VEGETABLE IMAGE CLASSIFICATION

Shyam Kumar Kanuru, Sri Rashmitha Boya, Ritwik Budhiraja
ENGR-E 503 Intro to Intelligent Systems

Abstract

This project aims to develop a deep learning framework that will classify images of vegetables. Throughout the timeline of this project, three convolution neural network models and three transfer learning models were built and compared. These models were used on a dataset of different vegetable images and some customs images added by us. The ResNet and MobileNet neural networks trained using transfer learning methods outperformed other models. Furthermore, an end-to-end application was also built to show the classification happening in real time.

Introduction

A vegetable production cycle generally includes selecting and sorting vegetables, labeling them, and placing them on shelves. These tasks are usually performed manually, therefore there has been an uprising demand for automating these processes. Amazon recently introduced a smart grocery store which uses the concept of a smart shopping cart. The Amazon dash cart uses computer vision algorithms and camera sensors to automatically detect the items being put in the cart and then upload the prices of these items in the virtual cart. This project presents an introductory step to achieving the goal of removing such tedious efforts and emulating Amazon's smart shopping idea.

Methods

The following sections represent how the classification of vegetable images was done.

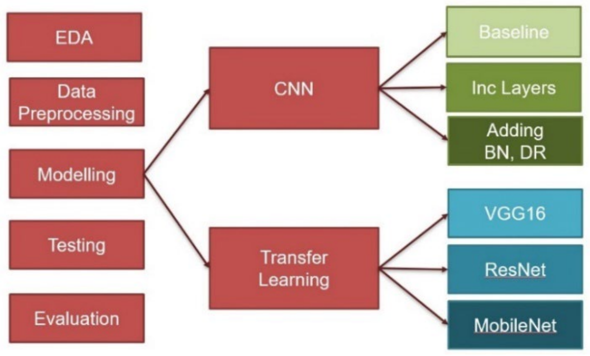


Figure 1: Flow chart

A. Exploratory Data Analysis (EDA)

In our dataset, there are 15 different common vegetable classifications, as seen in *Figure 2.*

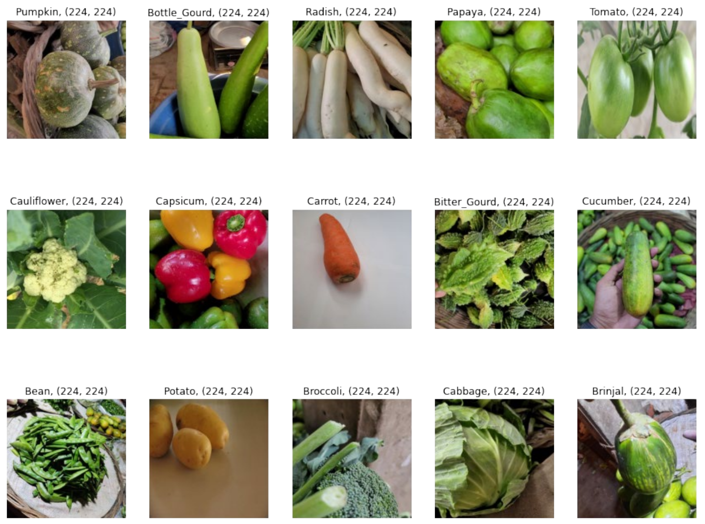


Figure 2: Types of Vegetables

B. Data Preprocessing

Image resizing:

Firstly, we performed one of the common preprocessing steps in Image Classification, i.e Image Resizing. This is done to guarantee that all the dataset's images are of the same size. In this case, we resized each image to 224*224

Image rescaling:

Then after the resizing of the images, image rescaling is done to change the scale of an image. Usually, this step is carried out to make sure that all the images have the same size and resolution.

Image augmentation:

After rescaling and resizing, we performed image augmentation tasks like random flip and random rotation to create a diverse dataset, so the models perform better. Here, we randomly rotated and flipped the images rather than increasing the size of our dataset

C. Modelling

After preprocessing the image, we constructed and trained our models to be able to categorize a given image into one of the 15 vegetable groups. The different models we used to classify our photographs are listed below. We compared several Convolutional neural networks (CNN) models and pre-trained CNN architectures using transfer learning. The TensorFlow Framework and Keras were utilized throughout the project to execute the models. Additionally, all the models make use of the following parameters.

- **Loss Function:** Cross Entropy
- **Optimizer:** Adam
- **Performance metric:** Accuracy

CNN Models

- *Baseline CNN*
- *CNN with increased layers*
- *CNN with Batch Normalization and Dropout layers*

Transfer Learning

Transfer Learning is a method in machine learning in which a pre-trained model is reused on a new problem. Then these architectures are compared with the traditional CNN architectures (above-discussed models). The following are the different models proposed using transfer learning.

- *VGG-16*
- *ResNet*
- *MobileNet*

D. Testing

After creating and training the above models, we tested our models on the test and validation datasets. This helped us assess the model's ability to generalize to new data. Also, we tested the above-mentioned models on custom data, i.e., we created a custom dataset by taking a few photos of real-life vegetables and then tested the models on that dataset.

E. Evaluation

A collection of measures can be used to gauge the effectiveness of the models during the evaluation phase. We used model accuracy as our performance metric. Metrics are used quantitatively to evaluate the model's performance and provide a basis for comparing different models or different versions (architectures) of the same model.

Experimental Setup

We shall go into more detail about our models' experimental setup below.

Baseline CNN

We developed a simple end-to-end Convolutional neural network (CNN) that acts as a benchmark for the complex models on which we worked later. *Figure 3* shows the architecture of the baseline CNN. The model has five layers: a convolution layer with 16 filters and a 3x3 kernel, and a max pooling layer, a flatten layer that converts the output of the max pooling layer into 1D array, a dense layer with 256 units(neurons), and a dense output layer with 15 units (15 classes) and a SoftMax activation function. The output of each layer is also shown.

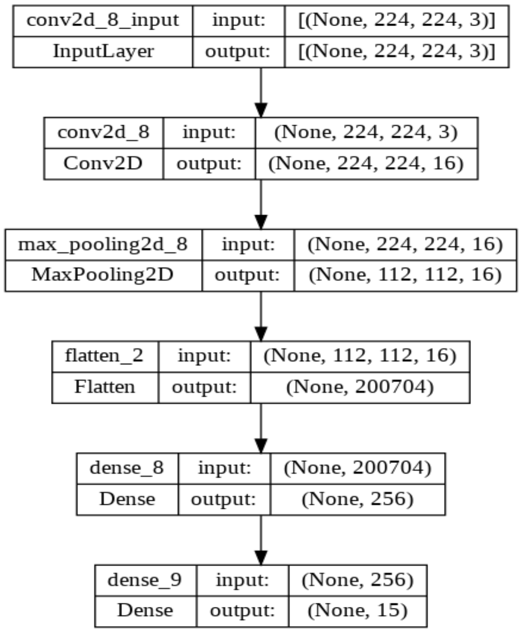

conv2d_8_input | input: | [(None, 224, 224, 3)]
InputLayer | output: | [(None, 224, 224, 3)]
conv2d_8 | input: | (None, 224, 224, 3)
Conv2D | output: | (None, 224, 224, 16)
max_pooling2d_8 | input: | (None, 224, 224, 16)
MaxPooling2D | output: | (None, 112, 112, 16)
flatten_2 | input: | (None, 112, 112, 16)
Flatten | output: | (None, 200704)
dense_8 | input: | (None, 200704)
Dense | output: | (None, 256)
dense_9 | input: | (None, 256)
Dense | output: | (None, 15)

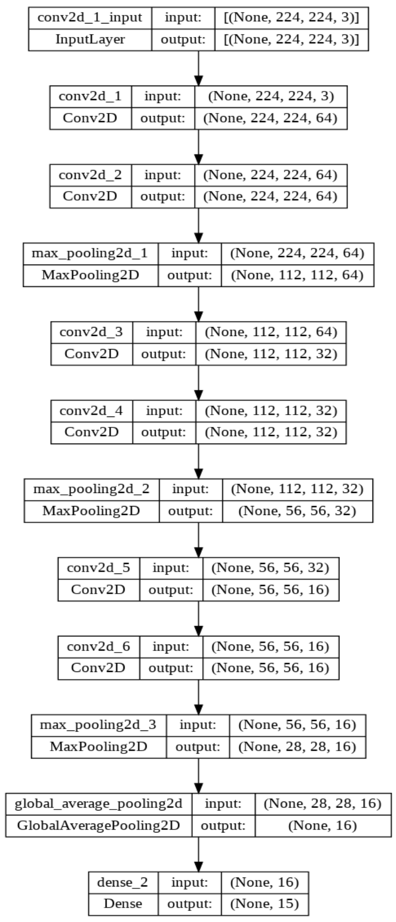
Figure 3: Baseline CNN architecture

CNN with increased layers

Then to improve our accuracy further, we tried Convolutional neural network (CNN) with increased convolutional and max pooling layers. Below is *Figure 4* which shows the architecture of the model. The model has 11 layers: 6 convolution layers, 3 max-pooling layers, 1 global average pooling layer, and 1 dense output layer. Each convolution layer has a specific number of filters (64, 32, and 16) and a 3x3 kernel. The max-pooling layers are used after every 2 convolution layers, which helps to down sample the input. The global average pooling layer then compresses the previous layers into a 1D array, which is then passed through the dense output layer with 15 units and a SoftMax activation function. Here to flatten the input, we used the global average pooling layer instead of a flattening layer because it applies average pooling across the entire input and helps us to overcome the overfitting problem.

CNN with Batch Normalization and Dropout layers

To the above-proposed architecture having 11 layers, we added 2 batch normalization layers which help to normalize the input layer by adjusting and scaling the activations, which in turn helps to reduce the training time and increase the performance of the model. We also added a Dropout Layer (Dropout rate = 0.25) after the global average pooling layer. This regularization technique (Dropout) works by randomly dropping out (setting to zero) a certain number of output features of a layer during training. This helped us to prevent overfitting the training data.


conv2d_1_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
conv2d_1 | Conv2D | input: (None, 224, 224, 3) | output: (None, 224, 224, 64)
conv2d_2 | Conv2D | input: (None, 224, 224, 64) | output: (None, 224, 224, 64)
max_pooling2d_1 | MaxPooling2D | input: (None, 224, 224, 64) | output: (None, 112, 112, 64)
conv2d_3 | Conv2D | input: (None, 112, 112, 64) | output: (None, 112, 112, 32)
conv2d_4 | Conv2D | input: (None, 112, 112, 32) | output: (None, 112, 112, 32)
max_pooling2d_2 | MaxPooling2D | input: (None, 112, 112, 32) | output: (None, 56, 56, 32)
conv2d_5 | Conv2D | input: (None, 56, 56, 32) | output: (None, 56, 56, 16)
conv2d_6 | Conv2D | input: (None, 56, 56, 16) | output: (None, 56, 56, 16)
max_pooling2d_3 | MaxPooling2D | input: (None, 56, 56, 16) | output: (None, 28, 28, 16)
global_average_pooling2d | GlobalAveragePooling2D | input: (None, 28, 28, 16) | output: (None, 16)
dense_2 | Dense | input: (None, 16) | output: (None, 15)
Figure 4: CNN with increased layers

Transfer learning

After experimenting with different CNN models, we moved on to the transfer learning method. As discussed earlier, Transfer learning is a technique in which a model trained on one task is reused as the starting point for a model on a second related task. This helps save a lot of training time because the weights of the model used as the starting point for the second task reach global minima much quicker.

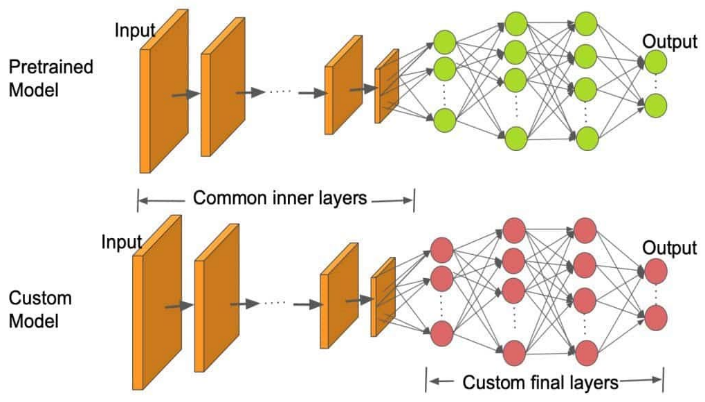
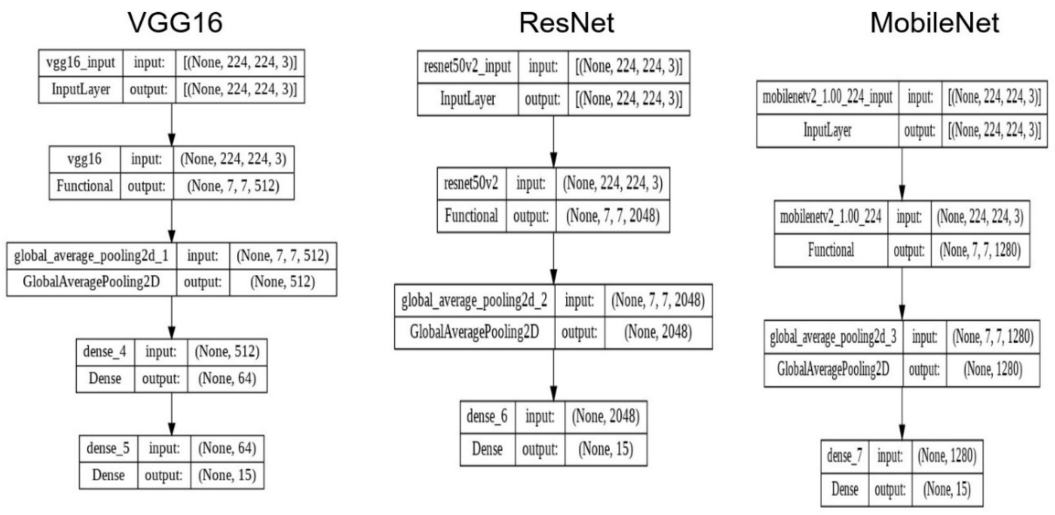


Figure 5: Transfer learning

Here in our case, the pre-trained models were already trained on the ImageNet [1] dataset, and these weights were passed onto the second task, i.e., custom layers, as shown in *figure 5.* The ImageNet dataset has almost 1000 classes, and a few of the classes may include vegetables. Hence, we expected the models trained using transfer learning to work well.

We started our experiments with VGG16 a popular state of the art deep learning model of the early 2010's. However, as the number of layers increases VGG16 is susceptible to vanishing gradients problem that makes the gradients harder to propagate till the initial input layers. To overcome this issue, we went with ResNet. It uses a residual learning framework, which allows the network to learn more effectively by adding the output of each layer to the input of the next layer, this makes the ResNet model quite expensive to train. Also, it is difficult to use ResNet for applications where computational efficiency is a concern, such as on mobile devices or in real time applications.

Since, we wanted to deploy the application on cloud and create an end-to-end real-time model that should be light weight for easy deployment but is also equally powerful we went with the MobileNet architecture. It uses depthwise separable convolutions to reduce the number of parameters and computations in the network, making it faster and more memory efficient.


VGG16
vgg16_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
vgg16 | Functional | input: (None, 224, 224, 3) | output: (None, 7, 7, 512)
global_average_pooling2d_1 | GlobalAveragePooling2D | input: (None, 7, 7, 512) | output: (None, 512)
dense_4 | Dense | input: (None, 512) | output: (None, 64)
dense_5 | Dense | input: (None, 64) | output: (None, 15)
ResNet
resnet50v2_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
resnet50v2 | Functional | input: (None, 224, 224, 3) | output: (None, 7, 7, 2048)
global_average_pooling2d_2 | GlobalAveragePooling2D | input: (None, 7, 7, 2048) | output: (None, 2048)
dense_6 | Dense | input: (None, 2048) | output: (None, 15)
MobileNet
mobilenetv2_1.00_224_input | InputLayer | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
mobilenetv2_1.00_224 | Functional | input: (None, 224, 224, 3) | output: (None, 7, 7, 1280)
global_average_pooling2d_3 | GlobalAveragePooling2D | input: (None, 7, 7, 1280) | output: (None, 1280)
dense_7 | Dense | input: (None, 1280) | output: (None, 15)
Figure 6: Architectures of different models

The dataset we used was taken from Kaggle [2]. It has a total of 21000 images from 15 classes, each class containing 1400 images. The dataset was split into three sections train (15000 images), test (3000 images) and validation (3000 images).

Results

Since the dataset we have chosen is balanced, accuracy was probably a good evaluation metric. So for each of the above-mentioned models, we calculated the training and test accuracy, which helped us know whether the model is prone to the problem of overfitting or not. Also, we calculated a custom accuracy, i.e., we created a dataset by taking a few pictures of real-life vegetables of a few classes and then tested the models on that dataset and calculated accuracy on that dataset. *Figure 7* clearly shows the accuracies of the different models and the number of parameters in each model.

Models	Train Accuracy	Test Accuracy	Trainable Parameters	Overfitting	Custom Accuracy
Baseline CNN	89.65	82.13	51,384,783	Yes	30.32
CNN with Increased Layers	95.30	90.23	73,631	Yes	35.69
CNN with Batch normalization and Dropout Layers	87.79	88.28	73,823	No	21.24
VGG16	96.80	97.10	33,807	No	63.64
ResNet	98.23	98.20	30,735	No	93.94
MobileNet	**98.37**	**98.90**	**19,215**	**No**	**87.88**

Figure 7: Scores for different models we created and trained

The Baseline CNN model with 5 layers gave us good accuracy on both the train and test data, but the model was overfitting. This was probably because of the larger number of trainable parameters (~51 million) as the model tries to learn more weights. To deal with this problem, we increased the number of convolution and max-pooling layers. Although the accuracy increased and the number of parameters reduced, the model was still overfitting. Then, after adding the Batch normalization and Dropout layers to the same architecture (model 2), we tried to prevent the model from overfitting, but the accuracy got significantly reduced. All three CNN models were not giving us good accuracy on the custom dataset due to the smaller size of the custom dataset.
Then we trained our dataset using the VGG16, ResNet, and MobileNet models through transfer learning. All three models were able to give us good accuracy on both the train and test datasets. Also, ResNet and MobileNet models performed better on the custom dataset. So, of all the models, we could say both ResNet and MobileNet outperformed the others. So, we decided to include plots related to ResNet and MobileNet models.

Figure 8 shows the loss and accuracy curves on train and test data for the ResNet Model. We can observe that as the number of epochs increases, the loss decreases while the accuracy increases. The model achieved a loss between 0.02 and 0.00 and an accuracy between 0.995 and 0.998. Likewise, figure *9* shows the loss and accuracy curves for the MobileNet model. The MobileNet model

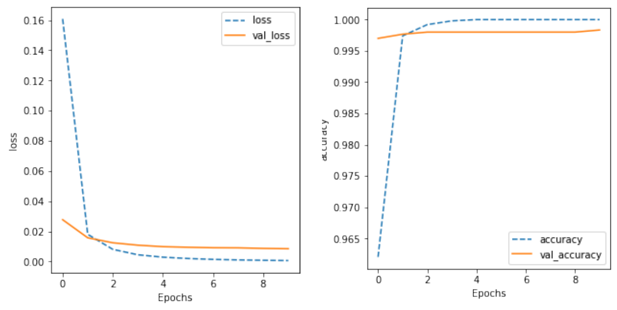


Figure 8: Loss and Accuracy vs Epochs for ResNet Model

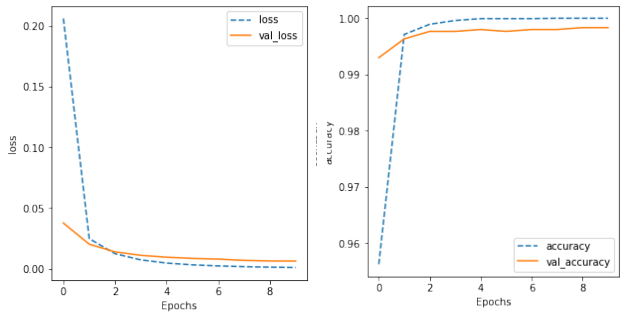
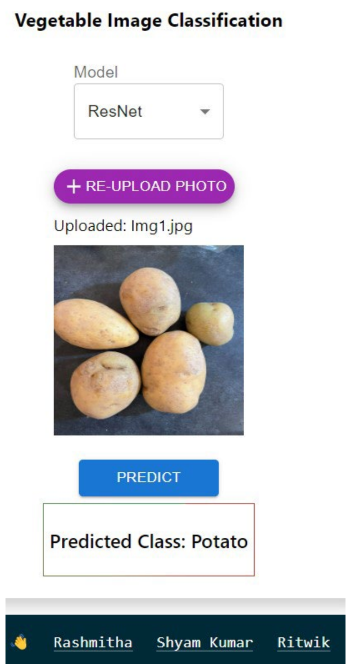


Figure 9: Loss and Accuracy vs Epochs for MobileNet Model

also achieved a loss between 0.04 and 0.01 and an accuracy between 0.99 and 1.00.

In Keras, the model.save() function[4] can be used to save a model's architecture, weights, and training configuration in a single file (H5 format). So, using this function and the training file obtained, we created an end-to-end application where we can upload an image to classify a vegetable in real-time using the models we trained. Below is a snapshot of the application created and how the ResNet model classifies the image as Potato.


Vegetable Image Classification

Model: ResNet

Uploaded: Img1.jpg

Predicted Class: Potato

Figure 10: Snapshot of the designed UI

Conclusion

Overall, after performing the experiments on various above-proposed CNN architectures, we could say that the models trained using transfer learning technique gave us better results than the traditional CNN models. This is because the dataset contains only 21000 images and the pre-trained CNN architectures perform better even on small datasets when compared to the baseline CNN models. Also, the pre-trained models performed relatively better than the CNN models on the custom dataset as well.

Key Findings

We observed that as the number of parameters in a network increases, the network becomes more complex and prone to overfitting, as seen in the case of a Baseline CNN model with nearly 50 million trainable parameters. This is because the model may accommodate noise in the data. Also, we expected the model to perform better after adding the Batch Normalization and Dropout layers (dropout rate = 0.25), but instead, the accuracy of the model decreased. This may be because random dropping out of activations may result in losing some information and can cause the model to underfit. So, tuning the dropout rate is important to find the right balance between overfitting and underfitting.

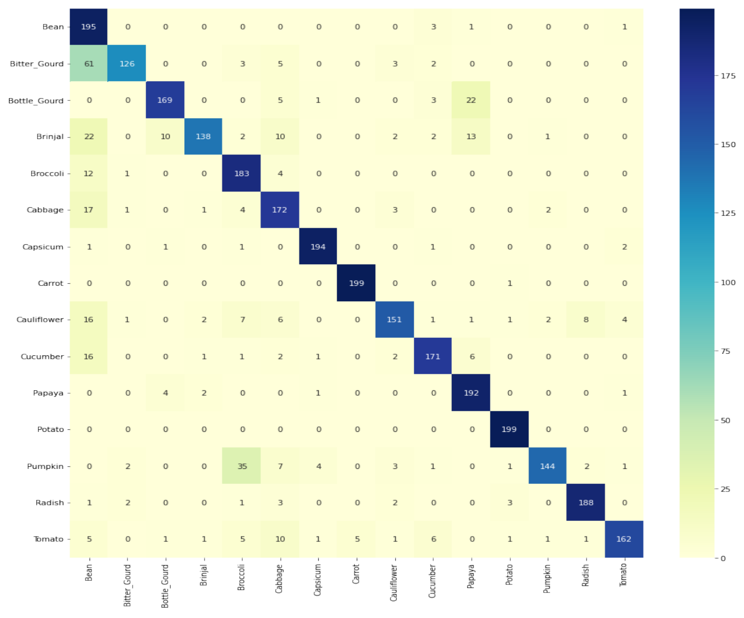


Figure 11: Confusion matrix of Baseline CNN model on test data

Of all the models, the lowest accuracy score on the test data was seen on the Baseline CNN. *Figure 11* shows us the confusion matrix of the Baseline CNN model on test data. We can see why the model didn't perform well on the test data. For example, the model classified most of the beans as bitter gourd because the model got confused as both the vegetables (*bean and bitter gourd*) are of the same color and size. Similarly, most pumpkins got classified as Broccoli. Hence, we can say that if there are multiple vegetables of the same color and shape, the models may classify the image incorrectly, specifically the traditional CNN models.

Limitations

The baseline CNN model was computationally expensive to train on our local machine. So, to reduce the training time, we used Google Colab (which has free access to GPU) to run the models and added more Convolution and max-pooling layers to reduce the number of trainable parameters and hence reduce the training time of the models. Also, if we test the model using a random image, let's say a human, the model classifies this image into one of the 15 classes, which shouldn't be the case. So, to overcome this problem, we can create a separate class as 'Not a Vegetable', and the model then should be able to classify the non-Vegetable images as 'Not a Vegetable'. Also, if we add a new class of vegetable to the dataset, we need to train the model on all the classes instead of the one class added.

Future Work

This project can be used to build a more complex model which can be used to classify 3D objects. That would require working with different sensors and exploring some IoT (Internet of Things) ideas. Images of rotten vegetables can also be added to the dataset, which can then be used to segregate such products from the fresh ones.

Team member contributions

Initial loading of the data, exploratory data analysis, data preprocessing, and building the first two CNN models was taken care of by Ritwik. Rashmitha was responsible for building the CNN model with batch normalization and dropout layers, VGG-16 and MobileNet transfer learning models. Shyam took care of making the ResNet model (the final model giving us the best accuracy), analyzing the results, and building the UI for real-time classification.

References

[1] https://www.image-net.org/
[2] https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset
[3] https://www.researchgate.net/publication/352846889_DCNN-Based_Vegetable_Image_Classification_Using_Transfer_Learning_A_Comparative_Study
[4] https://www.tensorflow.org/guide/keras/save_and_serialize
[5] https://www.atlantis-press.com/journals/ijcis/125941071
[6] https://www.researchgate.net/publication/352846889_DCNN-Based_Vegetable_Image_Classification_Using_Transfer_Learning_A_Comparative_Study
[7] https://arxiv.org/abs/1512.03385
[8] https://www.tensorflow.org/tutorials/images/classification