NIPES

Journal of Science and Technology Research

Journal homepage: www.nipesjournals.org.ng

Oil Palm Plantation Detection in Satellite Image Using Deep Learning

[1]*Fidelis Odinma Chete* and [1]*Vincent Akinwande*

[1]Department of Computer Science, University of Benin, Benin City

Article Info

Keywords:
Computerized Mapping, Convolutional Neural Network, Deep Learning, Oil Palm, Satellite Imagery

Received 4 July 2022
Revised 5 October 2022
Accepted 8 October 2022
Available online 3 Dec. 2022

https://doi.org/10.5281/zenodo.7393401

Abstract

The traditional approach, such as manual surveys, used by farmers and governments to detect where oil palms are planted has proven to be ineffective, tedious and time consuming. This problem can be addressed if images obtained from satellites can be used to scan through the forest to detect oil palm plantations. In this paper, we developed a web based deep learning system that is capable of processing a satellite image of land to detect the presence of oil palm plantations in Nigeria using high-resolution satellite imagery. This research designed a detection system which uses a convolutional neural network to extricate important features, and a classifier trained using satellite images. Results showed exceptional effectiveness with a training loss of 0.11 and an accuracy of 99.0%. Utilizing different images for validation taken from diverse elevations, the model reached a training loss of 0.245 and an accuracy of 82.9999% on validation data, while on test data we got 1.59 in loss and an accuracy of 87.5%. Thus, the proposed approach is seemingly effective within the field of precision agriculture.

1. Introduction

Oil palms are valuable and play an important role in the economies of some countries [1]. It is one of the most rapidly expanding crops in tropical regions due to its high economic value [2]. The oil palm industry is a dynamic sector due to the breadth of uses of palm oil produced from palm trees. The uses range from cooking to cleaning products, special greases and lubricants, personal hygiene and cosmetics, production of biodiesel and electrical energy [1]. Additional uses as submitted by [3] include the production of cooking oil, food additives, cosmetics, industrial lubricants, and biofuels. Historically, agriculture was the backbone of the Nigerian economy amid the pre-colonial and the colonial period and Nigeria was a driving exporter of palm kernel, and biggest producer and exporter of palm oil [4]. Given that the cultivation and harvesting of palm oil has its major advantages in Nigeria, it also comes with excessive problem of deforestation when demand is in very large quantities. According to a study by [4]**,** expansion of oil palm as the major driver of forest loss in Cross River State is having strong negative impacts on biodiversity and the livelihoods of local communities. Since plantations require the clearing of timberlands, which occurs within the adjustment and debasement of the environment, residents in the communities are then impoverished due to the loss of a forest as a source of income, land, social and cultural values [4]. To reduce the negative effects of numerous oil palm plantations, the rights and welfare of individuals must be prioritized in the quest for economic improvement. To achieve this, we leverage on the increasing use of Artificial Intelligence (AI) models for image analysis such as deep learning, which has attracted the attention of analysts and researchers. A deep learning application is one in which an algorithm is trained to automatically recognize an object (in this case the presence of oil palm in a satellite image), the result of this system can then be used for further analysis and planning like determining the appropriate location to cultivate oil palm. This study develops a web based deep learning system that is capable of processing a satellite image of land and running computation on it to detect the presence of oil palm plantations. The main objectives of this study include:(i) to gather data related to lands containing oil palm and those that don't have (ii) to create a neural networks model that is capable of predicting the presence of oil palm in an image with great accuracy and without over-fitting and convert results such as predictions and accuracy score into a form of Rest API that would be fed to a web application (iii) to connect the Rest API to a postman front end that can collect satellite data from users, send the data to the backend, and display the results in a presentable form.

The developed system can be used to detect the areas of land that contain oil palm and the ones that don't. The implementation of the system can help in solving the problem of deforestation and the negative impact it has on the environment and the people. The system may, however, contain some limitations and defects as it is a prototype. There are 195 countries in the world and each country has different states which also have different conditions that can in turn have effects on the satellite imagery. Consequently, regions, states and countries can have different variations of images with oil palm oil present or absent. All these variations can simply not be covered in this study; thus the scope of this study is limited to satellite imagery obtained from Kaggle, an online platform that allows users to find and publish data sets, explore and build models in a web-based data-science environment.

1.1 Related Work

Using high-resolution remote sensing images for Malaysia [5] proposed the first deep learning-based framework for oil palm tree detection and counting. The researchers used a number of manually interpreted samples to train and optimize the convolutional neural network (CNN), and predict labels for all the samples in an image dataset collected through the sliding window technique. The method achieved a detection accuracy of 96% in a pure oil palm plantation area in Malaysia.

[6] proposed a two-stage convolutional neural network (TS-CNN)-based oil palm detection method using high-resolution satellite images (i.e. Quickbird) in a large-scale study area located in the south of Malaysia. The TS-CNN consists of one CNN for land cover classification and one CNN for object classification. The two CNNs were trained and optimized independently based on 20,000 samples, in four classes, collected through human interpretation. The approach achieved a much higher average F1-score of 94.99% in the study

area compared with existing oil palm detection methods (87.95%, 81.80%, 80.61%, and 78.35% for single-stage CNN, Support Vector Machine (SVM), Random Forest (RF), and Artificial Neural Network (ANN), respectively.

In a study by [7], an automatic end-to-end method based on deep learning (DL), for the detection and counting of oil palm trees from images obtained from unmanned aerial vehicle (UAV) drone was proposed. In the study, the acquired images were first cropped and sampled into small size of sub-images, which were then divided into a training set, a validation set, and a testing set. The study employed a DL algorithm based on Faster-RCNN to build the model, extracted features from the images and identified the oil palm trees, and gave information on the respective locations. Thereafter, the model was trained and used to detect individual oil palm tree based on data from the testing set. The research then measured the overall accuracy of oil palm tree detection from three different sites. The results revealed 97.06%, 96.58%, and 97.79% correct oil palm detection. Based on these results, the study concluded that the proposed method is more effective, accurate in detection, and correctly counts the number of oil palm trees from the UAV images**.**

[8], in a study to detect young and mature oil palm, utilized two different convolution neural networks (CNNs) and also GIS during data processing and result storage process. The researchers exported prediction results to GIS software and created oil palm prediction map for mature and young oil palm. The results gave the accuracies for young and mature oil palm as 95.11% and 92.96%, respectively. The study concluded that the classifier performed well on previously unseen datasets, and was able to accurately detect oil palm from background.

In this research, we developed a web based deep learning system that is capable of processing a satellite image of land and running a computation on it to detect the presence of oil palm plantations.

2. Methodology

The image data would be gotten from the user through the desktop application platform provided using postman, which then transfers the data to the backend/ML model that computes for the presence of oil palm in the image and the accuracy level of such prediction. This research intends to utilize three tools in the development process. First, Pytorch, a deep learning framework for python will be used to train the model. The second is Flask, a python library that can be used for both back-end/frontend tasks. In this study, this library will be used to convert the results of the model to a Rest API format. Lastly, Postman, a collaboration platform for API development, will be utilized to provide a form-like platform for the user of the system to get data, send the data to the back end service and display the results.

2.1. Systems Analysis and Design

One of the easiest ways to locate oil palm is to manually mark the oil palm in the picture or use GPS to conduct a field survey to determine the location of the oil palm and display their location in the picture. However, where there are many oil palm plantations, which may be very large, and contain more than 1,000 oil palm trees, manual inspections and on-site inspections may be, tedious, time-consuming and costly. This is where remote sensing methods come into play. This work focuses on using high-resolution satellite images to identify and recognize oil palm images.

2.2. Proposed System

The proposed system performs high level land cover classification on images obtained from satellites. In general, the classification of land cover is carried out by classifying pixels of similar attribute/value depending on the used classifier. Thereafter, each pixel is assigned with a specific class across the image. This process is referred to as a pixel-based classification approach. Additionally, merging pixels with similar value results in regions of multiple scales to be clustered and classified based on its texture, context, and geometry [9]. Once the model training phase of the system is completed, the model can then be tested easily using series of images, sequential or in random order to determine if oil palm trees are present or not. Since each image fed to the model will be treated as independent images, the order would not matter and this is also an extra advantage given that training on initial size of the satellite of the image data would be computationally expensive; consequently we divided the images into parts, labeled them and trained independently, so even if a section of the divided image is missing it will still be able to detect the presence of oil palm. To aid the easy accessibility of this system, an Application Programmable Interface (API) was utilized; thus anyone can easily use the program in their existing software or in software they are currently building, and accessing it is as easy as just calling a line of code and passing the appropriate image parameter to it.

2.3. Data Collection

The dataset used for this study was obtained from Kaggle, an online platform for machine learning engineers, deep learning engineers, data scientists, data engineers and the likes. Kaggle's online platform created a competition that was intended to create a model that predicts the presence of oil palm plantations in satellite imagery, making available about 20,000 labeled satellite images from various locations. The dataset images are 3-meter spatial resolution and each is labeled with whether an oil palm plantation appears in the image or not (0 for no plantation, 1 for any presence of a plantation). Figure 1 depicts an image that contains oil palm plantation. Figure 2 shows an image that does not contain oil palm plantations**.**

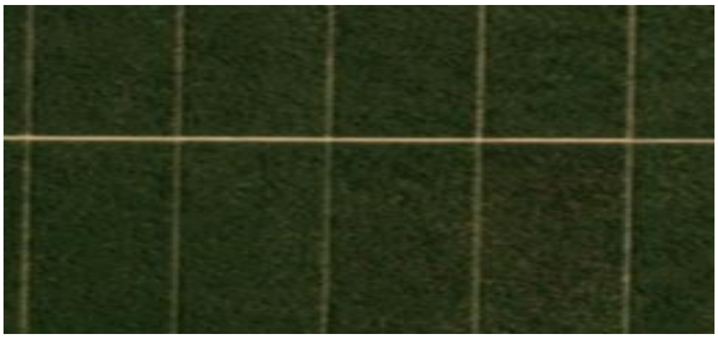


Figure 1: Sample of dataset with oil palm plantation**.**

(Source: https://www.kaggle.com/c/widsdatathon2019/data)

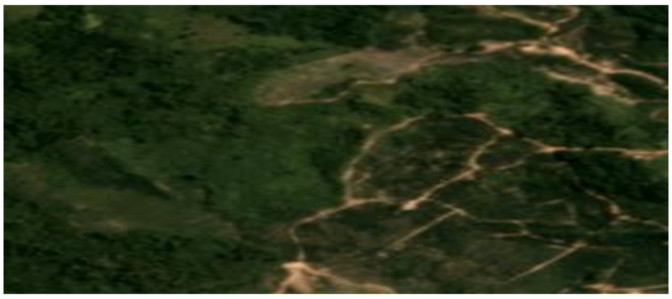


Figure2: Sample of dataset without oil palm plantation (Source**: https://www.kaggle.com/c/widsdatathon2019/data)**

2.4. Data Annotation

In deep learning, data annotation can be referred to as all the processes involved in assigning a class to an object or a data instance. This process is very important because it is the steps that determine the exact features the model is expected to learn. A well labeled dataset can bring about a very accurate model. In this work, the datasets was gotten from Kaggle and the classes for each image instance were already assigned. There are two major classes viz: the first is the class of images that has oil palm present in them, while the second class has no oil palm present in them. To distinguish between the two classes, we have a 'csv' format file that contains references to both classes. This file has both the reference to the images and label of the images. Each row in the 'csv' file is what would be used as a form of reference to the real images during the training and validation process. Table 1 depicts a dataset where a class (0 or 1) and score (0 - 1) is assigned to each image data.

Table1: Sample of dataset annotation in `csv` format (Source:**https://www.kaggle.com/c/widsdatathon2019/data)**

IMAGE_ID	HAS_OIL PALM	SCORE
img_000002017.jpg	0	0.7895
img_002012017.jpg	1	1
img_000022017.jpg	0	1
img_000072017.jpg	0	1
img_002232017.jpg	1	1
img_000092017.jpg	0	1

2.5. Data Splitting

The process in ML in which models are trained on all the available datasets, resulting in an accurate model that works great only on those datasets but not on other data is called over fitting. This is a very common problem in ML which is better avoided; one way to solve it is through data splitting. The concept behind data spitting is: we hold some part of the dataset by splitting it into different ratios, one part that would be used during training (training dataset), another part for testing as we train (testing dataset) and a final part for testing when training is done (validation dataset). In this study, we split the datasets into three parts as follows: 80% for the training dataset, 10% for testing dataset while the last 10% goes to the validation dataset.

2.6. Data Augmentation

In deep learning, when solving for a computer vision task, amongst lots of preprocessing that could be done to a data before training or testing, data augmentation is a very handy tool or preprocessing step to have as it helps create a stronger model [10]. It is worth noting that while data augmentation can be performed both in the training phase, testing phase and validation phase of a model, it is mostly used mainly in the training phase as the testing phase experiences the use of totally different and new datasets. Data Augmentation can range from just a few pixels shifting to rotating the image to changing the color space to flipping the image horizontally or vertically or randomly cropping some parts of the image. For this study, the process of the data augmentation is as follows: We used popular operations for Data Augmentation in frameworks (like pytorch or tensorflow) such as random image crop, random vertical flip, random horizontal flip, color jitter, random rotation, etc. Randomly cropping and image helps hide some aspects in the data while expecting the model to adapt and learn the important feature from the newly augmented image. Vertical Flip is the process of reversing the active layers of the image in the vertical direction; that is, from top to bottom but leaves the dimensions of the layer and the pixel information of the layer unchanged. Horizontal flip is the process of reversing the active layers of the image in the horizontal direction that is, from left to right but leaves the dimensions of the layer and the pixel information of the layer unchanged. When applying Data augmentation to an image, it needs to retain its image format but for other transformation steps like normalization, we first need to convert the image or augmented image into tensor. Tensor can generally be thought of as a vector or matrix and can easily be understood as a multidimensional array. Thus, converting an image to tensor simply means representing the image pixel with numerical values that can be manipulated using algebraic operations and can also have matrix operations executed on them. Since this is a required step before some operators like normalization can occur, this step comes first both in the training phase as well as the testing phase. When images are converted into tensors, they assume a range of values between 0 and 255, these values are then scaled to have a range of 0 - 1, this method is done using the equation given as follows:

$$\boldsymbol{x'} = \frac{(\boldsymbol{x - x\_min})}{(\boldsymbol{x\_max - x\_min})} \text{………………... (1) (https://stackoverflow.com/)}$$

where x′ is the new value and x is the input value. Ideally you would normalize values between [0, 1] then standardize by calculating the mean and std of your whole training set and applying it to all datasets (training, validation and test set) (https://stackoverflow.com/) **.**

After the Normalization is done, after the scaling process, we then normalize (standardize), for instance with the z-score, which makes mean(x') =0 and std (x') =1:

mean, std = x.mean(), x.std() which then gives

$$\boldsymbol{z} = \frac{(\boldsymbol{x - mean})}{\boldsymbol{std}} \text{………………… (2) (https://stackoverflow.com/)}$$

In this research, the mean value and standard deviation value were made constant at 0.5

(https://stackoverflow.com/), this helps convert the range from [(0) - (1)] to [(-1) - (1)] **.**

The reason for normalization in a deep learning research is because normalization helps the network converge faster.

2.7. Algorithm

The algorithm chosen for this system is the Deep Convolutional Neural Network (DCNN) algorithm. It is a deep learning algorithm and though difficult to interpret, has an edge over other similar algorithms on accuracy [11], for the type of data we wish to model

2.8. Feed Forward

After completing the setup of the model (either a simple neural network or a DCNN) and after completing the preprocessing of the dataset, the next step involved is the training step. To aid this, a concept called feed forward is used. Feed forward is the process by which processed data is fed into the network to mathematically compute the output from the network using weights and bias. This can be done using various forms of matrix operations like matrix multiplication, matrix addition, transpose etc. In this study, while training the proposed DCNN, 80% of the complete dataset went through an iterative process of the feed forward step where we continually aimed to get the output for each data and comparing it with the target to obtain the loss or error made by the current weights and bias in the network.

2.9. Loss Function

The Loss Function is a very important component in building a model using Neural Networks. Loss is an error in prediction made by the model and the method to calculate the loss is called Loss Function. In simple words, the Loss is used to calculate the gradients and in turn, we use the gradients to update the weights and biases of the model. There are various methods used when it comes to calculating the loss of a network or model. We have mean square error, binary cross entropy, categorical cross entropy, sparse categorical cross entropy etc. In this study, binary cross entropy was used because the output from the model is a binary result; it is either a zero (0) or a one (1).

2.10. Backpropagation

The real ability for a neural network to learn comes from a correctly implemented backpropagation algorithm. It can be considered to be the method of fine-tuning the weights of a neural network based on the error rate or gradients obtained in the previous epoch or iteration in the training step. The aim is to properly tune the weights and biases so as to reduce error rates and make the model reliable by increasing its generalization.

2.11. Deep Learning Architecture

ResNet (Residual Network) is an innovative neural network deep learning model that was first introduced by [12]. The Deep learning architecture used to train the model is the pre-trained Resnet-50 from the pytorch model zoo which is an implementation of the ResNet-50 model from the research by [12]. ResNet50 is a state of the art network that combines convolutional and recurrent components to capitalize on the spatial regularity of natural images and effectively learn the optimal depth of the network ([12] as cited in [13]). Figure 3 shows the Resnet Architecture as given by [14].

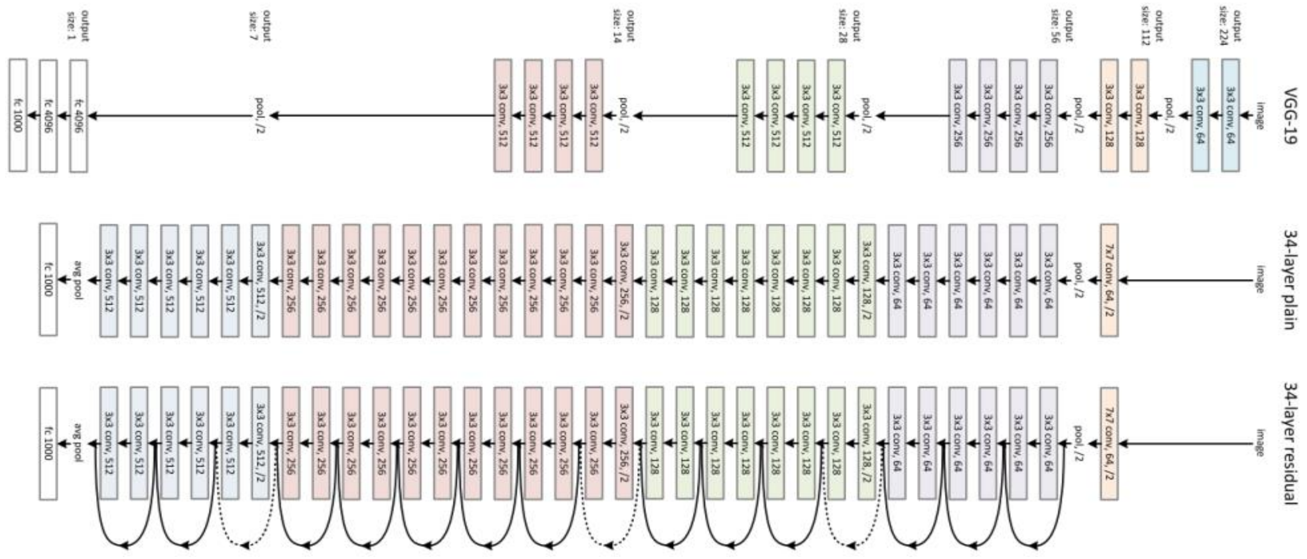


Figure 3: Resnet Architecture [14]

2.12. General System Architecture

Figure 4 shows the general system architecture of the proposed system.

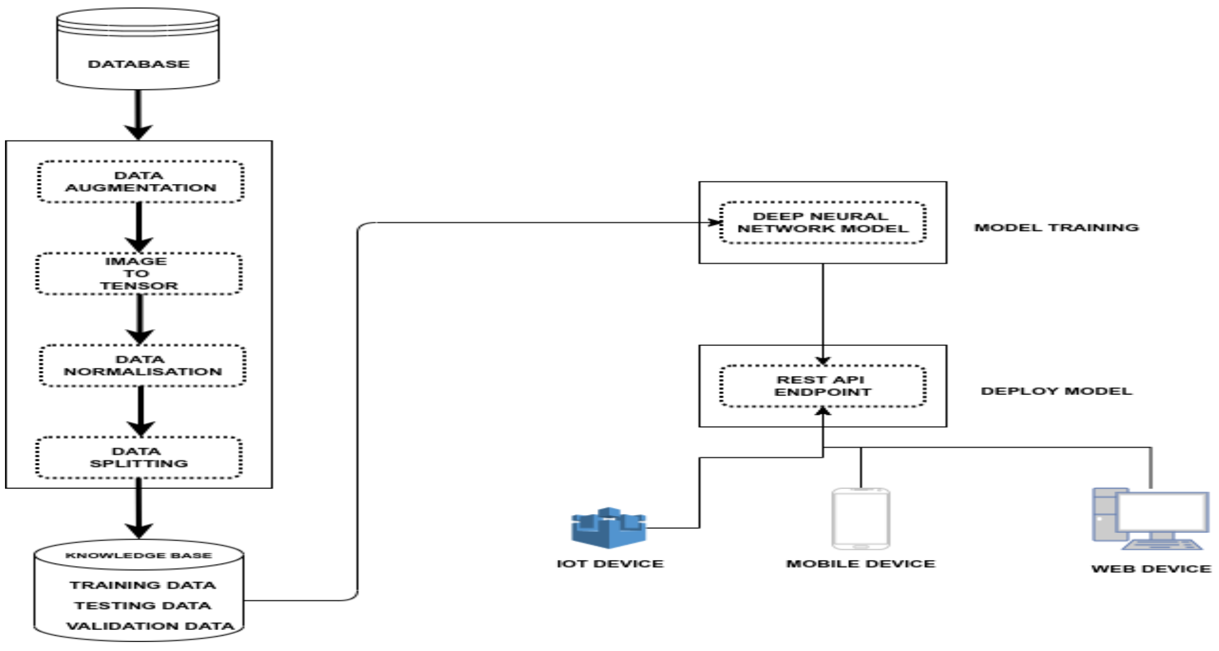


Figure 4: General System Architecture of Proposed System

The system architecture is connected in such a way that data comes into the system from the database into the data processing section where the output is divided into Train, Test and Validation dataset. These datasets then flow into the Deep learning architecture where training and testing takes place. If the results obtained from the deep learning model are satisfactory, the resultant model then flows into the rest API section which other devices can test from.

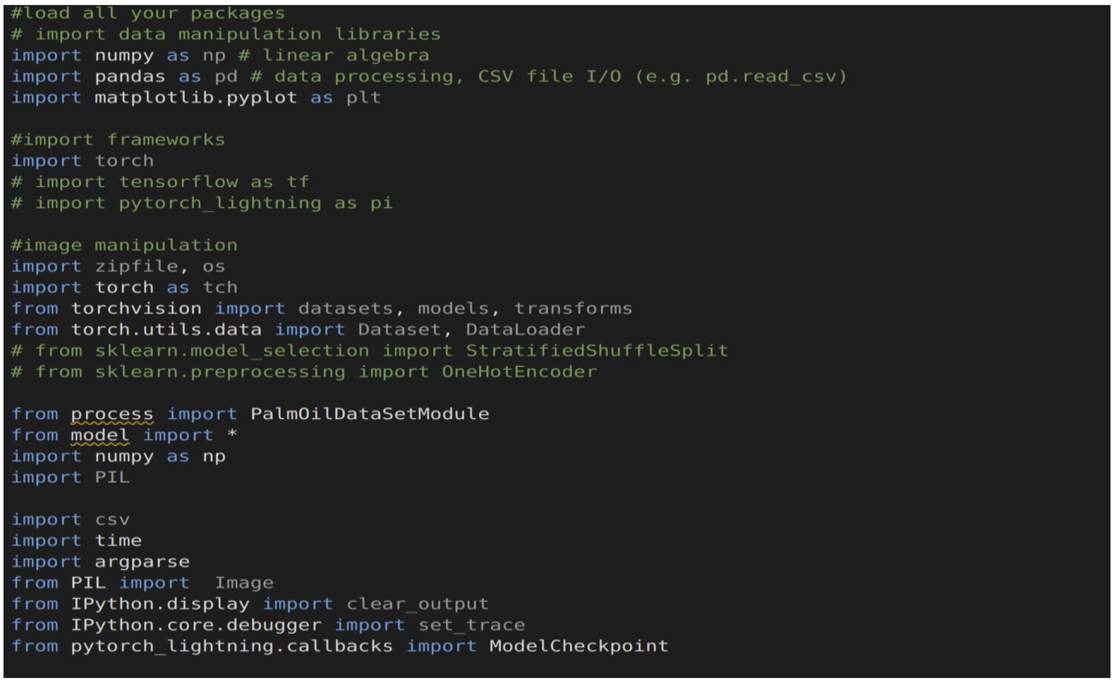
3. Implementation and Testing

3.1. Model Training

Considering the nature of this study, a proper modularization approach was taken in writing the code; at each level we made proper use of the class based approach of programming, creating mini backbones to be fed into the bigger system. The important modules and program files in this research are as follows: (i) Main.py python file (ii) Model.py python file (iii) Process.py python file (iv) Model.ckpt checkpoint

3.1.1 Main.Py (Training)

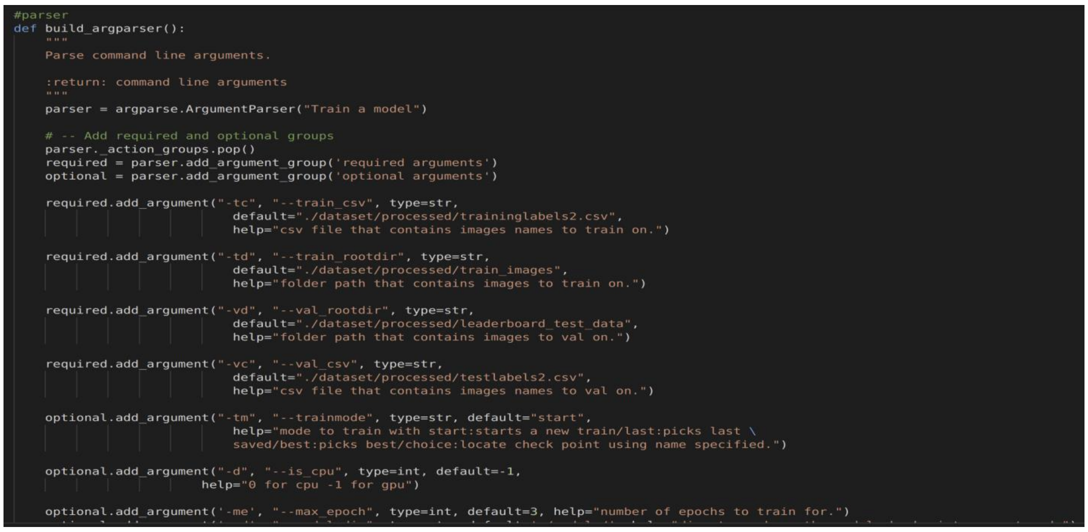
This is a high level file abstraction, where only the needed functions are made available to the user of the system. It basically contains how to train the model using major classes and backbone classes from program modules or files like `model.py` and `process.py`. To use this file a terminal based interaction is made available in vscode (one of the required softwares), where we can run the program to train the model using the command `python3 main.py` (this uses all the default set hyperparameters) or we can use the command `python3 main.py -h` to read the documentation on how to change default parameters. Figure 5 shows the libraries imported in the main.py program file.



```python
#load all your packages
# import data manipulation libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt

#import frameworks
import torch
# import tensorflow as tf
# import pytorch_lightning as pi

#image manipulation
import zipfile, os
import torch as tch
from torchvision import datasets, models, transforms
from torch.utils.data import Dataset, DataLoader
# from sklearn.model_selection import StratifiedShuffleSplit
# from sklearn.preprocessing import OneHotEncoder

from process import PalmOilDataSetModule
from model import *
import numpy as np
import PIL

import csv
import time
import argparse
from PIL import  Image
from IPython.display import clear_output
from IPython.core.debugger import set_trace
from pytorch_lightning.callbacks import ModelCheckpoint
```

Figure 5: Importing required libraries in the main.py program. (Source: screenshot from main.py program file)

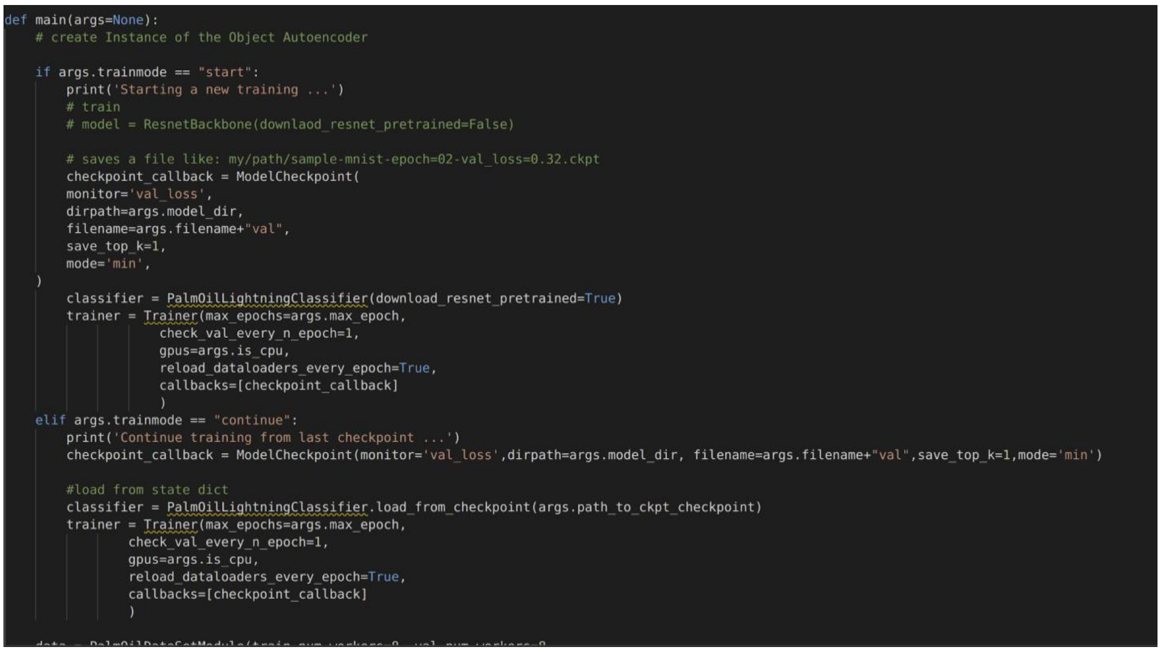
Figure 6 shows how input parameters sent to the main.py program file are handled.



```python
#parser
def build_argparser():
    """
    Parse command line arguments.

    :return: command line arguments
    """
    parser = argparse.ArgumentParser("Train a model")

    # -- Add required and optional groups
    parser._action_groups.pop()
    required = parser.add_argument_group('required arguments')
    optional = parser.add_argument_group('optional arguments')

    required.add_argument("-tc", "--train_csv", type=str,
                            default="./dataset/processed/traininglabels2.csv",
                            help="csv file that contains images names to train on.")

    required.add_argument("-td", "--train_rootdir", type=str,
                            default="./dataset/processed/train_images",
                            help="folder path that contains images to train on.")

    required.add_argument("-vd", "--val_rootdir", type=str,
                            default="./dataset/processed/leaderboard_test_data",
                            help="folder path that contains images to val on.")

    required.add_argument("-vc", "--val_csv", type=str,
                            default="./dataset/processed/testlabels2.csv",
                            help="csv file that contains images names to val on.")

    optional.add_argument("-tm", "--trainmode", type=str, default="start",
                            help="mode to train with start:starts a new train/last:picks last \
                            saved/best:picks best/choice:locate check point using name specified.")

    optional.add_argument("-d", "--is_cpu", type=int, default=-1,
                        help="0 for cpu -1 for gpu")

    optional.add_argument('-me', "--max_epoch", type=int, default=3, help="number of epochs to train for.")
```

Figure 6: Parameters sent for processing in the main.py program. (Source: screenshot from main.py program file)

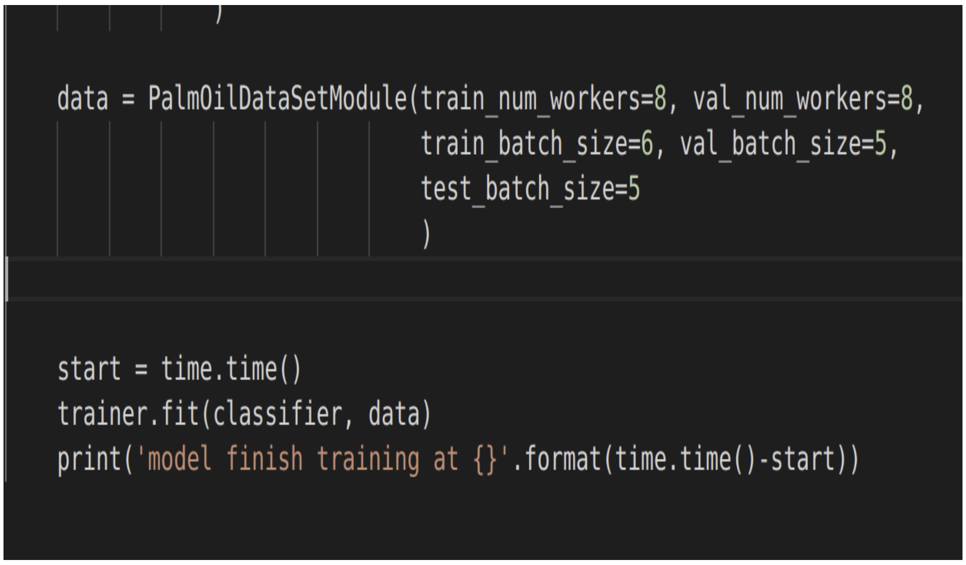
Figure 7 shows abstraction of the `PalmOilLightningClassifier` class used for model training in main.py program.



```python
def main(args=None):
    # create Instance of the Object Autoencoder

    if args.trainmode == "start":
        print('Starting a new training ...')
        # train
        # model = ResnetBackbone(downlaod_resnet_pretrained=False)

        # saves a file like: my/path/sample-mnist-epoch=02-val_loss=0.32.ckpt
        checkpoint_callback = ModelCheckpoint(
        monitor='val_loss',
        dirpath=args.model_dir,
        filename=args.filename+"val",
        save_top_k=1,
        mode='min',
    )
        classifier = PalmOilLightningClassifier(download_resnet_pretrained=True)
        trainer = Trainer(max_epochs=args.max_epoch,
                    check_val_every_n_epoch=1,
                    gpus=args.is_cpu,
                    reload_dataloaders_every_epoch=True,
                    callbacks=[checkpoint_callback]
                    )
    elif args.trainmode == "continue":
        print('Continue training from last checkpoint ...')
        checkpoint_callback = ModelCheckpoint(monitor='val_loss',dirpath=args.model_dir, filename=args.filename+"val",save_top_k=1,mode='min')

        #load from state dict
        classifier = PalmOilLightningClassifier.load_from_checkpoint(args.path_to_ckpt_checkpoint)
        trainer = Trainer(max_epochs=args.max_epoch,
                check_val_every_n_epoch=1,
                gpus=args.is_cpu,
                reload_dataloaders_every_epoch=True,
                callbacks=[checkpoint_callback]
                )
```

Figure 7: Abstraction of the `PalmOilLightningClassifier`class used for model training in main.py program (Source: screenshot from main.py program file).

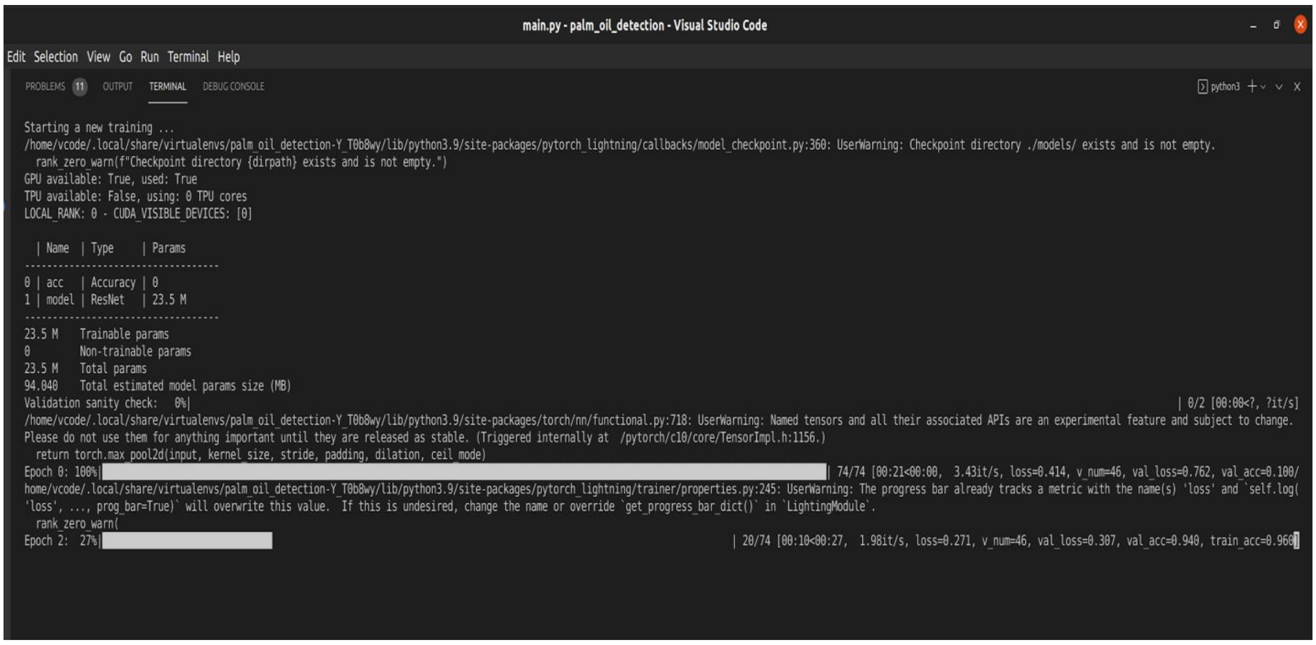
Figure 8 shows abstraction of the `PalmOilDataSetModule` class used for getting dataset during training in main.py program.



```python
                )

    data = PalmOilDataSetModule(train_num_workers=8, val_num_workers=8,
                                train_batch_size=6, val_batch_size=5,
                                test_batch_size=5
                                )


    start = time.time()
    trainer.fit(classifier, data)
    print('model finish training at {}'.format(time.time()-start))
```

Figure 8: Abstraction of the `PalmOilDataSetModule` class used for getting dataset during training in main.py program (Source: screenshot from main.py program file)

Figure 9 shows model training using the main.py program.

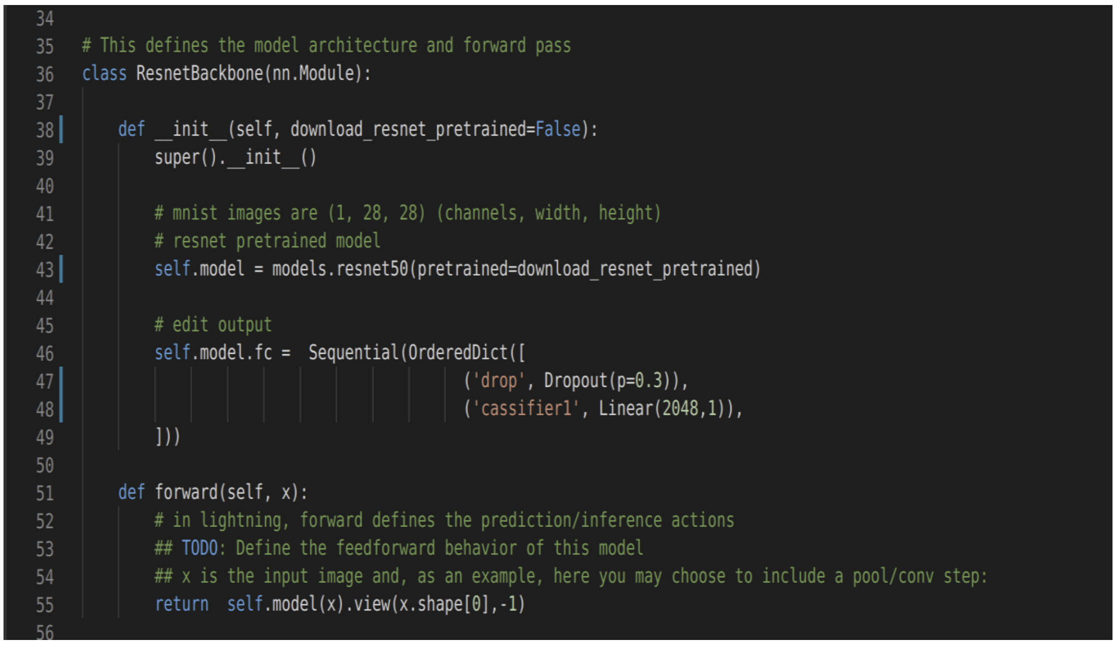

main.py - palm_oil_detection - Visual Studio Code
Edit Selection View Go Run Terminal Help
PROBLEMS 11
OUTPUT
TERMINAL
DEBUG CONSOLE
python3

```
Starting a new training ...
/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/pytorch_lightning/callbacks/model_checkpoint.py:360: UserWarning: Checkpoint directory ./models/ exists and is not empty.
  rank_zero_warn(f"Checkpoint directory {dirpath} exists and is not empty.")
GPU available: True, used: True
TPU available: False, using: 0 TPU cores
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name  | Type     | Params
-----------------------------------
0 | acc   | Accuracy | 0
1 | model | ResNet   | 23.5 M
-----------------------------------
23.5 M    Trainable params
0         Non-trainable params
23.5 M    Total params
94.040    Total estimated model params size (MB)
Validation sanity check:   0%|                                                                                     | 0/2 [00:00<?, ?it/s]
/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental feature and subject to change.
Please do not use them for anything important until they are released as stable. (Triggered internally at  /pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Epoch 0: 100%|██████████████████████████████████████| 74/74 [00:21<00:00,  3.43it/s, loss=0.414, v_num=46, val_loss=0.762, val_acc=0.100/
home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/pytorch_lightning/trainer/properties.py:245: UserWarning: The progress bar already tracks a metric with the name(s) 'loss' and `self.log(
'loss', ..., prog_bar=True)` will overwrite this value.  If this is undesired, change the name or override `get_progress_bar_dict()` in `LightingModule`.
  rank_zero_warn(
Epoch 2:  27%|██████████                            | 20/74 [00:10<00:27,  1.98it/s, loss=0.271, v_num=46, val_loss=0.307, val_acc=0.940, train_acc=0.960
```

Figure 9: Model training using main.py program (Source: screenshot from main.py program file)

3.1.2 Model.Py

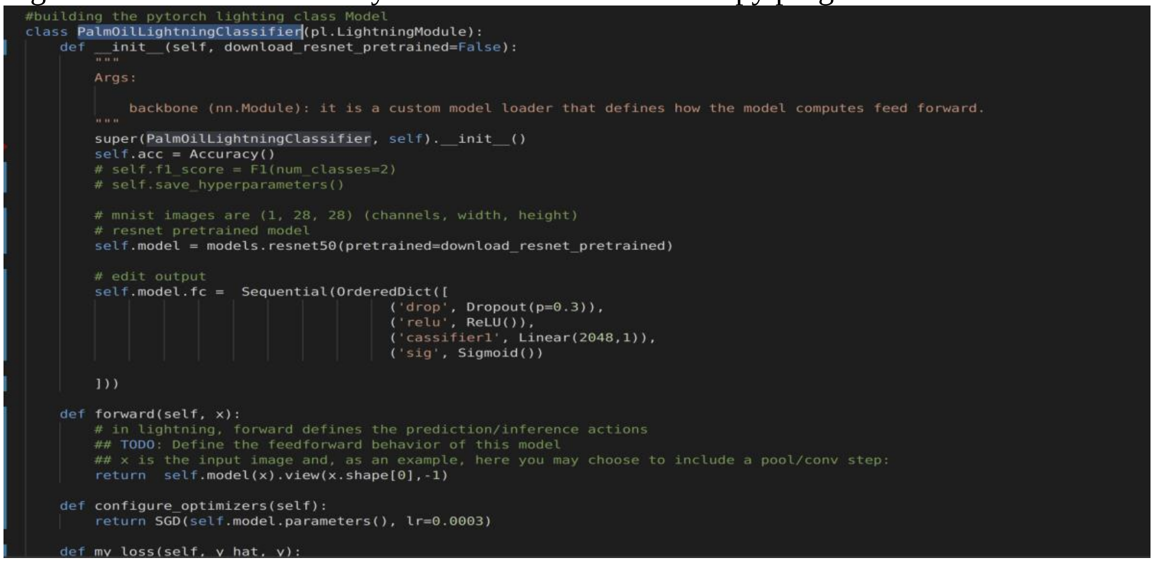
The architecture used to train the model was built on top of the famous Resnet architecture and every other preprocessing done to fine-tune the structure of the Resnet architecture sits in this program file. In this program file, all the rules and steps involved in the model training were defined and includes: splitting functionalities by abstracting out the model's structure into a backbone class called `ResnetBackbone` that contains the main pytorch based model architecture, a model system architecture called `PalmOilLightningClassifier` that takes the model backbone as a parameter in the training process, what the model should do while training, testing or validating the backbone.Figure 10 shows model backbone in model.py program.



```python
# This defines the model architecture and forward pass
class ResnetBackbone(nn.Module):

    def __init__(self, download_resnet_pretrained=False):
        super().__init__()

        # mnist images are (1, 28, 28) (channels, width, height)
        # resnet pretrained model
        self.model = models.resnet50(pretrained=download_resnet_pretrained)

        # edit output
        self.model.fc =  Sequential(OrderedDict([
                                  ('drop', Dropout(p=0.3)),
                                  ('cassifier1', Linear(2048,1)),
        ]))

    def forward(self, x):
        # in lightning, forward defines the prediction/inference actions
        ## TODO: Define the feedforward behavior of this model
        ## x is the input image and, as an example, here you may choose to include a pool/conv step:
        return  self.model(x).view(x.shape[0],-1)
```

Figure 10: model backbone in model.py program (Source: screenshot from model.py program file)

Figure 11 shows the model system architecture in model.py program.

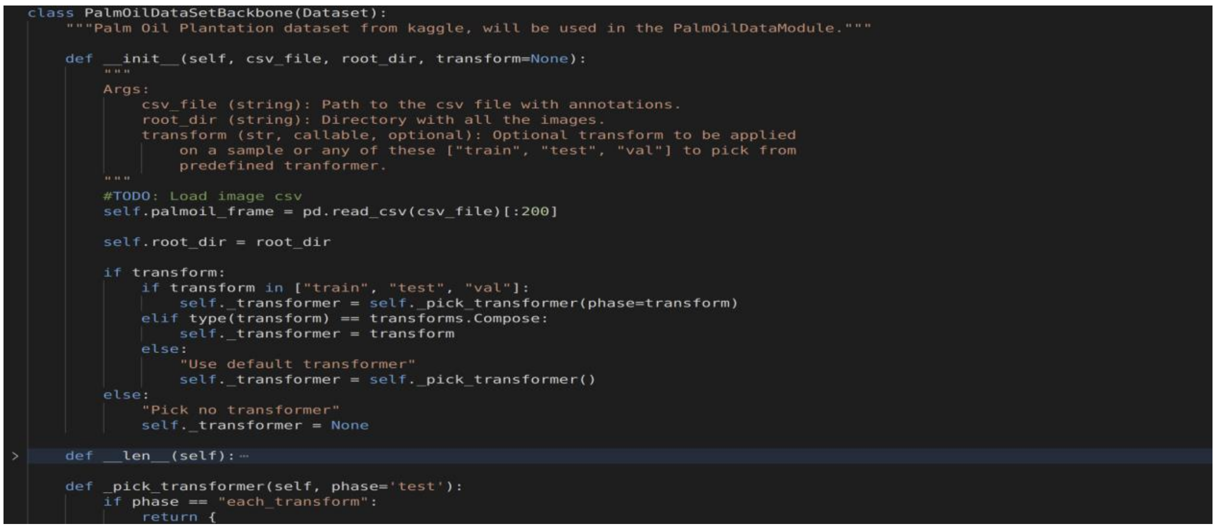


```python
#building the pytorch lighting class Model
class PalmOilLightningClassifier(pl.LightningModule):
    def __init__(self, download_resnet_pretrained=False):
        """
        Args:

            backbone (nn.Module): it is a custom model loader that defines how the model computes feed forward.
        """
        super(PalmOilLightningClassifier, self).__init__()
        self.acc = Accuracy()
        # self.f1_score = F1(num_classes=2)
        # self.save_hyperparameters()

        # mnist images are (1, 28, 28) (channels, width, height)
        # resnet pretrained model
        self.model = models.resnet50(pretrained=download_resnet_pretrained)

        # edit output
        self.model.fc =  Sequential(OrderedDict([
                                        ('drop', Dropout(p=0.3)),
                                        ('relu', ReLU()),
                                        ('cassifier1', Linear(2048,1)),
                                        ('sig', Sigmoid())

        ]))

    def forward(self, x):
        # in lightning, forward defines the prediction/inference actions
        ## TODO: Define the feedforward behavior of this model
        ## x is the input image and, as an example, here you may choose to include a pool/conv step:
        return  self.model(x).view(x.shape[0],-1)

    def configure_optimizers(self):
        return SGD(self.model.parameters(), lr=0.0003)

    def my_loss(self, y_hat, y):
```

Figure 11: model system architecture in model.py program (Source: screenshot from model.py program file)

3.1.3 Process.Py

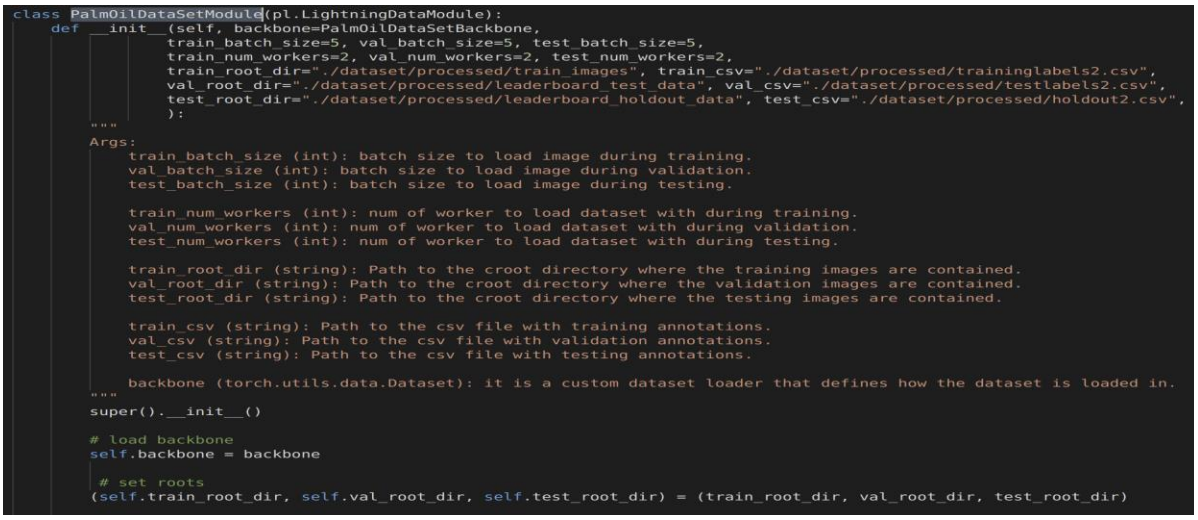
This study used a great deal of programming modularization by having the data section entirely detached from the main program. The data section was designed in such a way that data can be loaded as a group or individually and also to account for training data, testing data and validation data that are used by the model. It was also implemented in such a way that data can be displayed individually and randomly taking into account that preprocessing results should be visualized. All this functionalities packed so that we have two separate python classes to manage the whole process viz: the `Oil PalmDataSetBackbone` that defines the basic structure of the dataset such as how to get the data, how to display data with preprocessing, checking for missing data, saving data and handling missing data. To complement this class, we also have one other python class `OilPalmDataSetModule` class which takes care of every other function that deals with downloading the dataset, splitting into, train, test, validate and load up for the model system to use. Figure 12 shows the OilPalm Dataset backbone in the process.py program.



```python
class PalmOilDataSetBackbone(Dataset):
    """Palm Oil Plantation dataset from kaggle, will be used in the PalmOilDataModule."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string): Path to the csv file with annotations.
            root_dir (string): Directory with all the images.
            transform (str, callable, optional): Optional transform to be applied
                on a sample or any of these ["train", "test", "val"] to pick from
                predefined tranformer.
        """
        #TODO: Load image csv
        self.palmoil_frame = pd.read_csv(csv_file)[:200]

        self.root_dir = root_dir

        if transform:
            if transform in ["train", "test", "val"]:
                self._transformer = self._pick_transformer(phase=transform)
            elif type(transform) == transforms.Compose:
                self._transformer = transform
            else:
                "Use default transformer"
                self._transformer = self._pick_transformer()
        else:
            "Pick no transformer"
            self._transformer = None

>   def __len__(self):⋯

    def _pick_transformer(self, phase='test'):
        if phase == "each_transform":
            return {
```

Figure 12: Oil Palm Dataset backbone in process.py program (Source: screenshot from process.py program file)

Figure 13 shows the OilPalm Dataset system structure in the process.py program.



```python
class PalmOilDataSetModule(pl.LightningDataModule):
    def __init__(self, backbone=PalmOilDataSetBackbone,
                train_batch_size=5, val_batch_size=5, test_batch_size=5,
                train_num_workers=2, val_num_workers=2, test_num_workers=2,
                train_root_dir="./dataset/processed/train_images", train_csv="./dataset/processed/traininglabels2.csv",
                val_root_dir="./dataset/processed/leaderboard_test_data", val_csv="./dataset/processed/testlabels2.csv",
                test_root_dir="./dataset/processed/leaderboard_holdout_data", test_csv="./dataset/processed/holdout2.csv",
                ):
        """
        Args:
            train_batch_size (int): batch size to load image during training.
            val_batch_size (int): batch size to load image during validation.
            test_batch_size (int): batch size to load image during testing.

            train_num_workers (int): num of worker to load dataset with during training.
            val_num_workers (int): num of worker to load dataset with during validation.
            test_num_workers (int): num of worker to load dataset with during testing.

            train_root_dir (string): Path to the croot directory where the training images are contained.
            val_root_dir (string): Path to the croot directory where the validation images are contained.
            test_root_dir (string): Path to the croot directory where the testing images are contained.

            train_csv (string): Path to the csv file with training annotations.
            val_csv (string): Path to the csv file with validation annotations.
            test_csv (string): Path to the csv file with testing annotations.

            backbone (torch.utils.data.Dataset): it is a custom dataset loader that defines how the dataset is loaded in.
        """
        super().__init__()

        # load backbone
        self.backbone = backbone

        # set roots
        (self.train_root_dir, self.val_root_dir, self.test_root_dir) = (train_root_dir, val_root_dir, test_root_dir)
```

Figure 13: OilPalm Dataset system structure in process.py program (Source: screenshot from process.py program file)

3.1.4 Model.Ckpt

For Inference, the weight and parameters of a trained model are saved for later use. In this study, the weight and parameters of the model was saved if the results from the current training step is better than the result from the previous steps. Figure 14 shows the saved models after training.

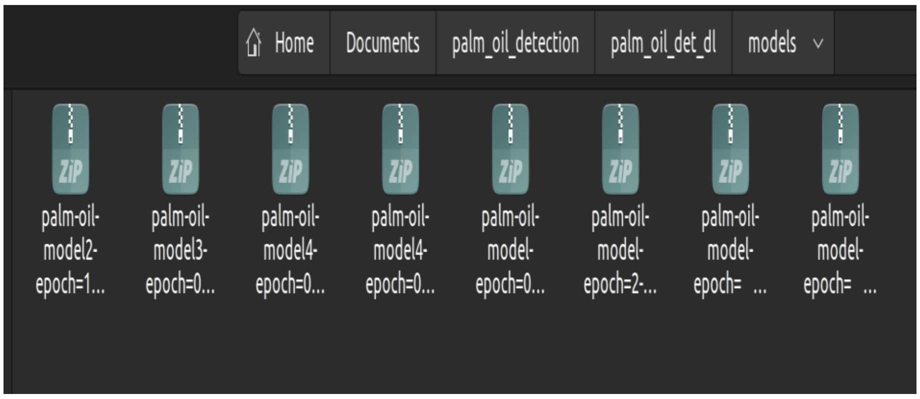

Home
Documents
palm_oil_detection
palm_oil_det_dl
models
palm-oil-model2-epoch=1...
palm-oil-model3-epoch=0...
palm-oil-model4-epoch=0...
palm-oil-model4-epoch=0...
palm-oil-model-epoch=0...
palm-oil-model-epoch=2-...
palm-oil-model-epoch= ...
palm-oil-model-epoch= ...

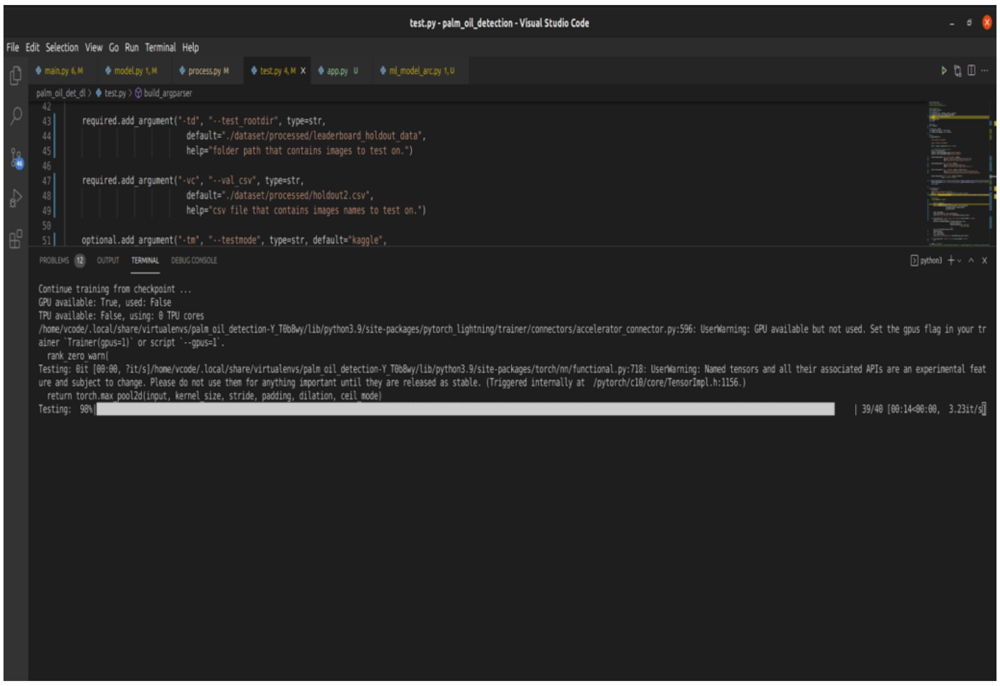
Figure 14: Saved models (Source: screenshot from models folder in the program)

3.2 Model Testing

Model testing is a process that involves checking if the model is not over-fitting. This process tests for the general model performance in three (3) phases viz: (i) testing done during the training phase: data used here is a separate dataset from the training data. (ii) testing done during the testing phase, also called the validation phase: data used here is also different from the data used to train and data used to test during the training phase. (iii) the last phase is the API stage where we convert the model into an API and other users can have access to it.

3.2.1 Test.Py

This is the program file used to test the model's performance after which training is completed. To test this file a terminal based interaction is made available in vscode (one of the required softwares) where we can run the program to train the model using the command `python3 test.py` (This uses all the default set hyperparameters), or we can use the command `python3 test.py -h` to read the documentation on how to change default parameters. Figure 15 shows the model during testing via terminal.

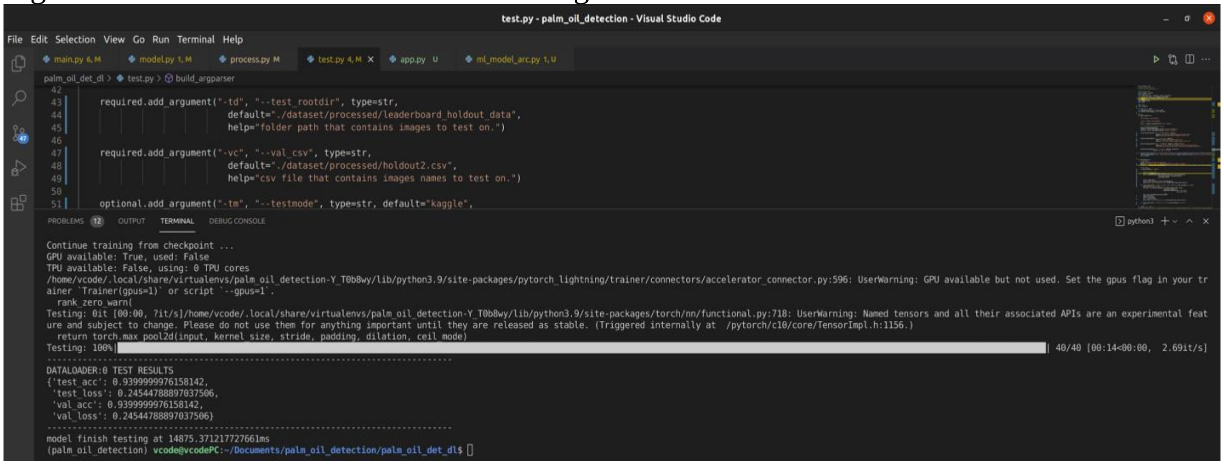

test.py - palm_oil_detection - Visual Studio Code
File Edit Selection View Go Run Terminal Help
main.py 6, M
model.py 1, M
process.py M
test.py 4, M
app.py U
ml_model_arc.py 1, U
palm_oil_det_dl › test.py › build_argparser
PROBLEMS 12
OUTPUT
TERMINAL
DEBUG CONSOLE
python3

```
    required.add_argument("-td", "--test_rootdir", type=str,
                          default="./dataset/processed/leaderboard_holdout_data",
                          help="folder path that contains images to test on.")

    required.add_argument("-vc", "--val_csv", type=str,
                          default="./dataset/processed/holdout2.csv",
                          help="csv file that contains images names to test on.")

    optional.add_argument("-tm", "--testmode", type=str, default="kaggle",
```

```
Continue training from checkpoint ...
GPU available: True, used: False
TPU available: False, using: 0 TPU cores
/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/pytorch_lightning/trainer/connectors/accelerator_connector.py:596: UserWarning: GPU available but not used. Set the gpus flag in your tr
ainer `Trainer(gpus=1)` or script `--gpus=1`.
  rank_zero_warn(
Testing: 0it [00:00, ?it/s]/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental feat
ure and subject to change. Please do not use them for anything important until they are released as stable. (Triggered internally at  /pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Testing:  98%|                                                            | 39/40 [00:14<00:00,  3.23it/s]
```

Figure 15: Model during testing via terminal. (Source: screenshot from test.py program file)

Figure 16 shows the model result after testing via terminal.


test.py - palm_oil_detection - Visual Studio Code
File Edit Selection View Go Run Terminal Help
main.py 6, M
model.py 1, M
process.py M
test.py 4, M
app.py U
ml_model_arc.py 1, U
palm_oil_det_dl › test.py › build_argparser
PROBLEMS 12
OUTPUT
TERMINAL
DEBUG CONSOLE
python3

```
    required.add_argument("-td", "--test_rootdir", type=str,
                          default="./dataset/processed/leaderboard_holdout_data",
                          help="folder path that contains images to test on.")

    required.add_argument("-vc", "--val_csv", type=str,
                          default="./dataset/processed/holdout2.csv",
                          help="csv file that contains images names to test on.")

    optional.add_argument("-tm", "--testmode", type=str, default="kaggle",
```

```
Continue training from checkpoint ...
GPU available: True, used: False
TPU available: False, using: 0 TPU cores
/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/pytorch_lightning/trainer/connectors/accelerator_connector.py:596: UserWarning: GPU available but not used. Set the gpus flag in your tr
ainer `Trainer(gpus=1)` or script `--gpus=1`.
  rank_zero_warn(
Testing: 0it [00:00, ?it/s]/home/vcode/.local/share/virtualenvs/palm_oil_detection-Y_T0b8wy/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their associated APIs are an experimental feat
ure and subject to change. Please do not use them for anything important until they are released as stable. (Triggered internally at  /pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Testing: 100%|                                                            | 40/40 [00:14<00:00,  2.69it/s]
--------------------------------------------------------------------------------
DATALOADER:0 TEST RESULTS
{'test_acc': 0.9399999976158142,
 'test_loss': 0.24544788897037506,
 'val_acc': 0.9399999976158142,
 'val_loss': 0.24544788897037506}
--------------------------------------------------------------------------------
model finish testing at 14875.371217727661ms
(palm_oil_detection) vcode@vcodePC:~/Documents/palm_oil_detection/palm_oil_det_dl$ 
```

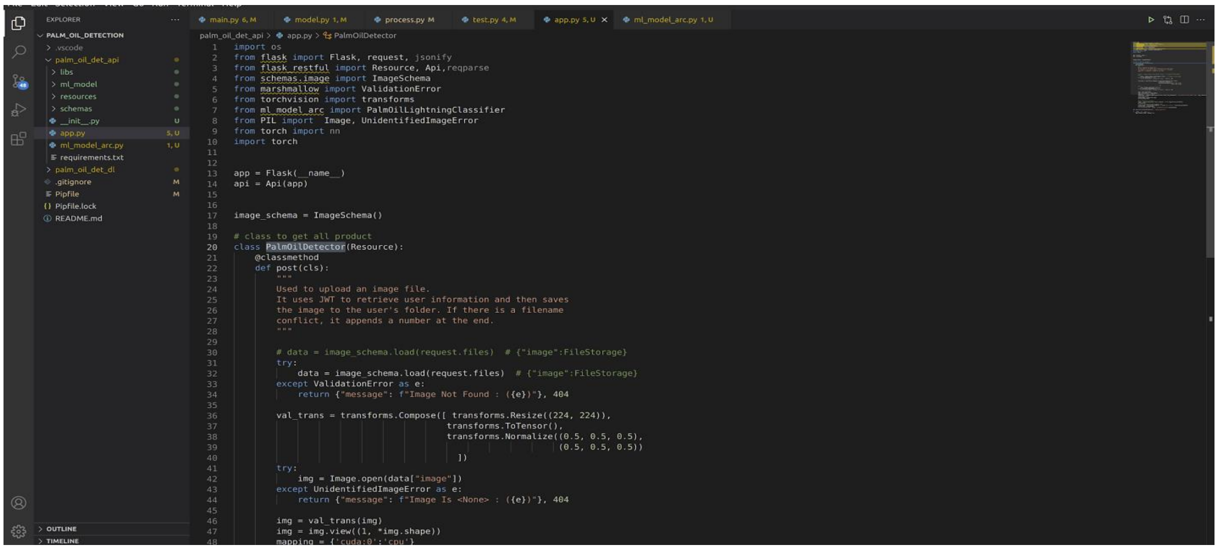
Figure 16: Model result after testing via terminal. (Source: screenshot from test.py program file)
The results of the test is depicted in Table 2.

Table 2. Test results

Dataset	Loss	Accuracy (%)	Prediction Threshold (%)
Train data	0.11	99.0	15
Test data	0.245	82.9999	15
Validation data	1.59	72.5000	15
Validation data	1.59	87.5	10

3.2.2 App.Py (Api)

To create the API that will enable other users connect to the model with, a flask framework was utilized. To develop this API application, we have the `OiPalmlDetector` class that handles the post request from the user to predict the presence of oil palm plantation in an image. We also have the ml_model folder where the model that has gone through the validation phase is placed for the API use. To run the API we only needed to run `python app.py` in the API folder. Figure 17 shows the App.py program file.



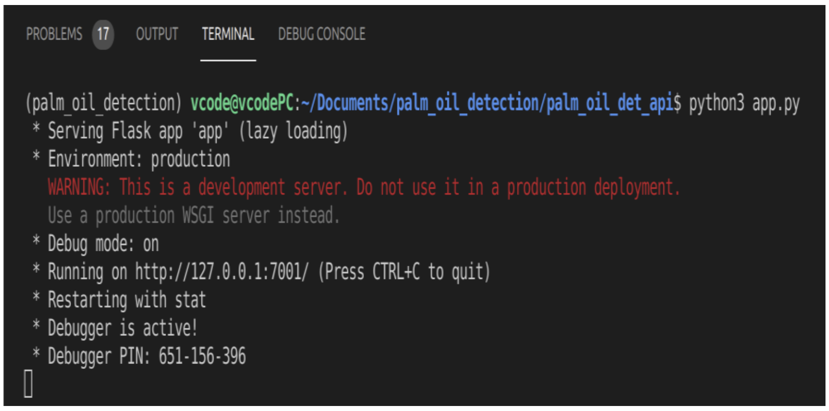
```python
import os
from flask import Flask, request, jsonify
from flask_restful import Resource, Api,reqparse
from schemas.image import ImageSchema
from marshmallow import ValidationError
from torchvision import transforms
from ml_model_arc import PalmOilLightningClassifier
from PIL import  Image, UnidentifiedImageError
from torch import nn
import torch


app = Flask(__name__)
api = Api(app)


image_schema = ImageSchema()

# class to get all product
class PalmOilDetector(Resource):
    @classmethod
    def post(cls):
        """
        Used to upload an image file.
        It uses JWT to retrieve user information and then saves
        the image to the user's folder. If there is a filename
        conflict, it appends a number at the end.
        """

        # data = image_schema.load(request.files)  # {"image":FileStorage}
        try:
            data = image_schema.load(request.files)  # {"image":FileStorage}
        except ValidationError as e:
            return {"message": f"Image Not Found : ({e})"}, 404

        val_trans = transforms.Compose([ transforms.Resize((224, 224)),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5),
                                                              (0.5, 0.5, 0.5))
                                          ])
        try:
            img = Image.open(data["image"])
        except UnidentifiedImageError as e:
            return {"message": f"Image Is <None> : ({e})"}, 404

        img = val_trans(img)
        img = img.view((1, *img.shape))
        mapping = {'cuda:0':'cpu'}
```

Figure 17 App.py program file. (Source: screenshot from app.py program file in the API section)

Figure 18 shows the folder containing Tested models.

Figure 18: Folder containing Tested models (Source: screenshot from ml_model folder in the API section).

Figure 19 shows the process of deploying the API on a local machine.



```
(palm_oil_detection) vcode@vcodePC:~/Documents/palm_oil_detection/palm_oil_det_api$ python3 app.py
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:7001/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 651-156-396
```

Figure 19: Deploying API on a local machine. (Source: screenshot from app.py program file in the API section)

3.2.3 Postman Form

To test the model, postman was used. Testing was done by running a post request in the postman application while sending the required image parameter along with the request. Figure 20 shows the one of the Images sent to API via postman.

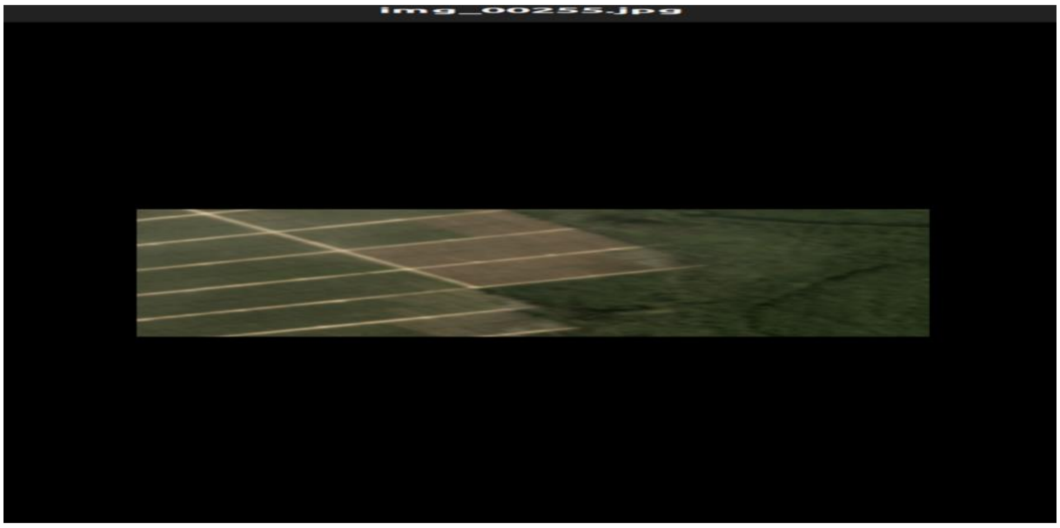
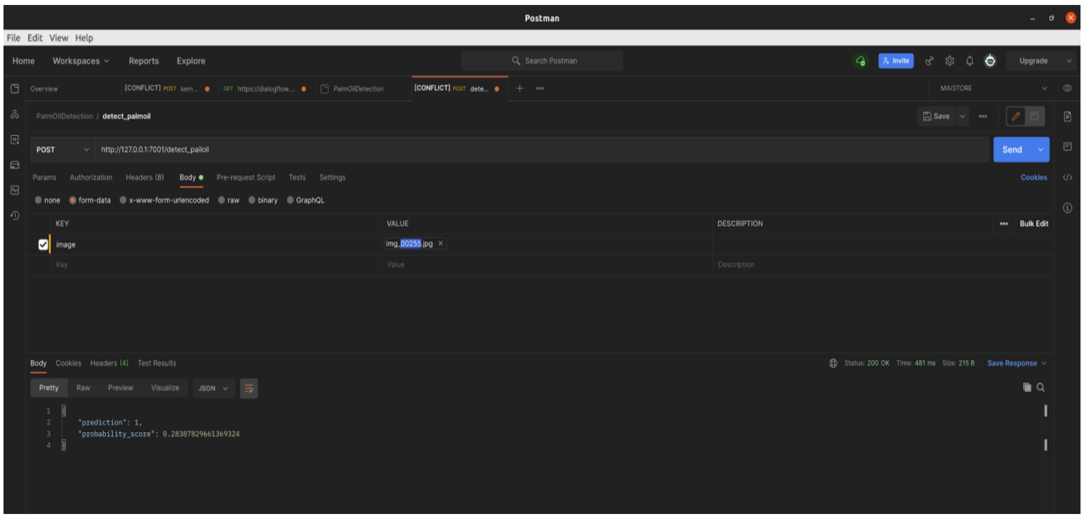


Figure 20: Image sent to API via postman. (Source: screenshot from dataset folder)

Figure 21 shows the prediction result of 1 and prediction confidence of 28.38% obtained from the model during testing via postman.


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Invite Upgrade
Overview [CONFLICT] POST kem... GET https://dialogflow.... PalmOilDetection [CONFLICT] POST dete...
MAISTORE
PalmOilDetection / detect_palmoil
Save
Send
Params Authorization Headers (8) Body Pre-request Script Tests Settings
Cookies
none form-data x-www-form-urlencoded raw binary GraphQL
Bulk Edit
Body Cookies Headers (4) Test Results
Save Response
Pretty Raw Preview Visualize JSON

POST http://127.0.0.1:7001/detect_pailoil

KEY	VALUE	DESCRIPTION
✓ image	img_00255.jpg	
Key	Value	Description

Status: 200 OK Time: 481 ms Size: 215 B

```
{
    "prediction": 1,
    "probability_score": 0.28387829661369324
}
```

Figure 21: Prediction result of 1 and prediction confidence of 28.38% obtained from the model (Source: screenshot from postman running on the local machine)

4. Conclusion

This research used a deep learning model to develop a system that can accurately predict the presence of oil palm plantations using satellite images. In the different data gathering process of the study, despite the large collection of data gotten, there was still a huge difference between the prediction score from the model and the prediction score from the data label. We found that the model could predict the correct label, but most times with low prediction score. To address that problem, the prediction threshold used in assigning classes in the system was reduced to 15%. This enabled us to correctly predict the class we are interested in without any major loss in accuracy; in fact a major increase in accuracy was experienced when threshold was dropped to 10%. It is important to remark that almost all deep learning development is a multiple iteration process that is aimed at fine tuning the model with variation in dataset, having each iteration experience new data from time to time, and thus resulting in the accomplishment of a more accurate and robust system. The study is a prototype of what could be a working system if adopted in Nigeria. However, since datasets for research in the oil palm area in Nigeria were not readily available at the time of carrying out this study, this research used dataset obtained from Kaggle, an online platform that allows users to find and publish data sets, explore and build models in a web-based data-science environment. It is hoped that further studies can be localized to solve the specific problems faced in Nigeria.

References

[1] I. Bonet, F. Caraffini, A. Pena, A. Puert, and M. Gongora (2020). "Oil palm detection via deep transfer learning." IEEE Congress on Evolutionary Computation (CEC), pp. 1-8

[2] Y. Cheng, L. Yu, Y. Xu, H. Lu, A.P. Cracknell, K. Kanniah and P. Gong (2018). "Mapping oil palm extent in Malaysia using ALOS-2 PALSAR-2 data". Int. J. Remote Sens. 39, 432–452.

[3] P. K. Lian and D. S. Wilcove (2007)." Cashing in palm oil for conservation". Nature 448, 993–4.

[4] G. U. Ojo, R. A. Offiong, S. Odion-Akhaine, A. Baiyewu-Teru, and F. Allen (2017) ." Oil palm plantations in forest landscapes: impacts, aspirations and ways forward in Nigeria". Wageningen, the Netherlands: Tropenbos International. 2017.

[5] W. Li, H. Fu, L.Yu and A. Cracknell (2016). "A deep learning based oil palm tree detection and counting for high resolution remote sensing images". Remote Sens. 9, 22.

[6] W. Li, R. Dong, H. Fu and L.Yu (2019). "Large-scale oil palm tree detection from high-resolution satellite images using two-stage convolutional neural networks". Remote Sensing, 2019, 11(1): 11.

[7] L. Xinni, H. G. Kamarul, H. Fengrong and I. M. Izzeldin (2021). "Automatic Detection of Oil Palm Tree from UAV Images Based on the Deep Learning Method". Applied Artificial Intelligence, 35(1), 13-24.

[8] N. A. Mubin, E. Nadarajoo, H. Z. M Shafri and A. Hamedianfar (2019). "Young and mature oil palm tree detection and counting using convolutional neural network deep learning method". International journal of remote sensing, 40(19),7500-7515

[9] T. Blaschke(2010) "Object based image analysis for remote sensing". ISPRS journal of photogrammetry and remote sensing, 65(1), 2-16.

[10] S. Chen, E. Dobriban and J. H. Lee (2020). "A group-theoretic framework for data augmentation". Journal of Machine Learning Research, 21(245), 1-71

[11] X. Yang, Z. Zeng, S. G. Teo L. Wang, V. Chandrasekhar and S. Hoi (2018). "Deep learning for practical image recognition: Case study on kaggle competitions". Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining, pp 923-931

[12] K. He, X. Zhang, S. Ren and J. Sun (2016). " Deep Residual Learning for Image Recognition". In Proceedings of the 2016 IEEE conference on computer vision and pattern recognition, pp 770-778, doi:10.1109/CVPR.2016.90

[13] M. Hamilton, S. Raghunathan, A. Annavajhala, D. Kirsanov, E. De Leon, E. Barzilay, I. Matiach, J. Davison, M. Busch, M. Oprescu, R. Sur, R. Astala, T. Wen and C. Park (2017). "Flexible and Scalable Deep Learning with MMLSpark" . Journal of Machine Learning Research, 1-48. Retrieved September 12, 2022 from: https://www.researchgate.net/figure/Schematic-overview-of-basic-transfer-learning-algorithm-We-use-ResNet50-and-truncate-the_fig3_324472161

[14] V. Feng (2017). "An Overview of ResNet and its Variants. Towards Data Science". Retrieved September 12, 2022 from https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035