Group 2:

Manish Kakarla

Pradeep Gummidipundi

Atiya Kailany

Henry Lao

P3 Documentation:

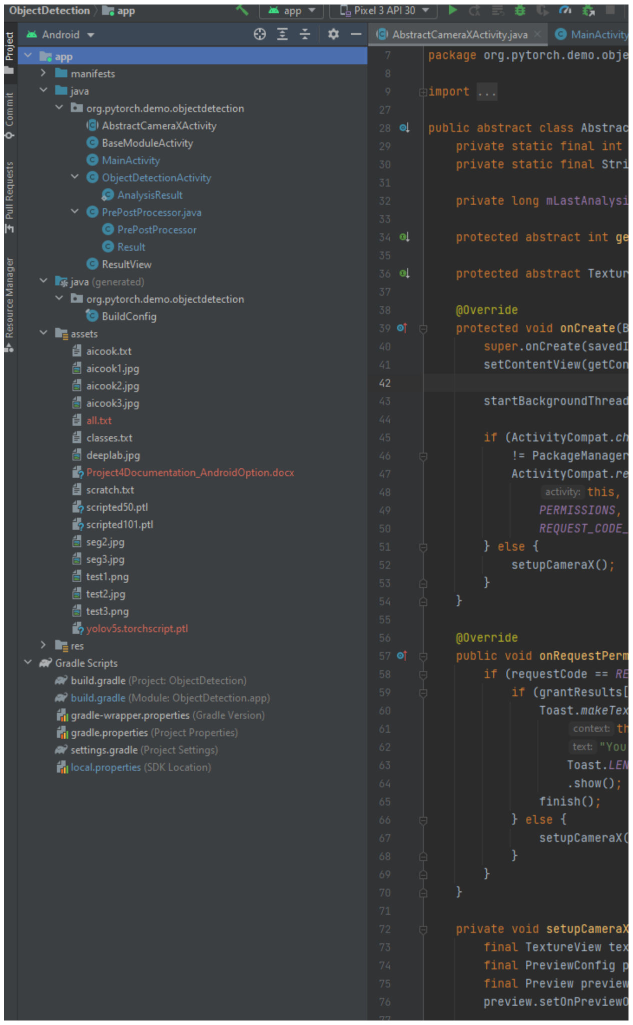
This project was designed to adapt the work of Liu et al. and convert it to a mobile device which would allow for in-the-field processing of images and much faster response time and lower network requirements versus a computing cluster that would typically be utilized for these sorts of tasks. While working on this adaptation, we utilized two separate methods to ensure flexibility of classification: a local method powered entirely by the Android device for those phones with the computing capacity to spare, and a REST-based method designed to take advantage of existing networks and send the image back to a computer for offsite processing, storage, and evaluation.

The following paper describes implementation, downloading and running instructions, screenshots, and finally comments/critiques.

Section 1:

Download (Frontend):

The app can be run like any other Android app. Firstly, the source code can be unzipped and the folder opened in Android Studio. After opening the app in Android Studio, select a suitable emulator or Android device connected to your local machine, and hit Run.



```
app
  manifests
  java
    org.pytorch.demo.objectdetection
      AbstractCameraXActivity
      BaseModuleActivity
      MainActivity
      ObjectDetectionActivity
        AnalysisResult
      PrePostProcessor.java
        PrePostProcessor
        Result
      ResultView
  java (generated)
    org.pytorch.demo.objectdetection
      BuildConfig
  assets
    aicook.txt
    aicook1.jpg
    aicook2.jpg
    aicook3.jpg
    all.txt
    classes.txt
    deeplab.jpg
    Project4Documentation_AndroidOption.docx
    scratch.txt
    scripted50.ptl
    scripted101.ptl
    seg2.jpg
    seg3.jpg
    test1.png
    test2.jpg
    test3.png
    yolov5s.torchscript.ptl
  res
Gradle Scripts
  build.gradle (Project: ObjectDetection)
  build.gradle (Module: ObjectDetection.app)
  gradle-wrapper.properties (Gradle Version)
  gradle.properties (Project Properties)
  settings.gradle (Project Settings)
  local.properties (SDK Location)
```

```java
package org.pytorch.demo.obje

import ...

public abstract class Abstrac
    private static final int
    private static final Stri

    private long mLastAnalysi

    protected abstract int ge

    protected abstract Textur

    @Override
    protected void onCreate(B
        super.onCreate(savedI
        setContentView(getCon

        startBackgroundThread

        if (ActivityCompat.ch
                != PackageManager
            ActivityCompat.re
                    activity: this,
                    PERMISSIONS,
                    REQUEST_CODE_
        } else {
            setupCameraX();
        }
    }

    @Override
    public void onRequestPerm
        if (requestCode == RE
            if (grantResults[
                Toast.makeTex
                        context: th
                        text: "You
                        Toast.LEN
                        .show();
                finish();
            } else {
                setupCameraX(
            }
        }
    }

    private void setupCameraX
        final TextureView tex
        final PreviewConfig p
        final Preview preview
        preview.setOnPreviewO
```

Figure 1: Android App file directory

The app can be ran in either an emulator or live device, but for easy of imagery, an emulated device is shown. The app must be allowed access to files/images on the device. To detect images, please use the "Detect" button to run the model.

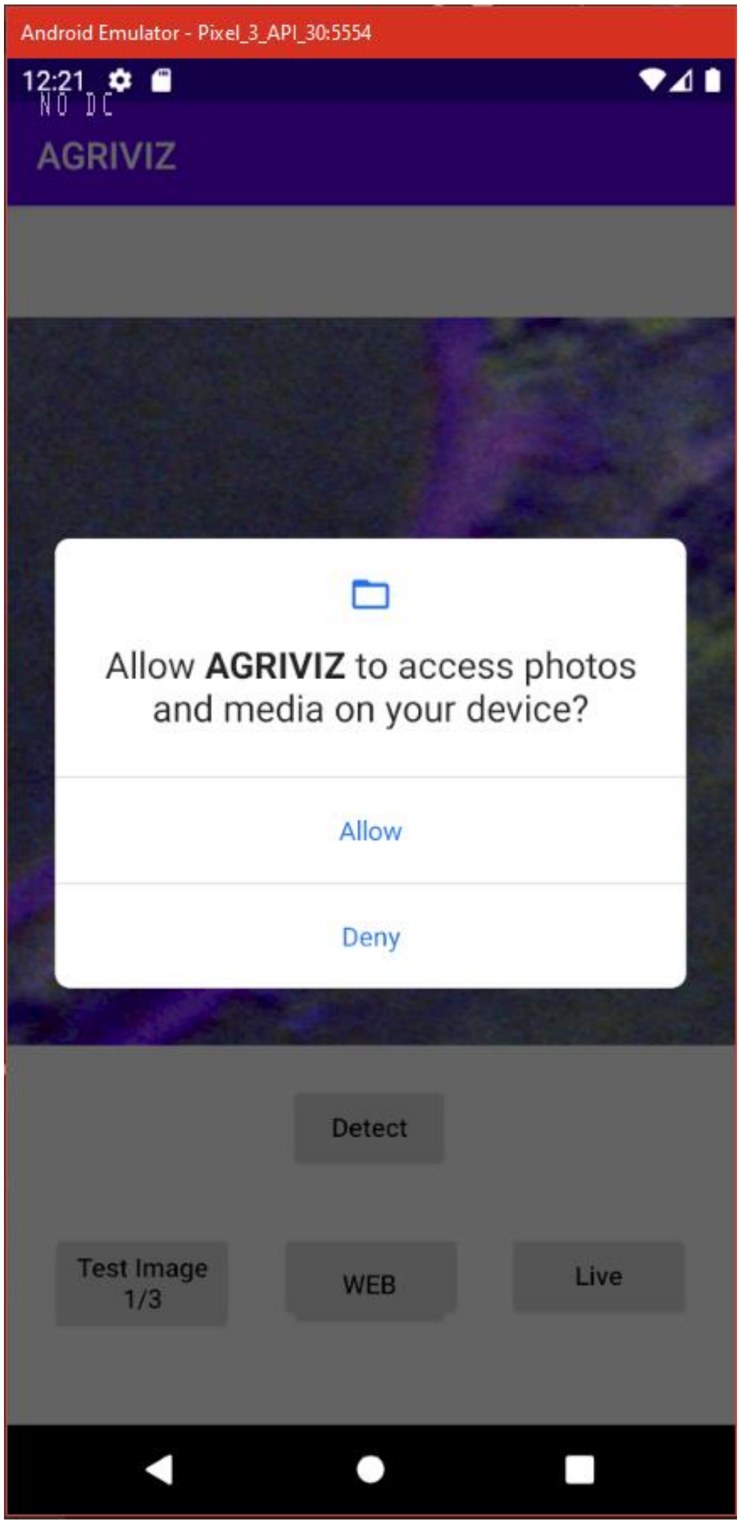

Android Emulator - Pixel_3_API_30:5554
12:21
AGRIVIZ
Allow **AGRIVIZ** to access photos and media on your device?
Allow
Deny
Detect
Test Image 1/3
WEB
Live

Figure 2: Startup screen, permissions

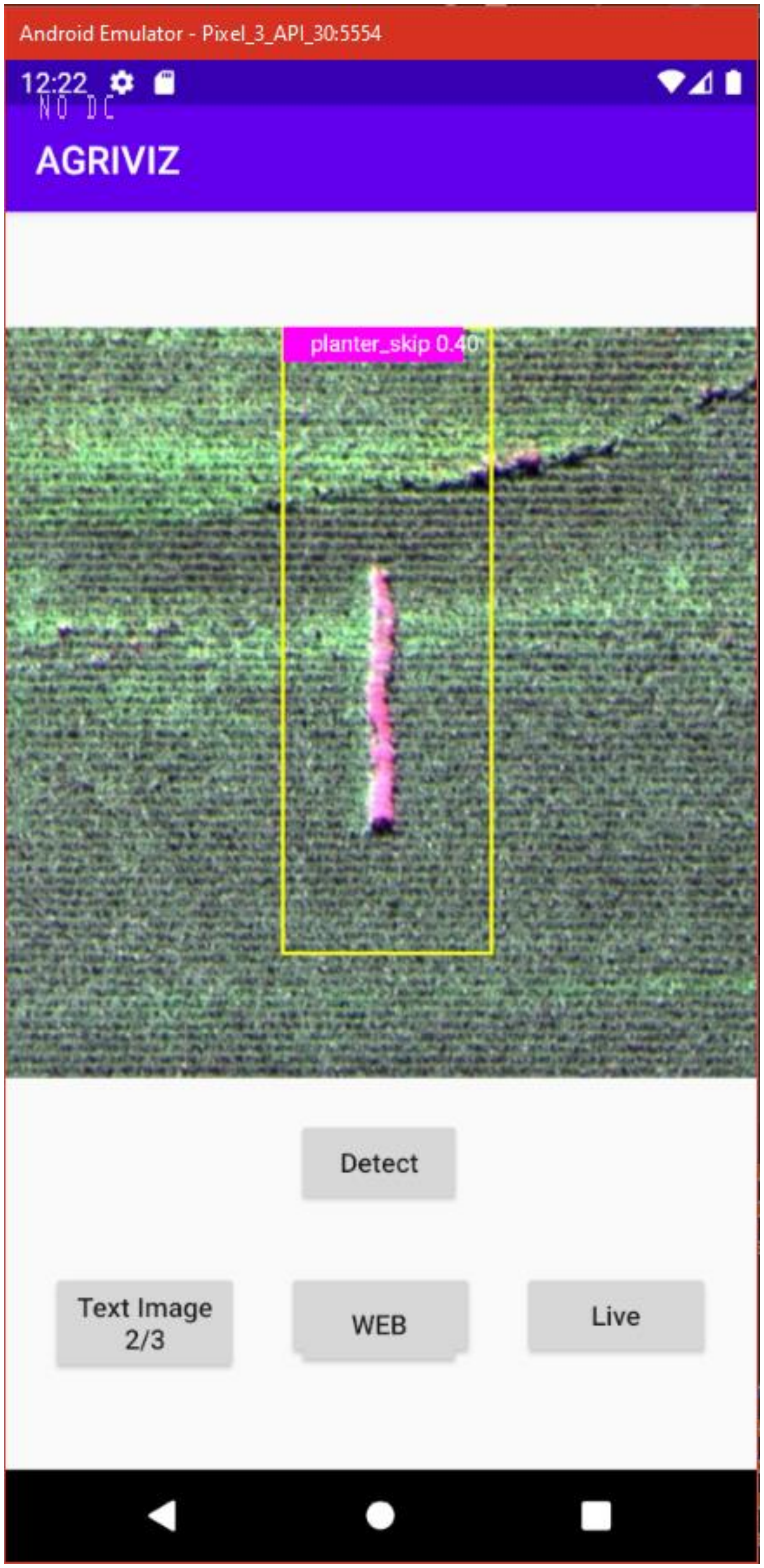


Figure 3: Successful detection of agricultural defect

Backend:

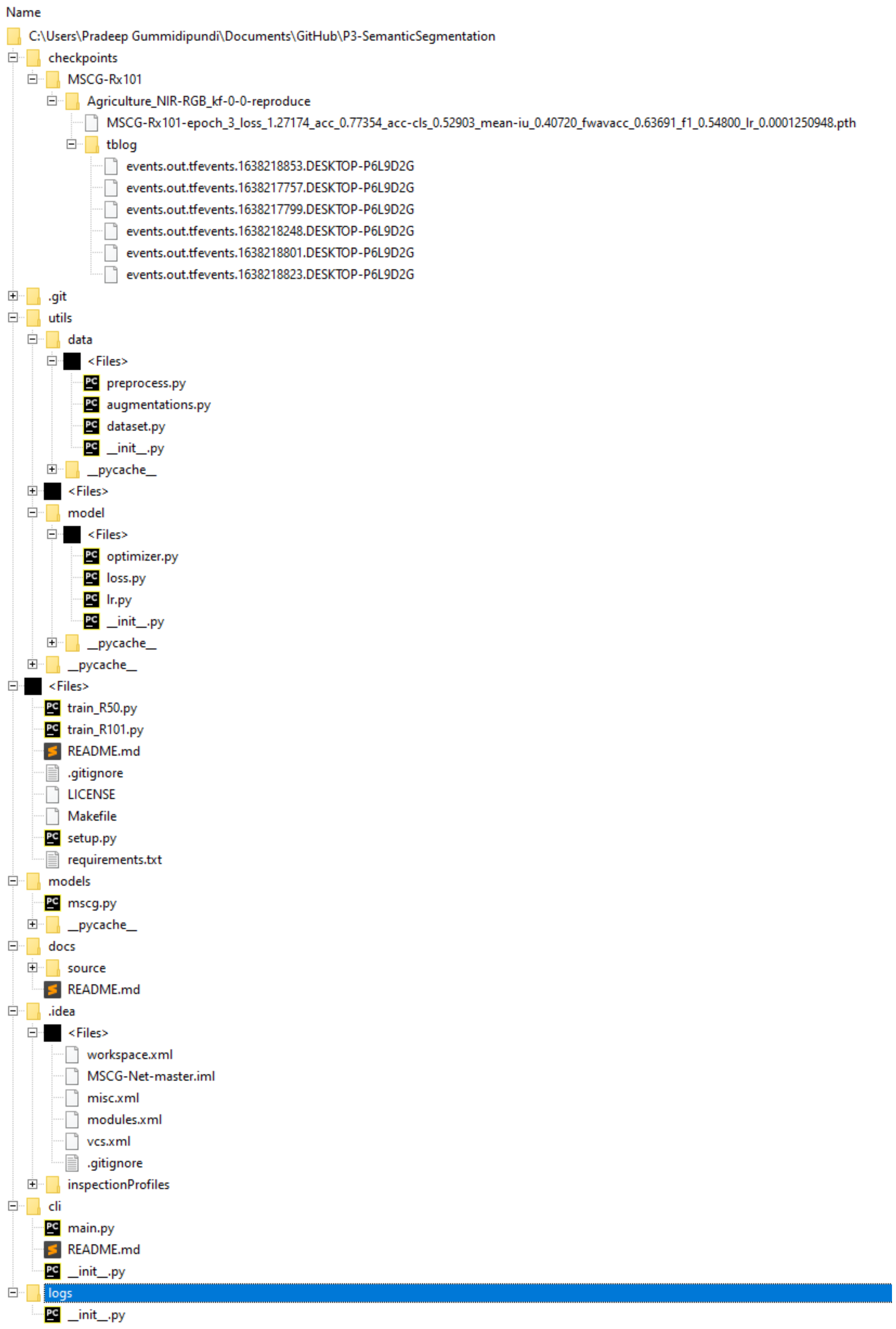
Several modifications must be made to file paths to ensure our program works on your local machine:

1. In preprocess.py under utils -> data, change DATASET_ROOT = '/home/usr/github/mscg-agriculture-dataset/2021/supervised/Agriculture-Vision-2021' to the current location of your unzipped dataset.
2. In model.py under utils -> model, change filepath = "/home/usr/github/P3-SemanticSegmentation/checkpoints/MSCG-Rx101/Agriculture_NIR-RGB_kf-0-0-reproduce/MSCG-Rx101-epoch_4_loss_1.64719_acc_0.73877_acc-cls_0.47181_mean-iu_0.34377_fwavacc_0.59123_f1_0.48653_lr_0.0001245001.pth" to the current location of the unzipped project. NOTE: This path is hardcoded to the name of the custom model produced by our team, and your model name will vary. Perform this step after training has been completed and a model produced that you intend to use.
3. In config.py under utils -> model, change root_path = Path("/home/usr/github/P3-SemanticSegmentation") to the root path of the project you've extracted/downloaded.
4. In both train_R101.py and train_R50.py, change checkpoint_path = "/home/hanz/github/P3-SemanticSegmentation/checkpoints/MSCG-Rx101/Agriculture_NIR-RGB_kf-0-0-reproduce/MSCG-Rx101-epoch_10_loss_1.62912_acc_0.75860_acc-cls_0.54120_mean-iu_0.36020_fwavacc_0.61867_f1_0.50060_lr_0.0001175102.pth" to your appropriate checkpoint folder.

Run setup.py to install all required packages for the program to function. This is a one-time function and does not need to be repeated on subsequent trainings. Alternatively, you can use Anaconda to install all packages manually, with the full list being found in requirements.txt.

Two models are created by the backend. Either one can be trained and created by simply running the corresponding .py file. To create a custom model using the pretrained model Se_ResNext50_32x4d, just run train_R50.py in PyCharm, or use the python command in CLI. To create a custom model using the pretrained model Se_ResNext101_32x4d, simply run train_R101.py using the same methods.

Training using such a large dataset can take a while for those without dedicated machines, so we've provided custom trained models in the checkpoints folder that will be auto-created.



```
Name
C:\Users\Pradeep Gummidipundi\Documents\GitHub\P3-SemanticSegmentation
⊟ checkpoints
    ⊟ MSCG-Rx101
        ⊟ Agriculture_NIR-RGB_kf-0-0-reproduce
            MSCG-Rx101-epoch_3_loss_1.27174_acc_0.77354_acc-cls_0.52903_mean-iu_0.40720_fwavacc_0.63691_f1_0.54800_lr_0.0001250948.pth
            ⊟ tblog
                events.out.tfevents.1638218853.DESKTOP-P6L9D2G
                events.out.tfevents.1638217757.DESKTOP-P6L9D2G
                events.out.tfevents.1638217799.DESKTOP-P6L9D2G
                events.out.tfevents.1638218248.DESKTOP-P6L9D2G
                events.out.tfevents.1638218801.DESKTOP-P6L9D2G
                events.out.tfevents.1638218823.DESKTOP-P6L9D2G
⊞ .git
⊟ utils
    ⊟ data
        ⊟ <Files>
            preprocess.py
            augmentations.py
            dataset.py
            __init__.py
        ⊞ __pycache__
    ⊞ <Files>
    ⊟ model
        ⊟ <Files>
            optimizer.py
            loss.py
            lr.py
            __init__.py
        ⊞ __pycache__
    ⊞ __pycache__
⊟ <Files>
    train_R50.py
    train_R101.py
    README.md
    .gitignore
    LICENSE
    Makefile
    setup.py
    requirements.txt
⊟ models
    mscg.py
    ⊞ __pycache__
⊟ docs
    ⊞ source
    README.md
⊟ .idea
    ⊟ <Files>
        workspace.xml
        MSCG-Net-master.iml
        misc.xml
        modules.xml
        vcs.xml
        .gitignore
    ⊞ inspectionProfiles
⊟ cli
    main.py
    README.md
    __init__.py
⊟ logs
    __init__.py
```

Figure 4: Unzipped P3-SemanticSegmentation file directory

Section 2:

The project is divided into two parts: a frontend and a backend that can function almost independently from each other. The backend is used to create a custom PyTorch model that is then converted to a PyTorch Lite model for use in our Android app.

Frontend:

MainActivity.java:

The driver function of every Android app. It houses buttons, calls, and calls our test images. It checks for permissions, determines the response to clicks, and finally calls our model to determine the prediction of the image it has called.

AbstractCameraXActivity.java:

Our camera driver utility, designed to request permissions for usage and start the camera. This was unused simply because the Android camera doesn't present 4d channels as our model needs.

ObjectDetectionActivityActivity.java:

Get the image provided by the camera and converts it to a bitmap object. This is then fed through a tensor and provided to PrePostProcessor utility functions, giving us our prediction with the highest level of accuracy.

PrePostProcessor.java:

Detects those bounding boxes that overlap with each other and provides the windows with the highest probability of a correct answer and serves to reduce the number of windows presented on an image (avoid data saturation).

BaseModuleActivity.java:

Monitors, creates, and destroys background threads.

ResultView.java:

Draws a window over the image highlighting the defect and displaying the type of defect detected, using OnDraw.

Backend:

setup.py:

Used to download and install the required packages for the project to run. This file isn't strictly necessary if the interpreter has already been configured for usage per the requirements.txt file. This can be done fairly easily in Anaconda, or alternatively using PyCharm's own native interpreter.

cli -> main.py:

Provides command line options to facilitate model training in CLI.

cli -> core -> process.py:

PROTOTYPE file to train a model with helper functions via CLI, in progress.

core -> net.py:

Define the parameters of the MSCG model with the difference being the ResNet model type. This uses the PyTorch nn.Module class, presets the initial layers to that of the se_resnext50_32x4d model, and the corresponding 101 model. Since this project uses a Graph Convolutional Network, we include a method to create GCNs and normalization of said GCN.

test_submission.py:

Uses checkpoints to get an accurate prediction of several sample images and apply bounding boxes to the found errors. The model makes a prediction which is compared to the actual answer for the test image provided. This file also provides a Test-Time-Augmentation (tta_real_test) function to apply transformations to test images and averages the result predicted by the ResNet custom model.

tests -> test_inference.py:

Designed to perform inference for the r50 custom models, still a work-in-progress.

utils -> trace -> checkpoint.py:

Loads predefined checkpoints that can be readily replaced and provides helper functions for the test-submission file to use in testing by loading test images.

utils -> metrics -> loss.py:

Provides the loss function (Adaptive Class Weighting) to calculate model error.

utils -> metrics -> lr.py:

Initializes the parameters of the learning rate and provides methods for adjustment.

utils -> metrics -> optimizer.py:

Implements the lookahead optimizer to train the model to find the lowest possible loss, using the loss function in loss.py.

utils -> metrics -> validate.py:

File to evaluate the collection of predictions given a set of ground-truths from the dataset.

utils -> export -> android.py(NEEDS TO BE UPDATED):

Converts the .pth model file to a .ptl format for use in Android applications. Loads the checkpointed model and uses Torch JIT to convert the file. While this was an adequate method to receive model results, we also decided to attempt a RESTful version of our app with an AWS, which was done to reduce computation on the mobile device in circumstances that don’t have that computational capacity.

utils -> trace -> gpu.py:

Returns information about the GPU usage of the machine, including available memory, available GPUs for training, and current GPU usage statistics.

utils -> trace -> logger.py:

Functions to create a logger for deeper insight during training, and has a function to print function call details.

validate.py:

Evaluates the predictions of the model given the values provided by a set of ground truth images which supports both train_ files.

utils -> config.py:

Configure the parameters of training the model. If required, reduce values of train_batch and val_batch to reduce GPU load at the cost of increased training time. This class allows for resuming training of checkpointed models.

util -> data -> dataset.py:

Defines the parameter of the CVPR Agriculture Dataset and provides augmentation/normalization methods for training and validation.

util -> data -> preprocess.py:

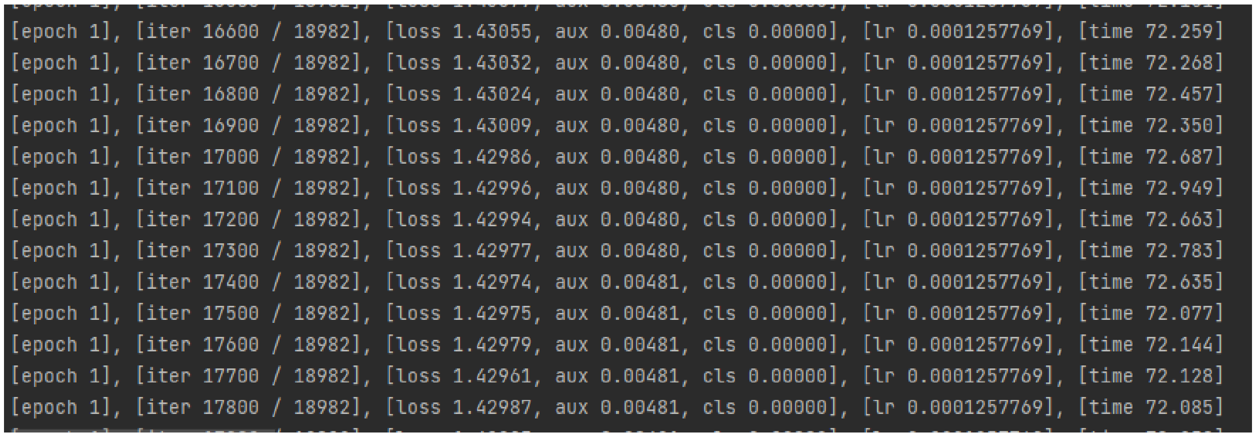
We define the root of the dataset here as a necessary first step, and define the labels as provided by the dataset. This file provides functions for producing and populating the ground truth folder in both training and validation subdirectories.

util -> data -> augmentation.py:

Provides helper functions to scale images, read them, perform augmentations on the images as required.

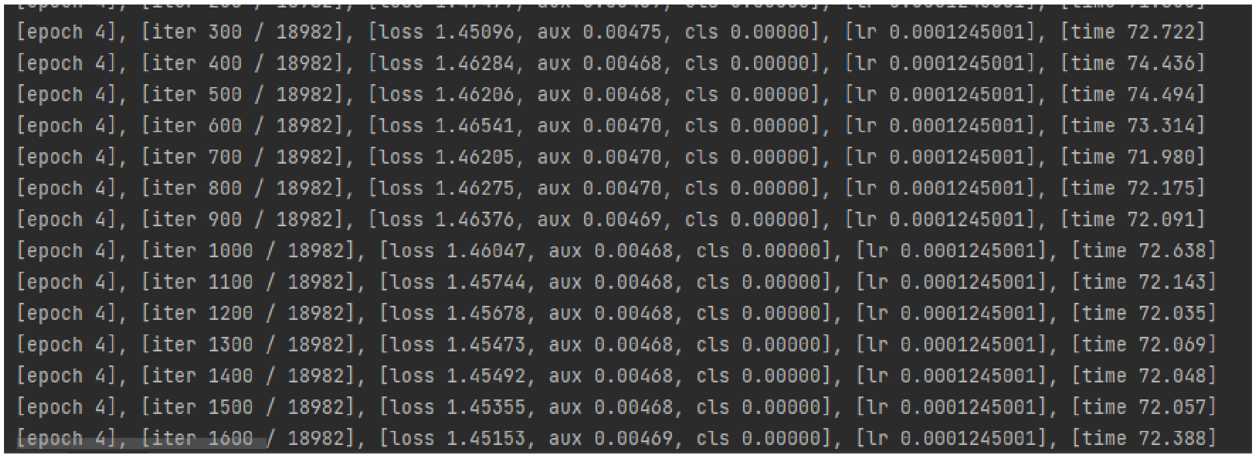
train_r50.py/ train_r101.py:

This is the main training function that validates the model and sets up and updates checkpoints. As mentioned before, this file can be avoided in favor of train_r101.py based on the model you're attempting to use. When running either file, please note that ADAM is used for the first 10 epochs, after which the optimizer must be switched manually to SGD. Thanks to our checkpointing system, the switch is extremely easy and only requires resuming training with the base_optimizer set to SGD. The number of epochs and batch sizes can be modified here quickly to suit your training purposes by modifying train_args.train_batch/train_args.val_batch, and epoch_current when using a partially trained model.



```
[epoch 1], [iter 16600 / 18982], [loss 1.43055, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.259]
[epoch 1], [iter 16700 / 18982], [loss 1.43032, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.268]
[epoch 1], [iter 16800 / 18982], [loss 1.43024, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.457]
[epoch 1], [iter 16900 / 18982], [loss 1.43009, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.350]
[epoch 1], [iter 17000 / 18982], [loss 1.42986, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.687]
[epoch 1], [iter 17100 / 18982], [loss 1.42996, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.949]
[epoch 1], [iter 17200 / 18982], [loss 1.42994, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.663]
[epoch 1], [iter 17300 / 18982], [loss 1.42977, aux 0.00480, cls 0.00000], [lr 0.0001257769], [time 72.783]
[epoch 1], [iter 17400 / 18982], [loss 1.42974, aux 0.00481, cls 0.00000], [lr 0.0001257769], [time 72.635]
[epoch 1], [iter 17500 / 18982], [loss 1.42975, aux 0.00481, cls 0.00000], [lr 0.0001257769], [time 72.077]
[epoch 1], [iter 17600 / 18982], [loss 1.42979, aux 0.00481, cls 0.00000], [lr 0.0001257769], [time 72.144]
[epoch 1], [iter 17700 / 18982], [loss 1.42961, aux 0.00481, cls 0.00000], [lr 0.0001257769], [time 72.128]
[epoch 1], [iter 17800 / 18982], [loss 1.42987, aux 0.00481, cls 0.00000], [lr 0.0001257769], [time 72.085]
```

Figure 5: Initial train at Epoch 1 of ResNet101



```
[epoch 4], [iter 300 / 18982], [loss 1.45096, aux 0.00475, cls 0.00000], [lr 0.0001245001], [time 72.722]
[epoch 4], [iter 400 / 18982], [loss 1.46284, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 74.436]
[epoch 4], [iter 500 / 18982], [loss 1.46206, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 74.494]
[epoch 4], [iter 600 / 18982], [loss 1.46541, aux 0.00470, cls 0.00000], [lr 0.0001245001], [time 73.314]
[epoch 4], [iter 700 / 18982], [loss 1.46205, aux 0.00470, cls 0.00000], [lr 0.0001245001], [time 71.980]
[epoch 4], [iter 800 / 18982], [loss 1.46275, aux 0.00470, cls 0.00000], [lr 0.0001245001], [time 72.175]
[epoch 4], [iter 900 / 18982], [loss 1.46376, aux 0.00469, cls 0.00000], [lr 0.0001245001], [time 72.091]
[epoch 4], [iter 1000 / 18982], [loss 1.46047, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.638]
[epoch 4], [iter 1100 / 18982], [loss 1.45744, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.143]
[epoch 4], [iter 1200 / 18982], [loss 1.45678, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.035]
[epoch 4], [iter 1300 / 18982], [loss 1.45473, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.069]
[epoch 4], [iter 1400 / 18982], [loss 1.45492, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.048]
[epoch 4], [iter 1500 / 18982], [loss 1.45355, aux 0.00468, cls 0.00000], [lr 0.0001245001], [time 72.057]
[epoch 4], [iter 1600 / 18982], [loss 1.45153, aux 0.00469, cls 0.00000], [lr 0.0001245001], [time 72.388]
```

Figure 6: Resumed train at Epoch 4 of ResNet101 using an Epoch 3 checkpointed model

Section 3:

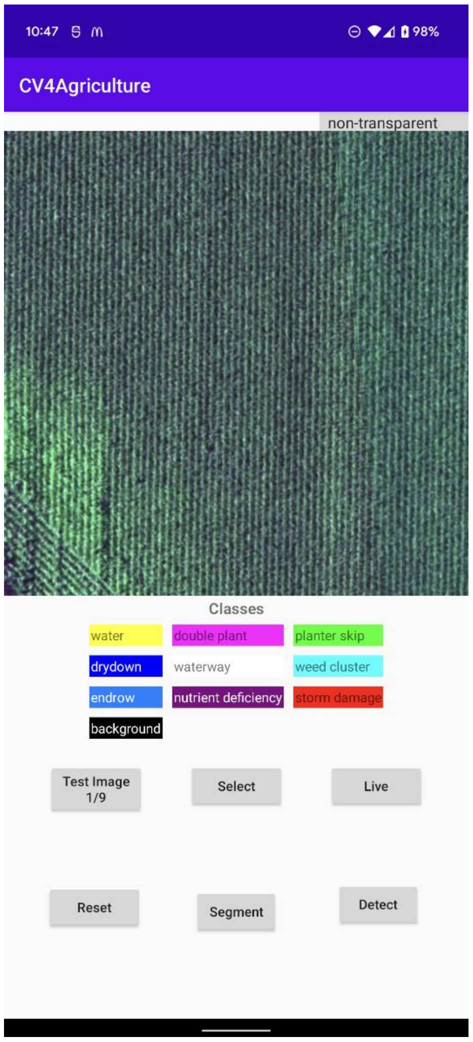


Figure 7a: Before running model

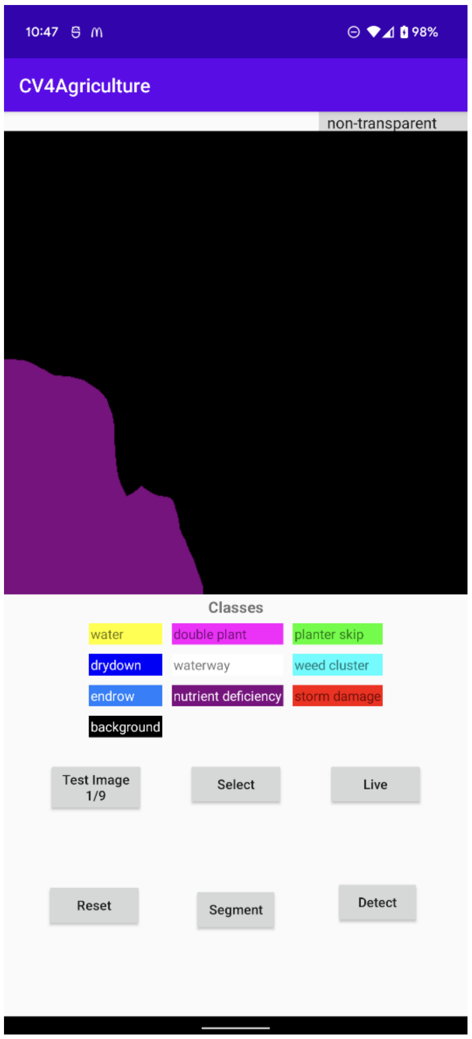


Figure 7b: After running model

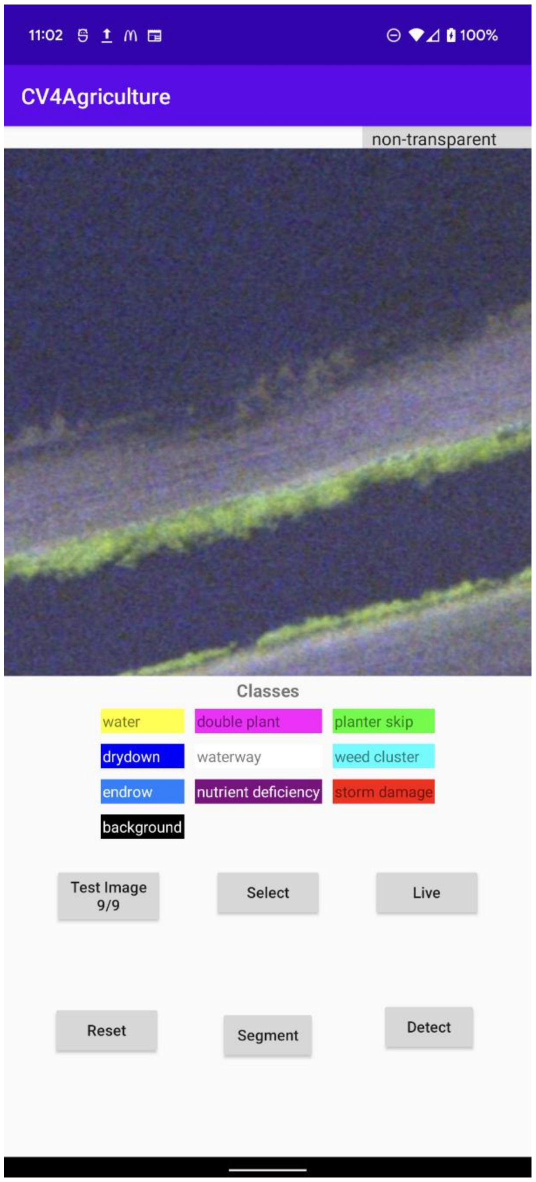


Figure 8:AndroidStartGUI

Section 4 (Comments):

After training, we had to resolve memory dependencies, there was a lot of hesitation in improving it because we had to layer multiple things together, training issues from the memory. Depth of understanding required to even clean the code, restructuring the code broke it in odd ways. While the project holders claimed they used a computing cluster, colab was not viable thanks to the size of the dataset and computational requirements. Lots of breakages resulted from minor changes, and several packages needed to be updated as well that initially provided some import issues.

Training on this model represented a significant time investment due to several factors: the massive size of the dataset and the limited computing capacity we had at our disposal. We initially thought we could train on Colab, but the dataset refused to extract properly (we were pushing the limits of free computing) so we resorted to local efforts to resolve our problems. Another issue was memory:

the project implementation loaded the entire dataset into memory, which meant we had to use a lot of RAM to keep the project running. Even with above average machines, we ran into memory access errors and out out-of-bounds issues. While the researchers claimed to have used a 1080Ti to run their model, their memory usage was exceedingly high and unrealistic for non-enterprise/academic-funded work.

Another factor was restructuring the code for readability and modification, which was necessary to get the project running. Several linkages broke with each modification, further slowing our progress as we debugged every corner of the project.

Finally, we ran into issues with our Android app. The Android camera wasn't designed to provide 4 channel images that our model was expecting from our dataset. The result was that we could serve images to it locally and from a network, but live/camera image processing was out of the question. The result was a nonfunctioning Android app that needs to be fixed in order to proceed.

In conclusion, the model itself performs exceedingly well through thorough training and testing, but several changes need to be made to our frontend to get the app serviceable and ready for widespread use.