CZ4041 Machine Learning

Group 18 Report

Kaggle - Plant Seedlings Classification

Rank **13** out of **833** (Top **1.56%**)

Team Members

Name	Matric number	Contribution
ACHARYA ATUL	U1923502C	Xception, Vision Transformer, Weighted Average Ensemble, Image Augmentation, Test Time Augmentation
AGRAWAL RACHITA	U1922919L	Image Segmentation, KNN Classifier, K-Means Clustering, Convolutional Neural Networks
TRAN HIEN VAN	U1920891J	Deep Neural Network (EfficientNet), GANs, Binary Classifier Error Correction
TAYAL AKS	U1922309J	Exploratory Data Analysis, Inception-ResNet-v2

Submitted to— Sinno Jialin PAN
School of Computer Science & Engineering
Nanyang Technological University, Singapore

Introduction

Background

Can you differentiate a weed from a crop seedling?

The ability to do so effectively can mean better crop yields and better stewardship of the environment.

The Aarhus University Signal Processing group, in collaboration with University of Southern Denmark, has recently released a dataset containing images of approximately 960 unique plants belonging to 12 species at several growth stages [1].

Objectives

The aim of this project is to build a classifier that can identify plant species from an image.
List of 12 different species:

- Black-grass
- Charlock
- Cleavers
- Common Chickweed
- Common wheat
- Fat Hen
- Loose Silky-bent
- Maize
- Scentless Mayweed
- Shepherds Purse
- Small-flowered Cranesbill
- Sugar beet

Evaluation on Kaggle

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}$$

Figure 1: F1 score

Submissions are evaluated on Mean Score, which at Kaggle is a micro-average F1-score.

Exploratory Data Analysis (EDA)

Overview

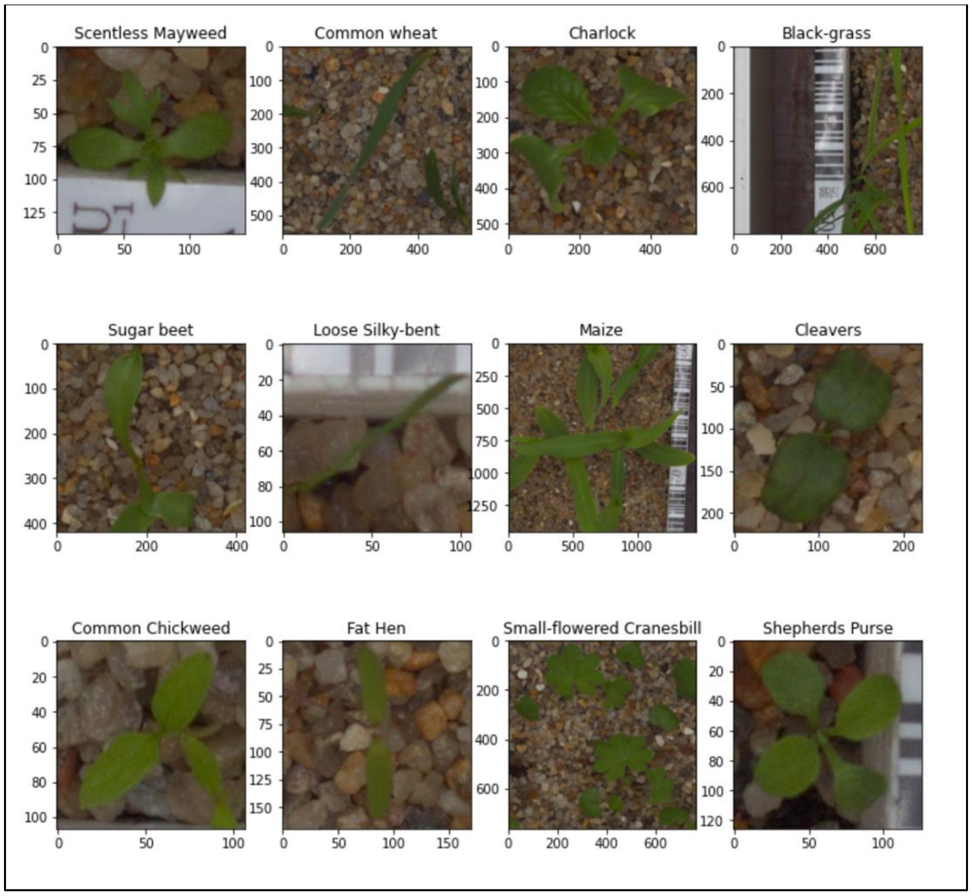


Figure 2: Sample plant images from 12 species

There are approximately 960 unique plants belonging to 12 species at several growth stages [1]. The dataset contains 4750 annotated RGB plant images as training set (80/20 for training/valid) and another 794 unlabeled RGB images as test set. All images have varied sizes even within the same class: 1135x1135, 1285x1285, and 154x154.

Data Distribution

The distribution of plant species in the training set is as follows:

Category	**# Images**
Black-grass	263
Charlock	390
Cleavers	287
Common Chickweed	611
Common wheat	221
Fat Hen	475
Loose Silky-bent	654
Maize	221

Scentless Mayweed	516
Shepherds Purse	231
Small-flowered Cranesbill	496
Sugar beet	385

Table 1: Number of images per class

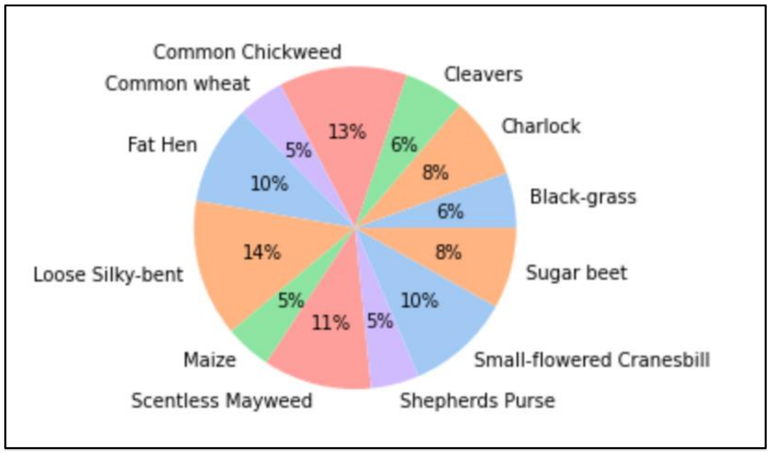


Figure 3: Pie-chart of classes in the dataset

After observing the distribution of data, there are some limitations for this dataset. Firstly, the dataset is imbalanced. For instance, there are 611 training samples (13% of the total training set) belonging to "Common Chickweed", while there are only 221 samples (5%) belonging to "Maize". Secondly, the number of training images is not large (4750 images) which may lead to overfitting the model.

PCA

PCA, or Principal Component Analysis (Jaadi, 2021) [2], is a dimensionality-reduction method. It is used to reduce the dimensionality of large data, by transforming it into a smaller one which still contains most of the key information in the large set. Here, we are analyzing our images which are of size 45 x 45 x 3, thus have 6075 dimensions. These are reduced to 2 dimensions since we are using 2 principal components. The way in which the first principal component is calculated is that it must have the largest variance of the dataset. The next principal component is calculated in a similar way, just that along with having the next high variance, it must now be uncorrelated to the first principal component.

Analysis: The graphs below show a scatter plot for each of the 12 categories, showing the distribution between the two PCA components. It can be noticed that by applying image segmentation (with sharpening) to the dataset, the PCA components have a higher variance. Thus, we will test our models with segmented images as well to see if we can boost classification.

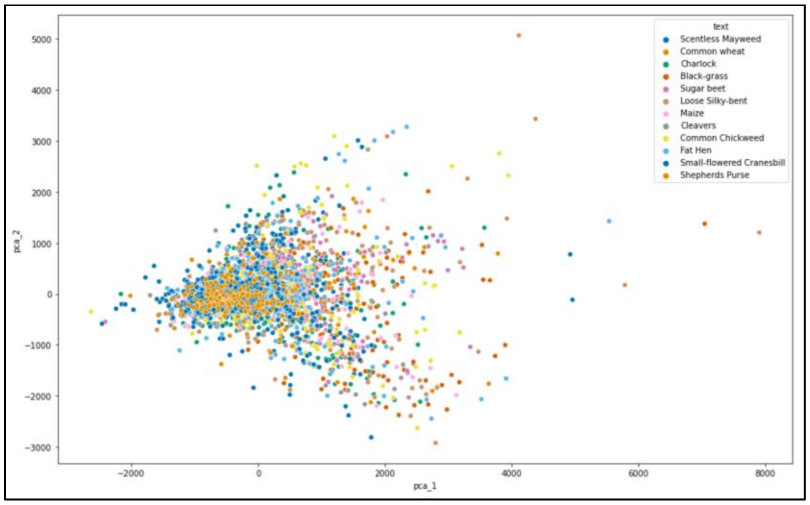


Figure 4: Distribution of first 2 PCA components on original images

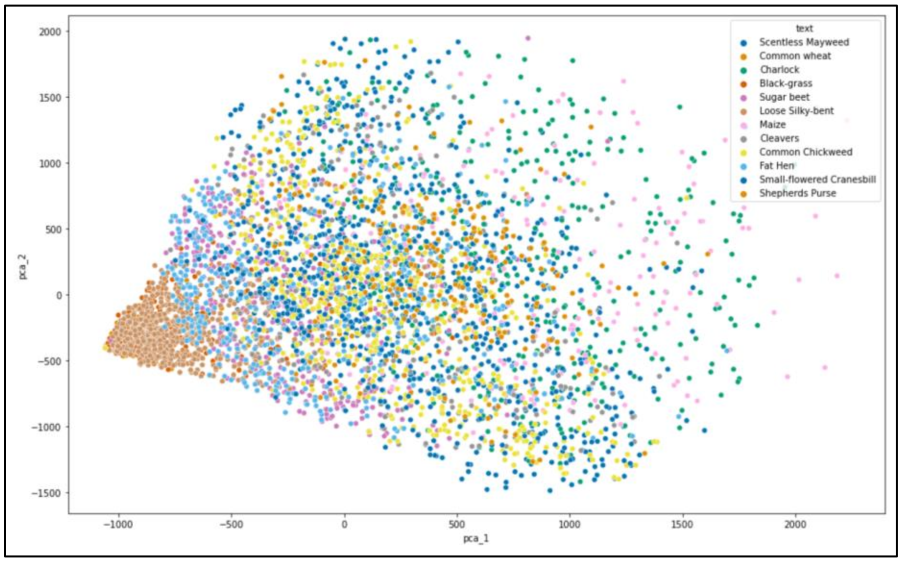


Figure 5: Distribution of first 2 PCA components for segmented images

Preprocessing

The preprocessing methods of the dataset included normalization, balancing 12 different classes, resizing and label encoding.

Resizing and Normalization

We applied these two steps to all our models:

Firstly, we normalize the pixel values to a [0,1] range. Having features on a similar scale could allow gradient descent to converge faster towards the minima for Gradient Descent Based Algorithms such as neural network. To scale data before implementing distance-based algorithms such as KNN and K-means makes all the features contribute equally to the result.

Secondly, all images are resized to 299x299 pixels as almost all machine learning models expect a fixed-size training dataset of images. For some models, we applied different sizing, which will be specifically indicated in the corresponding experiments. If there is no further information, the default image size will be 299x299.

Balancing dataset

As mentioned above, the number of data points are not large and unequally distributed among different classes. When it comes to domain-specific problems such as plant classification, the availability of data compared to other applications is significantly reduced for training deep learning models. We present two approaches to overcome this issue: Generative adversarial network (GAN) and Image Augmentation.

Generative Adversarial Networks

Generative models can be used to enhance small image datasets by producing artificial data samples that mimic properties of real data. GAN consists of two distinct models, a generator and a discriminator. The generator is responsible for producing 'fake' images that capture properties of real images. The job of the discriminator is to identify an image from the generator fake or real. During training, the generator becomes better at producing its fake images, while the discriminator becomes better at detection of real or fake images generated by the generator. The equilibrium of this game is when the discriminator can no longer give a correct classification for the image generated from the generator.

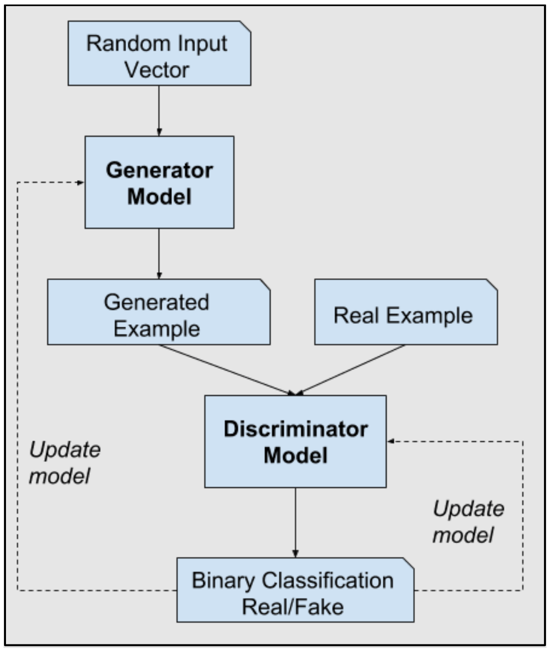


Figure 6: GAN architecture. [3]

In this project, we conducted experiments to generate new plant images with two GAN models: Deep Convolutional Generative Adversarial Network (DCGAN) and Conditional GAN (cGAN).

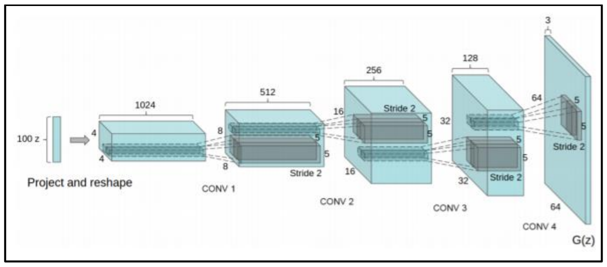


Figure 7: DCGAN's Generator Architecture [4]

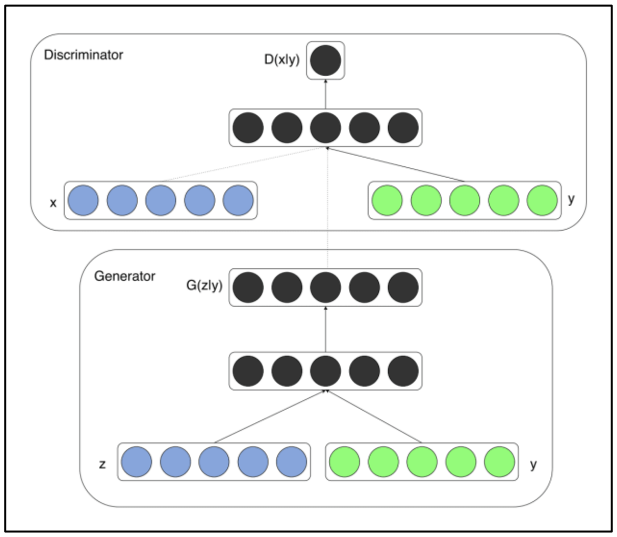


Figure 8: Example of a conditional generator and discriminator in a cGAN [5]

DCGAN explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It is a direct extension of GAN that could generate images from random classes, which would be a problem as we want to target some species with less data. We experimented with DCGAN to produce some fake images from random classes as shown in Figure 9.

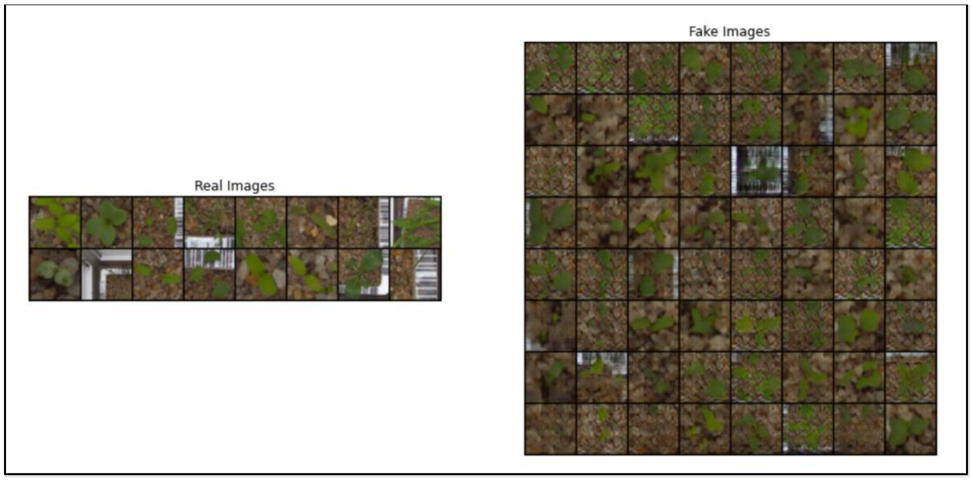


Figure 9: Example of real and fake images generated by DCGAN

Although, the generated images from DCGAN generally have a similar look to the real dataset, we aim at images generation of a given type, which DCGAN cannot be applied yet. Therefore, we trained a cGAN that makes use of class labels in addition to the image as input both to the generator and the discriminator models. The output results of this model, however, are not desirable as shown in the figure. The generated images from the trained cGAN mostly display white males and are unable to capture many other axes of variation in the real images.

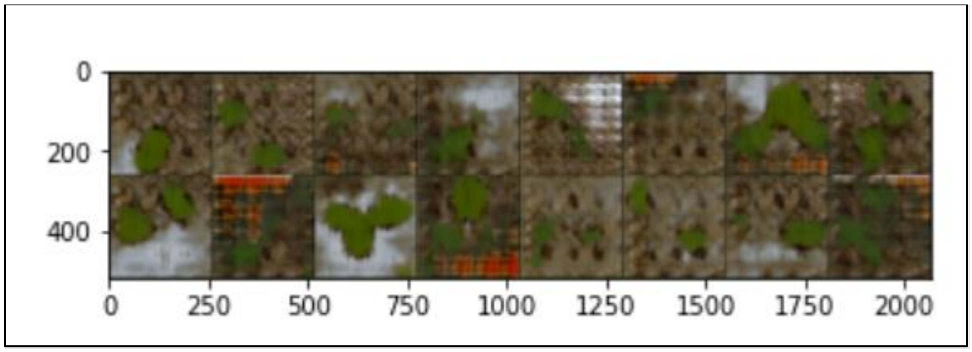


Figure 10: Fake Common wheat images generated by cGAN's generator with image size 64x64

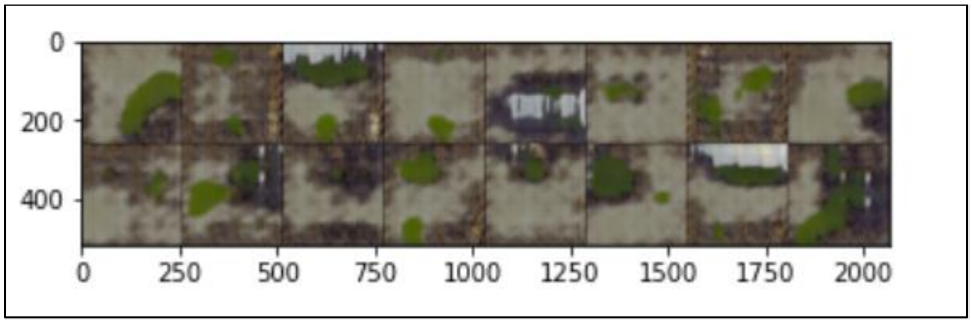


Figure 11: Fake Common wheat images generated by cGAN's generator with image size 128x128

The reason for these underfitting images is most likely due to the insufficient training instances for each class as well as a skew in the training data. The highest and lowest number of images per class is only 654 (Loose Silky-bent) and 221 (Common wheat). Therefore, for some classes with higher amount of data, the generated images are better than ones with lower number of images, but overall, the performance is not desirable. We could probably resolve this underfitting issue by making the generator deeper (i.e., increasing the number of deconvolutional kernels). However, our experiments are limited here due to computational restrictions. Therefore, we seek another option to overcome imbalanced dataset.

Image Augmentation

A popular technique for enhancing small datasets is image augmentation. It expands the training dataset by creating transformed versions of existing images. Such techniques can create variations of the images that can improve the performance and the ability of models to generalize.

These are steps that we used to augment all the images in preprocessing step:

1. Random perspective transformation of the given image with a given probability.
2. Random rotation

3. Random affine transformation of the image keeping center invariant.
4. Image flipping (vertical and horizontal)

Image Segmentation

Image Segmentation is used for partitioning images into different regions or segments under their respective class labels. By dividing the image into segments, we will be able to use the important segments of the image for training the model, I.e., highlighting these segments better for the model to pick up.

Methodology: For segmentation, we group the pixels of the image which have similar attributes. In this project, we choose to separate out the background pixels from the foreground. This means that the foreground pixels, I.e., the plant seedling itself, is highlighted. Thus, we remove the noisy background for the images. This is done by using the HSV (Hue-Saturation-Value) to create masks for plants to partition them. We also sharpened the segmentation mask to get better quality of images.

*Rationale for segmentation: T*his technique helps us to highlight to the models the 'most important part' of the image, thus hoping that that model can learn and classify the images better. We will be conducting experiments of the models, which we will talk about next, using the original as well as the Segmented dataset. This will help us draw conclusions as to what is the best technique to reach a high accuracy for this dataset.

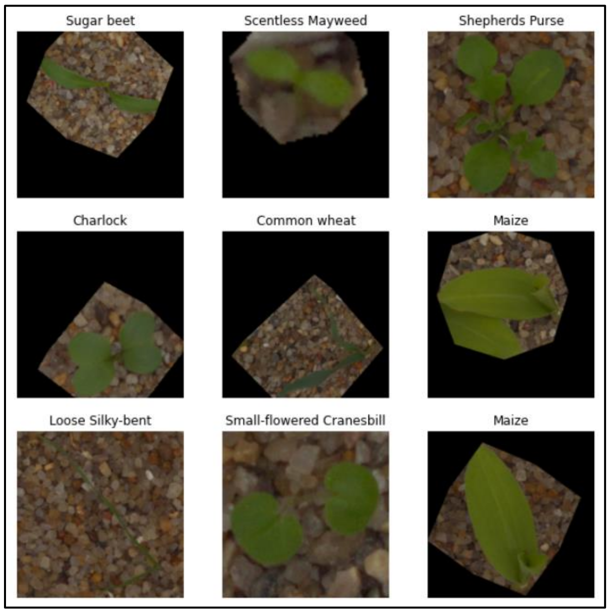


Figure 12: Example of augmented images

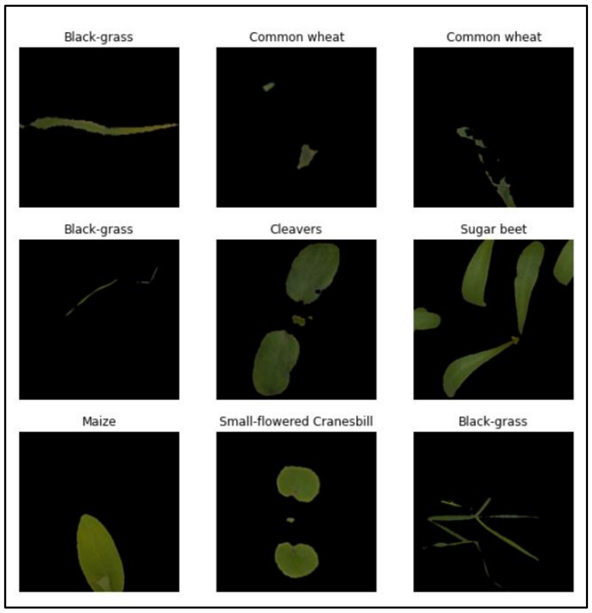


Figure 13: Example of segmented images

For future reference, we defined 4 types of datasets that we preprocessed for training as follows:

- Original dataset – the dataset provided in Kaggle
- Segmented dataset – the original dataset with segmentation
- Balanced dataset – the original dataset with augmentation to balance the number of images per class. The Balanced segmented dataset has a total of 12000 images, 1000 per class.

- Balanced segmented dataset – the original dataset with both augmentation and segmentation. After performing image augmentation and segmentation, our Balanced segmented dataset has a total of 12000 images, 1000 per class with an image size of 299x299.

Models

K-Means Clustering

Overview

K-means is an unsupervised learning algorithm for clustering data. Clustering is simply partitioning the entire data into "k" different clusters in which the data holds certain similar properties. In K-Means, each cluster is associated with a centroid. K-Means algorithm aims to minimize the sum of distances between the points and their respective cluster centroids.

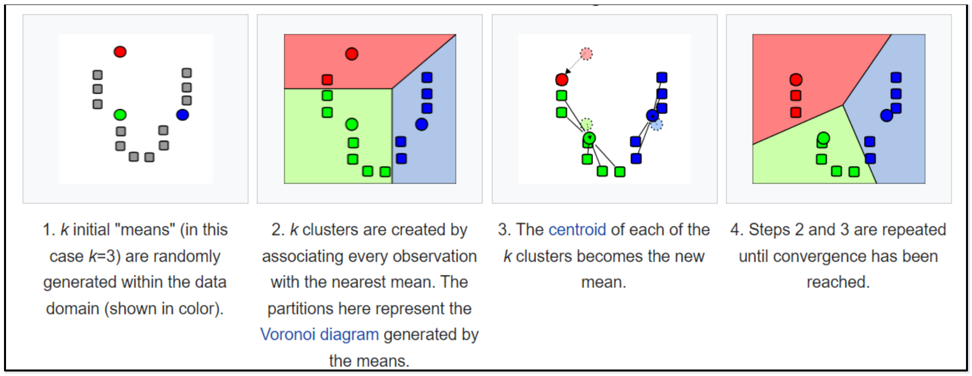

1. k initial "means" (in this case k=3) are randomly generated within the data domain (shown in color).
2. k clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.
3. The centroid of each of the k clusters becomes the new mean.
4. Steps 2 and 3 are repeated until convergence has been reached.

Figure 14: Demonstration of K-mean algorithm [6]

For image classification, K-mean only takes the images as input without labels and then groups them into clusters based on data patterns. Firstly, the algorithm will initialize the number of clusters "k". It will then randomly select the centroid for each cluster within the dataset. Every image is then assigned to each of the clusters by reducing the in-cluster sum of squares. The centroid is recomputed for each newly formed cluster. The process will continue until the centroids of newly grouped clusters are not changing, or each cluster remains the same data over iteration.

Experiments

For this experiment, a K-means model for k = 12 was chosen since we have 12 classes for which we need to classify. We experimented with both non-segmented and segmented images of both the balanced dataset as well as the unbalanced one. We can observe that the accuracy score for the Original Dataset is quite low, 16.84%, almost as if the model is doing guess work. But we can notice that when we use the Segmented Dataset, the accuracy improves considerably for this model (up to 30.99%). It can also be noted that for Balanced Dataset Segmented, the accuracy only increased by a small margin (12.83%). This may signify that segmentation of images only helps the model to classify only a few classes better, which have a higher number of images in the original dataset. So, the effect of which is not very visible in the Balanced Dataset.

Further, since running this algorithm with many clusters is quite memory expensive, we chose to run it for image sizes of 45 x 45 x 3. This was a suitable size for running it on the Kaggle platform.

Results

The following table summarizes the performance of our K-Means model when trained on different datasets. We chose not to submit the results due to the low training accuracies.

Dataset	Train Accuracy
Original Dataset	0.1684
Segmented Dataset	0.3099
Balanced Dataset	0.1296
Balanced Dataset Segmented	0.1983

Table 2: Accuracy of K-Means Clustering on different datasets

K-Nearest Neighbours (KNN)

Overview

K-nearest Neighbors is a supervised learning algorithm that can be used for both classification and regression tasks. The algorithm assumes that similar training instances exist in close proximity.

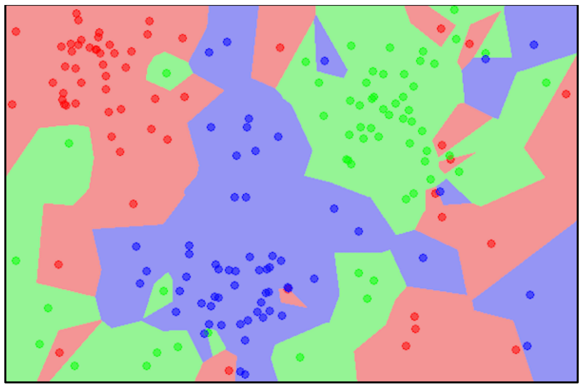


Figure 15: Demonstration of how similar data points typically exist near each other [7]

We choose "k", i.e., the number of neighbors. For each image instance, the algorithm computes the Euclidian distance between test data with training images and added in a sorted collection in ascending order. After the iteration, for each entry, the first "k" entries are selected, and the most frequent labels among their labels are returned.

Experiments

Since our classification problem is multi-class, we used the OneVsRestClassifier combined with the KNeighboursClassifier. OneVsRestClassifier splits a multi-class classification into one binary classification problem per class. In our case since we have 12 classes, we would have 12 such binary classifications, where we will be classifying one plant seedling class with the rest of the classes combined as one. This could cause an issue with larger datasets (in our case the Balanced Dataset) since it can get quite memory intensive. Therefore, we chose to run out model for image sizes of 45 x 45 x 3, which was a suitable size for running it on the Kaggle platform without running into an out of memory error.

We can see that our model consistently performs better for the segmented data in both the original as well as in the balanced dataset. Thus, unlike in the previous case where the K-Means algorithms only improved for a certain class after segmentation, here the KNN algorithm helped in learning the classes better overall.

Results

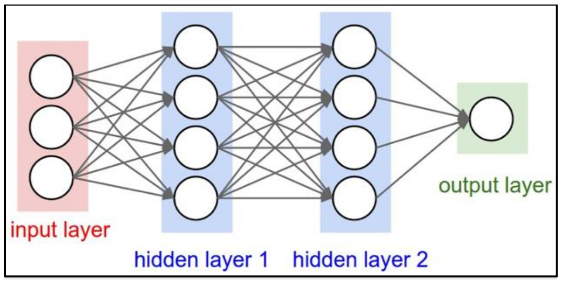
The following table summarizes the performance of our KNN model when trained on different datasets:

Dataset	Public score	Private Score
Original Dataset	0.22670	0.22670
Segmented Dataset	0.51196	0.51196
Balanced Dataset	0.22670	0.22670
Balanced Dataset Segmented	0.49937	0.49937

Table 3: Scores of KNN Classifier on different datasets

Convolutional Neural Network (CNN)

Overview



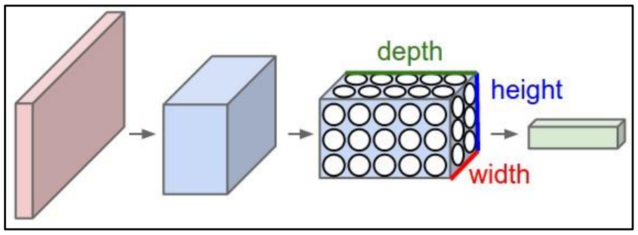


Figure 16: Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. [8]

Convolutional Neural Network (CNN) is a type of deep learning neural network which is a popular method applied in many computer vision applications. When it comes to image data, regular neural networks don't scale well to full images, especially ones with large size. Also, it may not be able to capture a lot of spatial information from images. Therefore, we applied CNN in this task as our input data is images.

There are three main types of blocks to build CNN architectures: Convolutional Layer, Pooling Layer, and Fully Connected Layer. It is designed to learn spatial patterns through a backpropagation algorithm.

Experiments

In this experiment, we tried to train a custom CNN model from scratch. We have conducted several experiments with how the architecture is supposed to be like, what increases accuracy and what works for our model. We will briefly be talking about the final architecture and the rationale behind the various components which were chosen carefully after a step-by-step method of adding each component one by one and observing the accuracy.

<u>Data preparation:</u> Due to the large size of our model, we did not have the resources to run the model for image size 299x299. Thus, for this experiment we have tried out image size up to only 128x128, followed by a series of image augmentations such as image rotation, flipping, etc. This helps to improve

the performance of the model since the training data basically has some new and different examples to train on. The train validation split used for this model follows an 80:20 ratio.

The batch size was experimented with and the optimal size of 16 was chosen. We also considered that since our dataset has 12 classes to classify into, our batch size should not be smaller than 12 to ensure that at each step our model has enough data to learn from. The number of epochs was set to a large value of around 175, and we applied callbacks to monitor the validation loss. The patience for this callback was set to 10 epochs. So, as soon as the validation accuracy would stop increasing for more than 10 epochs, the training would stop.

<u>*Model summary:*</u> A basic block of our model includes (calling it a CNN block):

- Convolutional layer
- Batch Normalization
- Leaky Relu

We have two such blocks after which we have a Max pooling layer, to extract the sharpest features for the image data. This structure (CNN block, CNN block, Max pooling) is repeated twice more, after which we flatten the outputs.

This is followed by 2 sets of:

- Dropout
- Dense
- Batch normalization
- Activation

The detailed model architecture diagram for the CNN model is shown later in this section.

<u>*Rationale:*</u> The number of filters for the convolutional blocks gradually increased by a factor of 2, starting from 64 in one CNN block, CNN block, Max pooling set, reaching a maximum of 256. The kernel size was fixed to 3x3 so that we would not lose the meaning of the image pixels as the model progressed.

Batch normalization is a technique for training very deep neural networks which standardizes the inputs to a layer for each mini batch. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

We are using Leaky Relu here as it substitutes zero values with some small value say 0.0001. Thus, the gradient descent will have a non-zero value, so it will be able to learn without reaching an end. Hence, leaky ReLU proves to be better than just a simple ReLU function.

Dropout is a regularization technique for neural network models where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. The effect is that the network becomes less sensitive to the specific weights of neurons. This in turn results in a network that is capable of better generalization and is less likely to overfit the training data. Here, we used a small value for dropouts, I.e., 0.1.

The last set of Dense layer has 12 outputs, which is set according to the number of classes in which we want to classify into, thus our output dimension is changed to 12.

We use the Softmax activation function for this last Dense layer since it helps to predict a multinomial probability distribution, which is necessary for a multi-class classification problem.

Some techniques that helped in boosting the accuracy of this model include:

- Optimizers: Instead of just tuning the learning rate for the optimizer (the chosen optimizer was Adam as it proved to be better by experiments), we chose to tune the beta 1, beta 2 and epsilon values as well to control the momentum of the optimizer.
- Callbacks: We implemented ReduceLROnPlateau and monitored the validation accuracy. As the name suggests, it helps in reducing the learning rate when a metric, in this case validation accuracy, has stopped improving. It reduces by the value of Epsilon that is set for this metric.

Detailed model architecture:

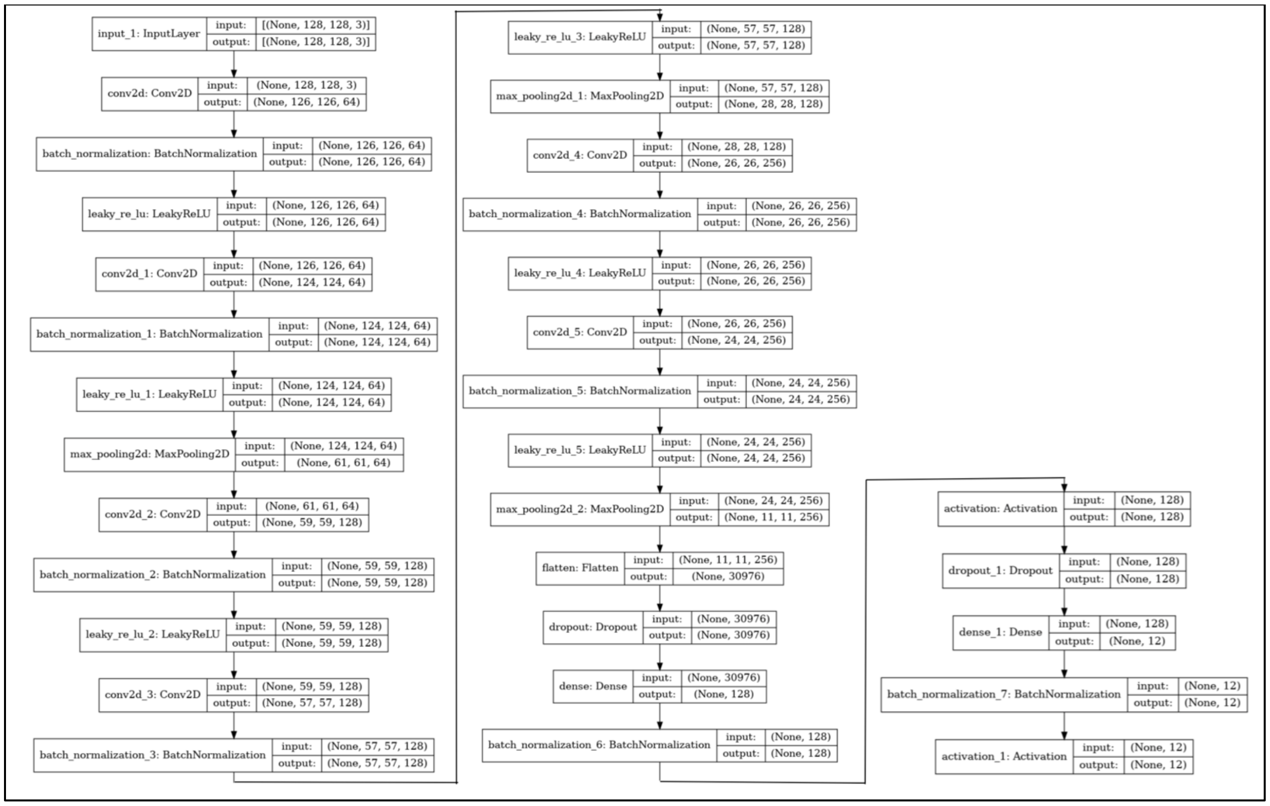

input_1: InputLayer | input: [(None, 128, 128, 3)] | output: [(None, 128, 128, 3)]
conv2d: Conv2D | input: (None, 128, 128, 3) | output: (None, 126, 126, 64)
batch_normalization: BatchNormalization | input: (None, 126, 126, 64) | output: (None, 126, 126, 64)
leaky_re_lu: LeakyReLU | input: (None, 126, 126, 64) | output: (None, 126, 126, 64)
conv2d_1: Conv2D | input: (None, 126, 126, 64) | output: (None, 124, 124, 64)
batch_normalization_1: BatchNormalization | input: (None, 124, 124, 64) | output: (None, 124, 124, 64)
leaky_re_lu_1: LeakyReLU | input: (None, 124, 124, 64) | output: (None, 124, 124, 64)
max_pooling2d: MaxPooling2D | input: (None, 124, 124, 64) | output: (None, 61, 61, 64)
conv2d_2: Conv2D | input: (None, 61, 61, 64) | output: (None, 59, 59, 128)
batch_normalization_2: BatchNormalization | input: (None, 59, 59, 128) | output: (None, 59, 59, 128)
leaky_re_lu_2: LeakyReLU | input: (None, 59, 59, 128) | output: (None, 59, 59, 128)
conv2d_3: Conv2D | input: (None, 59, 59, 128) | output: (None, 57, 57, 128)
batch_normalization_3: BatchNormalization | input: (None, 57, 57, 128) | output: (None, 57, 57, 128)
leaky_re_lu_3: LeakyReLU | input: (None, 57, 57, 128) | output: (None, 57, 57, 128)
max_pooling2d_1: MaxPooling2D | input: (None, 57, 57, 128) | output: (None, 28, 28, 128)
conv2d_4: Conv2D | input: (None, 28, 28, 128) | output: (None, 26, 26, 256)
batch_normalization_4: BatchNormalization | input: (None, 26, 26, 256) | output: (None, 26, 26, 256)
leaky_re_lu_4: LeakyReLU | input: (None, 26, 26, 256) | output: (None, 26, 26, 256)
conv2d_5: Conv2D | input: (None, 26, 26, 256) | output: (None, 24, 24, 256)
batch_normalization_5: BatchNormalization | input: (None, 24, 24, 256) | output: (None, 24, 24, 256)
leaky_re_lu_5: LeakyReLU | input: (None, 24, 24, 256) | output: (None, 24, 24, 256)
max_pooling2d_2: MaxPooling2D | input: (None, 24, 24, 256) | output: (None, 11, 11, 256)
flatten: Flatten | input: (None, 11, 11, 256) | output: (None, 30976)
dropout: Dropout | input: (None, 30976) | output: (None, 30976)
dense: Dense | input: (None, 30976) | output: (None, 128)
batch_normalization_6: BatchNormalization | input: (None, 128) | output: (None, 128)
activation: Activation | input: (None, 128) | output: (None, 128)
dropout_1: Dropout | input: (None, 128) | output: (None, 128)
dense_1: Dense | input: (None, 128) | output: (None, 12)
batch_normalization_7: BatchNormalization | input: (None, 12) | output: (None, 12)
activation_1: Activation | input: (None, 12) | output: (None, 12)

Figure 17: Convolutional Neural Network Architecture

Accuracy and Loss graphs:

(This is for the best model, which is the one trained on the Balanced Dataset.)

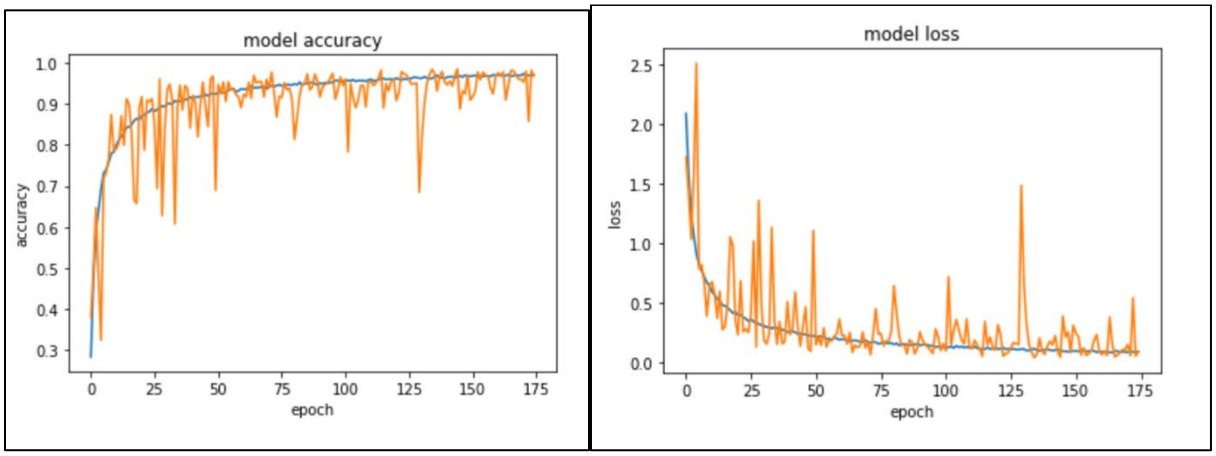
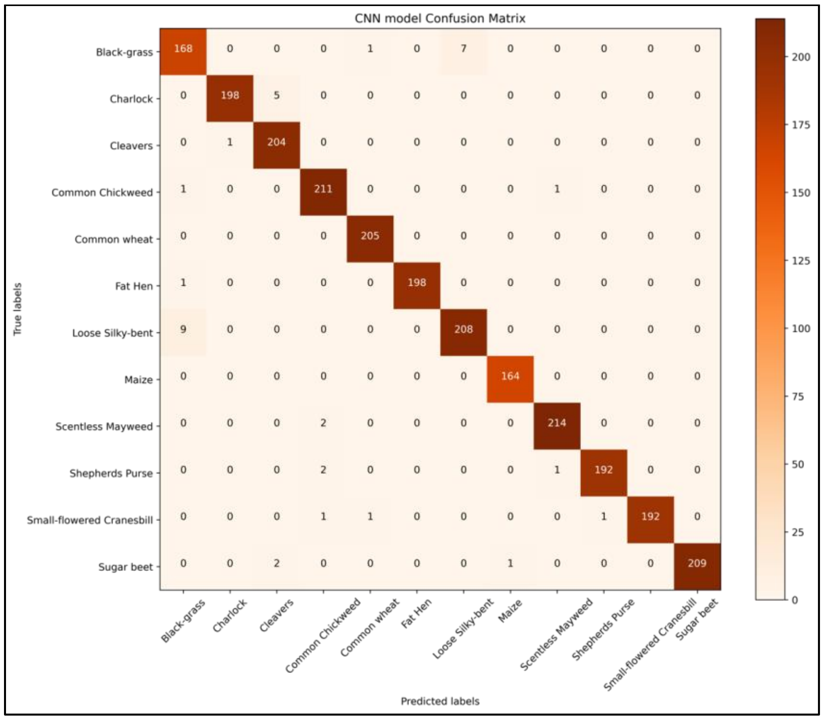


Figure 18: Left: Accuracy of CNN on train and validation data. Right: Loss on train and validation data

Analysis of the Accuracy and Loss graphs: It can be observed that overfitting is very minimal since the train and validation loss and accuracy move together. Though, it can be observed that there are sudden spikes in these graphs, the cause of which can be the learning rate, which may be a little too high for our model. Though, precautions were taken to reduce the learning rate and thus increasing the number of epochs for the model to be able to learn better. The effect of such spikes was much larger before we implemented the ReduceLROnPlateau function.

Confusion matrix:

(For the same model as described above)



	Black-grass	Charlock	Cleavers	Common Chickweed	Common wheat	Fat Hen	Loose Silky-bent	Maize	Scentless Mayweed	Shepherds Purse	Small-flowered Cranesbill	Sugar beet
Black-grass	168	0	0	0	1	0	7	0	0	0	0	0
Charlock	0	198	5	0	0	0	0	0	0	0	0	0
Cleavers	0	1	204	0	0	0	0	0	0	0	0	0
Common Chickweed	1	0	0	211	0	0	0	0	1	0	0	0
Common wheat	0	0	0	0	205	0	0	0	0	0	0	0
Fat Hen	1	0	0	0	0	198	0	0	0	0	0	0
Loose Silky-bent	9	0	0	0	0	0	208	0	0	0	0	0
Maize	0	0	0	0	0	0	0	164	0	0	0	0
Scentless Mayweed	0	0	0	2	0	0	0	0	214	0	0	0
Shepherds Purse	0	0	0	2	0	0	0	0	1	192	0	0
Small-flowered Cranesbill	0	0	0	1	1	0	0	0	0	1	192	0
Sugar beet	0	0	2	0	0	0	0	1	0	0	0	209

Figure 19: Confusion matrix on validation sample of Balanced Dataset

Analysis of the Confusion Matrix: On observing the confusion matrix, an obvious misclassification error that can be observed is that there are two classes- Black Grass and Loose Silky-bent being misclassified as each other. We will later try to solve this issue by training a separate binary classifier for these two classes and then perform multi-class classification for the entire dataset.

Results

The following table summarizes the performance of our CNN model when trained on different datasets:

Dataset	**Public score**	**Private Score**
Original Dataset	0.96473	0.96473
Segmented Dataset	0.88790	0.88790
Balanced Dataset	**0.96725**	**0.96725**
Balanced Dataset Segmented	0.89420	0.89420

Table 4: Scores of Convolutional Neural Network on different datasets

Analysis of the overall results: It can be noticed that the CNN model does not perform well on the Segmented Image data, possibly because of the loss of background information which is caused by the segmentation. This shows that the CNN model learns better with all the pixel information rather than just the information of the plant seedling. Further, amongst the scores for the original and balanced dataset, we can notice that the Balanced Dataset further helps in improving the performance of the model.

Transfer Learning

Transfer Learning is a machine learning method where we reuse a pre-trained model as the starting point for a model on a new task. A model trained on one task is repurposed on a second, related task as an optimization that allows rapid progress when modeling the second task. The intuition behind transfer learning for image classification is that if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world. It allows us to take advantage of these learned feature maps without having to start from scratch by training a large model on a large dataset. By applying transfer learning to a new task, one can achieve significantly higher performance than training with only a small amount of data. In this project, we carried out experiments with 3 different pre-trained models: EfficientNet, Xception, and Inception-Resnet.

The pretrained model can be used in 3 ways:

1. Freeze all layers of the pretrained model and train the classification head
2. Unfreeze some or all the layers and train them along with the classification head
3. Use the pretrained model as a feature extractor for other Machine Learning Algorithms (SVM, RandomForest, XGBoost, MLPClassifier, etc)

We use the 2nd technique and unfreeze all the layers of the pretrained model. This increases the training time since more layers are trainable. However, we achieve better results when unfreezing these layers because they learn better features for the downstream task.

For better results, we also implemented some layers following the original model architectures: Batch Normalization and Dropout. The benefits of such techniques are discussed above in the CNN section.

To tune hyperparameters such as learning rate, we added some callback functions to improve the performance. Firstly, the EarlyStopping callback was used to stop the training once the model stops improving. We also use the ReduceLROnPlateau callback to reduce the learning rate if the loss increases over a period. This technique helps the model converge at the local minimum as smaller learning rates will prevent large updates to the weights as the weights approach their local minimum.

We trained all the pre-trained models with inputs as images from our 4 datasets mentioned above: original, segmented, balanced, and balanced segmented, respectively to observe the effects of each dataset on the results. The validation size is 10% of the dataset size.

EfficientNet

Overview

EfficientNet was introduced in Tan (2019) [9] which reaches State-of-the-Art accuracy on many image classification transfer learning tasks. It is a CNN-based model scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient.

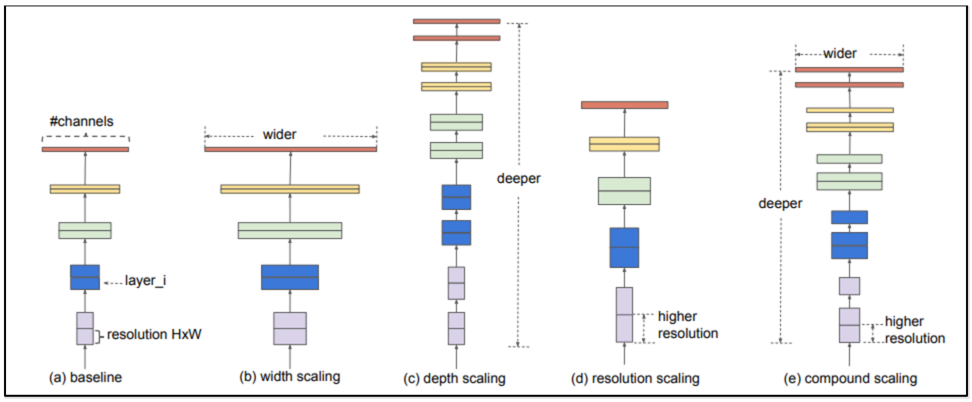


Figure 20: Model Scaling. (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network [9]

Differently from traditional methods that scale different dimensions of networks: resolution, depth, and width, EfficientNet scales each dimension with a fixed set of scaling coefficients uniformly. As mentioned in the original model, the choice of such dimensions is also limited by many factors. Therefore, instead of allowing arbitrary choices of these parameters, there are only 8 variants of EfficientNet models (B0 to B7) with manually chosen parameters that are proven to achieve good outputs.

Experiment

In this experiment, we made use of the pretrained EfficientNetB3 model with architecture as shown in figure 20. From the EfficientNet family, we selected this variant because it provides a good compromise between accuracy and computational resources. We use Global Average Pooling instead of a Flatten Layer since it spatially averages each feature map and ensures that each feature map can be considered as a confidence map. This is followed by a Dense layer with 256 neurons and the ReLU activation function.

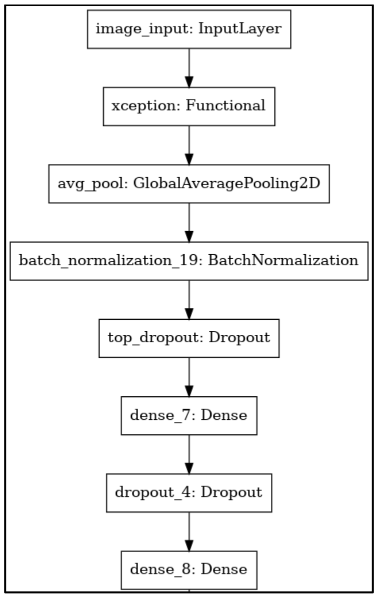


Figure 21: EfficientNetB3 Model Architecture

We trained the model on 4 datasets (original, segmented, balanced and balanced segmented), each is training with 40 epochs and a batch size 16. The Loss and Accuracy Plot of the model trained on the …. dataset (best) is shown below.

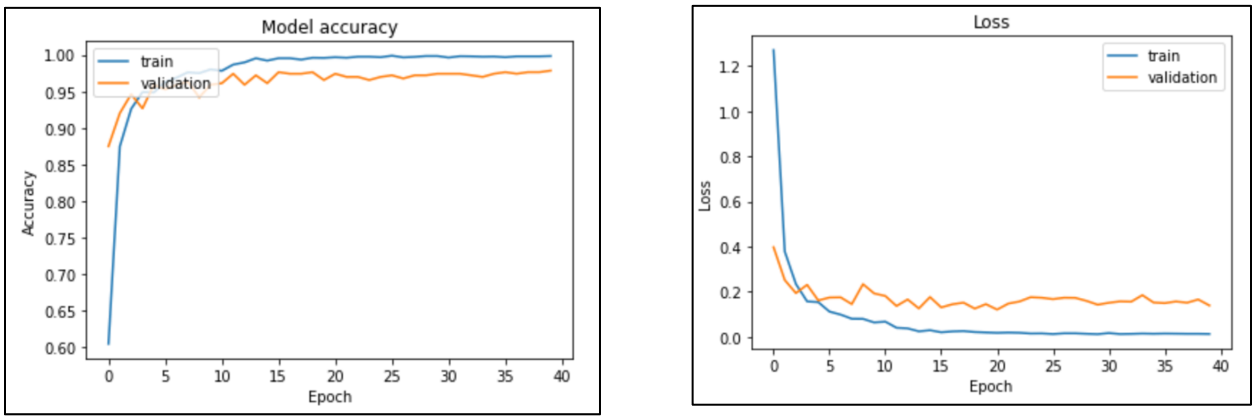


Figure 22: Left: Accuracy of EfficientNetB3 on train and validation data. Right: Loss on train and validation data

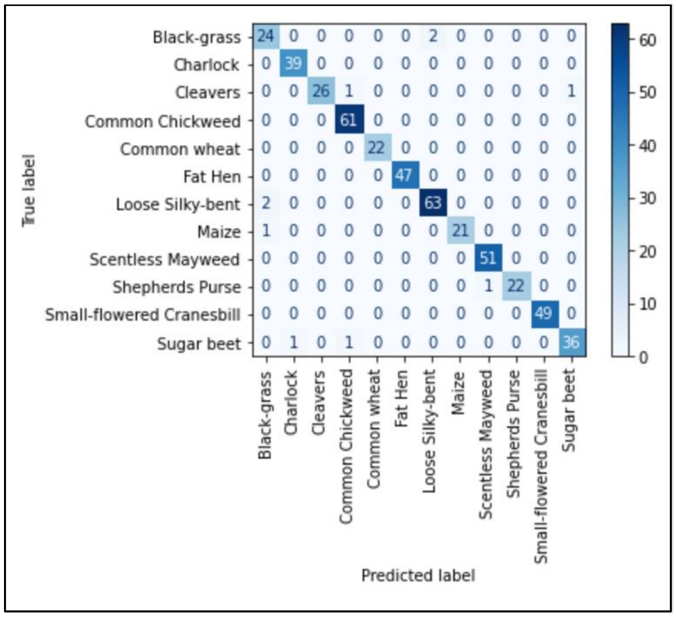


Figure 23: Confusion matrix on validation sample of Original Dataset

Results

The following table summarizes the performance of EfficientNetB3 when trained on different datasets.

Dataset	Public score	Private Score
Original Dataset	**0.98236**	**0.98236**
Segmented Dataset	0.96725	0.96725
Balanced Dataset	0.98110	0.98110
Balanced Dataset Segmented	0.97858	0.97858

Table 5: Scores of EfficientNetB3 on different datasets

The EfficientNetB3 model fine-tuned on the Original Dataset gives the highest score in the competition.

Xception

Overview

Xception is a popular deep Convolutional Neural Network (CNN) developed by Google. The Xception model has been shown to outperform VGG-16, ResNet-152, as well as Inception V3 on the ImageNet dataset (consisting of around 14 million images) [10]. Xception is an extension of the Inception architecture which replaces the standard Inception modules with depth-wise separable convolutions. In traditional CNNs, convolutional layers learn correlations across both space (across a channel) as well as depth (between the multiple channels e.g., RGB). The Xception model challenges the concept that we need to consider both at the same time. The Inception model introduced a slight separation between the two. It used 1x1 convolutions to project the original image into separate smaller input spaces. Then, a different type of filter was used to transform those input spaces into 3D blocks of data. Xception improves upon this method. Instead of separating the data into several chunks, it transforms the spatial correlation for each output channel separately. It then does a 1x1 depth-wise convolution to capture cross-channel correlation. This new technique is known as depth-wise separable convolution which consists of a depth-wise convolution followed by a point-wise convolution. The result is that the

Xception model slightly outperforms VGG-16, ResNet-152, and InceptionV3. It also has the same number of parameters as Inception V3 which means it is computationally more efficient.

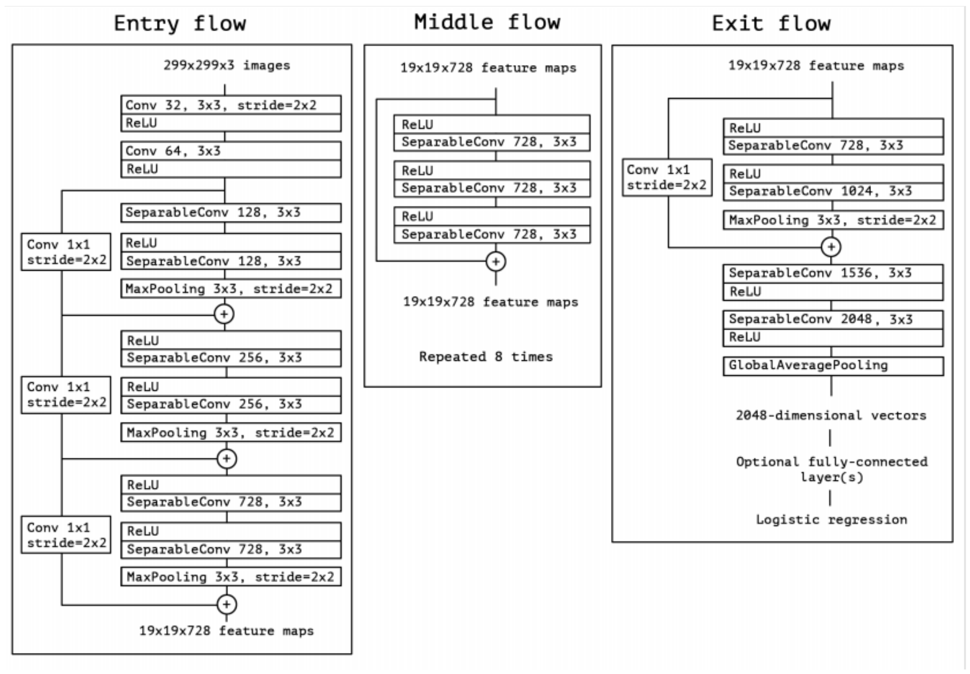


Figure 24: Xception Architecture [10]

Experiments

In this experiment we use a pretrained Xception model which has been trained on more than a million images from the ImageNet database. We unfreeze all the layers and add our custom layers on top of it This is followed by a Dense layer with 256 neurons and the SELU activation function. Scaled Exponential Linear Units, or SELUs, are activation functions that induce self-normalizing properties. The SELU activation function is given by $f(x) = \lambda x \; if \; x \geq 0$ and $f(x) = \lambda\alpha(e^x - 1) \; if \; x < 0$ with $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

This is followed by a Dropout layer with probability 0.2 and another Dense layer with 128 neurons and the SELU activation function. The final classification layer contains 12 neurons corresponding to the 12 classes along with the softmax activation function.

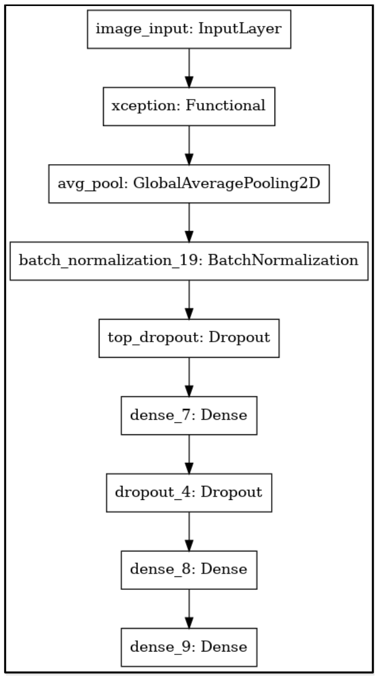


Figure 25: Xception Model Architecture

We trained the model on 4 datasets (original, segmented, balanced and balanced segmented). The Loss and Accuracy Plot of the model trained on the balanced segmented dataset is shown below.

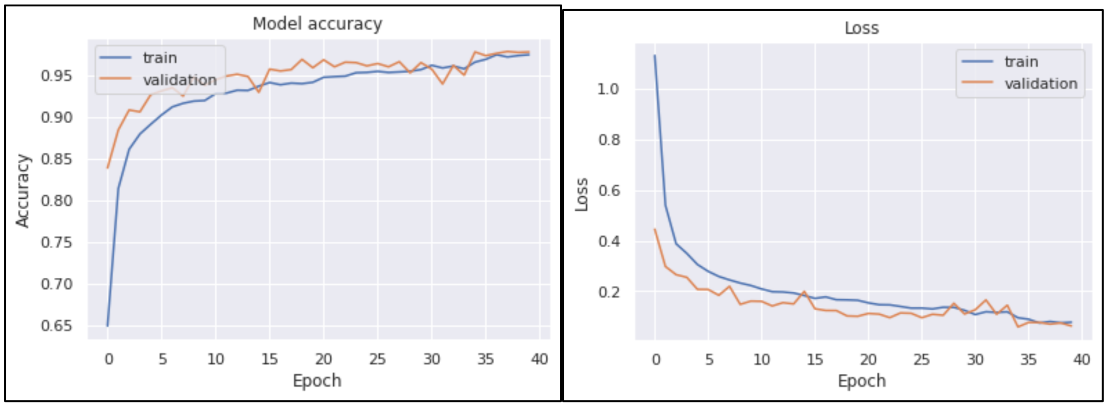


Figure 26: Left: Accuracy of Xception on train and validation data. Right: Loss on train and validation data

The confusion matrix is shown in Figure 27. It is interesting to note that some images of Black-grass and Loose Silky-bent have been misclassified as the other.

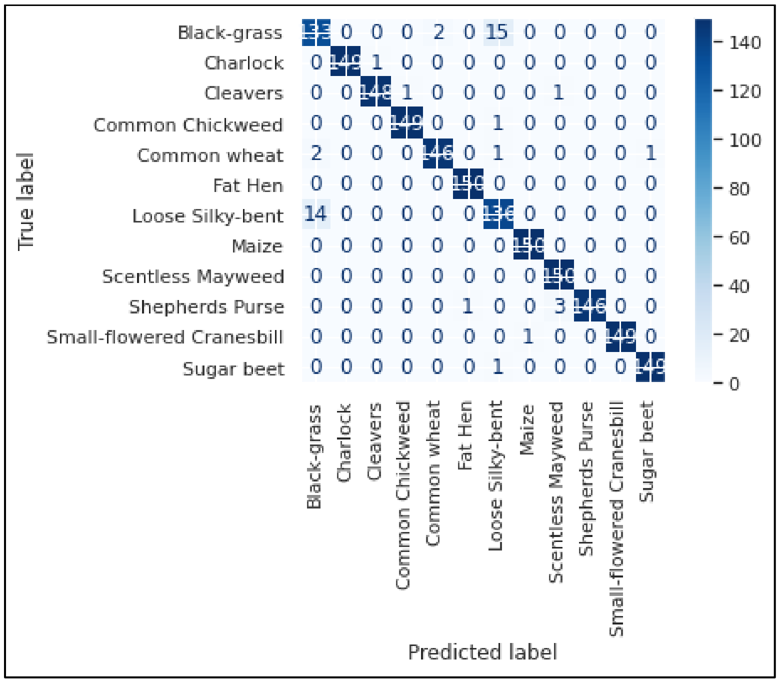


Figure 27: Confusion matrix on validation sample of Balanced Segmented Dataset

Results

The following table summarizes the performance of Xception when trained on different datasets.

Dataset	Public score	Private Score
Original Dataset	0.96851	0.96851
Segmented Dataset	0.97607	0.97607
Balanced Dataset	0.96851	0.96851
Balanced Dataset Segmented	**0.97858**	**0.97858**

Table 6: Scores of Xception on different datasets

The Xception model fine-tuned on the Balanced Segmented Dataset gives the highest score in the competition.

Inception-ResNet-v2

Inception-ResNet-v2 [11] is a convolutional neural architecture that builds on the Inception family of architectures but incorporates residual connections (replacing the filter concatenation stage of the Inception architecture). In the Inception-Resnet block, multiple sized convolutional filters are combined with residual connections. The usage of residual connections not only avoids the degradation problem caused by deep structures but also reduces the training time.

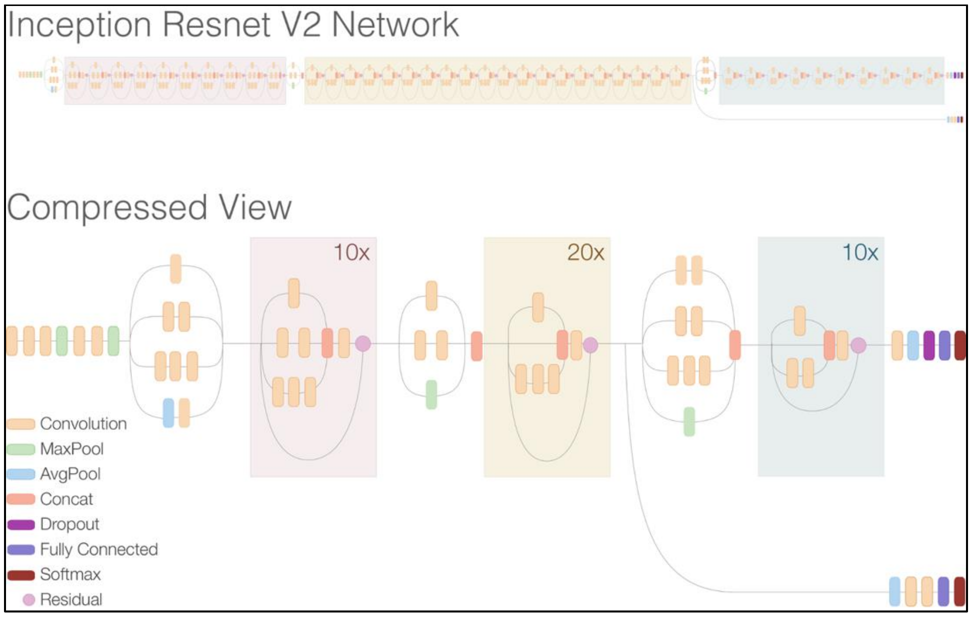


Figure 28: Inception-ResNet-v2 Architecture [11]

Experiments

In this experiment we use a pretrained Inception-ResNet-v2 model which has been trained on more than a million images from the ImageNet database. We use Global Average Pooling instead of a Flatten Layer since it spatially averages each feature map and ensures that each feature map can be considered as a confidence map. We then utilize a Batch Normalization layer followed by a Dense layer with 256 neurons and the mish activation function. The mish activation function is given by $f(x) = x \cdot \tanh(\ln(1 + e^{x}))$ where $\ln(1 + e^{x})$ is the softplus activation function. The mish activation function has been shown to outperform ReLU most of the time[citation]. This is followed by a Batch Normalization layer and a Dropout layer with probability 0.25. The final classification layer contains 12 neurons corresponding to the 12 classes along with the softmax activation function.

The inputs, callbacks and validation size used are the same as what has been mentioned in the Transfer Learning Overview.

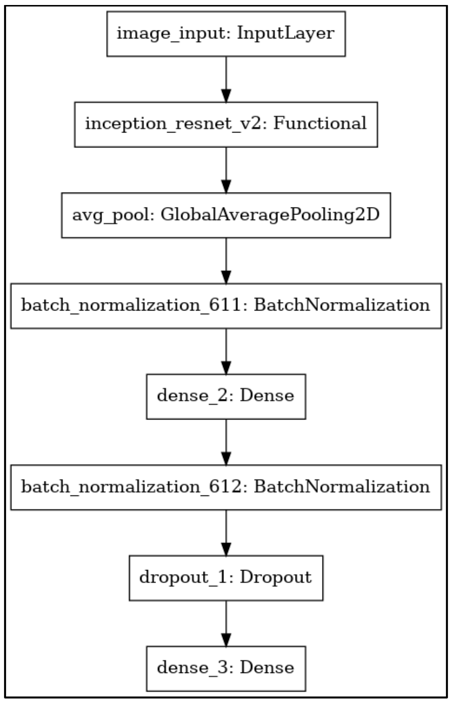

image_input: InputLayer
inception_resnet_v2: Functional
avg_pool: GlobalAveragePooling2D
batch_normalization_611: BatchNormalization
dense_2: Dense
batch_normalization_612: BatchNormalization
dropout_1: Dropout
dense_3: Dense

Figure 29: Inception-ResNet-v2 Model Architecture

We trained the model on 4 datasets (original, segmented, balanced, and balanced segmented). The Loss and Accuracy Plot of the model trained on the original dataset is shown below.

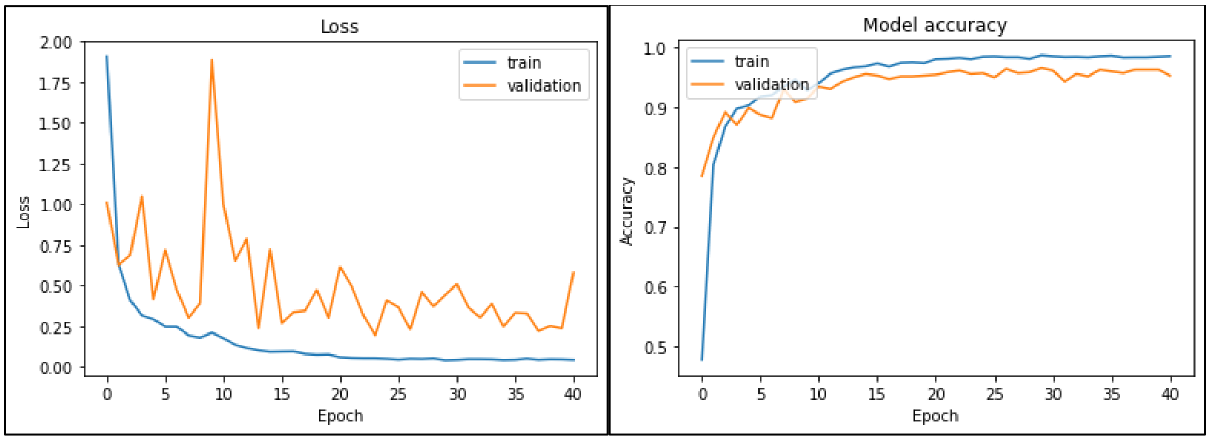


Figure 30: Left: Accuracy of Inception-ResNet-v2 on train and validation data. Right: Loss on train and validation data

The confusion matrix is shown in Figure 27. It is interesting to note that some images of Black-grass and Loose Silky-bent have been misclassified as the other.

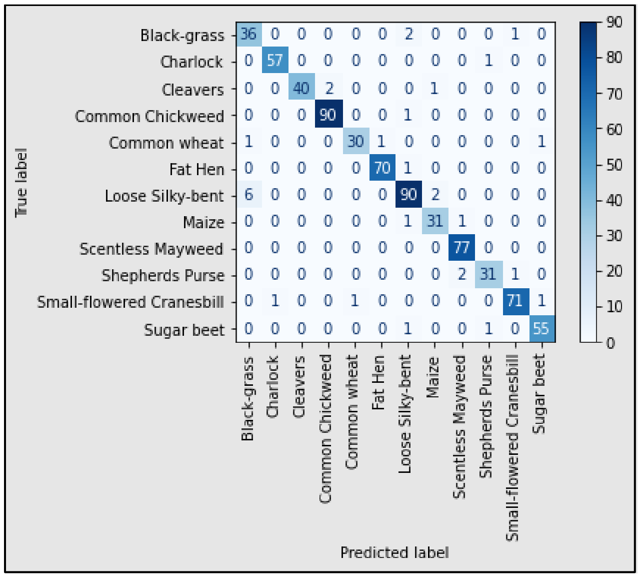


Figure 31: Confusion matrix on validation sample of Original Dataset

Results

The following table summarizes the performance of Inception-ResNet-v2 when trained on different datasets.

Dataset	Public score	Private Score
Original Dataset	0.98362	0.98362
Segmented Dataset	0.96599	0.96599
Balanced Dataset	**0.98488**	**0.98488**
Balanced Dataset Segmented	0.96851	0.96851

Table 7: Scores of Inception-ResNet-v2 on different datasets

The Inception-ResNet-v2 model fine-tuned on the Balanced Dataset gives the highest score in the competition.

Transformers

A transformer model is a neural network that learns context and thus meaning by tracking relationships in sequential data like the words in this sentence. Transformer models have become the de facto status quo in natural language processing (NLP). In computer vision research, there has recently been a rise in interest in Vision Transformers (ViTs). In the following sections we will explore more about vision transformers and how they perform in the Plant Seedling Classification task.

Vision Transformers

Overview

The Vision Transformer (ViT) model was proposed in Dosovitskiy (2020) [12]. It's the first paper that successfully trains a Transformer encoder on ImageNet, attaining very good results compared to familiar convolutional architectures. The Vision Transformer, or ViT, is a model for image classification that employs a Transformer-like architecture over patches of the image. An image is split into fixed-size patches, each of them are then linearly embedded, position embeddings are added, and the resulting sequence of vectors is fed to a standard Transformer encoder. To perform classification, the standard approach of adding an extra learnable "classification token" to the sequence is used. Recently, Vision Transformers (ViT) have achieved highly competitive performance in benchmarks for several computer vision applications, such as image classification, object detection, and semantic image segmentation. ViT exhibits an extraordinary performance when trained on enough data, breaking the performance of a similar state-of-art CNN with 4x fewer computational resources.

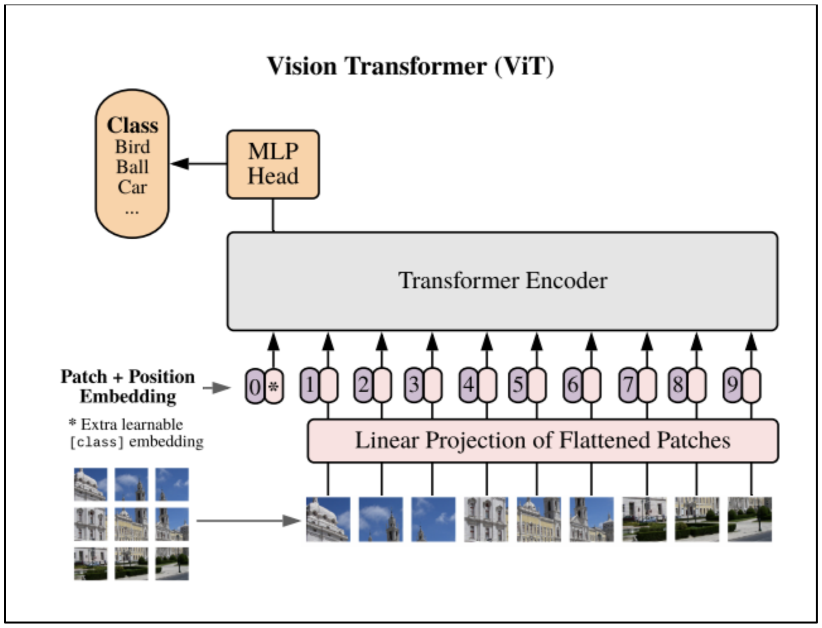

Vision Transformer (ViT)
Class
Bird
Ball
Car
...
MLP Head
Transformer Encoder
Patch + Position Embedding
* Extra learnable [class] embedding
0*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Figure 32: Vision Transformer Architecture [12]

Experiments

We first apply a series of transformations (Resize image to 224x224, Randomized Crop, Randomized Horizontal Flip, Pixel Normalization) to help our model generalize and prevent overfitting. We then trained a Vision Transformer using pytorch-lightning wrapper with the hyperparameters mentioned in table 8. We used dropouts to prevent overfitting and callbacks to save the best model.

Hyper-parameters	**Value**
Embedding dimensions	256
Hidden Dimensions	512
Number of heads	8
Number of layers	6
Patch size	16x16
Number of channels	3
Number of patches	196
Dropout Rate	25%
Max epochs	60
Batch Size	16
Learning Rate	0.0003
Validation Size	10% of dataset

Table 8: Hyperparameter settings for training vision transformer

Result

The following table summarizes the performance of the ViT when trained on different datasets.

Dataset	Public score	Private Score
Original Dataset	**0.87027**	**0.87027**
Segmented Dataset	0.80100	0.80100
Balanced Dataset	0.84256	0.84256
Balanced Dataset Segmented	**0.87027**	**0.87027**

Table 9: Scores of Vision Transformer on different datasets

The ViT model fine-tuned on the Original Dataset and the Balanced Dataset with Segmentation gives the highest score in the competition.

Vision Transformers Transfer Learning and Fine-Tuning

Overview

While the ViT full-transformer architecture is a promising option for vision processing tasks, the performance of ViTs is still inferior to that of similar-sized CNN alternatives (such as ResNet) when trained from scratch on a mid-sized dataset such as ImageNet.

The vision transformer model is trained on a huge dataset even before the process of fine-tuning. The only change is to disregard the MLP layer and add a new D times KD*K layer, where K is the number of classes of the small dataset.

Experiments

We first apply a series of transformations (Resize image to 224x224, Randomized Crop, Randomized Horizontal Flip, Pixel Normalization across the RGB channels with mean (0.5, 0.5, 0.5) and standard deviation (0.5, 0.5, 0.5) to help our model generalize and prevent overfitting. We then trained a Vision Transformer (ViT) model pre-trained on ImageNet-21k (14 million images, 21,843 classes) with image resolution 224x224 using pytorch-lightning wrapper with the hyperparameters mentioned in table 10. We used the EarlyStopping callback to stop the training process if the loss doesn't improve due to overfitting. The pretrained weights are from https://huggingface.co/google/vit-base-patch16-224-in21k

Hyper-parameters	Value
Max epochs	100
Batch Size	16
Learning Rate	0.0001
Validation Size	10% of dataset

Table 10: Hyperparameter settings for training pre-trained vision transformer

Result

The following table summarizes the performance of the pretrained ViT when trained on different datasets.

Dataset	Public score	Private Score
Original Dataset	0.97229	0.97229
Segmented Dataset	0.93198	0.93198
Balanced Dataset	**0.98110**	**0.98110**
Balanced Dataset Segmented	0.96473	0.96473

Table 11: Scores of pre-trained vision transformer on different datasets

The Pre-trained ViT fine-tuned on the Balanced Dataset gives the highest score in the competition.

Ensembling

Overview

Ensemble methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone. When we ensemble multiple algorithms to adapt the prediction process to combine multiple models, we need an aggregating method. Three main techniques can be used:

1. Max Voting: The final prediction in this technique is made based on majority voting for classification problems.
2. Averaging: Typically used for regression problems where predictions are averaged. Probability can be used as well, for instance, in averaging the final classification.
3. Weighted Average: Sometimes, we need to give weights to some models/algorithms when producing the final predictions.

Due to the low cardinality of the dataset, we avoid Stacking and Blending ensembles since they require a lot of data to train the base classifiers as well as the meta classifiers without overfitting. We experiment with the Weighted Average aggregation technique and ensemble the Xception, Inception-ResNetv2 and EfficientnetB3 models by taking a weighted average of their probabilities before making the final prediction. Note that Averaging is a special case of Weighted Average where all the weights/multipliers are equal and sum up to 1.

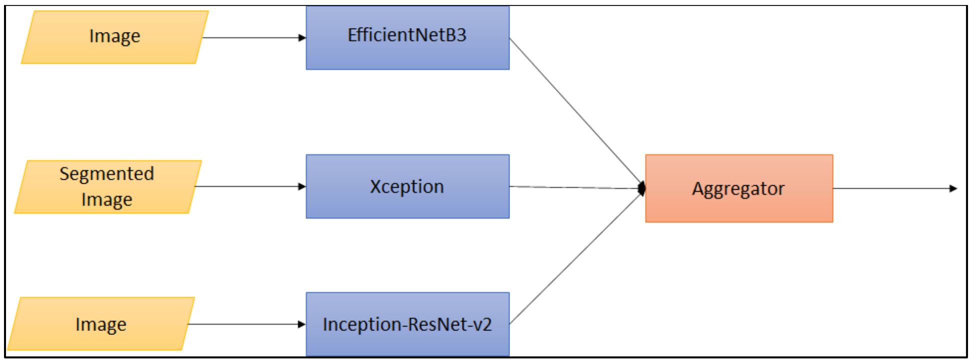


Figure 33: Weighted Average Ensemble Classifier

Experiments and Results

The best performing models are chosen as the base classifiers and the weighted average of the probabilities are calculated for the final prediction. The multipliers are chosen such that they add up to 1. The final probabilities are given as follows.

$$P(x) = w_1 \cdot EfficientNetB3(x) + w_2 \cdot InceptionResNetV2(x) + w3 \cdot Xception(x)$$

The following table summarizes the results of the weighted average ensemble. Different multipliers have been used for different classifiers and the Leaderboard rank is observed with these different multipliers.

EfficientNetB3	Inception-ResNet-v2	Xception	Public score	Private Score
1.0	0.0	0.0	0.98236	0.98236
0.0	1.0	0.0	0.98488	0.98488
0.0	0.0	1.0	0.97858	0.97858
0.5	0.5	0.0	0.98614	0.98614
0.0	0.5	0.5	0.98740	0.98740
0.5	0.0	0.5	0.98362	0.98362
0.33	0.33	0.33	0.98614	0.98614
0.25	0.5	0.25	0.98992	0.98992
0.1	**0.5**	**0.4**	**0.99118**	**0.99118**
0.3	0.5	0.2	0.98992	0.98992
0.3	0.6	0.1	0.98740	0.98740
0.2	0.6	0.2	0.98866	0.98866

Table 12: Impact of weights on final score

The Weighted Average Ensemble (0.9918) outperformed the individual classifiers — EfficientnetB3 (0.98236), Inception-ResNet-v2 (0.98488) and Xception (0.97858). Note that all the weighted average ensembles beat the individual classifiers involved in the ensemble (a weight of 0 indicates that the classifier is not involved in the ensemble).

The aggregation that gives the best result is:

$$P(x) = 0.1 \cdot EfficientNetB3(x) + 0.5 \cdot InceptionResNetV2(x) + 0.4 \cdot Xception(x)$$

Other Techniques

Test Time Augmentation

Overview

Like what Data Augmentation is doing to the training set, the purpose of Test Time Augmentation is to perform random modifications to the test images. Thus, instead of showing the regular, "clean" images, only once to the trained model, the model will see the augmented images several times. The predictions of each corresponding image are averaged, and the final prediction is calculated from the average value.

Experiments and Results

During inference we created 20 different augmentations for each test image and passed it to the model for prediction. The final prediction is the average of all these predictions.

The following table summarizes the results of inference of different models when test time augmentation is applied.

Model	Public Score	Private Score	Public Score with TTA	Private score with TTA
Convolutional Neural Network	0.96725	0.96725	**0.96851**	**0.96851**
Xception	0.97481	0.97481	**0.98362**	**0.98362**
Inception-Resnet-v2	0.98488	0.98488	**0.98614**	**0.98614**
EfficientnetB3	**0.98362**	**0.98362**	0.97984	0.97984
ViT	**0.98110**	**0.98110**	0.97355	0.97355

Table 13: Comparison of model performance with Test-Time Augmentation

From Table 13 we can see that applying Test-Time Augmentation during inference improved the scores of most of our models. However, it did result in lower scores for the EfficientNetB3 and Vision Transformer.

Binary Classifier

Overview

From confusion matrices of the convolutional neural networks (CNN, Xception and Inception-ResNet-v2) we observed that the class labels ‘Black-grass’ and ‘Loose Silky-bent' often get misclassified as one another. On reinspection of the dataset, Black-grass and Loose Silky-bent images are indistinguishable for normal humans and only a domain expert can easily distinguish between them. This is the reason why our models can’t classify these images accurately.

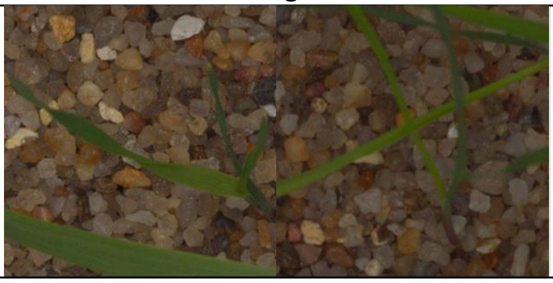
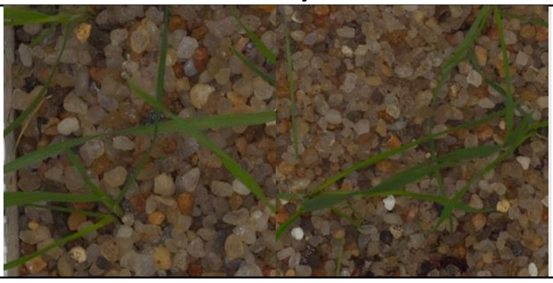
Black-grass	**Loose Silky-bent**
	

Table 14: Comparison of images from class labels ‘Black-grass’ and ‘Loose Silky-bent'

To correct this error, we propose a binary classifier that changes the predictions of the main model if the prediction is ‘Black-grass’ or ‘Loose Silky-bent'. The binary classifier will be trained only on images from these 2 classes. Our hypothesis is that the binary classifier will better distinguish between these 2 classes than our main models. The proposed idea is illustrated in figure 34.

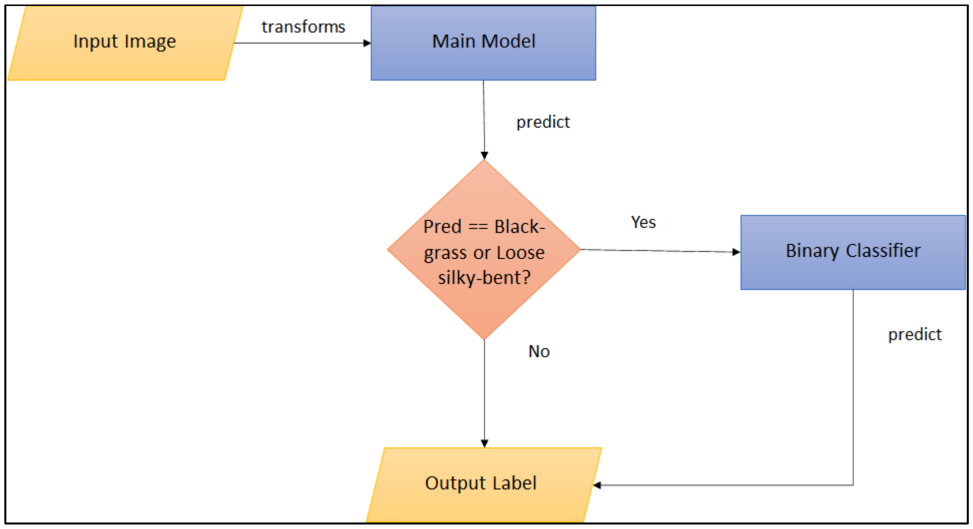


Figure 34: Binary Classification Error Correction Process

Experiments and Results

We train a binary classifier on only the 2 class labels ‘Black-grass’ and ‘Loose Silky-bent' from the Balanced Dataset. The architecture we use is the EfficientNetB3 with a final layer containing 1 neuron with the sigmoid activation function. The architecture of the model is described in figure 33.

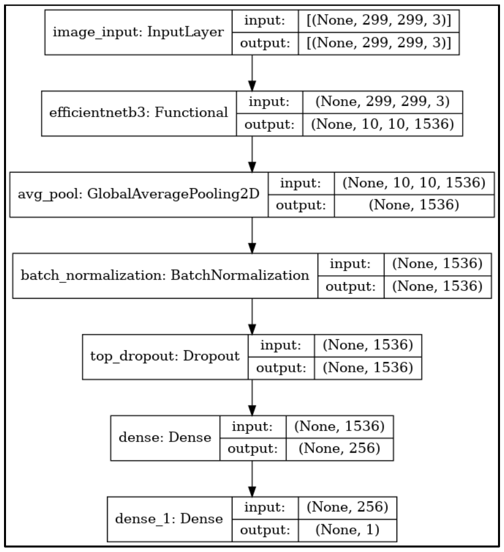

image_input: InputLayer | input: [(None, 299, 299, 3)] | output: [(None, 299, 299, 3)]
efficientnetb3: Functional | input: (None, 299, 299, 3) | output: (None, 10, 10, 1536)
avg_pool: GlobalAveragePooling2D | input: (None, 10, 10, 1536) | output: (None, 1536)
batch_normalization: BatchNormalization | input: (None, 1536) | output: (None, 1536)
top_dropout: Dropout | input: (None, 1536) | output: (None, 1536)
dense: Dense | input: (None, 1536) | output: (None, 256)
dense_1: Dense | input: (None, 256) | output: (None, 1)

Figure 35: Binary Classifier Architecture

The Loss and Accuracy plots are shown below

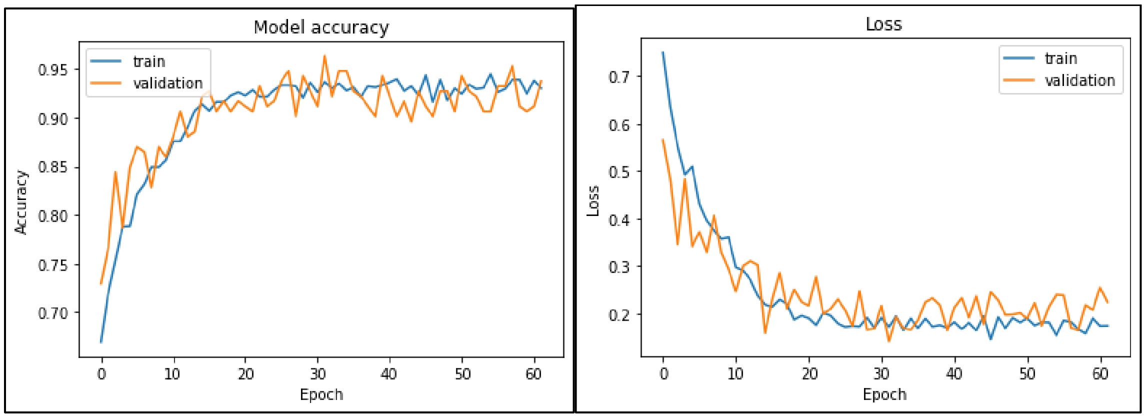


Figure 36: Left: Accuracy of Binary Classifier on train and validation data. Right: Loss on train and validation data

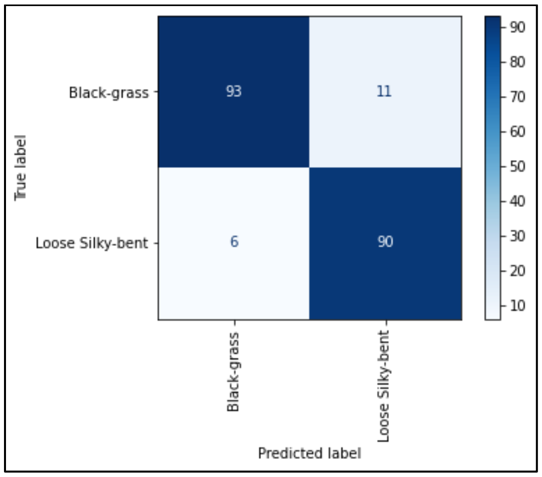


Figure 37: Confusion matrix produced by binary classifier

The table below summarizes the performance of our models when the binary classifier is used to correct the errors of misclassification.

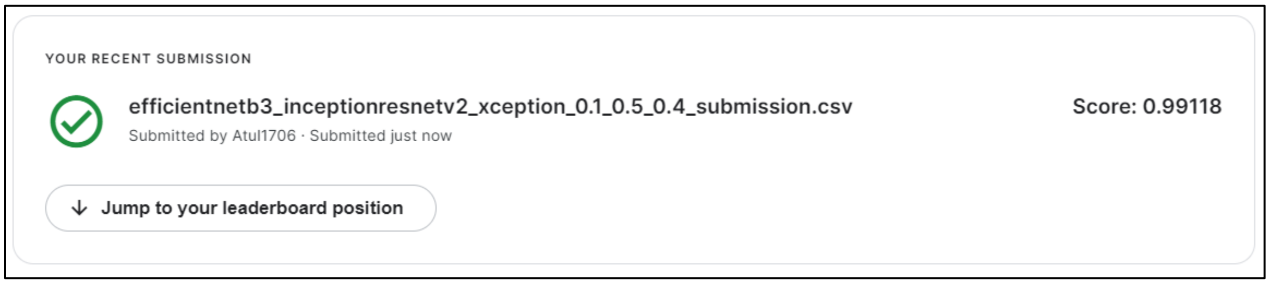
Model	Public Score	Private Score	Public Score with Binary Classifier	Private score with Binary Classifier
Convolutional Neural Network	**0.96725**	**0.96725**	0.85012	0.85012
Xception	**0.97858**	**0.97858**	0.96725	0.96725
Inception-Resnet-v2	**0.98488**	**0.98488**	0.97607	0.97607
EfficientnetB3	**0.98362**	**0.98362**	0.97355	0.97355

Table 15: Comparison of model performance with Binary Classifier

The binary classifier worsened the performance of all our major models. The binary classifier struggles to differentiate between the 2 classes. This could be due to multiple reasons such as bad architecture and hyperparameter choices of binary classifier and some images between the 2 classes may be almost impossible to distinguish even for a domain expert.

Leaderboard

The following is our leaderboard score with the weighted average ensemble. We scored 0.99118 and reached the Top 1.56% (Rank 13/833).


YOUR RECENT SUBMISSION

efficientnetb3_inceptionresnetv2_xception_0.1_0.5_0.4_submission.csv

Submitted by Atul1706 · Submitted just now

Score: 0.99118

Figure 38: Leaderboard Score

Solution Novelty

Enhanced activate functions and layers

As we trained with very deep neural networks such as Xception, we will use Batch normalizatio n as a method to standardize the inputs to a layer for each mini batch. Also, instead of using RELU, we applied Scaled Exponential Linear Unit (SELU) or Mish activation as they can learn faster and better than RELU, even when they are combined with batch normalization. Mish is a recent activate function which is unbounded above and bounded below. Such attributes prevent models from saturation due to capping and compared to a hard zero bound like RELU, Mish allows better gradient flow. On the other hand, SELU is an activation that introduces self-normalization property which can preserve mean and variance from previous layers. This is called "internal normalization" that allows the network to converge faster. The original paper also proves that this activation function can prevent the network from vanishing and exploding gradient problem.

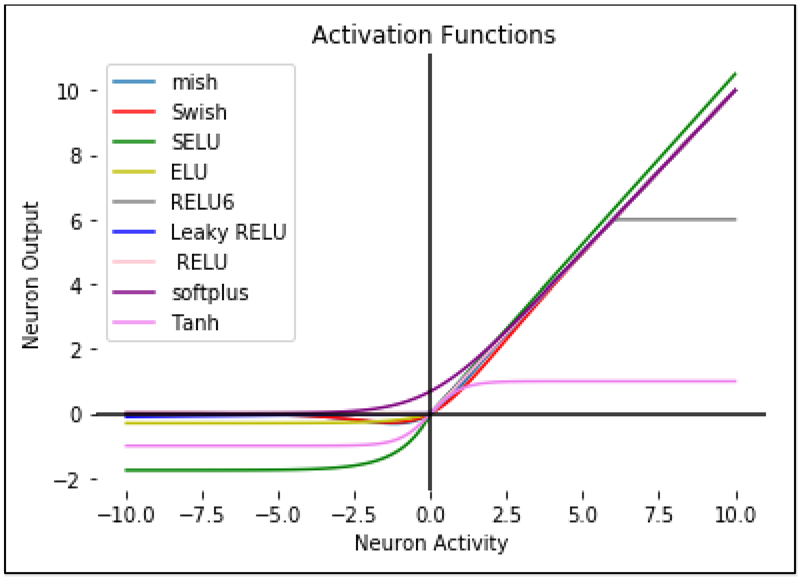
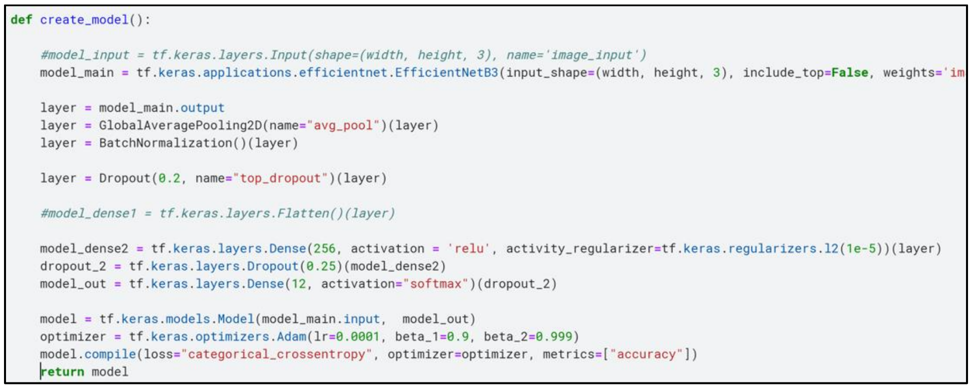


Figure 39: Graphs of common activation functions

Dropout and L2 Regularization

We applied several regularization techniques including Dropout and L2 Regularization. Drop out is used to ignore randomly selected neurons during training with a specified probability. On the other hand, L2 Regularization penalizes the model a cost to keep the model weight small. Such techniques help us to evade model overfitting during training, especially when we have big datasets like our balanced dataset (12000 images).



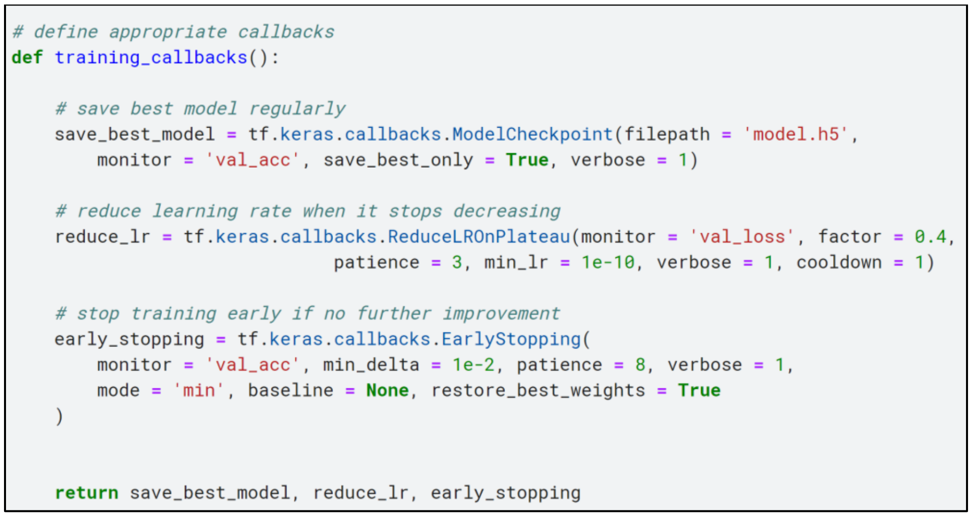
```
def create_model():

    #model_input = tf.keras.layers.Input(shape=(width, height, 3), name='image_input')
    model_main = tf.keras.applications.efficientnet.EfficientNetB3(input_shape=(width, height, 3), include_top=False, weights='im

    layer = model_main.output
    layer = GlobalAveragePooling2D(name="avg_pool")(layer)
    layer = BatchNormalization()(layer)

    layer = Dropout(0.2, name="top_dropout")(layer)

    #model_dense1 = tf.keras.layers.Flatten()(layer)

    model_dense2 = tf.keras.layers.Dense(256, activation = 'relu', activity_regularizer=tf.keras.regularizers.l2(1e-5))(layer)
    dropout_2 = tf.keras.layers.Dropout(0.25)(model_dense2)
    model_out = tf.keras.layers.Dense(12, activation="softmax")(dropout_2)

    model = tf.keras.models.Model(model_main.input,  model_out)
    optimizer = tf.keras.optimizers.Adam(lr=0.0001, beta_1=0.9, beta_2=0.999)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["accuracy"])
    return model
```

Figure 40: Code snippet implementing dropouts and L2 Regularization

EarlyStopping and ReduceLROnPlateau callbacks in Keras

In this project, we utilized Keras' powerful callbacks: EarlyStopping and ReduceLROnPlateau which will be called in each epoch during training. EarlyStopping callback will stop the training when the loss is not improved as it has coverage and the model started overfitting. In addition, we implemented ReduceLROnPlateau to reduce the learning rate if the loss stops improving after several epochs. It is proved that a larger learning rate could cause the loss to oscillate around its minimum.

Such callbacks prevent learning from stagnation without overfitting, which helps to achieve higher accuracy.

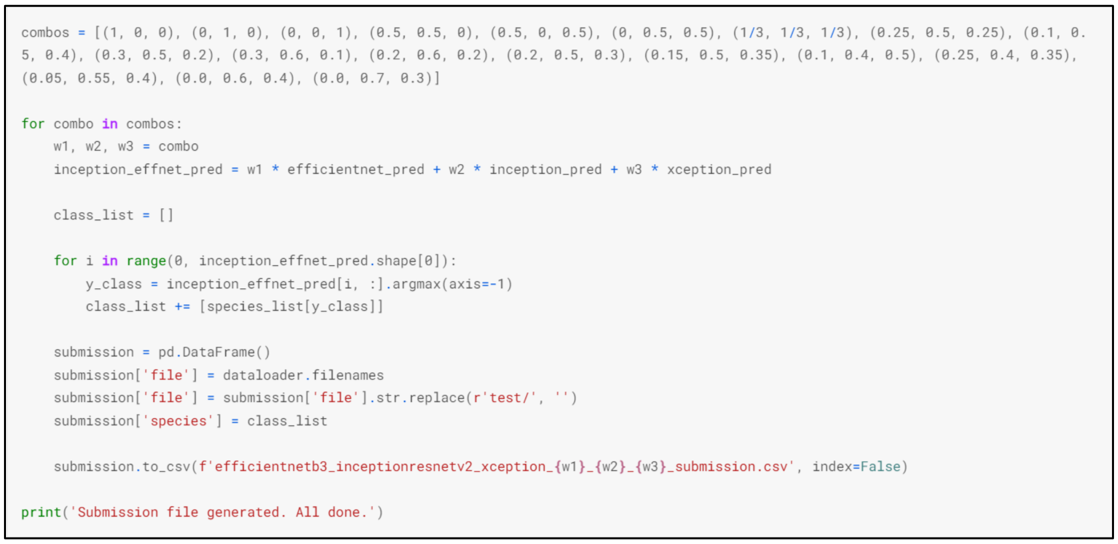


```
# define appropriate callbacks
def training_callbacks():

    # save best model regularly
    save_best_model = tf.keras.callbacks.ModelCheckpoint(filepath = 'model.h5',
        monitor = 'val_acc', save_best_only = True, verbose = 1)

    # reduce learning rate when it stops decreasing
    reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor = 'val_loss', factor = 0.4,
                              patience = 3, min_lr = 1e-10, verbose = 1, cooldown = 1)

    # stop training early if no further improvement
    early_stopping = tf.keras.callbacks.EarlyStopping(
        monitor = 'val_acc', min_delta = 1e-2, patience = 8, verbose = 1,
        mode = 'min', baseline = None, restore_best_weights = True
    )


    return save_best_model, reduce_lr, early_stopping
```

Figure 41: Code snippet implementing callbacks

Ensembling

We trained multiple base models which scored decently well in the competition. We were not able to cross the Top 10% rank and we decided to ensemble our best models. Ensembling is a highly effective technique that allowed us to cross the Top 10% rank and reach the Top 1.56% Rank just by using the weighted average of the probabilities generated by our best models. Ensembling reduced the errors from our individual models, and we were able to increase our score from 0.98488 (Inception-Resnet-v2) to 0.99118 (Weighted Average Ensemble).



```
combos = [(1, 0, 0), (0, 1, 0), (0, 0, 1), (0.5, 0.5, 0), (0.5, 0, 0.5), (0, 0.5, 0.5), (1/3, 1/3, 1/3), (0.25, 0.5, 0.25), (0.1, 0.
5, 0.4), (0.3, 0.5, 0.2), (0.3, 0.6, 0.1), (0.2, 0.6, 0.2), (0.2, 0.5, 0.3), (0.15, 0.5, 0.35), (0.1, 0.4, 0.5), (0.25, 0.4, 0.35),
(0.05, 0.55, 0.4), (0.0, 0.6, 0.4), (0.0, 0.7, 0.3)]

for combo in combos:
    w1, w2, w3 = combo
    inception_effnet_pred = w1 * efficientnet_pred + w2 * inception_pred + w3 * xception_pred

    class_list = []

    for i in range(0, inception_effnet_pred.shape[0]):
        y_class = inception_effnet_pred[i, :].argmax(axis=-1)
        class_list += [species_list[y_class]]

    submission = pd.DataFrame()
    submission['file'] = dataloader.filenames
    submission['file'] = submission['file'].str.replace(r'test/', '')
    submission['species'] = class_list

    submission.to_csv(f'efficientnetb3_inceptionresnetv2_xception_{w1}_{w2}_{w3}_submission.csv', index=False)

print('Submission file generated. All done.')
```

Figure 42: Code sippet implementing Weighted Average Aggregator

Vision Transformers

We created our own Vision Transformer and experimented with a pre-trained Vision Transformer. Vision Transformers have not been used in the competition before. This could be because Vision Transformers did not exist during this competition. We showed that a pre-trained Vision Transformed fine-tuned on the Plant Seedling Classification challenge can achieve scores and reach the Top 12.48%.

Conclusion

Apart from being able to get our hands dirty trying out different machine learning techniques, one of our key takeaways from this project has been how to approach an image classification problem. We were able to understand the importance of the exploratory data analysis and pre-processing steps. It is crucial to have a good understanding of the data at hand so that we can choose appropriate machine learning techniques to tackle the problem. Our initial experiments helped us verify that one of our image pre-processing steps, namely image segmentation, might prove to help boost model accuracies. We realized the importance of Convolutional layers to handle tasks related to images and understood the effectiveness of Transfer Learning. Thus, it is necessary to start with smaller experiments and set a baseline before moving on to apply advanced techniques such as model ensembling, test time augmentation etc.

References

[1] Giselsson, T. M. (2017, November 15). A Public Image Database for Benchmark of Plant Seedling. arXiv.Org. https://arxiv.org/abs/1711.05458

[2] Jaadi, Z. (2021). Step-by-step explanation principal component analysis. Builtin. https://builtin.com/data-science/step-step-explanation-principal-component-analysis

[3] Brownlee, J. (2019, July 19). A gentle introduction to generative adversarial networks (GANs). Machine Learning Mastery. https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/

[4] Radford, A. (2015, November 19). Unsupervised representation learning with deep Convolutional. arXiv.Org. https://arxiv.org/abs/1511.06434

[5] Mirza, M. (2014, November 6). Conditional Generative Adversarial Nets. arXiv.Org. https://arxiv.org/abs/1411.1784

[6] Wikipedia contributors. (2022, March 20). K-means clustering. Wikipedia. https://en.wikipedia.org/wiki/K-means_clustering

[7] Brownlee, J. (2020, February 23). Develop k-Nearest Neighbors in Python From Scratch. Machine Learning Mastery. https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

[8] CS231n Convolutional Neural Networks for Visual Recognition. (2022). CS231n: Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/neural-networks-1/

[9] Tan, M. (2019, May 28). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv.Org. https://arxiv.org/abs/1905.11946v5

[10] Chollet, F. (2016, October 7). Xception: Deep Learning with Depthwise Separable Convolutions. arXiv.Org. https://arxiv.org/abs/1610.02357

[11] Szegedy, C. (2016, February 23). Inception-v4, Inception-ResNet and the Impact of Residual. . . arXiv.Org. https://arxiv.org/abs/1602.07261v2

[12] Dosovitskiy, A. (2020, October 22). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv.Org. https://arxiv.org/abs/2010.11929