VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Multidisciplinary Project (CO3107)

Project Report (Semester 222) - Group 1

CASSAVA LEAF DISEASE DETECTION USING COMPUTER VISION

Instructor: Dr. Phạm Hoàng Anh

Students: Dương Ngọc Quang Huy - 2052489
Nguyễn Huy Hoàng - 2053014
Trần Phạm Minh Hùng - 2053067
Hồ Thanh Bình - 2010929

HO CHI MINH CITY, APRIL 2023

Contents

1 Project Introduction **3**
1.1 Motivation . . . 3
1.2 Dataset Description . . . 4

2 Methodologies **5**
2.1 Machine Learning Framework . . . 5
2.2 Brief introduction about CNN . . . 6
2.2.1 What are convolutions? . . . 6
2.2.2 Pooling layer . . . 8
2.3 EfficientNetV2 . . . 9
2.3.1 Introduction . . . 9
2.3.2 EfficientNetV2 . . . 12
2.3.3 Problems with EfficientNet (Version 1) . . . 13
2.3.4 EfficientNetV2 — changes made to overcome the problems and further improvements . . . 14
2.3.4.a Adding a combination of MBConv and Fused-MBConv blocks . . . 14
2.3.4.b NAS search to optimize Accuracy, Parameter Efficiency, and Training Efficiency . . . 15
2.3.4.c Intelligent Model Scaling . . . 16
2.3.4.d Progressive Learning . . . 17
2.3.5 Conclusion . . . 18
2.4 Implementation . . . 18

3 Product Outcome **20**
3.1 Local Website . . . 20
3.1.1 Farm Summary . . . 22
3.1.2 Detailed Info of Each Status . . . 23
3.1.3 Direct Input Classifier . . . 23
3.2 APIs View . . . 23
3.3 Hardware setup . . . 24

4 Further Improvement **28**

List of Figures

1 Cassava . . . 3
2 Dataset images . . . 4
3 The input dataset . . . 5
4 Feature Mapping . . . 6
5 Example 1 . . . 7
6 Example 2 . . . 7
7 Pooling Demo . . . 8
8 Pooling Example . . . 9
9 An overview of Neural Architecture Search . . . 10
10 All the different parameters that the controller searched for in each layer of the network . . . 10
11 Parameters sampled in MnasNet architecture. All the contents written in blue are searched using RL . . . 11
12 Workflow to find model architecture, considering both accuracy and latency to decide the final reward for the controller . . . 11
13 Scaling the depth, width, and image resolution to create different variations of the EfficientNet model . . . 12
14 Training and Parameter efficiency of the EfficientNetV2 model compared with other state-of-the-art models . . . 13
15 Structure of MBConv and Fused-MBConv blocks . . . 15
16 The architecture of EfficientNetV2-S . . . 16
17 The architecture of EfficientNet-B0 . . . 16
18 The algorithm for progressive learning with adaptive regularization . . . 17
19 Progressive learning with adaptive regularization visual explanation . . . 18
20 System Workflow . . . 19
21 FastAPI . . . 19
22 Farm Status Summary . . . 21
23 Farm Summary . . . 22
24 Detailed Info . . . 23
25 Direct Input Classifier . . . 23
26 API endpoints view . . . 24
27 ESP32-CAM model . . . 25
28 ESP32-CAM connect to power source . . . 26
29 ESP-32 web server view . . . 27

1 Project Introduction

1.1 Motivation

As it can resist difficult conditions, cassava, the second-largest source of carbohydrates in Africa, is a crucial food security crop raised by smallholder farmers. Over 800 million people worldwide depend on it as a staple food, and the crop also supports the livelihoods of numerous rural communities and is a source of income for farmers. This starchy root is grown on at least 80% home farms in Sub-Saharan Africa, although viral infections are a major cause of the low yields.

According to the disease's severity and the stage of the crop's growth, cassava leaf diseases can result in output losses of 30% to 100%. For farmers, this may lead to lower income and food security, which may have a domino impact on their families and communities.

Cassava leaf infections can be managed and prevented from spreading by farmers with the aid of early detection and correct diagnosis. This can involve choosing resistant types, using pesticides selectively, and using the right cultural practices. Furthermore, millions of people who depend on cassava as their main source of food might have their food security increased by prompt disease detection and control, which can drastically minimize crop losses.

Figure 1: Cassava

1.2 Dataset Description

The dataset is a collection of 21,367 annotated photos that were gathered during a routine study in Uganda. The majority of the images were obtained from farmers who took pictures of their gardens and were crowdsourced; they were then annotated by professionals from the National Crops Resources Research Institute (NaCRRI), in partnership with the AI lab at Makerere University in Kampala. This is presented in a format that most accurately depicts what farmers would need to identify in the field.[1]

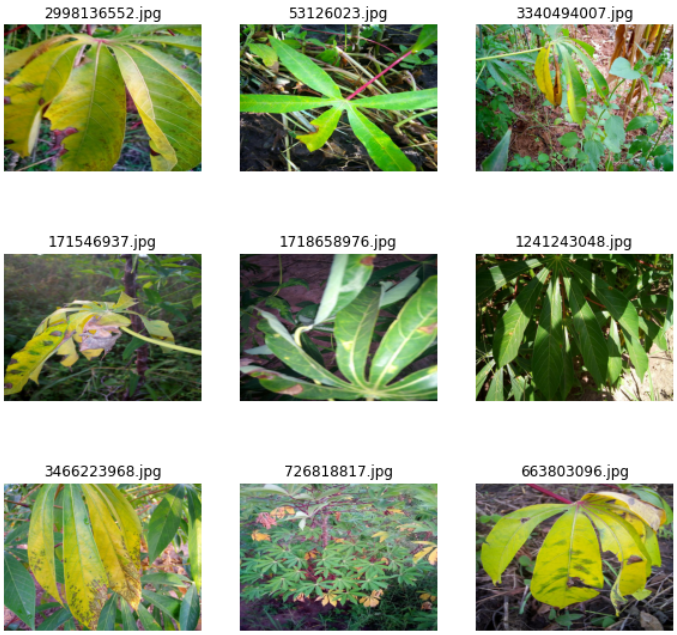


Figure 2: Dataset images

The pictures are really fascinating. several of the photos were shot in the fields, while others were taken indoors (cassava leaves are just spread out on a piece of paper in several of the photos). Different lighting effects can be seen in images. That should be considered in the model. There are many zoom levels on the leaves. Sometimes the leaves are so far away that it is difficult, even with human eyes, to tell if the plant is in trouble.

2 Methodologies

In this project, we will build an image classification model with the Cassava Leaf Disease dataset. The Cassava Leaf Disease dataset is a collection of images of cassava leaves, which are labeled with one of five possible classes of disease. We also divide the dataset 80% to train the model and the other 20% for evaluation of the model. The model aims to predict the disease class of a given cassava leaf image which can be directly uploaded from the computer or from the camera.

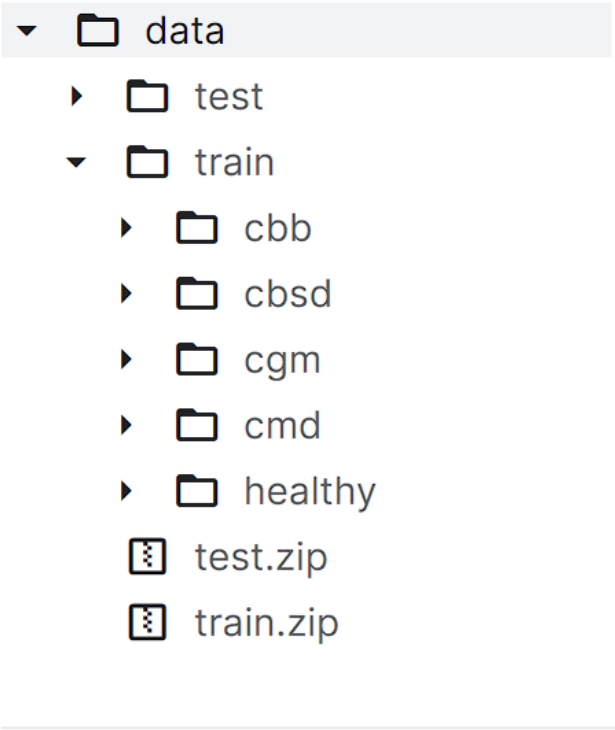


Figure 3: The input dataset

2.1 Machine Learning Framework

TensorFlow 2.x framework is our choice for the machine learning framework. TensorFlow is an open-source machine learning framework that was developed by Google and is widely used in the industry for building and deploying machine learning models. TensorFlow 2.x is the latest version of the framework, which includes several improvements over the previous versions, such as improved ease of use and increased flexibility, providing a powerful and flexible framework for building and deploying machine learning models, with a focus on ease of use and flexibility.

2.2 Brief introduction about CNN

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

2.2.1 What are convolutions?

A convolution is a filter that passes over an image, processes it, and extracts its important features. If you have an image of a person wearing a sneaker, in order to detect that a sneaker is present in the image, we will have to extract the important features and blur the inessential features. This is called feature mapping.

The feature mapping process is theoretically simple. we will scan every pixel in the image and then look at its neighboring pixels. we multiply the values of those pixels by the equivalent weights in a filter.

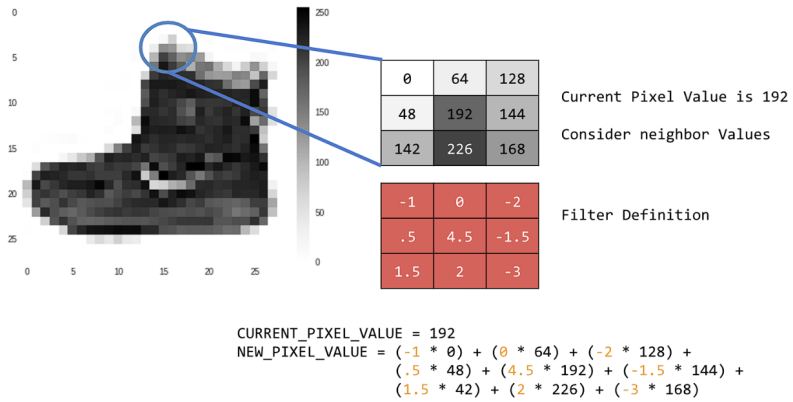


Figure 4: Feature Mapping

In this case, a 3x3 convolution matrix, or image kernel, is specified. The current pixel value is 192. We can calculate the value of the new pixel by looking at the neighbor values, multiplying them by the values specified in the filter, and making the new pixel value the final amount.

Consider the following filter values and their impact on the image.

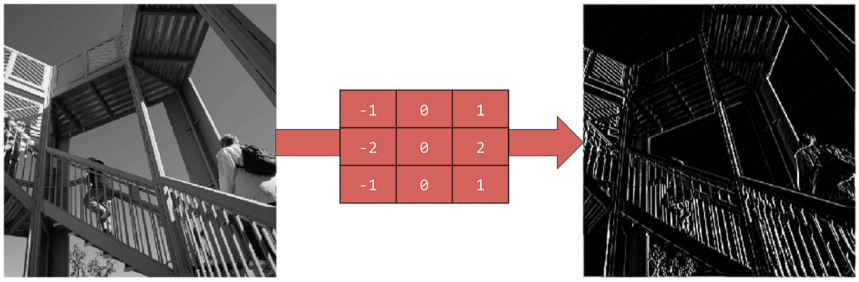


Figure 5: Example 1

Using [-1,0,1,-2,0,2,-1,0,1] gives you a very strong set of vertical lines.

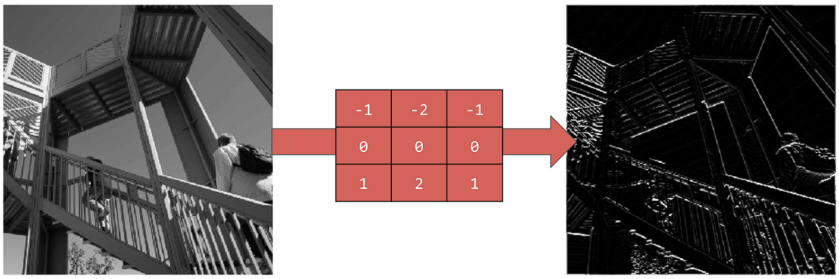


Figure 6: Example 2

Using [-1,-2,-1,0,0,0,1,2,1] gives you horizontal lines.

2.2.2 Pooling layer

Similar to convolutions, pooling greatly helps with detecting features. Pooling layers reduce the overall amount of information in an image while maintaining the features that are detected as present.

There are a number of different types of pooling, but we will use one called Maximum (Max) Pooling.

Iterate over the image and, at each point, consider the pixel and its immediate neighbors to the right, beneath, and right-beneath. Take the largest of those (hence max pooling) and load it into the new image. Thus, the new image will be one-fourth the size of the old.

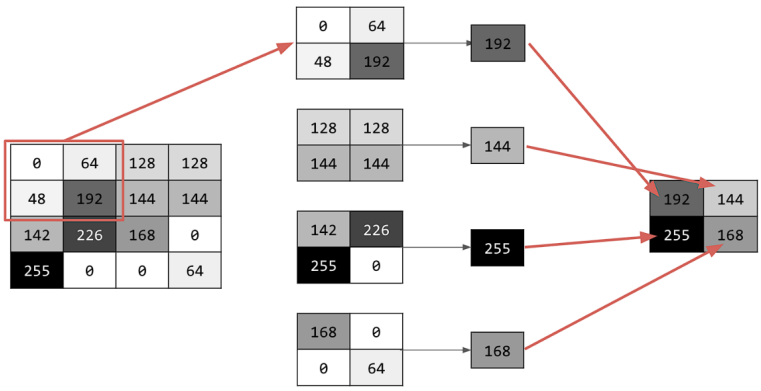


Figure 7: Pooling Demo

The image is now 256x256, one-fourth of its original size, and the detected features have been enhanced despite less data now being in the image.

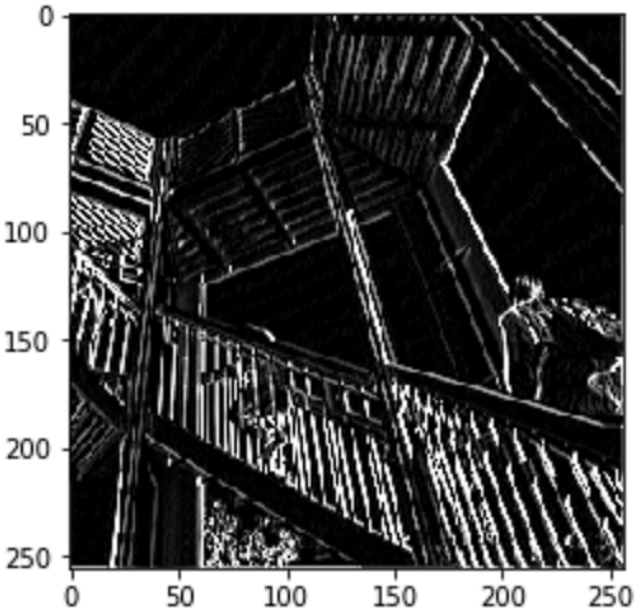


Figure 8: Pooling Example

2.3 EfficientNetV2

2.3.1 Introduction

EfficientNets are currently one of the most powerful convolutional neural network (CNN) models. With the rise of Vision Transformers, which achieved even higher accuracies than EfficientNets, the question arose whether CNNs are now dying. EfficientNetV2 proves this wrong by not just improving accuracies but by also reducing training time and latency.

The EfficientNet models are designed using neural architecture search(NAS). The idea is to use a controller (a network such as an RNN) and sample network architectures from a search space with probability 'p'. This architecture is then evaluated by first training the network, and then validating it on a test set to get the accuracy 'R'. The gradient of 'p' is calculated and scaled by the accuracy of 'R'. The result (reward) is fed to the controller RNN. The controller acts as the agent, the training and testing of the network act as the environment, and the result acts as the reward. This is the common Reinforcement learning (RL) loop. This loop runs multiple times till the controller finds the network architecture which gives a high reward (high test accuracy).

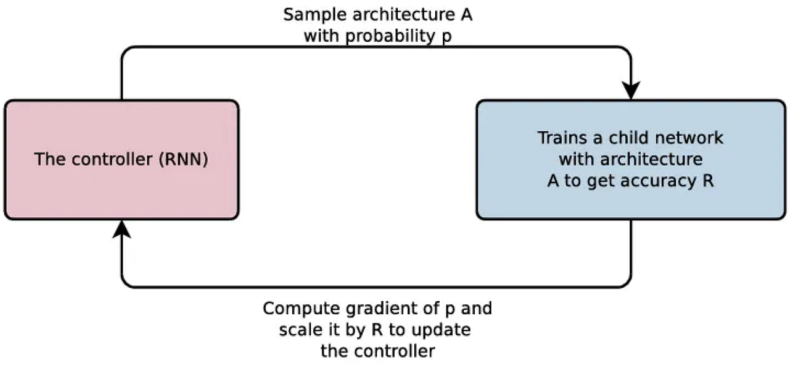


Figure 9: An overview of Neural Architecture Search

The controller RNN samples various network architecture parameters — such as the number of filters, filter height, filter width, stride height, and stride width for each layer. These parameters can be different for each layer of the network. Finally, the network with the highest reward is chosen as the final network architecture.

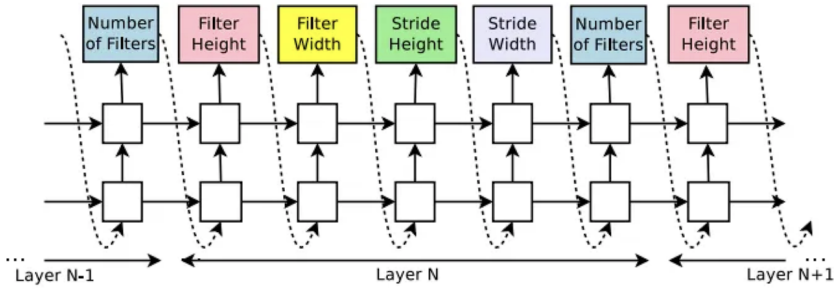


Figure 10: All the different parameters that the controller searched for in each layer of the network

Even though this method worked well, one of the problems with this method was that this required a huge amount of computing power as well as time.

To overcome this problem, the authors looked into previously famous Convolutional Neural Network (CNN) architectures such as VGG or ResNet, and figured, that these architectures do not have different parameters in each layer, but rather have a block with multiple convolutional and pooling layers, and throughout the network architecture, these blocks are used multiple times. The authors used this idea to find such blocks using the RL controller and just repeated

these blocks N times to create the scalable NASNet architecture.

In this network, the authors chose 7 blocks, and one layer of a block was sampled and repeated for each block.

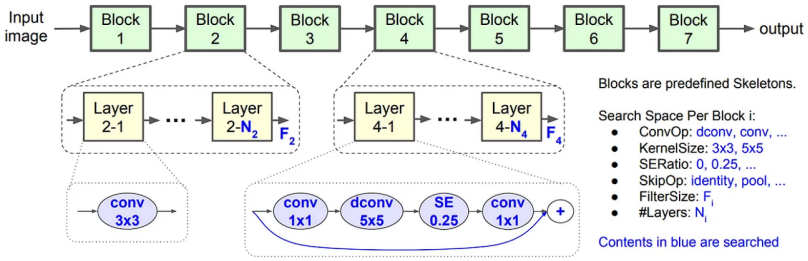


Figure 11: Parameters sampled in MnasNet architecture. All the contents written in blue are searched using RL

In addition to these parameters, one more very important parameter was considered while deciding the reward, which went into the controller, and that was 'latency'. So for MnasNet, the authors considered both the accuracy and latency to find the best model architecture. This made the architecture small, and it could run on mobile or edge devices.

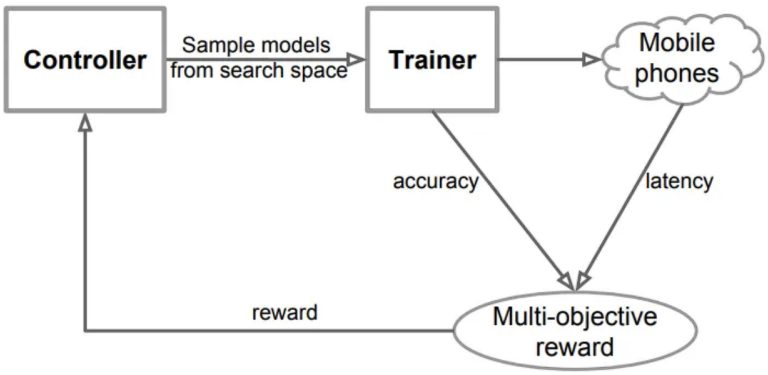


Figure 12: Workflow to find model architecture, considering both accuracy and latency to decide the final reward for the controller

The workflow for finding the EfficientNet architecture was very similar to the MnasNet, but instead of considering 'latency' as a reward parameter, 'FLOPs (floating point operations per second)' were considered. This criteria search gave the authors a base model, which they called EfficientNetB0. Next, they scaled up the base models' depth, width, and image resolution (using grid search) to create 6 more models, from EfficientNetB1 to EfficientNetB7.

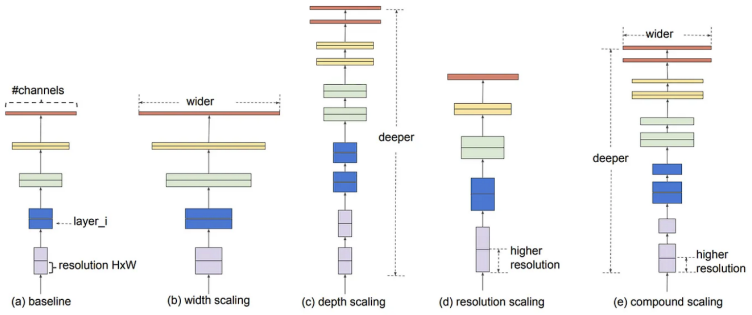
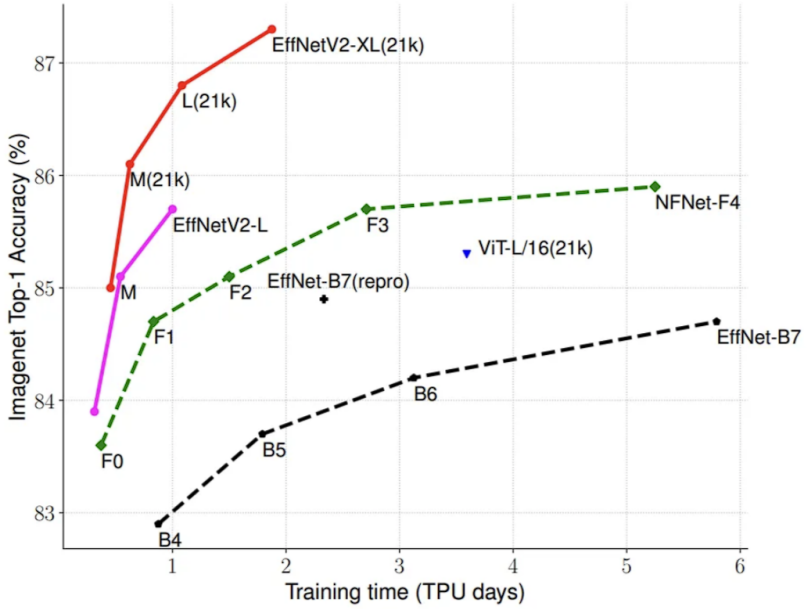


Figure 13: Scaling the depth, width, and image resolution to create different variations of the EfficientNet model

2.3.2 EfficientNetV2

EfficientNetV2 goes one step further than EfficientNet to increase training speed and parameter efficiency. This network is generated by using a combination of scaling (width, depth, resolution) and neural architecture search. The main goal is to optimize training speed and parameter efficiency. Also, this time the search space also included new convolutional blocks such as Fused-MBConv. In the end, the authors obtained the EfficientNetV2 architecture which is much faster than previous and newer state-of-the-art models and is much smaller (up to 6.8x times).



(a) Training efficiency.

	EfficientNet (2019)	ResNet-RS (2021)	DeiT/ViT (2021)	EfficientNetV2 (ours)
Top-1 Acc.	84.3%	84.0%	83.1%	83.9%
Parameters	43M	164M	86M	24M

(b) Parameter efficiency.

Figure 14: Training and Parameter efficiency of the EfficientNetV2 model compared with other state-of-the-art models

Figure 14(b) clearly shows that The EfficientnetV2 has 24 million parameters, while a Vision Transformer (ViT) has 86 million parameters. The V2 version also has nearly half the parameters of the original EfficientNet. While it does reduce the parameter size significantly, it maintains similar or higher accuracies than the other models on the ImageNet dataset.

The authors also perform progressive learning, that is, a method to progressively increase image size along with regularizations such as dropout and data augmentation. This method further speeds up training.

2.3.3 Problems with EfficientNet (Version 1)

EfficientNets are generally faster to train than other large CNN models. But, when large image resolution was used to train the models (B6 or B7 models), the training was slow. This is because larger EfficientNet models require larger image sizes to get optimal results. When larger images are used, the batch size needs to be lowered to fit these images in the GPU/TPU memory, making the overall process slow.

In the early layers of the network architecture, depthwise convolutional layers (MBConv) were slow. Depthwise convolutional layers generally have fewer parameters than regular convolutional layers, but the problem is that they cannot fully make use of modern accelerators. To overcome this problem EfficientNetV2 uses a combination of MBConv and Fused MBConv to complete the training faster without increasing the parameters.

Equal scaling was applied to the height, width, and image resolution to create the various EfficientNet models from B0 to B7. This equal scaling of all layers is not optimal. For example, if the depth is scaled by 2, all the blocks in the network get scaled up 2 times, making the network very large/deep. It might be more optimal to scale one block two times and the other 1.5 times (non-uniform scaling), to reduce the model size while maintaining good accuracy.

2.3.4 EfficientNetV2 — changes made to overcome the problems and further improvements

2.3.4.a Adding a combination of MBConv and Fused-MBConv blocks

MBConv block often cannot fully make use of modern accelerators. Fused-MBConv layers can better utilize server/mobile accelerators. The only differences between the structures of MBConv and the Fused-MBConv are the last two blocks. While the MBConv uses a depthwise convolution (3x3) followed by a 1x1 convolution layer, the Fused-MBConv replaces/fuses these two layers with a simple 3x3 convolutional layer.

Fused MBConv-layers can make training faster with only a small increase in the number of parameters, but if many of these blocks are used, it can drastically slow down training with many more added parameters. To overcome this problem, the authors passed both MBConv and Fused-MBConv in the neural architecture search, which automatically decides the best combination of these blocks for the best performance and training speed.

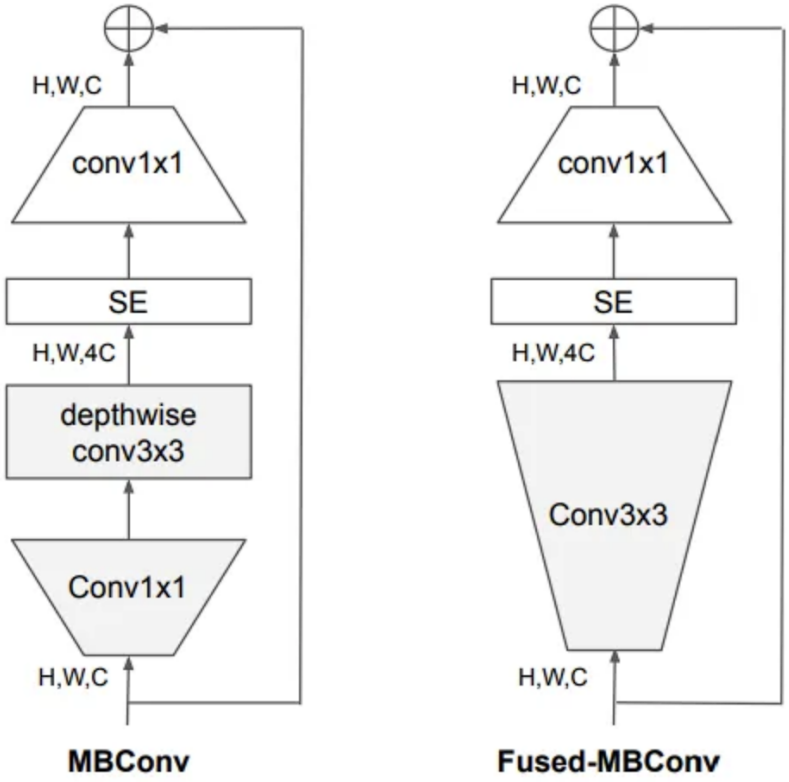


Figure 15: Structure of MBConv and Fused-MBConv blocks

2.3.4.b NAS search to optimize Accuracy, Parameter Efficiency, and Training Efficiency

The EfficientNet model was used as a backbone, and the search was conducted with varying design choices such as — convolutional blocks, number of layers, filter size, expansion ratio, and so on. Nearly 1000 models were samples and trained for 10 epochs and their results were compared. The model which optimized best for accuracy, training step time, and parameter size was chosen as the final base model for EfficientNetV2.

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Figure 16: The architecture of EfficientNetV2-S

Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1

Figure 17: The architecture of EfficientNet-B0

Figure 16 shows the base model architecture of the EfficientNetV2 model (EfficientNetV2-S). The model contains Fused-MBConv layers in the beginning but later switches to MBConv layers. For comparison, Figure 17 shows the architecture of the previous EfficientNet. The previous version only has MBConv layers and no Fused-MBConv layers.

EfficientNetV2-S also has a smaller expansion ratio as compared to EfficientNet-B0. EfficeinetNetV2 does not use 5x5 filter matrix and only uses 3x3 filter matrix.

2.3.4.c Intelligent Model Scaling

Once the EfficientNetV2-S model was obtained, it was then scaled up to obtain the EfficientNetV2-M and EfficientNetV2-L models. A compound scaling method was used, similar to the EfficientNet, but some more changes were made to make the models smaller and faster.

1. maximum image size was restricted to 480x480 pixels to reduce GPU/TPU memory usage, hence increasing training speed.

2. more layers were added to later stages (stages 5 and 6 in Figure 16), to increase network capacity without increasing much runtime overhead.

2.3.4.d Progressive Learning

Larger image sizes generally tend to give better training results but increase training time. Some papers have previously proposed dynamically changing image size, but it often leads to a loss in training accuracy.

The authors of EfficientNetV2 show that as the image size is dynamically changed while training the network, so should the regularization be changed accordingly. Changing the image size, but keeping the same regularization leads to a loss in accuracy. Furthermore, larger models require more regularization than smaller models.

The authors of EfficientNetV2 used Progressive Learning with Adaptive Regularization. The idea is very simple. In the earlier steps, the network was trained on small images and weak regularization. This allows the network to learn the features fast. Then the image sizes are gradually increased, and so are the regularizations. This makes it hard for the network to learn. Overall this method, gives higher accuracy, faster training speed, and less overfitting.

Algorithm 1 Progressive learning with adaptive regularization.

Input: Initial image size S_0 and regularization $\{\phi_0^k\}$.
Input: Final image size S_e and regularization $\{\phi_e^k\}$.
Input: Number of total training steps N and stages M.
for $i = 0$ **to** $M - 1$ **do**
  Image size: $S_i \leftarrow S_0 + (S_e - S_0) \cdot \frac{i}{M-1}$
  Regularization: $R_i \leftarrow \{\phi_i^k = \phi_0^k + (\phi_e^k - \phi_0^k) \cdot \frac{i}{M-1}\}$
  Train the model for $\frac{N}{M}$ steps with S_i and R_i.
end for

Figure 18: The algorithm for progressive learning with adaptive regularization

Figure 19: Progressive learning with adaptive regularization visual explanation

As the number of epochs increases the image size and the augmentations also increase gradually. EfficicentNetV2 uses three types of regularization — Dropout, RandAugment, and Mixup.

2.3.5 Conclusion

EfficientNetV2 models are smaller and faster than most state-of-the-art models. This CNN model shows that even though Vision Transformers have taken the computer vision world by storm, by getting higher accuracies than other CNNs, better-structured CNN models with improved training methods can still achieve faster and better results than transformers, proving that CNNs are here to stay. [2]

2.4 Implementation

For our system, we are implementing the system using the client-server approach with the model will be deployed on a server and the clients which are the hardware/mobile app will send requests to the server through APIs.

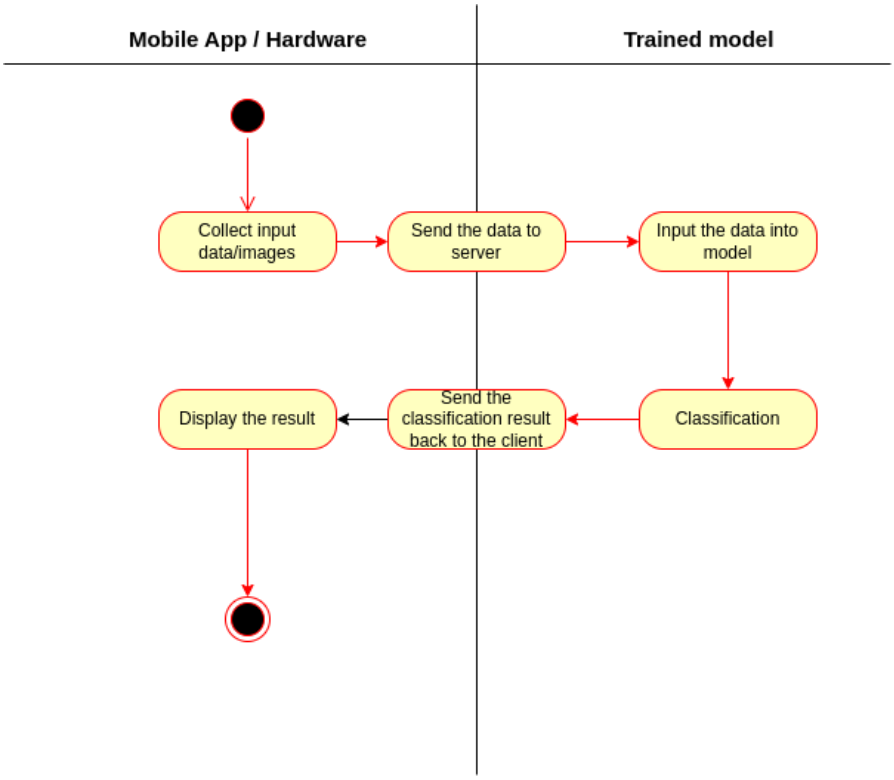


Figure 20: System Workflow

The request and communication between the model and the client will be handled by FastAPI. FastAPI is a modern, high-performance web framework for building APIs with Python. Its advantages consist of high performance, ease of use, and standard-based.

Figure 21: FastAPI

3 Product Outcome

For a cassava farm, we will divide it into many sections and install an ESP32 camera for each section. The cameras then capture the pictures of these sections and send them to our model periodically to classify whether a section is healthy or not. The result is displayed on a local webpage for visualization as well as simple statistic.

3.1 Local Website

Below is the fullscreen capture of our webpage:

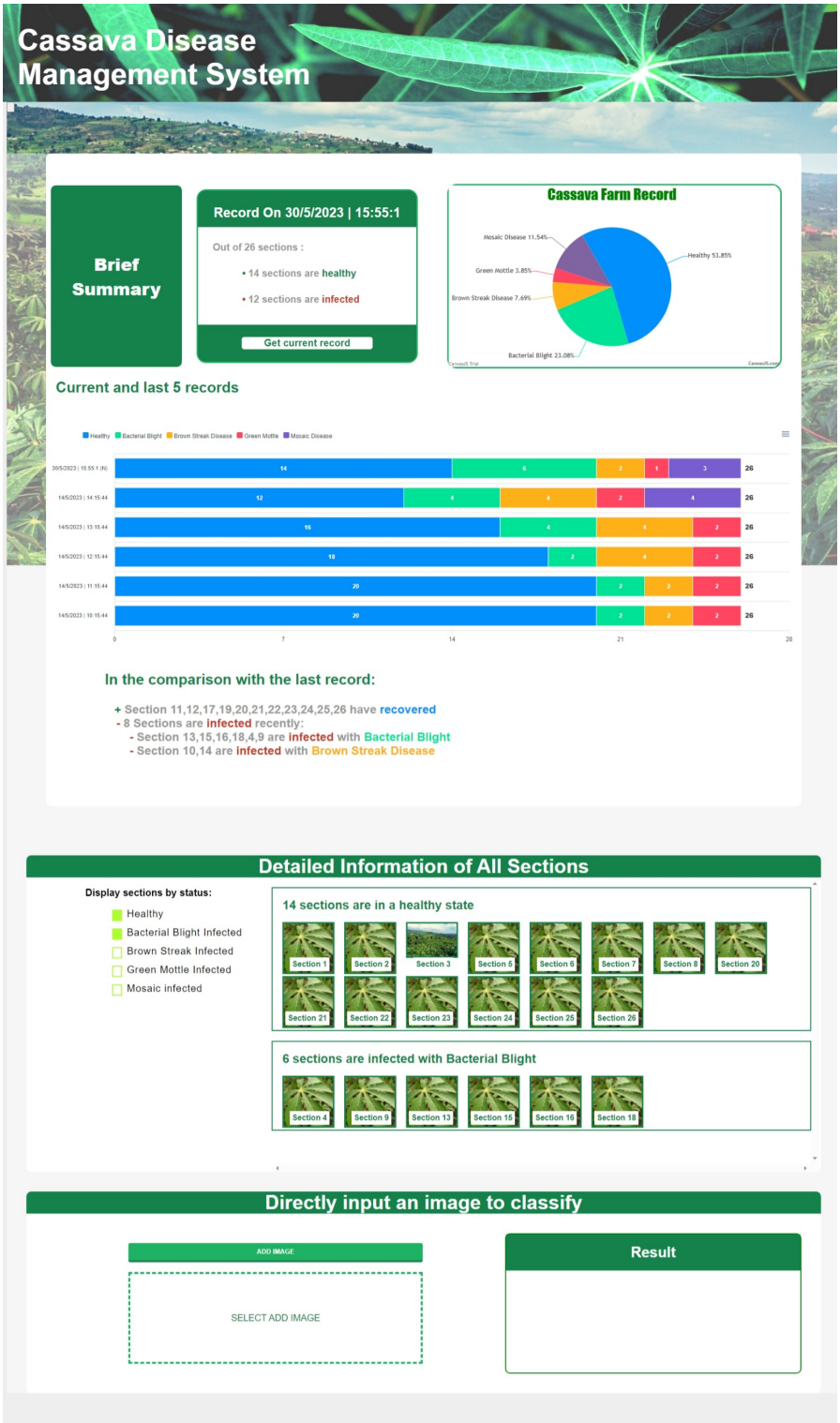


Figure 22: Farm Status Summary

Our webpage consists of three parts, the summary of our farm, the detailed information of all sections and the custom input classification system. The data used for this webpage is entirely retrieved from our FastAPI server.

3.1.1 Farm Summary

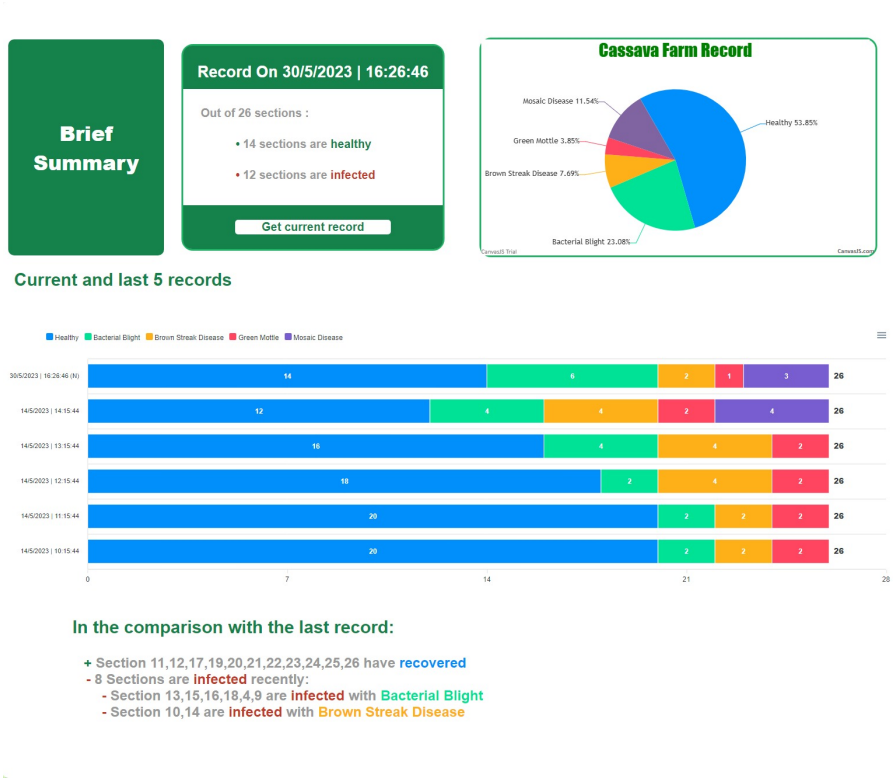


Figure 23: Farm Summary

In our farm's status summary, we will display the current number of healthy and infected sections as well as the number of sections of each status. We also compare the current record with the latest 5 previous records and displayed newly recovered/infected sections.

3.1.2 Detailed Info of Each Status

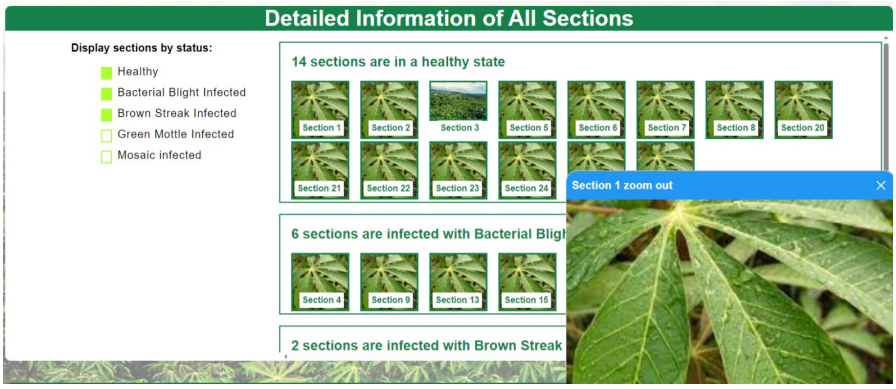


Figure 24: Detailed Info

This part displayed the current pictures of all sections divided by their status. User can choose which status to be displayed as well as zoom out a section with a zoombox.

3.1.3 Direct Input Classifier

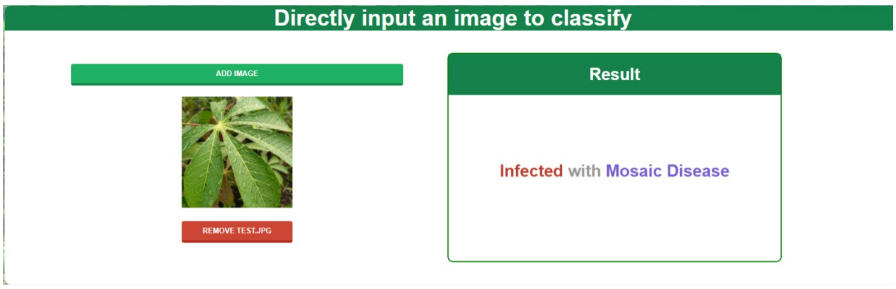


Figure 25: Direct Input Classifier

With the direct input classifier, we can upload an image from our own computer to determine whether the cassava leaf in the image is infected or not.

3.2 APIs View

The client webpage and server machine learning model will interact through APIs endpoint, which include features such as:

Classification: Manual classification by uploading the image to the model and receive the classification result as response.

Remote Classification: This feature will take the image from our set up cameras and load into the model and send the classification results as response.

Set/Get Records: Read and write the classification records feature.

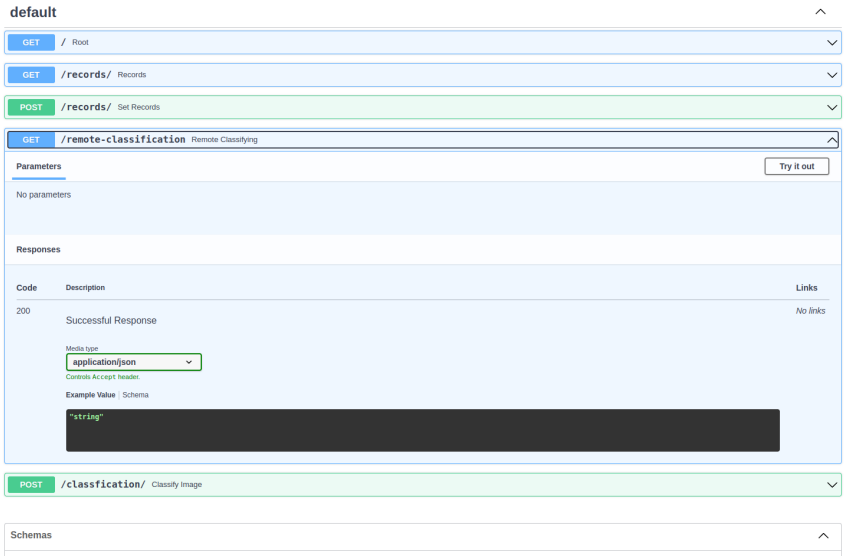


Figure 26: API endpoints view

3.3 Hardware setup

We built a web using client-server architecture, which allow user to upload images of the leafs in doubt of disease to check through result after running the presented detection model on those images. User can use any device with camera to capture and upload image, which in most scenario likely to be a phone, however, this manner require user to be in a certain distance and only capture the state of limited number of leafs. A remedy come up to this matter is our web's remote leaf monitoring system(RLMS) provide up to 26 surveillance camera.

The camera device we use in our implementation of RLMS is ESP32-CAM model. This is an incredible strong IoT device with affordable price that come in Wi-Fi compatible and programmable.

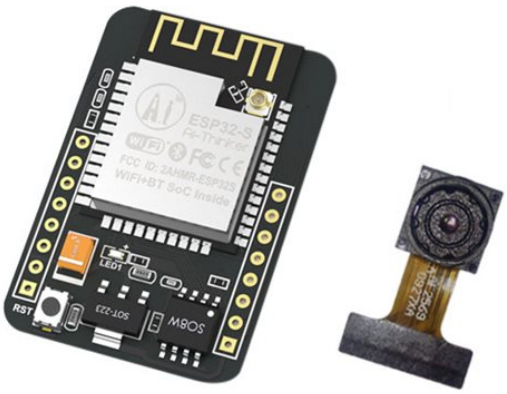


Figure 27: ESP32-CAM model

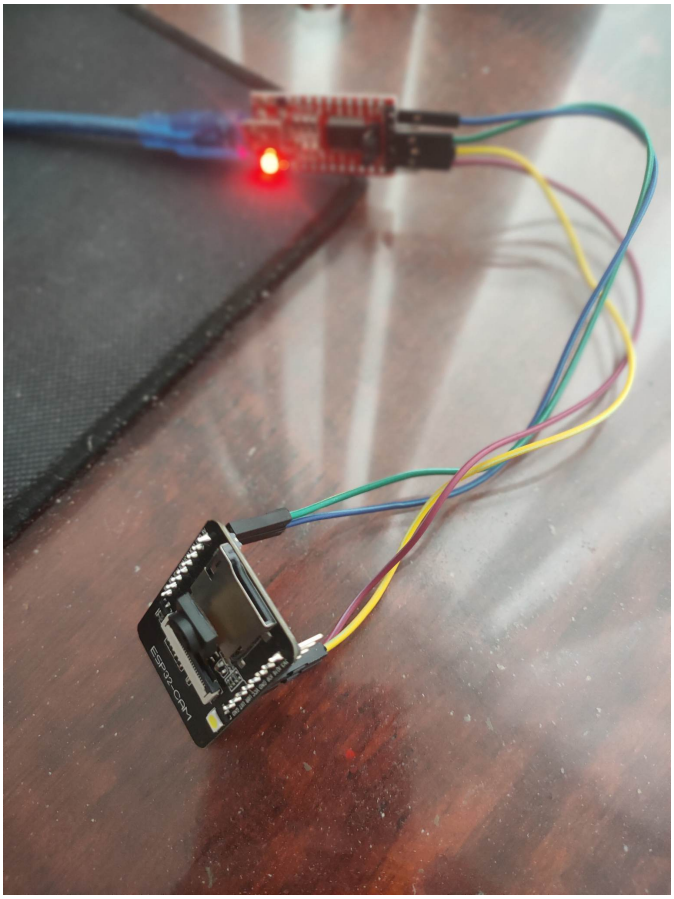


Figure 28: ESP32-CAM connect to power source

We program the ESP32 to connect to Wi-Fi and play a role of a web server, that receive request from client then capture and save image in flash memory. Captured image will be display on our web and the wed server of ESP32, which you can access to through IP provided by the router ESP32 connect to. User then user captured image to run checking, this process will be further explain bellow. Now following an explanation on some part of the code used to the ESP32.

```
15  // Replace with the network credentials you want to connect
16  const char* ssid ="REPLACE_WITH_YOUR_SSID";
17  const char* password = "REPLACE_WITH_YOUR_PASSWORD";
18
19  // Create AsyncWebServer object on port. We choose port 5050 to avoid any conflict
20  AsyncWebServer server(5050);
```

```
//building ESP32 web server
const char index_html[] PROGMEM = R"rawliteral(
<!DOCTYPE HTML><html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <style>
    body { text-align:center; }
    .vert { margin-bottom: 10%; }
    .hori{ margin-bottom: 0%; }
  </style>
</head>
<body>
  <div id="container">
    <h2>ESP32-CAM Last Photo</h2>
    <p>It might take more than 5 seconds to capture a photo.</p>
    <p>
      <button onclick="rotatePhoto();">ROTATE</button>
      <button onclick="capturePhoto()">CAPTURE PHOTO</button>
      <button onclick="location.reload();">REFRESH PAGE</button>
    </p>
  </div>
  <div><img src="saved-photo" id="photo" width="70%"></div>
</body>
<script>
  var deg = 0;
  function capturePhoto() {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', "/capture", true);
    xhr.send();
  }
  function rotatePhoto() {
    var img = document.getElementById("photo");
    deg += 90;
    if(isOdd(deg/90)){ document.getElementById("container").className = "vert"; }
    else{ document.getElementById("container").className = "hori"; }
    img.style.transform = "rotate(" + deg + "deg)";
  }
  function isOdd(n) { return Math.abs(n % 2) == 1; }
</script>
</html>)rawliteral";
```

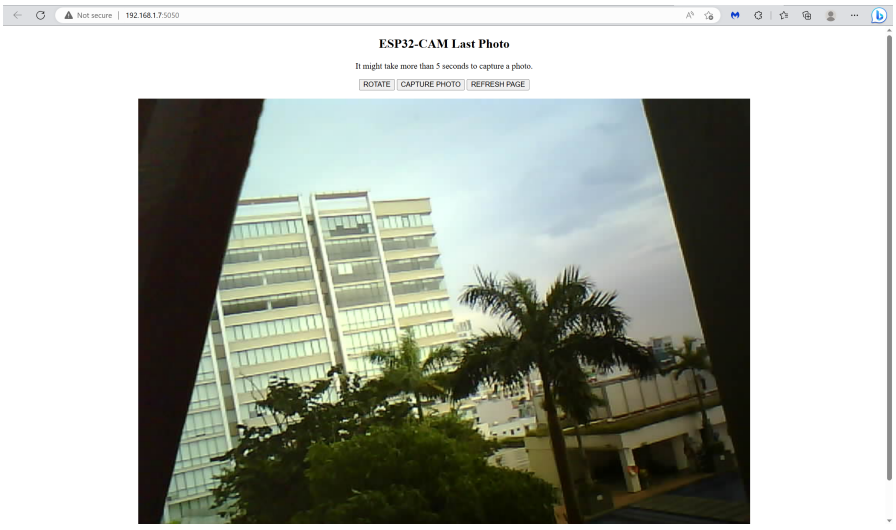


Figure 29: ESP-32 web server view

```
160  //invoke to take new photo and request captured photo
161  server.on("/saved-photo", HTTP_GET, [](AsyncWebServerRequest * request) {
162    takeNewPhoto = true;
163    request->send(SPIFFS, FILE_PHOTO, "image/jpg", false);
164  });
```

```
134  // Select lower framesize if the camera doesn't support PSRAM
135  if (psramFound()) {
136    config.frame_size = FRAMESIZE_SVGA; //800 x 600
137    config.jpeg_quality = 10; //10-63 lower number means higher quality
138    config.fb_count = 2;
139  } else {
140    config.frame_size = FRAMESIZE_VGA; //640 x 480
141    config.jpeg_quality = 12;
142    config.fb_count = 1;
143  }
```

4 Further Improvement

As for further updates, we plan on perform improvements and optimizations on the model classification accuracy by tuning the hyperparameters, perform transfer learning, update the dataset and its accuracy, etc.

Currently, our camera has only been setup to be access from the same network. However, this approach cause some inconveniences as the system only work when the server and camera are on the same network. Some solutions for this problems as we can think of are:

- Forward the camera port through gateway router(this method is simple to implement yet can cause some security issues).
- Create a NGROK tunnel to the camera video stream port(preferred approach).

References

[1] Makerere University AI Lab. Cassava leaf disease classification. https://www.kaggle.com/competitions/cassava-leaf-disease-classification/overview, 2021. Last accessed 22 April 2023.

[2] Arjun Sarkar. Efficientnetv2: Faster, smaller, and higher accuracy than vision transformers. https://towardsdatascience.com/efficientnetv2-faster-smaller-and-higher-accuracy-than-vision-transformers-98e23587bf04, 2022. Last accessed 27 May 2023.

–END–