Machine Learning Engineer Nanodegree

Capstone Project

Praveen Maniyan
May 14, 2018

I. Definition

Project Overview

In this world, there are some thousands to millions species of plants. In the field of Botany, there have been many research work in the field of Identifying a Plant species with the use of plant leaf. Due to this fact, a lot of researchers and Botanists, have been collecting a lot of plant leaf and have been documenting them as leaf databases. One form of this database is the photographs taken of the leaf for each species in a specific angle, position and light exposure. Such a Database/Dataset exists in the UCI (University of California, Irvine) Machine Learning Repository, known as Folio Data Set.

My project is to explore such a leaf database, and try to implement a Machine Learning technique/algorithm, especially a Deep Learning technique. After the project is implemented, the algorithm will be able to provide an accuracy score based on the training and test/validation data.

Problem Statement

The Plant species classification can be best achieved by using the leaf samples, as they are unique to species and is a challenging task. The application of this research work is in identifying a plant species from a leaf sample, identify how close a very new leaf sample might belong to a known species, and during fossil extraction, to which variety can this fossil be linked to the current active species. We can try to use Neural Networks with computer vision to solve the classification problem.

The solution would be built by following the below mentioned steps:

1. Download the Dataset from https://archive.ics.uci.edu/ml/machine-learning-databases/00338/Folio%20Leaf%20Dataset.rar
2. Organize the data/images in the downloaded folder into train and test/validation folders
3. Train a VGG16 (a Deep Convolutional Networks for Large-Scale Image Recognition) Pre-trained model in Keras
4. Measure the loss and accuracy and report it

The final application is expected to be useful in showing that such an approach can be used to for identifying a plant species and also how close a new/unknown plant could be associated to a know plan species.

Metrics

Binary Cross Entropy is a measure for the loss calculation in CNN Deep Learning. The lower the loss, the better the model is trying to learn and best fits the data. The formula used for this calculation is:

$$crossentropy(t,o) = -(t \cdot log(o) + (1-t) \cdot log(1-o))$$ or

$$L_{CE} \equiv t\log(o) + (1-t)\log(1-o)$$

The final binary cross entropy of the final model is calculated as:

$$\mathcal{L}(\hat{\mathrm{y}},\mathrm{y}) = -\frac{1}{N}\sum_{i}^{N}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

Justification:

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label.

In binary classification, where the number of classes M equals 2, cross-entropy can be calculated as the first formula show above.

If M>2 (i.e. multiclass classification), we calculate a separate loss for each class label per observation and sum the result as in the second formula shown above.

Since, the is a species prediction problem, multi-class classification has to be used and employed. Hence, the binary cross-entropy has been used.

II. Analysis

Data Exploration

The Dataset which can be used for this project can be found at the UCI Machine Learning Repository at the following link.

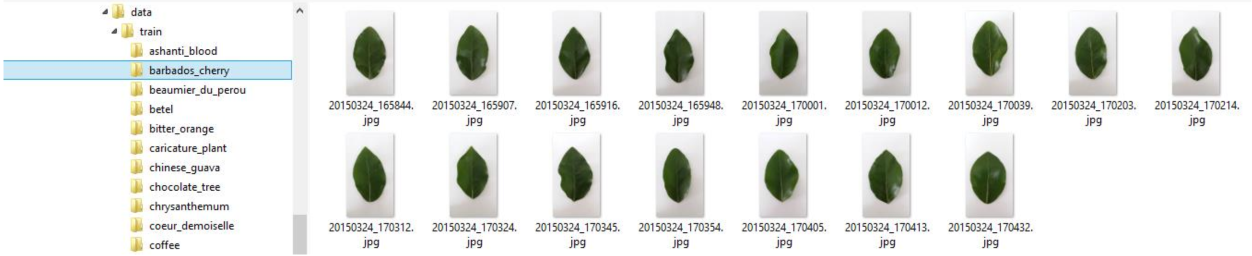
https://archive.ics.uci.edu/ml/datasets/Folio

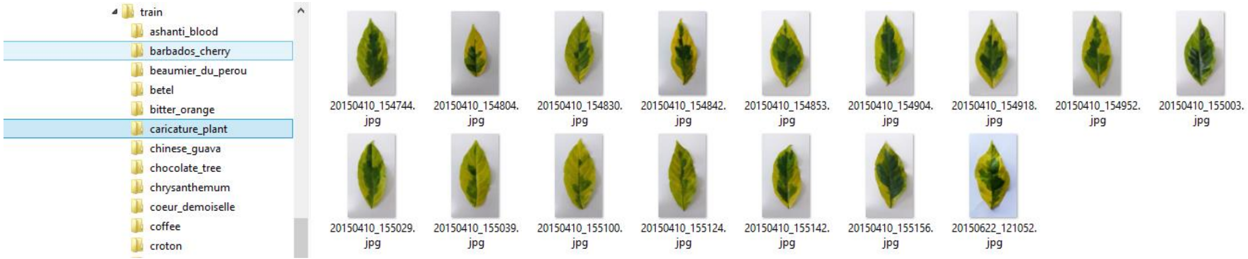
The Datase is called the Folio dataset and comprises of 20 photos of leaves for each of 32 different species. From the above set, the data set is described as below:

Data Set Characteristics:	Multivariate	**Number of Instances:**	637	**Area:**	N/A
Attribute Characteristics:	N/A	**Number of Attributes:**	20	**Date Donated**	2015-07-05
Associated Tasks:	Classification, Clustering	**Missing Values?**	Yes	**Number of Web Hits:**	36774

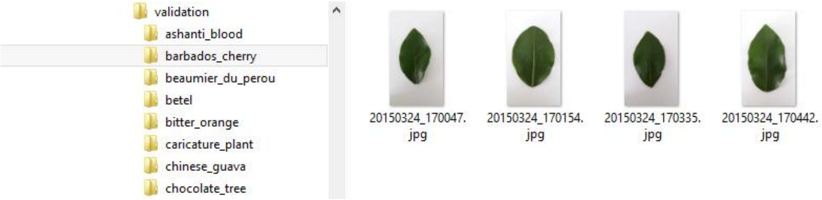
Exploratory Visualization

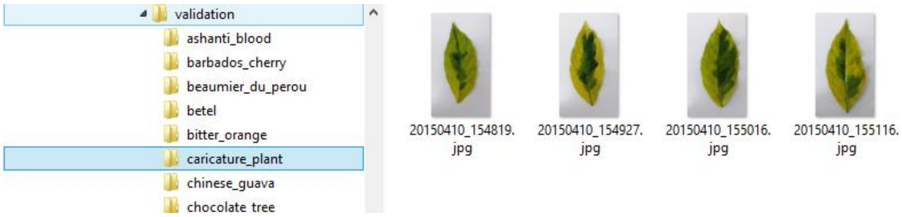
The training set contains 509 (training database) files in 32 (species) folders. The validation set contains 128 (test database) files in 32 (species) folders. The training data folder is organized as follows:



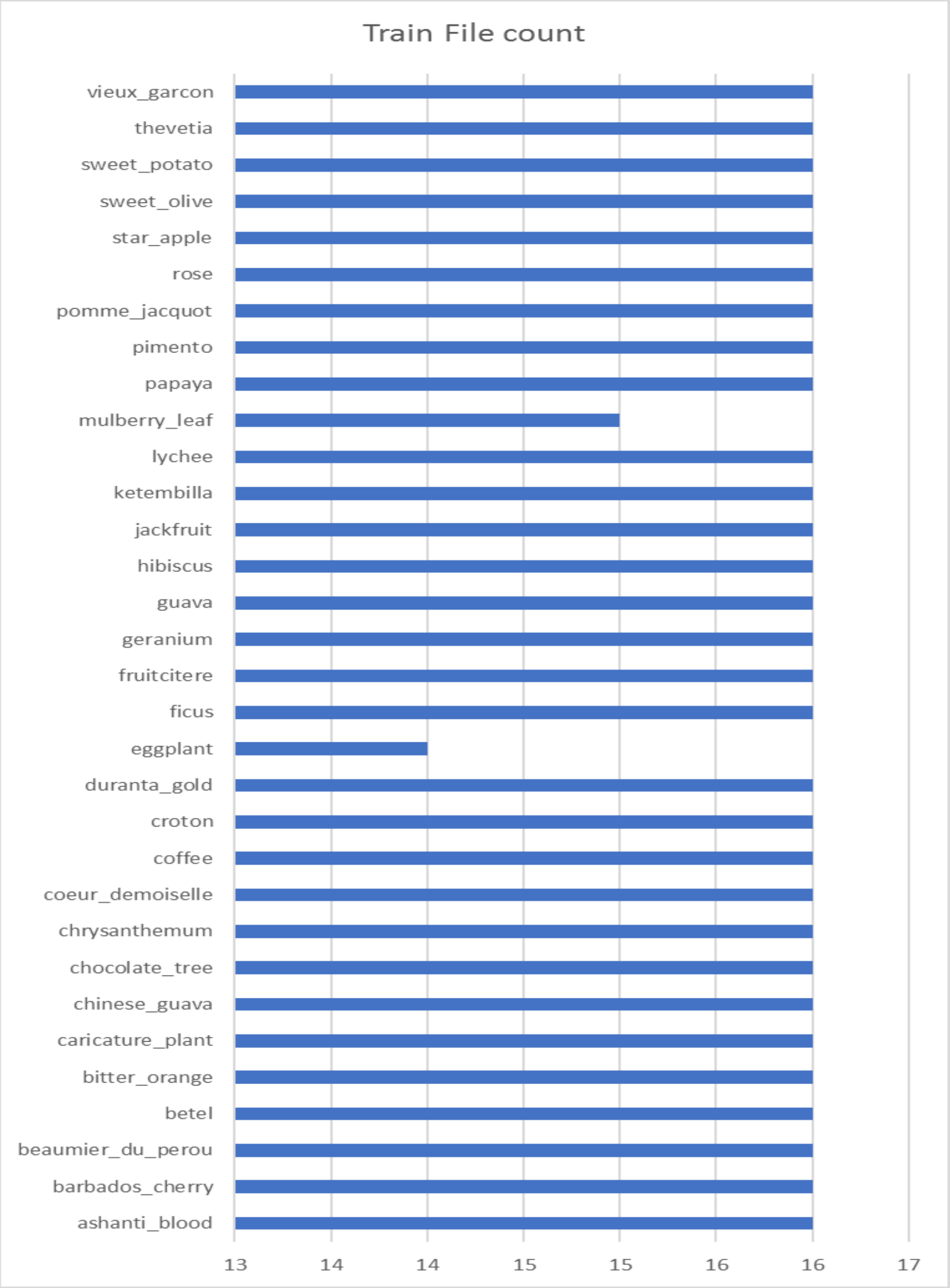


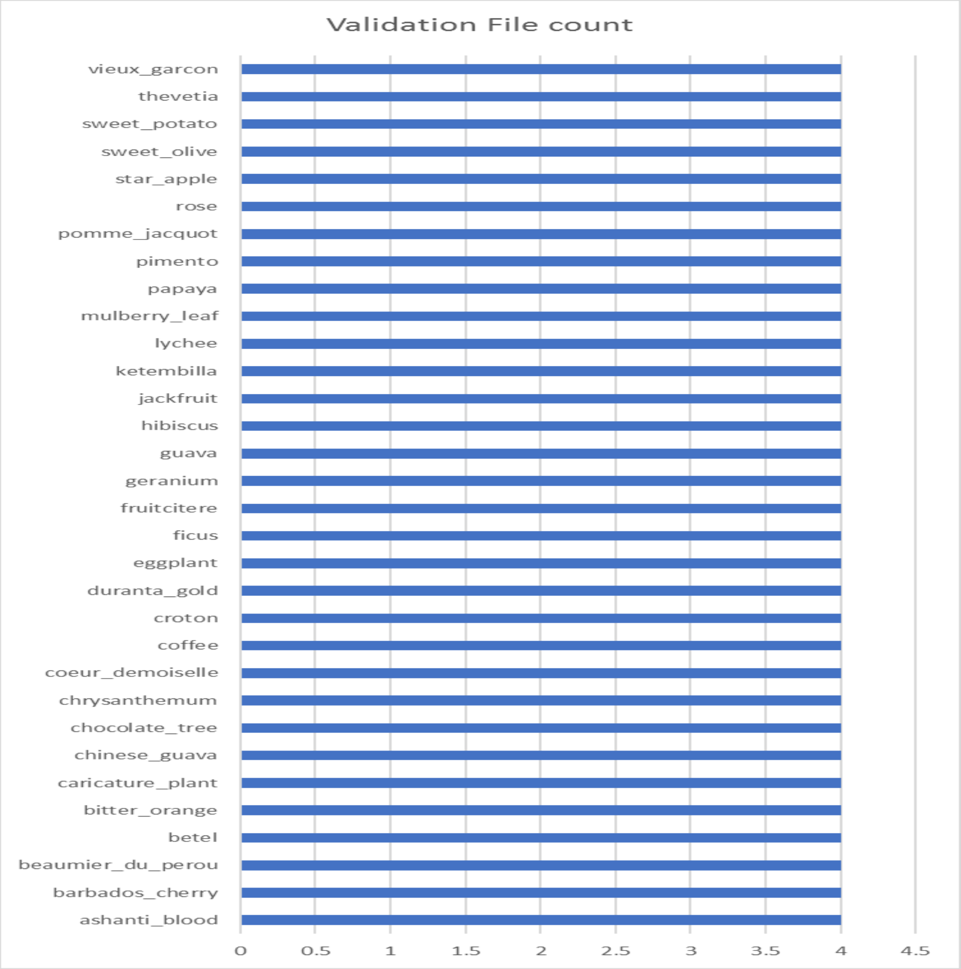
The validation/test data folder is organized as follows:





Please find below the Frequency Distribution of the Count of Files for each species for Training and Validation.


Train File count
vieux_garcon
thevetia
sweet_potato
sweet_olive
star_apple
rose
pomme_jacquot
pimento
papaya
mulberry_leaf
lychee
ketembilla
jackfruit
hibiscus
guava
geranium
fruitcitere
ficus
eggplant
duranta_gold
croton
coffee
coeur_demoiselle
chrysanthemum
chocolate_tree
chinese_guava
caricature_plant
bitter_orange
betel
beaumier_du_perou
barbados_cherry
ashanti_blood
13
14
14
15
15
16
16
17



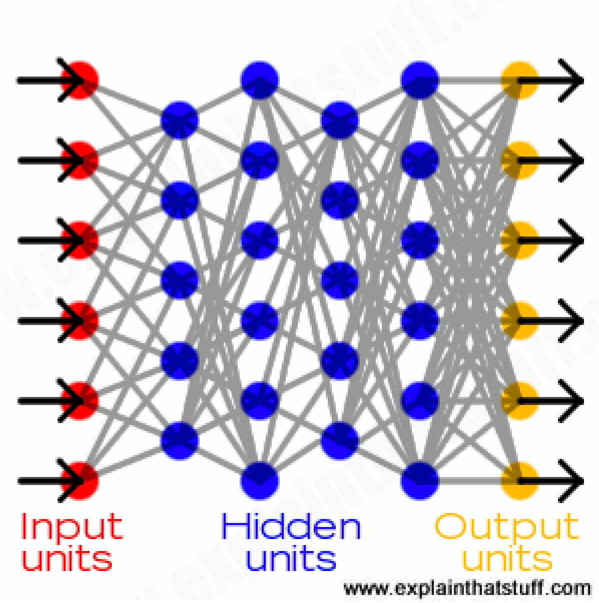
Algorithms and Techniques

I have used the technique of Transfer Learning to implement this algorithm. The basic idea involved are Neural Networks, Convolution Neural Network , and Transfer Learning. Convolution Neural Network classifier, which is the state-of-the art algorithm for Image classification. The details are mentioned below.

Neural Network

The basic idea behind a neural network is to simulate (copy in a simplified but reasonably faithful way) lots of densely interconnected brain cells inside a computer so you can get it to learn things, recognize patterns, and make decisions in a humanlike way. The amazing thing about a neural network is that you don't have to program it to learn explicitly: it learns all by itself, just like a brain.

A typical neural network has anything from a few dozen to hundreds, thousands, or even millions of artificial neurons called units arranged in a series of layers, each of which connects to the layers on either side. Some of them, known as input units, are designed to receive various forms of information from the outside world that the network will attempt to learn about, recognize, or otherwise process. Other units sit on the opposite side of the network and signal how it responds to the information it's learned; those are known as output units. In between the input units and output units are one or more layers of hidden units, which, together, form the majority of the artificial brain. Most neural networks are fully connected, which means each hidden unit and each output unit is connected to every unit in the layers either side. The connections between one unit and another are represented by a number called a weight, which can be either positive (if one unit excites another) or negative (if one unit suppresses or inhibits another). The higher the weight, the more influence one unit has on another. (This corresponds to the way actual brain cells trigger one another across tiny gaps called synapses.)



Convolutional neural network

VGG16 is a famous pre-trained CNN available in Keras module and produces a fair enough acceptable module with acceptable accuracy.

The Configuration of the VGG16 model, a summary, a visualization on how it can be perceived are given below.

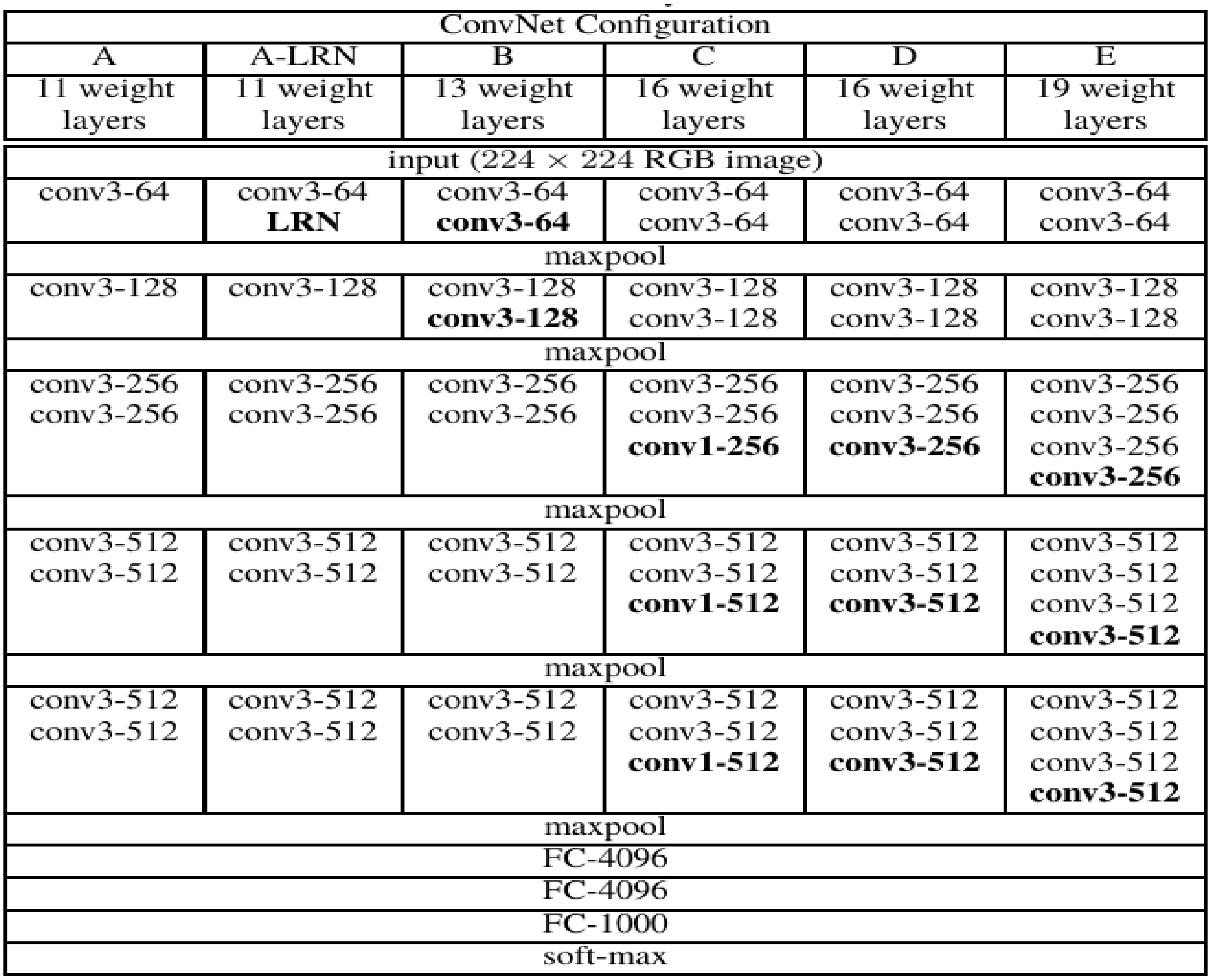
Application:

- Given image → find object name in the image
- It can detect any one of 1000 images

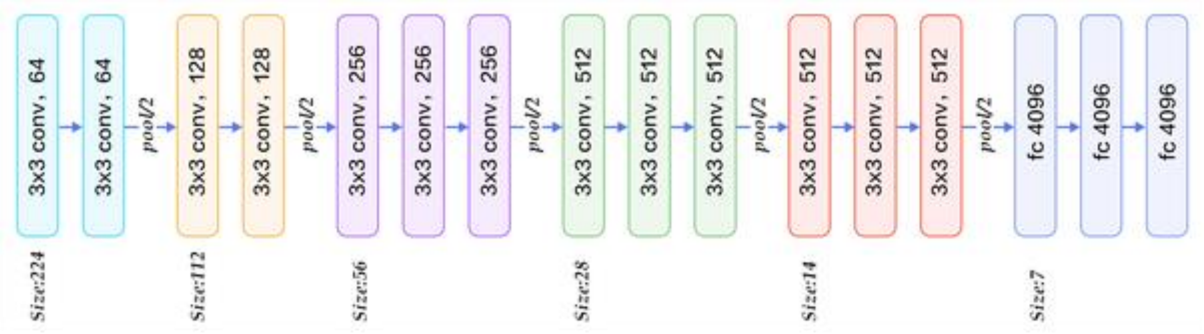
- It takes input image of size 224 * 244 * 3 (RGB image)

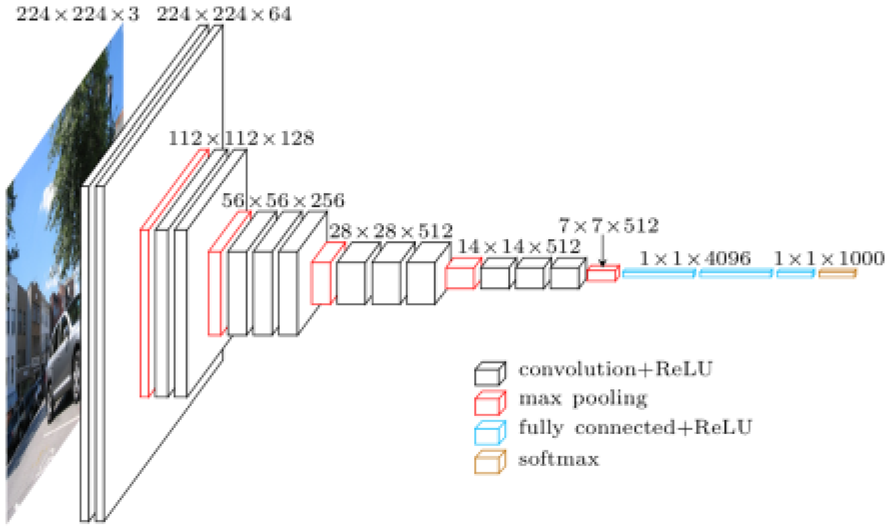
Built using:

- Convolutions layers (used only 3*3 size)
- Max pooling layers (used only 2*2 size)
- Fully connected layers at end
- Total 16 layers



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					





Transfer Learning

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task.

It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems.

Benchmark

The internal benchmark for this project from my side would be the following:

- The base bare metal model should be able to achieve an accuracy of more than 75%
- The processing time of the images, training and validation of the CNN model should not take more than 30 minutes on an AWS GPU enabled machine

III. Methodology

Data Preprocessing

Since the Data was organized in good format, and also the image capture was also in good quality, whitish background, with clear photograph taken, not much of pre-processing steps was required to do any feature extraction or feature generation. The existing pre-trained VGG16 model was itself able to generate the necessary features.

However, the image files had to be organized a little. This was to facilitate Keras to split the data into training and validation/test sets. The steps followed were:

- Create two folders under a parent folder, say "data", namely "train" and "validation"
- Re-create the folder structure, as from the original dataset into each of the above folders
- From the original dataset, pick 4 random files and move it to the validation (into the corresponding species) folder, and the rest (20 – 4) 16, into the training's corresponding species folder

Implementation

The algorithm was implemented in the following manner. The following two websites were referred mainly for the implementation.

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

https://www.codesofinterest.com/2017/08/bottleneck-features-multi-class-classification-keras.html

- Import the necessary Keras, OpenCV and Numpy libraries
- Initialize program parameters such as, train sample size, validation sample size, batch size for each iteration of each epoch, number of epochs
- Load the VGG16 pre-trained model of Keras
- Use the ImageDataGenerator and flow_from_directory features of Keras to generate the features Bottleneck features for the Train datasets, and store these features in a numpy matrix and store them on disk
- Do the above set for the Validation data set also

- Load the Train numpy matrix and store it in a variable
- Convert the species directory names from the train data directory into Categorical Variable
- Do the above step for validation data directory classes also
- Create a CNN Sequential Model
- Add a Flattened Layer to the model
- Add another Dense Layer to the model with 256 output units and a relu activation function
- Add a Dropout Layer
- Add another Dense layer with output unit as the number of classes and a sigmoid activation function. This is out last output layer
- Compile the above model
- Train it using the batch size, epochs train data and validation data
- Save the weights and model
- Use the Model's evaluate function to get the validation loss and accuracy of the model
- Plot the above in a epoch graph

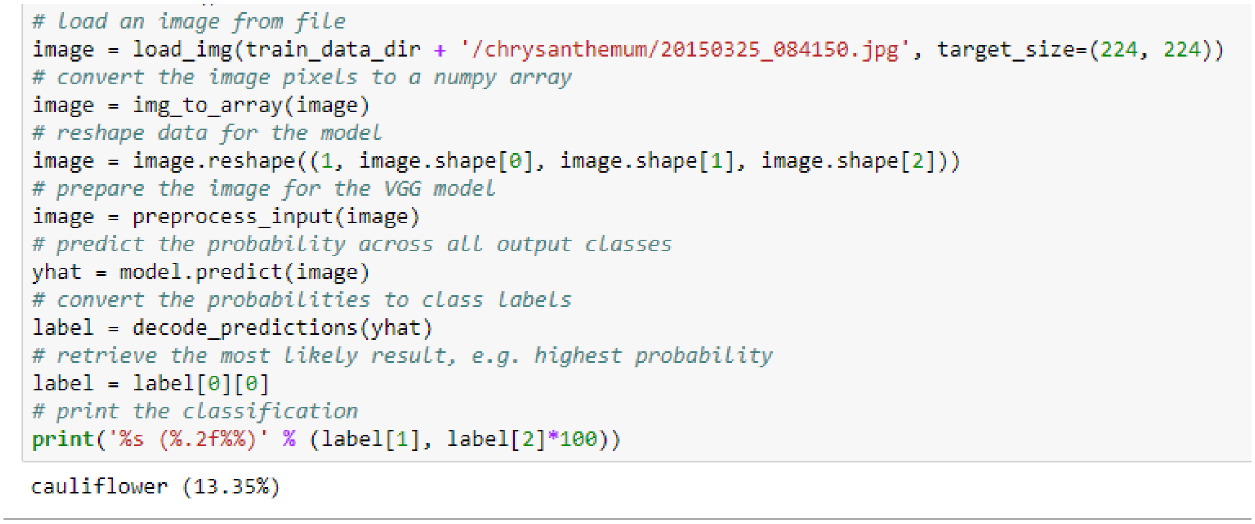
The following are few of the issues that were encountered and solved:

- The training and validation number of samples were calculated dynamically. This was creating a problem while adding the Flatter layer to the model (input_shape parameter was not being passed correctly. Hence, had to hard-code the sample sizes and adjust the input_shape parameter.
- In the Bottleneck features generations and saving functions, these were written for very balanced input samples. Hence, the calculation of the predict size train was not correct. The input number of samples and the predicted features sample (number or rows) were incorrect. The root cause of this was the calculation of the predict_size_train and predict_size_validation. These were corrected.

Refinement

I first tried to use the Vanilla VGG16 model to see if it could use its existing learned features to predict the species of the plant, or some plat like features. Unfortunately, it utterly failed. Please find below two instances of the trial.

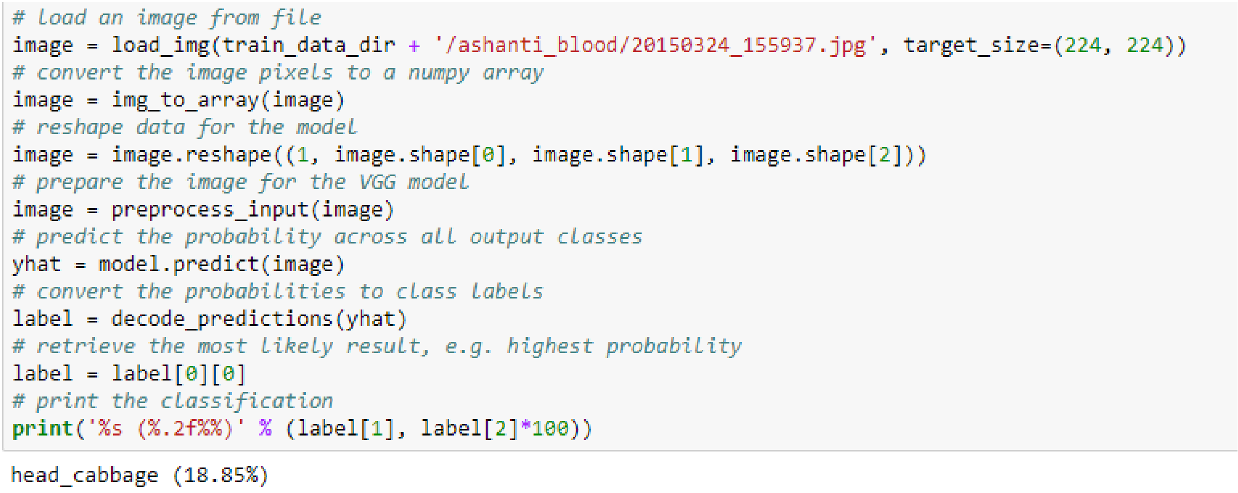
In the following trial, I was loading a Chrysanthemum leaves image, but the VGG16 model predicted it to be "cauliflower".



```
# load an image from file
image = load_img(train_data_dir + '/chrysanthemum/20150325_084150.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

```
cauliflower (13.35%)
```

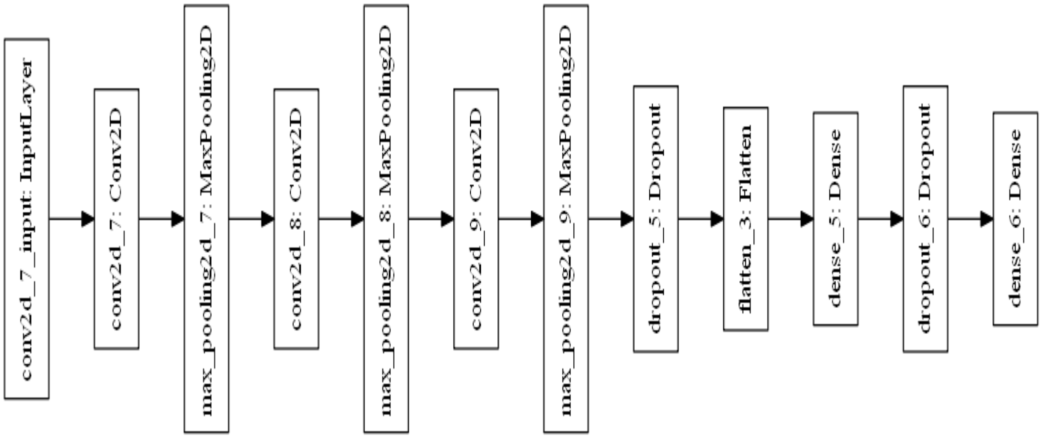
In the following trial, I was loading a Ashanti Blood leaves image, but the VGG16 model predicted it to be “head_cabbage”.



```
# load an image from file
image = load_img(train_data_dir + '/ashanti_blood/20150324_155937.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

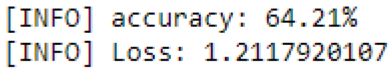
```
head_cabbage (18.85%)
```

The second trial I did was with a custom Conv2D model as shown below.

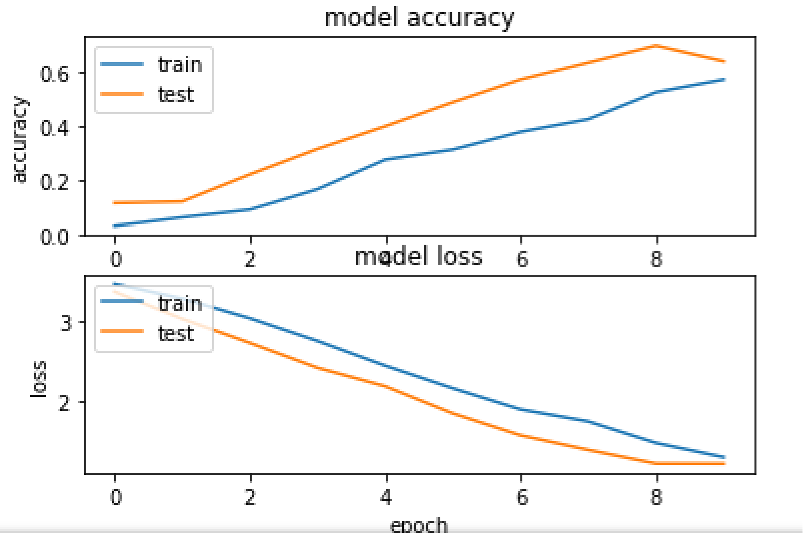


The accuracy and loss results with a three fold K-Cross Validation results are shown below:

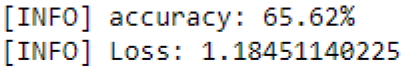
```
Running Fold 1 / 3
Train on 319 samples, validate on 190 samples
```



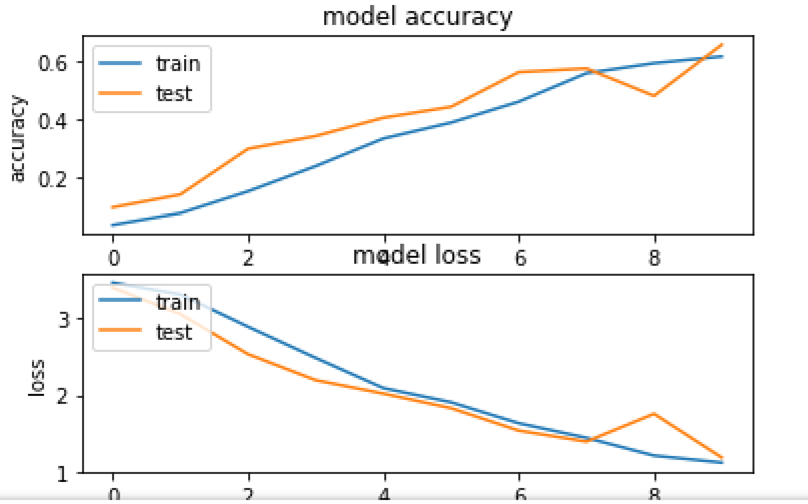
```
[INFO] accuracy: 64.21%
[INFO] Loss: 1.2117920107
```



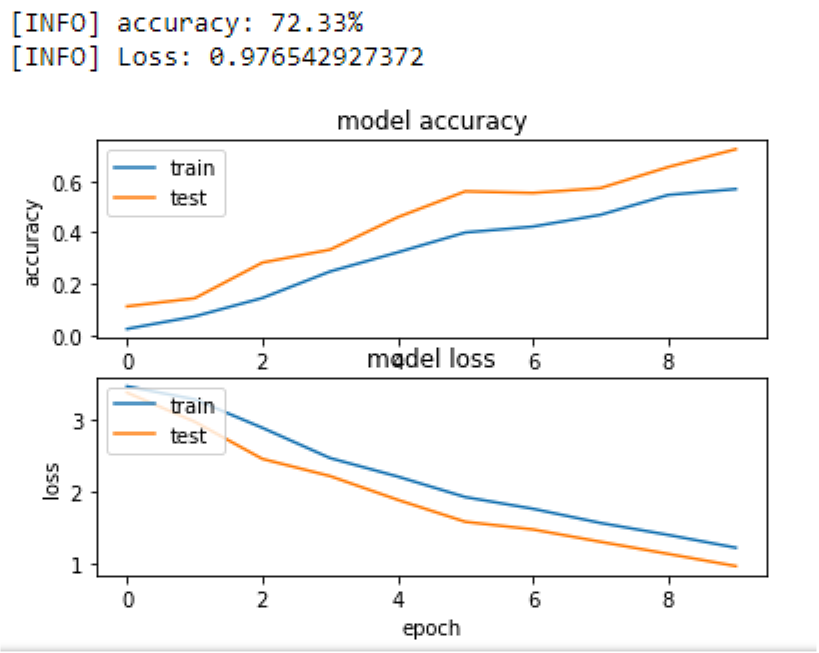
```
Running Fold 2 / 3
Train on 349 samples, validate on 160 samples
```



```
[INFO] accuracy: 65.62%
[INFO] Loss: 1.18451140225
```



```
Running Fold 3 / 3
Train on 350 samples, validate on 159 samples
```



```
[INFO] accuracy: 72.33%
[INFO] Loss: 0.976542927372
```

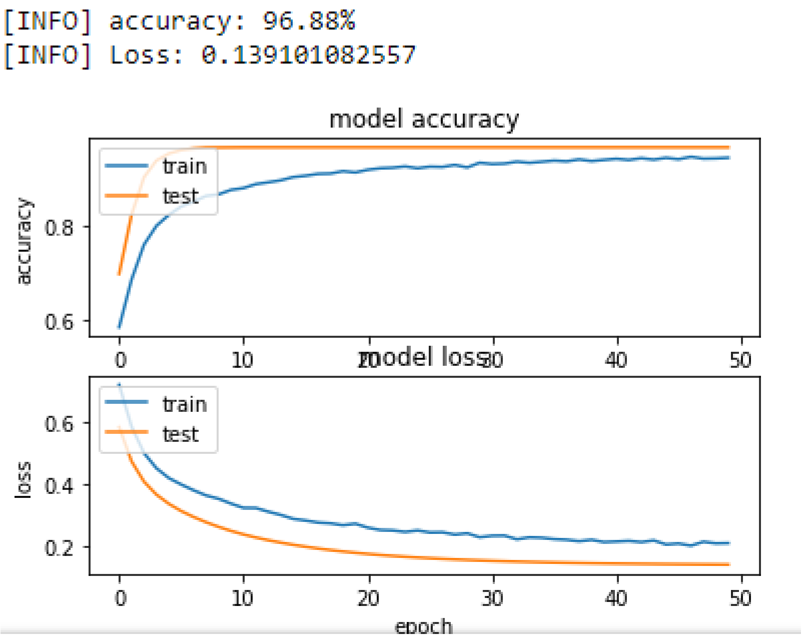
From the above results, it was clear that existing model will not work and the accuracy were also my internal benchmark of 75%. Hence, I moved onto the Transfer Learning technique.

IV. Results

Model Evaluation and Validation

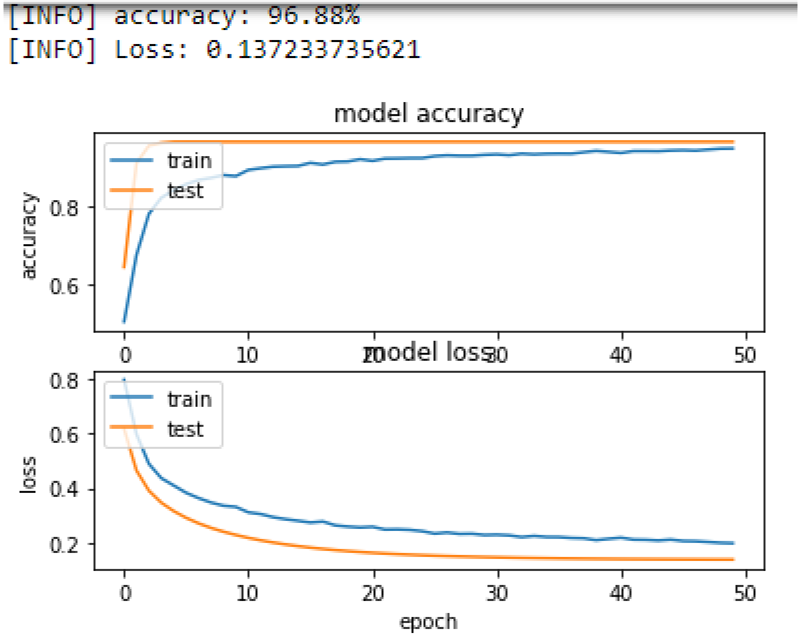
The final model was evaluated with a 3 Fold K-Fold Cross Validation. The results of which are shown below:

```
Running Fold 1 / 3
Train on 319 samples, validate on 190 samples
```



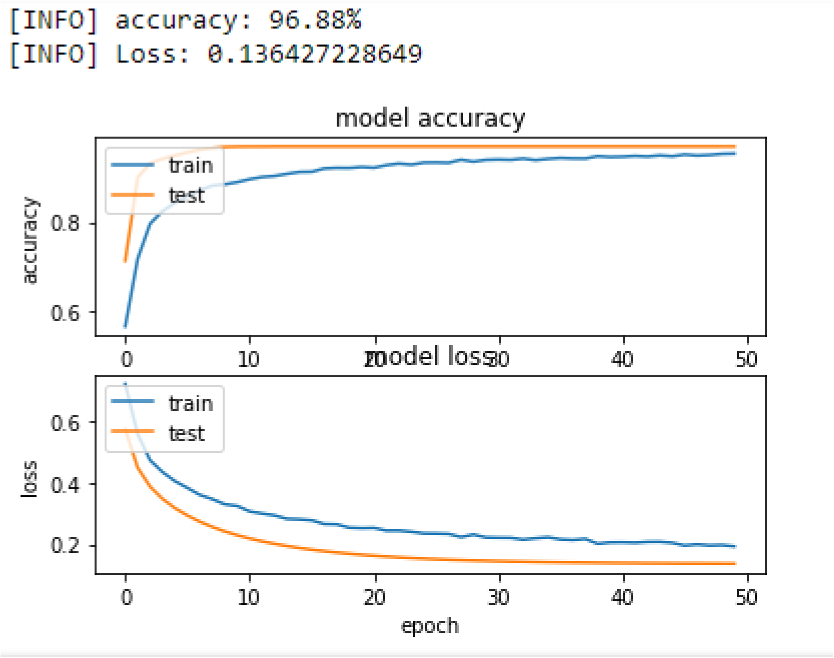
```
[INFO] accuracy: 96.88%
[INFO] Loss: 0.139101082557
```

```
Running Fold 2 / 3
Train on 349 samples, validate on 160 samples
```



```
[INFO] accuracy: 96.88%
[INFO] Loss: 0.137233735621
```

```
Running Fold 3 / 3
Train on 350 samples, validate on 159 samples
```



```
[INFO] accuracy: 96.88%
[INFO] Loss: 0.136427228649
```

In all the runs, the average accuracy is 96.88% and the average loss value is 0.137587349.

The final model has the following features and capabilities:

- This is the bare metal model using the pre-trained VGG16 CNN layers and using Transfer Learning Technique
- Has an accuracy of 96.88%
- Is very fast on a GPU enabled AWS system
- Seems to learn from a small epoch run of 21-23 runs and have a low validation loss of 0.1375

Justification

In the benchmark solution, http://www.ai.rug.nl/~mwiering/GROUP/ARTICLES/plantaugment.pdf, if we refer the section 4.1, Table 2, the Original (no flip) Scratch accuracy results with both AlexNet and GoogleNet are lower than 90%.

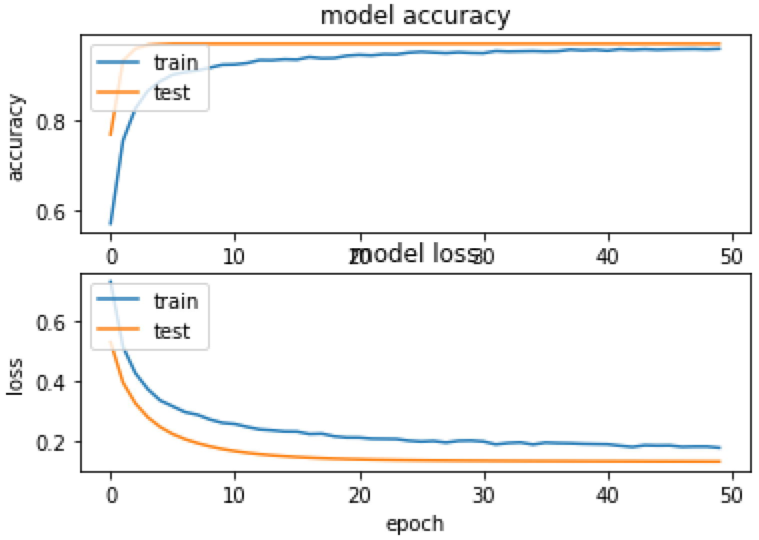
In my implementation, with the bare metal model, and no pre-processing, I achieve an accuracy of 96.88%. Hence, this model should be better than the above said benchmark model.

If we, do some extra pre-processing, and do fine tuning, the model should be able achieve equivalent or better results than the benchmark model.

V. Conclusion

Free-Form Visualization

Please find below the Model Accuracy and Model Loss against the epoch run for the final model.



Reflection

The problem that I wanted to solve was of, Plant species classification by using the leaf samples. Neural Networks with computer vision can be used to solve the classification problem. I was trying to implement a Machine Learning technique/algorithm, especially a Deep Learning technique. the algorithm will be able to provide an accuracy score based on the training and test/validation data based on it's learning.

The steps followed are:

- First get a dataset that has good images taken of the leaves of various species
- Organize the data in proper folder for training, validation and classification

- Pick some random 20% of the data and assign it as validation/test data
- Download and Load a pre-trained VGG16 model file
- Run the training data and validation data through the VGG16 model, and store the resultant matrix is some format
- Create another Sequential CNN model, add a Flatten and Dense layer, and an output layer with sigmoid activation function
- Train this newly created model, and validate the model against the validation/test set
- Report the accuracy and loss values

The interesting part while doing this project was, how a pre-trained model, even though whose base idea was different, could be used for a different purpose.

Another interesting part was the bare metal model, could produce a such a good results.

The final model fits my expectations for this project work based upon my understanding of Deep Learning and CNN, and can definitely be used to solve these kind of problems.

Improvement

A few improvements that I can propose or think of is:

- More data could be added to improve the accuracy
- Will have to fine-tune the model by adding some more layers or using some other loss entropy calculations or other optimizers
- Will like to pre-process the data and check the effects of rotation, scaling, illumination, etc.
- Will like to pre-process the data and apply a Ridge Filter using the Hessian Matrix calculation. This will identify the vein structure of the leaf, and then this image files can be fed into the CNN model

References

https://arxiv.org/pdf/1409.1556.pdf

http://deeplearning.net/software/theano/library/tensor/nnet/nnet.html#theano.tensor.nnet.nnet.binary_crossentropy http://deeplearning.net/software/pylearn/formulas.html#pylearn.formulas.costs.binary_crossentropy

https://github.com/keras-team/keras/issues/6444

https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

https://github.com/keras-team/keras/issues/3296

https://www.codesofinterest.com/2017/08/bottleneck-features-multi-class-classification-keras.html

https://stackoverflow.com/questions/47535596/how-do-i-get-the-true-labels-when-i-use-a-imagedatagenerator-flow-from-directory?rq=1

https://stackoverflow.com/questions/45806669/keras-how-to-use-predict-generator-with-imagedatagenerator

https://keras.io/preprocessing/image/

https://www.kaggle.com/keras/vgg16

https://archive.ics.uci.edu/ml/datasets/Folio

https://www.kaggle.com/c/leaf-classification#description

https://www.quora.com/in/What-is-the-VGG-neural-network

http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy

http://www.explainthatstuff.com/introduction-to-neural-networks.html

https://machinelearningmastery.com/transfer-learning-for-deep-learning/