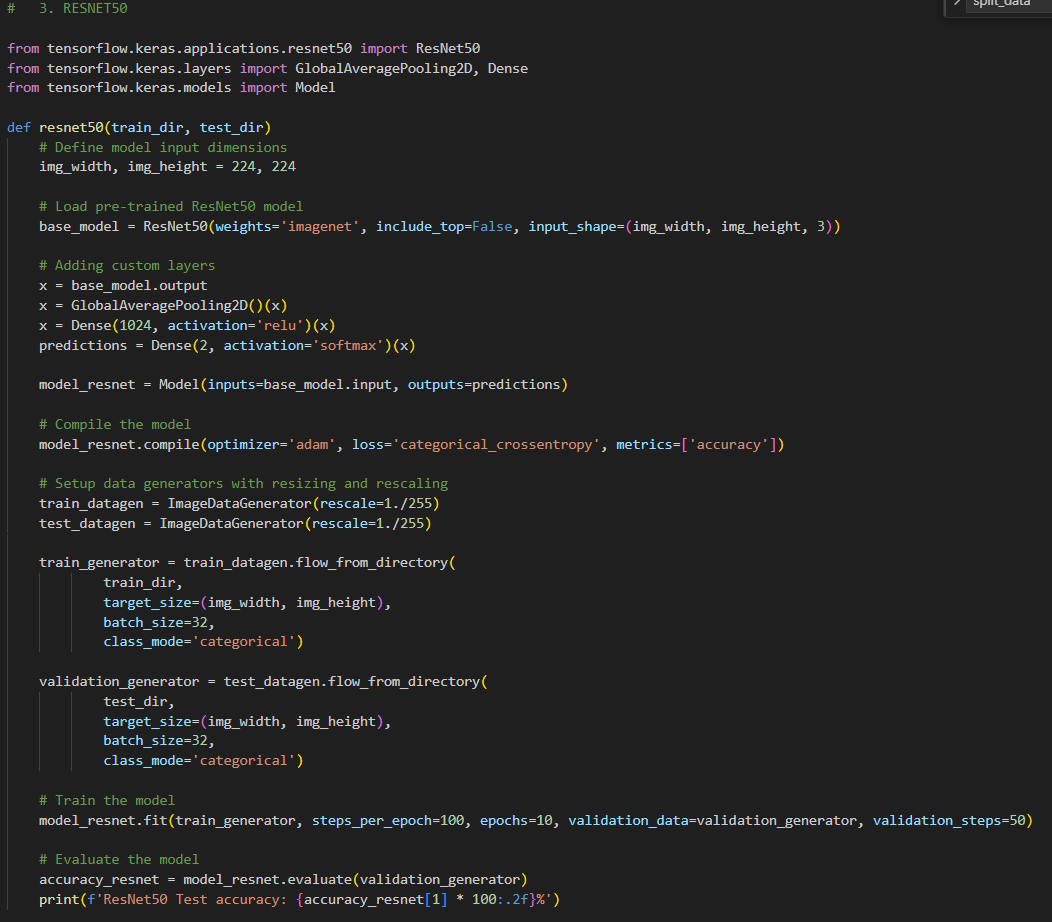
Using the same prepared dataset from Corrosion, I have also set up, trained and tested the Resnet50 model. Actually, I have defined a base model - a pre-trained one, with parameters as follows:

* weights: “imagenet”, which is a large visual database containinng over 14 million annotated images categorized into over 20,000 categories.
* input\_shapes is (224, 224, 3), with 224x224 is the average dimensions of images data observed by me, while 3 represents the three basic color: Red, Green and Blue.

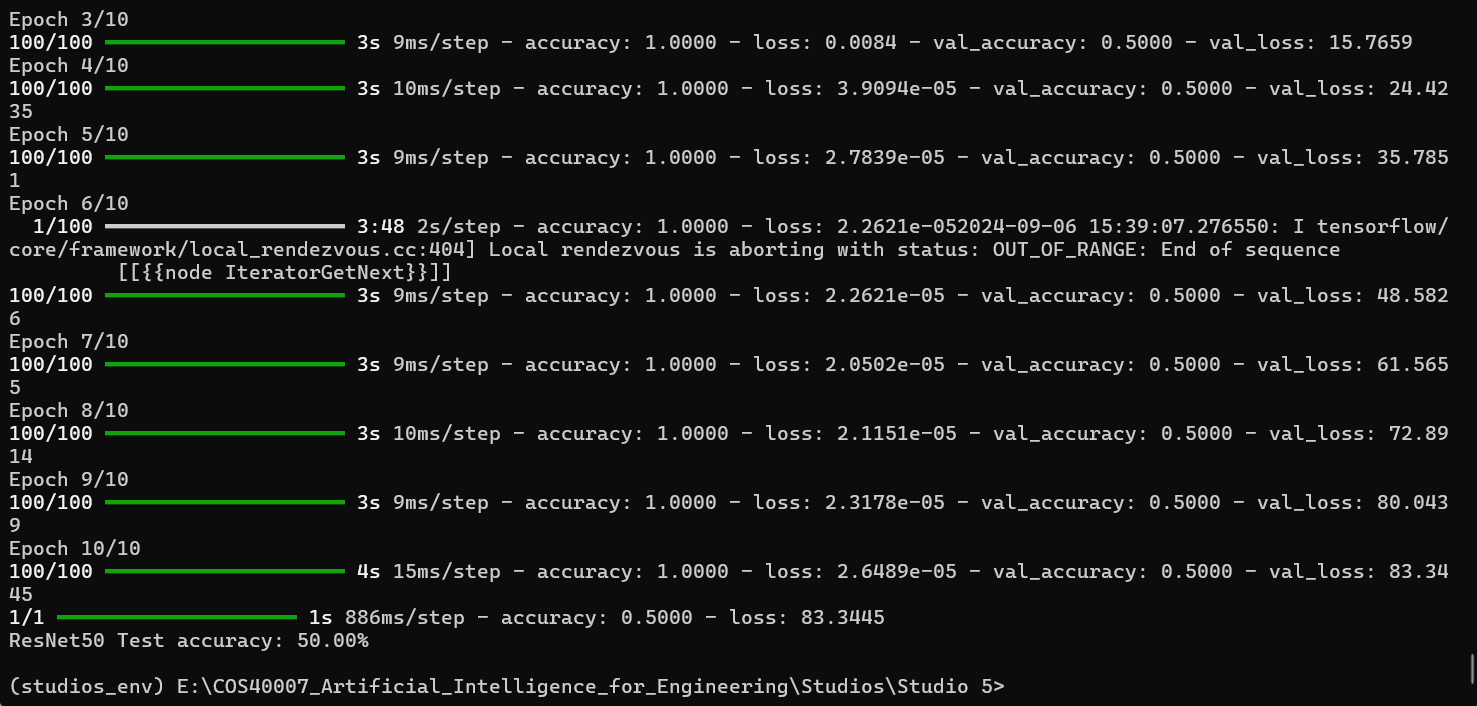
I then customized this base model with the following layers:

* a dense layer with 1024 units (ReLU activation function).
* A dense layse as the final layer (Softmax)m outputting probabilities for two classe similar to the above CNN

Then, I set up my final Resnet50 model (model\_resnet), use the input from the base model. The train and test data is the same train\_generator and validation\_generator like the above CNN, with the epoch of 10 in training process. Here is the implementation in the resnet50 method:



Here is the output of training/testing ResNet50:



Similar to the above CNN model, the test dataset is quite small (10 images), so the accuracy can be from 50% to 60% while running multiple times

# Step 2: Develop Mask RCNN for detecting log

## Prepare data

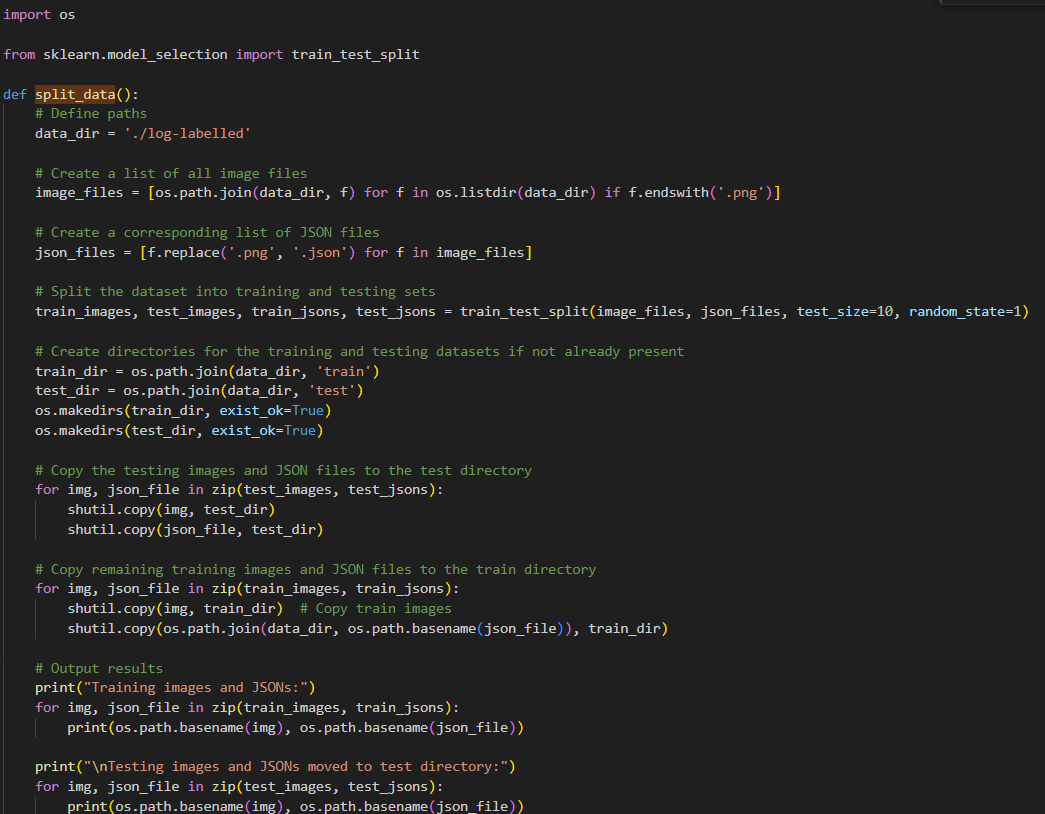
### Split data

The data for this task is from the directory log-labelled (<https://liveswinburneeduau-my.sharepoint.com/personal/fforkan_swin_edu_au/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Ffforkan%5Fswin%5Fedu%5Fau%2FDocuments%2FCOS40007%2Flog%2Dlabelled&ga=1>). This dataset included 600 image files, and 600 corresponding labelled JSON files for those images

Firstly, I have splitted data into test set and train set. The test set includes 10 images and 10 corresponding JSON labelled files, while the train set includes the other 590 pairs. You can view the log-labelled/train and log-labelled/test in the links below:

* log-labelled/train:
* log-labelled/test:

Here is the implementation for the splitting task, in the method split\_data:



### Convert to appropriate format for Touchvision’s Mask RCNN

The library for Mask RCNN model is torchvision.models.detection. To set up this model, I first have to convert the train/test dataset to appropiate format. The method generate\_coco\_annotations use the pairs of image and corresponding labelled JSON files, to generate one final annotations JSON file. (That means, one annotations file for test dataset and one annotations file for train set).

A COCO annotations file provides the following key information:

* Images: meta data of each image in the dataset (file name, id, size, ...)
* Categories: This is different types of objects that could be recognized.
* Annotations:
  + Image ID
  + Categori ID show the type of annotated object
  + Segmentation: For segmentation tasks, it offers the object's pixel-level shape. This could take the shape of a binary mask or polygon.
  + Bounding box: specify the coordinate of the rectangle enclosing each object
  + Area
  + IsCrowd: if the objects distribution is crowd or not

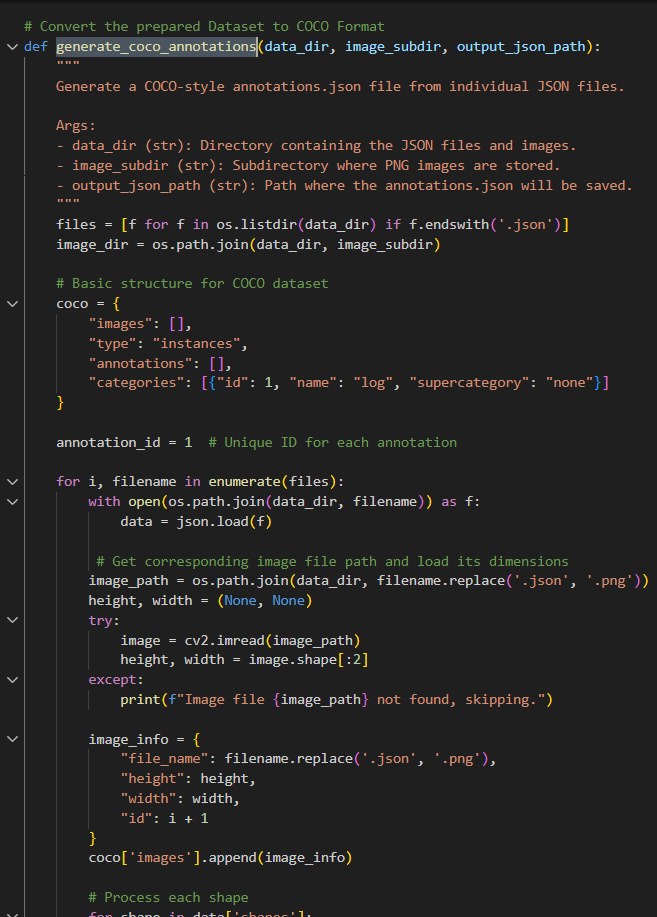
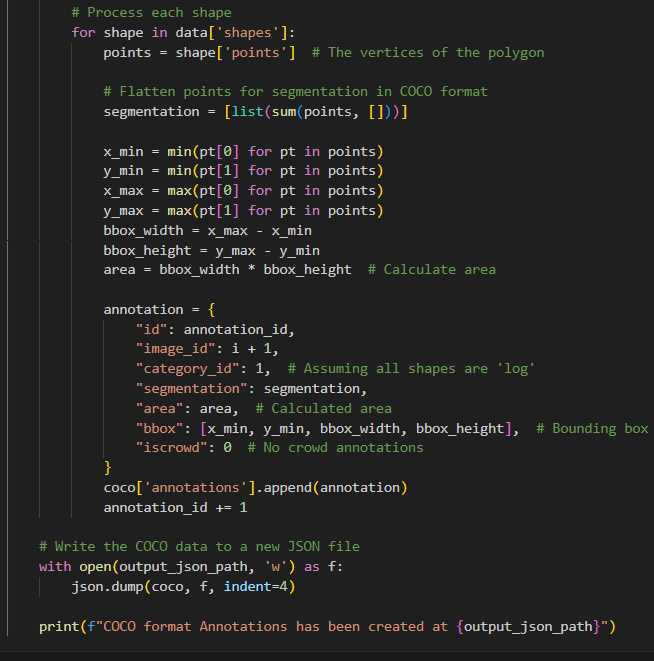
In terms of inplementation, after loading images and corresponding labelled jjson files with os, I have intialized the COCO structure, which is a dictionary (coco) including keys like images, type, annotations and categories

There is only one category in the dataset, which is “log”. I have also specified the corresponding id for this is 1.

Then I processed the annotations and converted to the format that COCO required:

* Information about segmentation is compressed and kept.
* determines the minimum and maximum x and y coordinates from the points to compute the bounding box (bbox), after which the width and height are determined.
* determines the bounding box's area.
* Adds the following annotation data to coco['annotations']: id, calculated area, bbox, and iscrowd set to 0 (signifying that these are individual object instances rather than crowd annotations).

Here is the implementation of method generate\_coco\_annotations, which is used for both train set and test set:

The two created annotations with coco format is saved in “log-labelled/train/” and “log-labelled/test/” directories respectively. Here are the links for those:

* train\_annotations.json:
* test\_annotation.json:

Also, I have create a method call setup\_image\_directories, using shutil and os to move all the images from each dataset (train/test) to separate folders of “images”, just for easier model implementing later. Here are the links for:

* “log-labelled/test/images”:
* “log-labelled/train/images”:

Here is the console output for COCO-suite format converting:



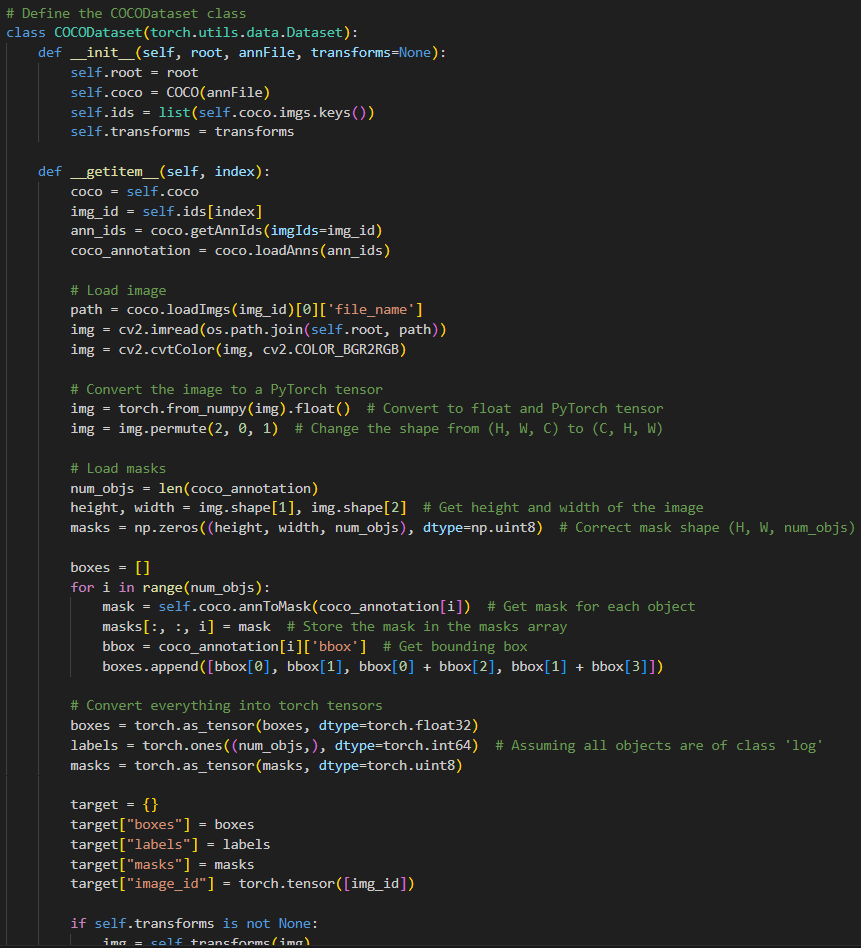
## Mask RCNN model setting and traininig

The train\_model method is responsible for this task.

Initially, I have implement the class of COCODataset (inheritted from torch.utils.data.Dataset), to summarise the images saved in the images file and the coco-suite annotations JSON file, from each dataset (train/test). This class has three methods that have been overidden by me:

* Constructer (\_\_init\_\_): The annotation file, dataset root directory, and any image transformations to be applied are loaded
* \_\_getitem\_\_: This fetchs a single data sample (e.g. when we index the dataset)
* \_\_len\_\_: simply return the length of the dataset

Here is the implementation for this class:



So, with the train dataset in the log-labelled folder, I have create the object of COCODataset for this.

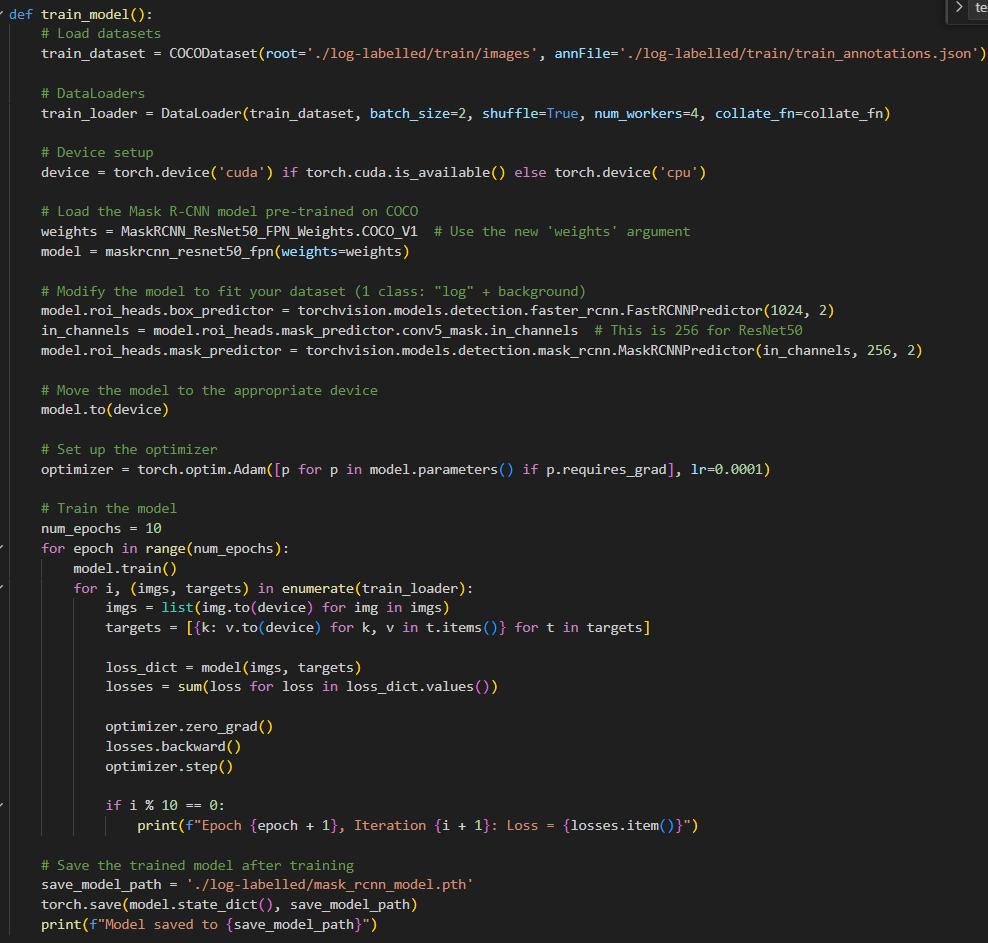
The COCODataset instance cannot be used to input to the used mask rcnn model (maskrcnn\_resnet50\_fpn in torchvision.models.detection), so I have wraped them in the corresponding torch.utils.data.DataLoader instance. This class also allows to set up batch sizes splitting, which I have input the value of 2

The optimizer for the training process is also Adam (torch.optim.Adam)

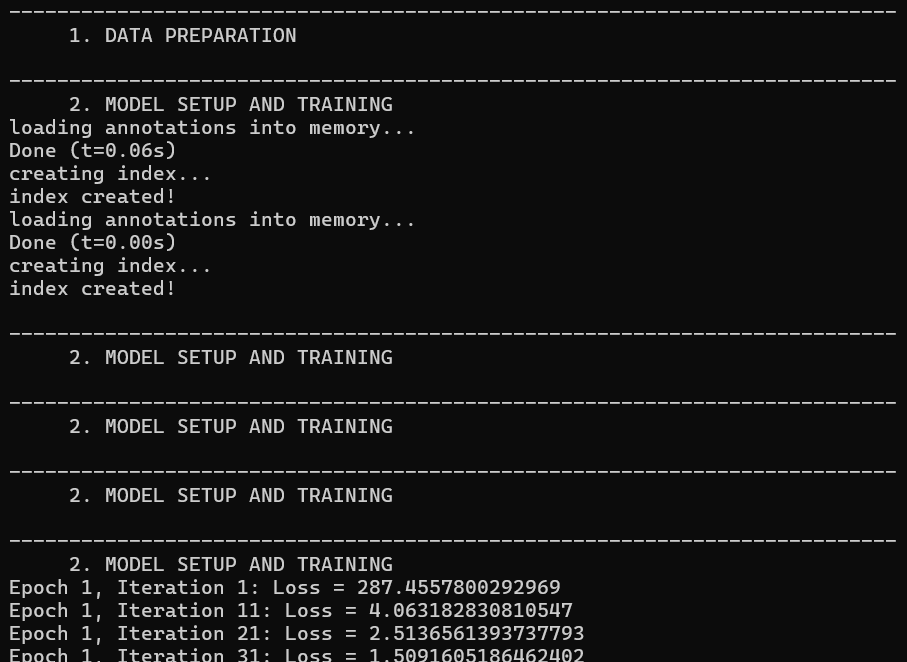
The number of epoch is 10.

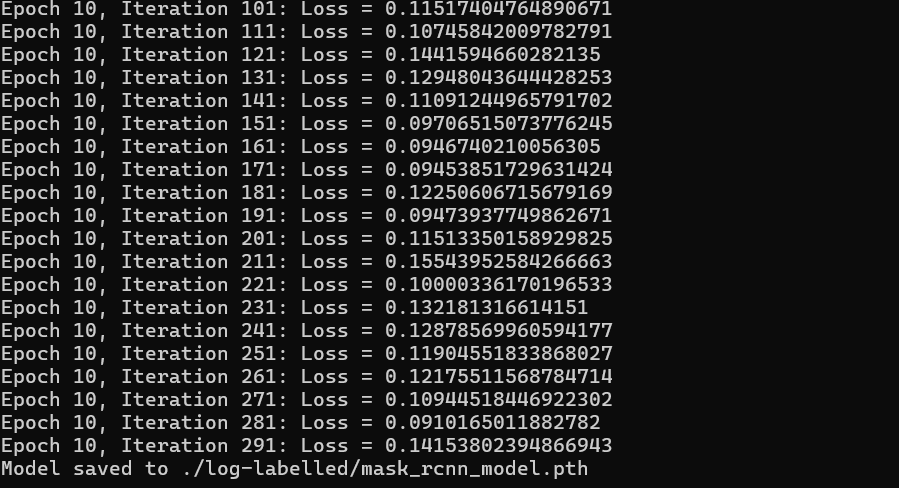
The model is then saved (using torch.save) to the log-labelled folder, so that I can test it in the future without the need for re-training.

Here is the link for the implementation:



And, these screenshots are the console outputs for the training process:





Here is the link for the trained one:

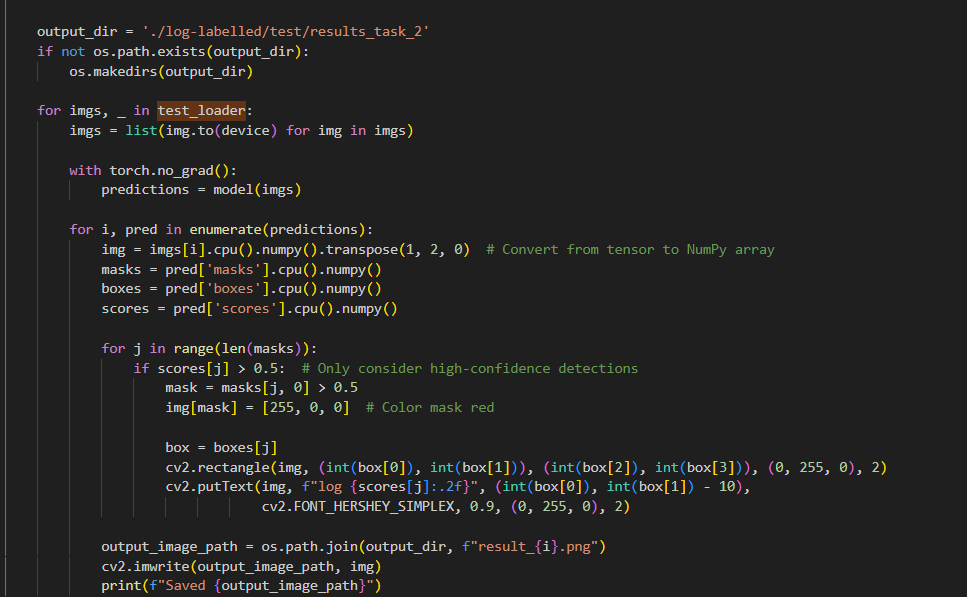
## Mask RCNN Testing and Evaluating

For testing the above model, I have implemented a method named “test\_model” in the file studio5\_task2.py.

* The function starts by loading the saved weights of the Mask R-CNN model, which named “mask\_rcnn\_model.pth” trained in the above stage
* Create COCODataset and DataLoader instance for test dataset
* Then I performed inference on test images (located in log-labelled/test directory). The method iterates over the test dataset, retrieving batches of images from test\_loader
* The model predicts bounding boxes, masks, and confidence scores for each image.
* Visualization:
  + Check the confidence score. If detection is high-confidence (confidence score > 0.5), then we will process the visualization for that output
  + Color the detected log in red.
  + A greenbounding box is drawn around the detected object
  + The output images are saved in the directory ‘log-labelled/test/results\_task\_2’

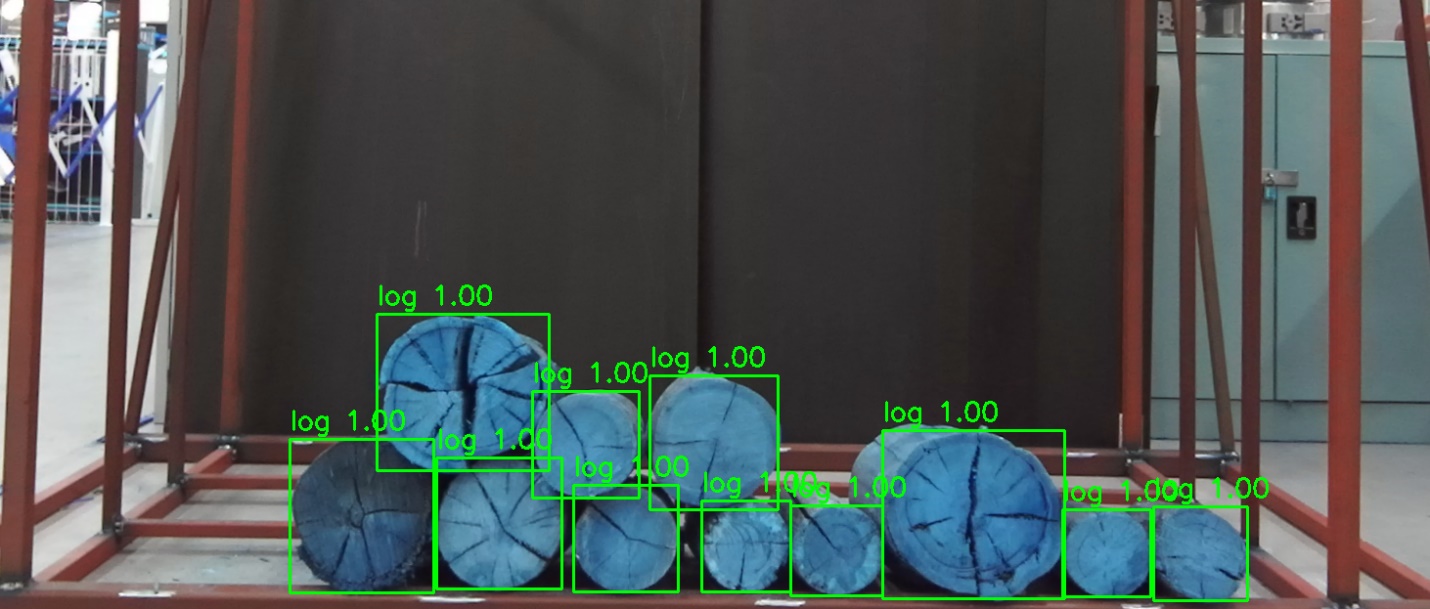
Here is the implementation for test\_model





Here is the part of the console output for testing stage:

And, finally, some output generate images include:



*result\_0.png*



*result\_1.png*

You can find all of those via this link of ‘log-labelled/test/results\_task\_2’ directory:

# Step 3: Extending log labelling to another class