2019

Species Classification Using Computer Vision

RESEARCH PROJECT B BYABIR BHATTACHARYYA
GUIDED BY- PROF. DR. RONNY HARTANTO

HOCHSCHULE RHEIN WAAL | ID:24107

Contents

Acknowledgment

I would like to thank my parents. They have supported me throughout my entire life encouraging me to push myself more to become a better person. Without their support I would not be writing this in the first place.

Right after them comes Doctor Professor Ronny Hartanto who has been my guide and mentor since the day I have enrolled in the prestigious Hochschule Rhine-Waal University of applied Science. Starting from my learning agreement till today, he has been guiding me patiently with all my doubts, queries and insane ideas. Without him I wouldn't have found this wonderful field of machine learning or my passion in this field.

I would also like to mention Doctor Professor Achim Kehrein for encouraging me to learn programing and helping me with some valuable resources.

Finally, I would like to thank my project partner Lennart Jensen who collected the dataset for this project and all my other friends and professors of the department of technology and bionics for helping me to complete the project in time.

A special thanks to Miss Elisabeth Banken for patiently hearing out my really long drawn, wildly vivid and extravagant ideas out of my imaginations and putting them down as politely as humanly possible. Also for being a good friend throughout not only my university days but all my good and bad days.

Lists of Abbreviations used

AI- Artificial Intelligence
DL- Deep Learning
DRL-Deep reinforced Learning
ML-Machine Learning
MLP-Multi-Layer Perceptron
ROI-Region of Interest
GPU-Graphical processing unit
CV- Computer Vision
IDE- Integrated Development Environment
CNN- Convolution Neural Network
CNTK-Cognitive toolkit
HSV- Hue Separation Value
CPU-Central processing unit

List of Tables

List of Figures

Preface

This is a continuation work of my previous research project A which was titled “Drone based species classification”. As a quick recap in the previous project, a detailed literature review was explained about how a drone based species classification work and how it has become an important tool for researchers for species classification and bio-diversity identification all around the world.

Along with the literature survey some of image pre-processing work had also been done such as pixel manipulation for successfully taking out a particular species. The algorithms however were really basic such gray scale imaging and pixel manipulation. Matlab was used to perform all the algorithm the results and details can be shared upon request.

In this project we step up our game, instead of just pixel manipulation, the field of computer vision is highly focused upon. Starting from algorithm used from pre-processing of an image to forming deep neural network to classify a plant with a really good accuracy. This project takes a deeper dive on the machine learning aspect of the process of classification. All the algorithm in this project are however tested on python. A quick installation guide of the environment and libraries used are explained in the chapter pre-installations if anyone reading this wants to try on their own. However, snippets of codes are given instead of the whole code. So, if anyone is interested in knowing the details can contact me or my guiding professor Dr. Ronny Hartanto without whom I would not have been introduced to the wonderful world of computer Vision.

A lot of the basics are taken from the book “Computer Vision- Algorithms and Applications” by Richard Szeliski. While the algorithms are formulated by using various python 3.6 libraries and there documentations.

1. Introduction

As mentioned in the preface a lot of the work is carried forward from the previous work which was done. In this report the field of computer vison and artificial intelligence is heavily focused. A lot of algorithms on python are tested and tried whose results. The goal of the project is to classify two or more species using machine learning techniques specifically deep learning. The difference between AI, ML and DL is addressed in one of the chapters. If you are wondering if they are the same, they are not and we will see why. The report is basically divided into the following sections-

1. Introduction and pre-installation.

2. Pre-Processing of images using python.

3. Deep Learning for species recognition using python.

4. Real Time classification using advance computer vision techniques.

The goal of the project is to successfully classify species of plants using deep learning techniques and in doing so obtain a high accuracy. As we speak, further work is being going on to include the algorithm to make real time detection using advance computer vision algorithms such as You only look once, haarcascades and single shot detection. Although, the results might not be available before the report submission deadline but the mechanism and theory behind the algorithms are explained at the end.

The reason behind this project is we want to see how precisely we can classify species. If we are being able to do so in real time we can use it for various purpose such as autonomous drones for species classification, detecting diseases in plants, and detecting weeds or unwanted plants in a cultivation.

1.1 Pre-Installation

All the algorithms are performed on Python 3.6. If it is installed in the system along with the list of libraries then the algorithms can be tried out if not it is advisable to follow the instructions below.

1.1.1 Anaconda Distribution

Anaconda distribution has been used to install python. It is recommended because of the ease of forming the environment. The software can be download from this link here- https://www.anaconda.com/ . Please note that they have updated to python 3.7 but tensorflow currently (as of 22.2.2019) supports upto 3.6 if you have trouble installing tensorflow please make sure that the version of python installed is correct by running in the terminal.

```
conda search python
```

If your version is 3.7 or higher downgrade by typing the following code in the anaconda prompt.

```
conda install python=3.6.0
```

Once it is installed the other libraries installation and descriptions are given below-

1.1.2 List of Libraries

1. Numpy
2. Pandas
3. Open CV
4. Sci-kit learn
5. Matplot lib
6. Keras
7. Pytorch

1. Numpy- It is a basic tool for scientific computing and good to work with matrices and n-dimensional arrays. Since pictures are basically 2-D arrays of numbers it is a really important to install it [1]. Running the following code in the conda(all the codes below needs to run in the conda prompt) prompt will install it automatically.

```
pip install numpy
```

2. Pandas- Pandas are a data analysis tool which can be used to visualize the arrays in data frames and a few useful operations can be done easily using pandas. This library is not used that much in this project but it is advisable to install it as it is heavily used in machine learning specially if the labels of the picture are given in csv or tsv form [2]. The below code installs pandas.

```
pip install pandas
```

3. Open CV- This is the most important library as the project uses a lot of pre-processing tools from this library. It has got all the image processing functions of Matlab and can be used both for C++ and python [3]. The below code installs it.

```
pip install open-cv python
```

4. Sci-Kit Learn- or sk-learn as popularly known is a powerful machine learning tool and contains all the algorithm to implement. There are a lot of algorithms to choose from to make a machine learning model depending on our data set [4]. The below code installs it.

```
 pip install -U scikit-learn
```

5. Matplot-lib- This is a third-party python library which is used for visualization of data and also give reference coordinates in images [5]. To install

```
pip install matplotlib
```

6. Keras- This is perhaps the most important library of them all. This library enables the user to implement an idea to a deep neural model fast. This runs on tenserflow, theano or CNTK as a backend. These are different libraries which have longer and complex rules to work with. Thus, this libraries simplifies the work a lot [6].
Important note the current version of python supported by this as of 22nd February is python 3.6. Below is the command to install it.

```
pip install keras
```

7. PyTorch – This is a really useful library for deep learning in research and quick prototyping of deep neural network. To install it one needs to go to the home page https://pytorch.org/ . A specification with a unique conda installation will be given based on the system specs. For this project the system specs were [7]-

OS: Windows, Package: Conda, Language: Python 3.6, CUDA: 9.0. Based on this the installation code was-

```
conda install pytorch torchvision cudatoolkit=9.0 -c pytorch
```

With these prerequisites cleared all the experiments which were done are explained in the following chapters.

Apart from this there are two important IDE's are used namely Jupyter NoteBook and Spyder both of these IDEs are good as machine learning tools as data variable storage and visualization can be done in great details.

In the next chapter we will discuss the preprocessing of images. This is necessary in the deep learning phase for image augmentation. Since the amount of images are limited due to space or limited number of dataset, image augmentation is used to increase the number of samples by introducing variation and convolution in it. These concepts are explained in details in the next chapters.

2. Image Pre-processing

For image pre-processing, we mainly use the open CV library and matplot-lib in python. All the algorithms are inspired from the book- “Computer Vision- Algorithms and Applications” by Richard Szeliski. The series of experiments involve

1. HSV conversion
2. Binary Bit wise Operation
3. Convolutions- Filters, Canny
4. ROI extraction.
4. Edge Detection.
5. Hough Transform.

The results of one image is shown for the entire process.
To load the images we have to first import the library and use the following commands-

```
import cv2
img1=cv2.imread(“grass.jpg”)
img2=cv2.imread(“pipe.jpg”)
```

Figure1:(Left) Img1 as grass.jpg and (Right) Img2 as pipe.jpg

2.1 HSV Image

As we know images are basically a combination of pixel. Each pixel is composed of red, green and blue values. This combination can be represented in another way called hue, saturation or intensity or value. This gives an abstract value of the various shades that may be present in an image along with the intensity. Below is the code for hsv applied to the previously recorded and loaded image.

```
hsv1=cv2.cvtColor(img1, cv2.COLOR_BGR2HSV)
hsv2=cv2.cvtColor(img2,cv2.COLOR_BGR2HSV)
```

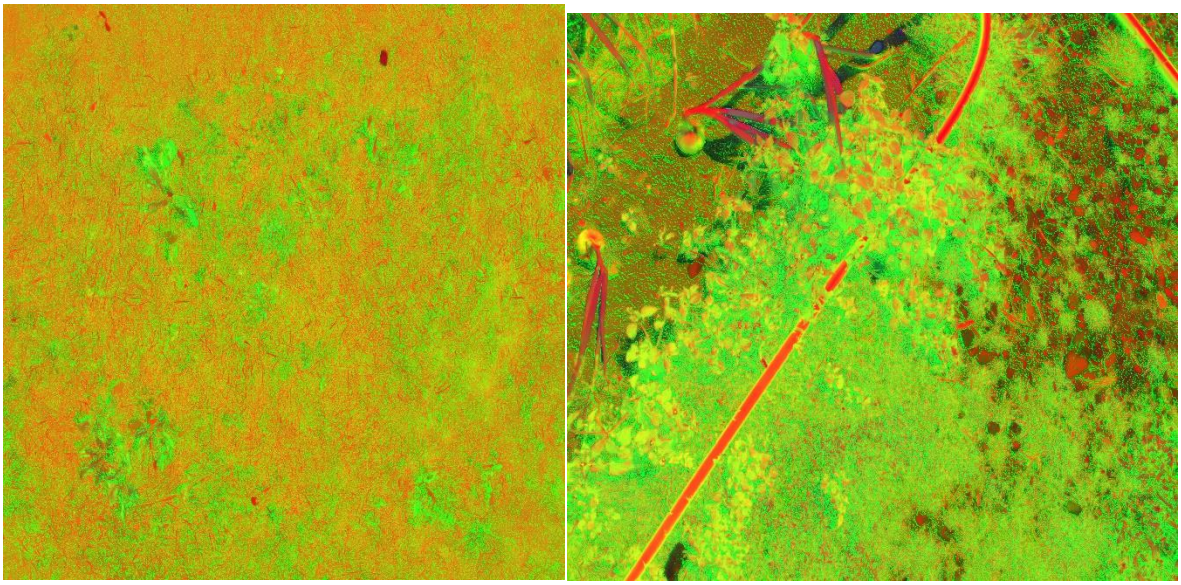


Figure2: (Left) HSV of grass.jpg and (Right) HSV of pipe.jpg

2.2 Binary Bit Wise operation

In these operations there are only two values given that is 0 and 1. In this the entire image is converted to pixels composed of 0 and 1. For this to take effect we must do what is called thresh-holding. It is the process by which we choose a range within which the pixels can be either one or zero. We can see the result below-

```
ret,thresh1 = cv2.threshold(gray1,90,180,cv2.THRESH_BINARY)
ret,thresh2= cv2.threshold(gray2,90,180,cv2.THRESH_BINARY)
```

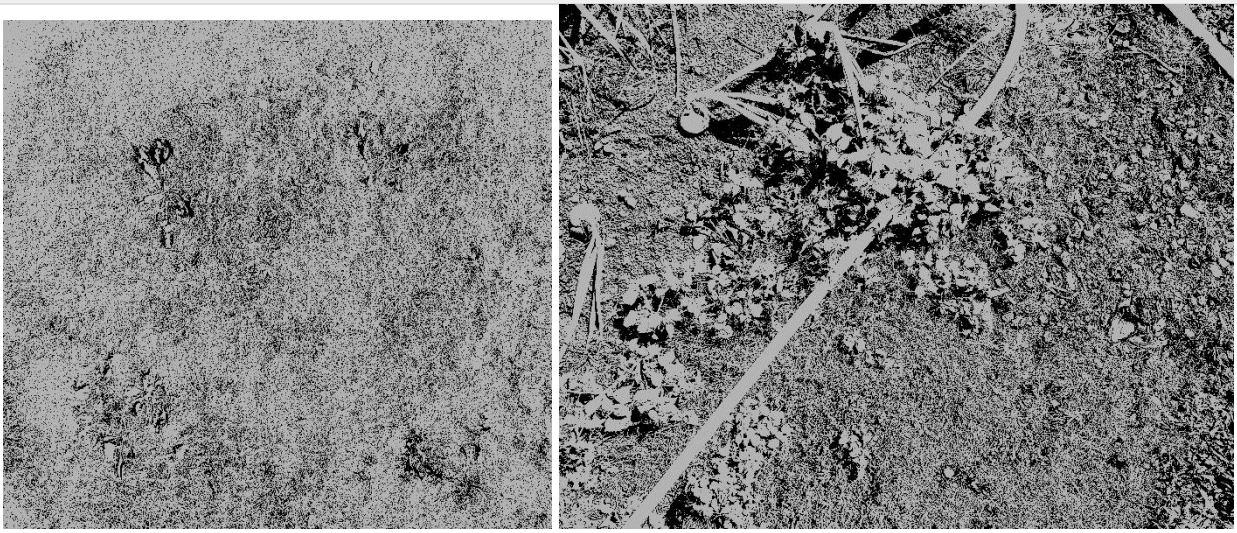


Figure3: (Left) Binary image of grass.jpg and (Right) pipe.jpg

The variable gray1 and gray2 indicates that the image is converted to grayscale first. The integers 90 and 180 indicates the range in which we want to convert the pixels to 1 which indicates white and the rest is 0 indicating black.

2.3 Convolution

Convolution might be a reoccurring theme throughout the rest of the report. Convolution is a special operation in which a selected kernel is convolved with the original image to get variety of results. Convolution is not entirely matrix multiplication but it is matrix multiplication with addition. Something like a scalar product but instead of just rows or columns the entire matrices are multiplied with the corresponding element and all the elements are summed over . For example we can choose a kernel of 3X3 matrix to convolve with our 11X11 matrix this will give us the same 11X11 image but with its feature somewhat enhanced. Two types of convolution is explained in this section. There are a lot more to choose from. [8]

2.3.1 Gaussian Blur

It is a smoothing filter which when convolved with our original image will give us a smooth image by decreasing the edges. The gausian kernel is a 5X5 matrix of the form [3]-

$$K = C * \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Where K= kernel; C= constant can be any positive or negative floating point number. This kernel is then scanned throughout the original image making one or two step iterations each time till it scans the entire image. Which can be seen below.

```
blur_img1=cv2.GaussianBlur(gray1,(5,5),0)
blur_img2=cv2.GaussianBlur(gray2,(5,5),0)
```

Here gray1 and gray2 are the grayscale image. The Gaussian Blur is the function which forms the matrix and applies it to the image. (5, 5) is the matrix dimension and 0 represents the standard deviation. The results are shown below-

Figure 4: (Left) smooth grass.jpg and (Right) Pipe.jpg

2.3.2 Canny Image

Canny is a popular edge detection kernel which when convolved with the gray scale image gives the edges. It works on the principle that there is always a change in the intensity along the edges in a picture. This change in intensity is called a gradient and the canny edge detects all these gradients to successfully detect the edges. It works in the principle that there is an intensity gradient change at the edges of any picture. When applied with the right kernels they can be effectively used to detect the edge. In canny edge detection at first the Gaussian blur kernel is applied which is then acted upon by another kernel [8].

```
def make_canny(image):
    gray_img=cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    blur_img=cv2.GaussianBlur(gray_img,(5,5),0)
    canny_img=cv2.Canny(blur_img,20,180)
    return canny_img
```

Here we make a function called make_canny which takes in one argument which is the original image. The image is first converted to grayscale, a Gaussian blur is then applied to it and finally the function Canny which is provided by the CV2 library is acted upon. The arguments 20, 180 is the threshold limits which the algorithm takes to detect the edges successfully. Tuning these two parameters can yield better or worse results depending on the quality of the pictures.

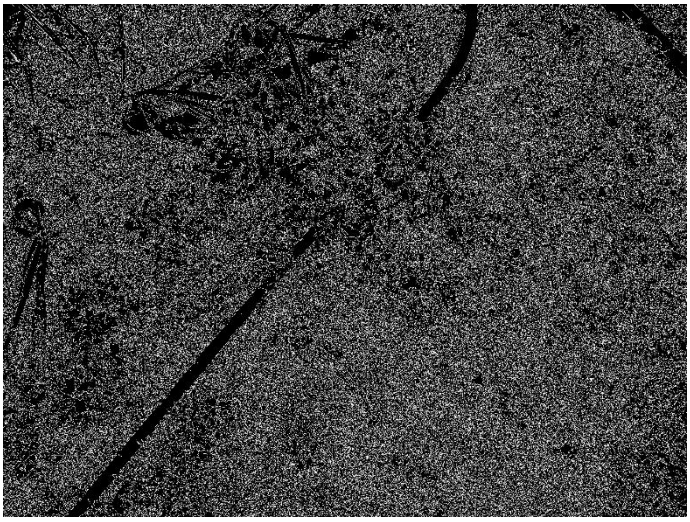


Figure 5: Canny edge detection on pipe.jpg

We apply the canny edge to only in one of our image because there is no particular edged to be detected. As wee cans see the pipe is detected very clearly along with some leaves and onions in the side. In the other image as everything was fused we got dark leaves in sparse spots in the picture as shown in figure 6. But this proved to be very successful as it was able to detect the plants of our interest.

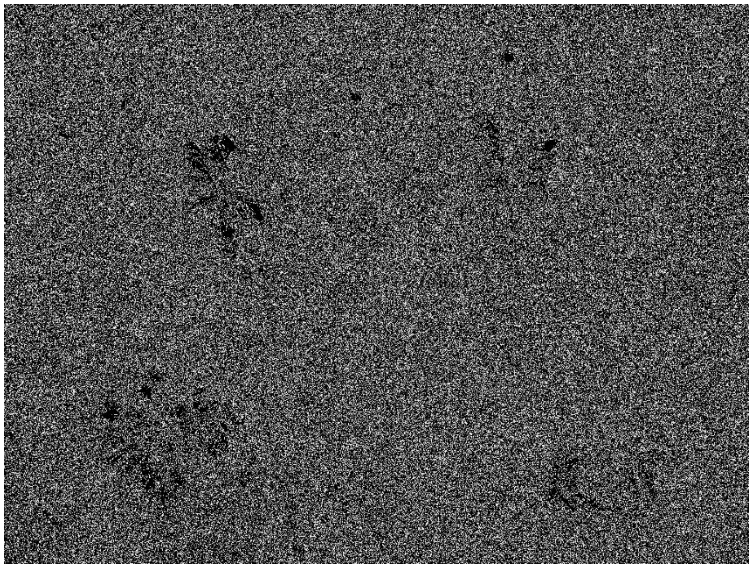


Figure 6: Canny edge of grass.jpg

As seen above the algorithm with that threshold was able to take out the leaves and showed where the species is present in the picture. We were not expecting this result as we thought that it would be difficult for us to take out the edges within a fused mass as there would be no real gradient present but we were proven wrong.

2.4 Region of Interest

This is perhaps one of the most important pre-processing algorithm. In an image there may be redundant data along with import information. We want to focus our attention to the region which contains the most relevant data. For this we need to be able to extract the region in our picture as shown below.

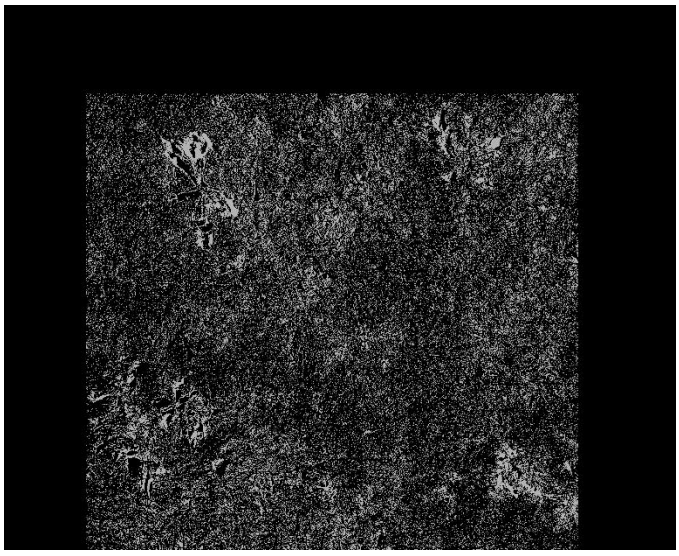


Figure 7: ROI of grass.jpg.

Here we have taken the image in the form of a square but we can use it in the form of circle, triangle or any other form of polygon. The code below shows how it was achieved.

```
def roi(img):
    h=img.shape[0]
    rectangle=np.array([[(500,h),(500,500),(3500,500),(3500,h)]])
    mask=np.zeros_like(img)
    cv2.fillPoly(mask,rectangle,255)
    masked_img=cv2.bitwise_and(img,mask)
    return masked_img
```

We again make a function for our convenience calle roi. It takes one argument which is our image. The variable rectangle is a numpy array of the coordinates which we are interested in. As we know an image coordinate systems origin is placed at the top right corner of the image and the positive x-axis extends length-wise towards right while the positive y-axis extends downwards height-wise.
The parameters in the variable rectangle gives the co-ordinate points of the corners of a rectangle which is our region of interest.
The variable mask is basically a black image of the same size as our original image.
Fill poly is the function provided by CV2 library. The masked image is then added to the original image using bitwise and operation. To know how a Bitwise and operation works please take a look at the appendix section.
At the end we get the region we are interested in.

2.4 Conclusion

Image preprocessing techniques are very useful especially if we are dealing with some highly sensitive or noisy data. Also they prove to be a valuable way to evaluate static images or video which do not have that many changes. Although the techniques mentioned here are not explicitly used during the building of the architecture of the convolutional neural network but are used as hidden parameters during the training of the images. It is therefore good to be familiar with these process specially the concept of convolution.

One important pre-processing technique called Hough-Transform has not been described in this section because we do not need to take out straight edges from our image. The Hough-transform is very useful during lane detection or connecting markers. A brief overview is given in the appendix section.

In the next section we will dive right into the deep learning aspect of the project.

3. Deep Learning

Before we move into convolution neural network it is important to understand the concept of deep neural network specifically the concept of perceptron first. As it is the basis of any machine learning algorithm. This chapter might be a bit theory based but is one of the most important chapter for the project as all the core concept of computer vision using deep learning is encompassed here.

3.1 The Perceptron

Just like the nerve cell in human brain in artificial neural network uses a perceptron to process information. The schematic diagram of a simple perceptron is shown in figure 8.

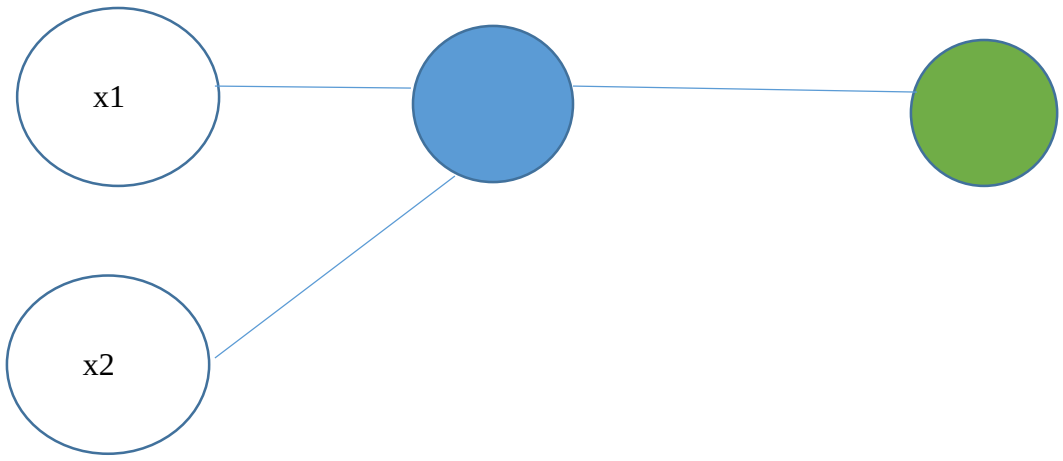


Figure 8: The perceptron

In a simple perceptron there are three layers the first layer which is in white is called the input layer. This is usually the variables in our dataset. The middle blue circle is called the hidden layer. In the hidden layer usually the weights w1 and w2 are multiplied with the input node and summed after which a function is applied to get a particular value and standardize the result. This is then given out in the output layer. [9]
This is the simplest explanation of a perceptron. All data scientist and many researchers have compared this perceptron to a brain neuron cell as its functionality is more or less similar. However unlike electrochemical action reaction in a neuron in a perceptron it is more math specifically regression and error minimization to give an optimized output prediction.[9]

In all deep learning application the perceptron forms the heart of all neural networks. Be it self-driving cars or an algorithm to predict the stock market, neural networks has seen a great spike in applications in the past decade. With modern fast processors and technical advancements it application has become countless. [9]

In a deep neural network the only difference with a perceptron is that it involves more hidden layers and the number of weights are significantly increased. The two main mathematical concepts are regression and error minimization which are described briefly in the next sections.

3.2 Regression

It is a very important concept in statistics. Given a set of points there exists a line which passes through the points fitting or going through as many points as possible and separating the points into definite regions [10]. As shown in fugure 9. Let us have the following data points say-

Table1: Data points

X1	X2	Label
2	0	0
4.5	.5	0
6	.5	0
6	2	0
1	1.5	1
2	4	1
6	4	1

Let us assume the weights w1=-.43 and w2= .34 and bias b= 1.26. These are randomly selected and the result obtained is as shown in figure 9 below-

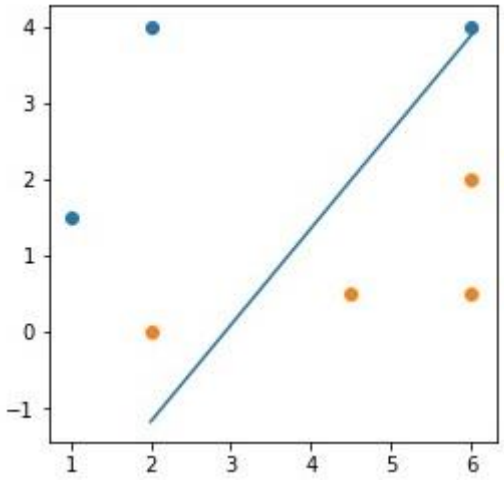


Figure 9: Regression Plot

The plot is given by the equation [10]:

$$y = w1 * x1 + w2 * x2 + b$$

In general if there are more than two inputs the general formula can be given by [10]:

$$y = w1 * x1 + w2 * x2 + \cdots + wn * xn + b$$

As we can see above some points are misclassified the main goal of a neural network or any machine learning algorithm is to reduce this error using different processes some of the process include gradient descent and cross entropy. We will discuss the process of gradient descent in the

following section. But before we move to the error minimization we take a look at some actiation function and there use.

3.3 Activation Functions

As we can see plugging in X1 and X2 in the regression equation doesn't yield us a value between 0 and 1 it can be any value. To make the value between 0 and 1 we need special functions these functions convert the result to the appropriate value accordingly. The math behind each of the function are not defined but the use of a few typical activation functions are discussed [9]

1. **Step function**: This is a simple function it takes in a threshold say in our case any value above 2 will be treated as a 1 and anything other wise will be zero.

$$f(x) = \begin{matrix} 1 \; if \; w1x+b>2 \\ 0 \; otherwise \end{matrix}$$

2. **Sigmoid function**: The step function is not that ideal for doing gradient decent as there is no gradient change with this function but a steep spike at one spot. Therefor other activation functions are used. Sigmoid function uses exponential function and gives a value between 0 and 1.

$$f(x) = \frac{1}{1 + e^{-x}}$$

3**. Radial Basis Function**- This is another type of activation function which is used to smooth a function which is not linear. A number of such rbf functions can be added to fit polynomial functions. It takes μ as mean and σ as variance.

$$f(x) = \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right)$$

4. **Relu Function**- One major drawback of sigmoid function is that the curve can be manipulated to give a steep result. Doing so may make our model stuck at some local minima and we won't be able to find the global minima. For this relu function is being used. The value is 0 when value is 0 or less than 0 and the value itself other wise. It is like a proportional function. This is very popularly used in convolutional networks.

$$f(x) = \max(0, x)$$

3.3 Gradient descent

We will not focus on the math behind the gradient descent but try to break the steps involved as simply as possible. In any machine learning algorithm the main task is to decrease the error of the misclassified points. In the misclassified point if the activation function has an output of 0 but the actual label of the point is 1. The point will try to push away the line while for the other case the point will be try to be as close to the line as possible. This is done by looking at the error [9][12]. There are mainly two ways of looking at the errors one is the mean of the sum of absolute error while the other is the mean of sum of squared errors. In both the cases the main goal of the algorithm is to adjust the weights of the line in such a way that final result has the minimum error that is possible [11][12]. Although in one forward pass of the perceptron we get one error and based on that error a step is taken in any direction if in a few steps the error remains same then the algorithm have reached the minimum value but sometimes if the activation function used be like step function it will be impossible for gradient descent that is why other functions are used instead of step function in machine learning [12]. From the above figure we apply the gradient descent and after a few epochs we see the final result in figure 10.

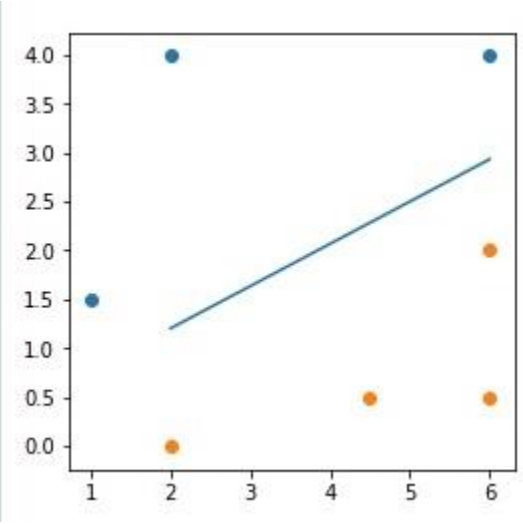


Figure 10: Gradient descent

Thus as we can see a clear cut separation of both classes of points. This is more or less the whole concept of a perceptron later we will see how these concepts evolve in deep neural network. A connection of two or more perceptron is called a multi-layered perceptron it is a subset of deep neural network. The way it differs is by gradient descent which we will see in the next section.

3.4 Deep Neural Network

A deep neural network consists of more than one hidden layer and instead of one output there may be multiple outputs. It can also be termed as multilayer perceptron as it is basically collection of number of perceptrons but there is a subtle difference between the two which results in falling of the MLP inside DNN. The major difference is in a deep neural network there occurs what is known as back-propagation instead of gradient descent. And also in a deep neural network an array of multiple weights are adjusted instead of just which are connected to it. Thus making it a bit computational expensive [9]. A structure of DNN is shown in Figure 11.

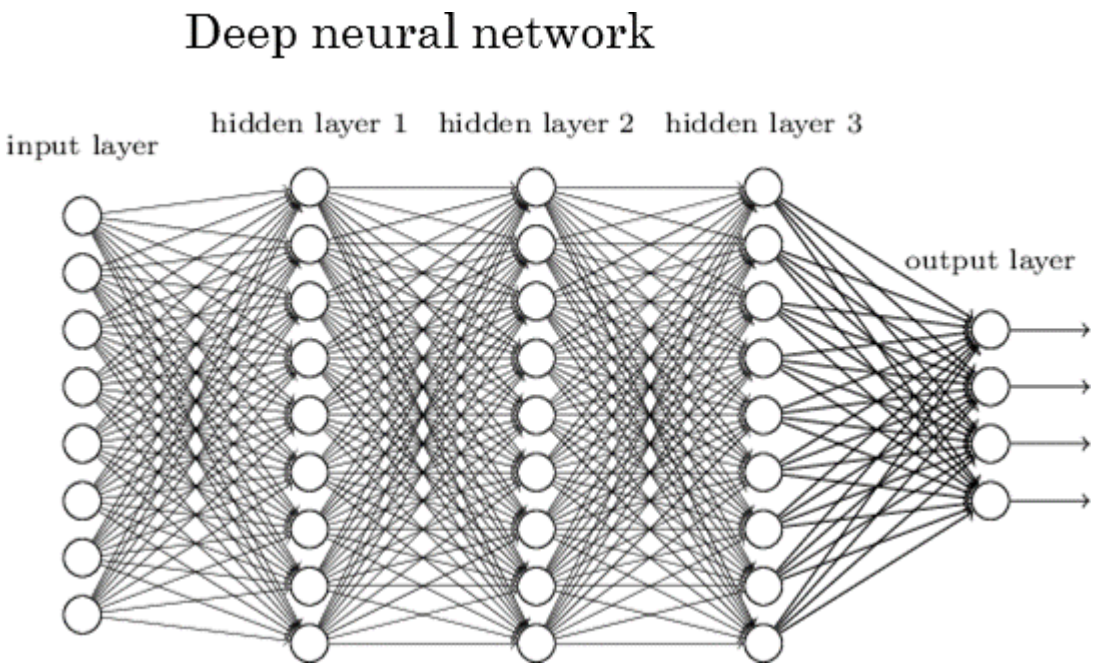


Figure 11: Deep Neural Architecture (Source:dataworrior.wordpress)

As we can see in a deep neural network there is a dense architecture unlike a simple perceptron, with each node connected with other nodes and thus forming a complex structure. The main goal of a deep neural network is similar to that of a perceptron that is to decrease the error in predicting the output. However, this is done by backpropagation which is like gradient descent but a bit complex and is known as cross-entropy error.[9][13]

3.5 Conclusion

In this chapter we briefly skimmed through some of the machine learning concepts we have used. This was not even the tip of the iceberg. During the research phase for this project we encountered and applied countless machine learning algorithms first before moving into deep learning. These algorithms and techniques are not included in the report as they were not related to our projects as machine learning algorithms deals with data which are in tabular form such as the set of point's example shown here in this chapter.

Although one might argue saying it's the same but machine learning paves the way for deep learning and should not be mixed. Deep learning in turn has different branches in turn based on the field of application. There are heavy research going on in the field of voice recognition, natural language processing, text to speech, autonomous robots and of course computer vision. We will see how deep learning is used in computer vision in the next chapter.

4. Convolutional Neural Network

This is a special kind of neural network. A convolutional network consists of a deep neural network attached to at the end. Convolutional network is a special kind of deep learning which is applied heavily in computer vision. It has been used for a variety of purpose. From differentiating detecting skin cancer with better accuracy than a dermatologist to self-driving cars convolutional neural network are being popularly used to solve repetitive tasks as well as innovate better and newer technology. [14]

The heart of a CNN is obviously convolutions which was discussed in chapter 2. This is done mainly to reduce the spatial information that an image provide. As we know an image consists of number of pixels arranged in 2-D space they may have a single dimension such as grayscale image or may consists of 3 channels which includes red, blue and green channel. Thus, as we can see in a single image there can be easily more than a million data points available. Which can be time consuming to process even one image. Thus, the convolution layer along with pooling layer helps to find relevant pattern in the pictures so that only the relevant pixels are passed into the deep learning layer. This reduces the number of input node that we saw in the previous chapter significantly and can reduce the training time remarkably.[14] A typical architecture of a CNN is shown in figure 12.

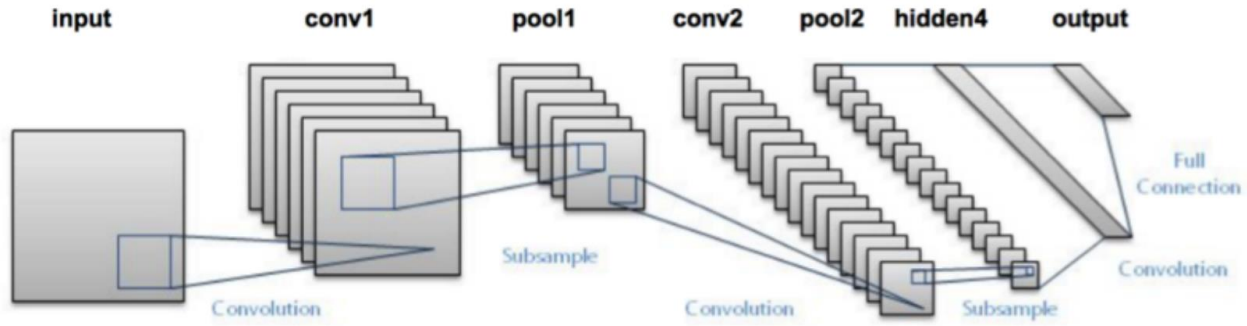


Figure 12: CNN architecture (source: blog.goodaudience.com)

We will now break down each layer.

4.1 Input layer

It is the input image it can be a small in pixels like the MNIST data set which is only 28X28 and has only one channel [16]. This can be used to directly and can be trained in a so called multi-layer perceptron. It is nothing but the fully connected layer or the deep neural network at the end of the picture. There is no need of convolution in such a simple case. Thus, depending on our dataset the architecture of a deep neural network varies significantly.

4.2 Convolutional Layer

As described earlier in a convolutional layer a kernel is imposed or convolved on to the input image and then a specific pattern is obtained depending on different kernels chosen there can be a number of convolved layer. Each of the output consisting of a certain pattern in them for example if we are developing an algorithm to detect dogs one kernel can be used to detect the ears while another the nose and depending on the number of feature we want to detect these kernels can be as many as we choose [14]. Further, more layers can be applied which acts upon

this convoluted outputs to find deeper patterns. Continuing our example this can be patterns within the eyes or the ears. As the number of convolutions increase the size keeps decreasing.

In the convolved layer further features can be enhanced using activation functions such as the Relu function. This ensures that the negative values in the images are retained while the positive features are passed on. [9]

The kernels which are used as filters for convolution are randomly chosen with random values. These values ae then adjusted in every epoch in the backpropagation stage.

4.3 Pooling Layers

The size of a convolutional layer can further be decreased using pooling layers. The pooling layers are introduced in between two convolution layers to reduce the dimension and spatial size of the layer while at the same time keeping the important information. There are two main methods of pooling. [17]

a) Max pooling- In this a filter of a given size is chosen which moves across the entire image. In each stride it gives the maximum value of all the points it covers. Thus, reducing the size of the convolved layer and also reducing the dimension.[17]

b)Global Average pooling- If we are dealing with 3 channels in our convolutional layer this method is used to reduce the size of the image to 3-single points which are the respective average of the red, blue and green channel. This is a bit extreme as it reduces the entire image to a single point so it is preferred mainly as the final layer before flattening.[17]

4.4 Flattening

In this the final layer all the convolutional layer are stacked on top of each other and "unwrapped". The term unwrapped is used because in this flattening process the nXn matrix at the end is made into a long chain of input nodes. This is then connected to a fully connected network to complete our architecture. [16][17]

4.5 Full connection

In full connection it consists of the hidden layers of a deep neural network along with the output. This forms the deep neural network with the input nodes as the flattened node.

4.6 Example and Conclusion

An example can be taken from the keras package itself called the cfar10 algorithm. The dataset of it consists of random pictures of objects, animals and birds. The goal is to classify the images correctly according to the category. Figure 13 shows the train test and validation split of the model. [6]

```
In [4]: from keras.utils import np_utils

        # one-hot encode the labels
        num_classes = len(np.unique(y_train))
        y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)

        # break training set into training and validation sets
        (x_train, x_valid) = x_train[5000:], x_train[:5000]
        (y_train, y_valid) = y_train[5000:], y_train[:5000]

        # print shape of training set
        print('x_train shape:', x_train.shape)

        # print number of training, validation, and test images
        print(x_train.shape[0], 'train samples')
        print(x_test.shape[0], 'test samples')
        print(x_valid.shape[0], 'validation samples')

x_train shape: (45000, 32, 32, 3)
45000 train samples
10000 test samples
5000 validation samples
```

Figure 13: Train test split of the cifar10 dataset

Here, we can see the input shape of each image as 32X32 with 3 channels that is all the three red, blue and green channel are present in the images.

Table 2 shows the architecture for the classification.

Table 2: cifar10 Architecture

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 16)	208
max_pooling2d_1 (MaxPooling2	(None, 16, 16, 16)	0
conv2d_2 (Conv2D)	(None, 16, 16, 32)	2080
max_pooling2d_2 (MaxPooling2	(None, 8, 8, 32)	0
conv2d_3 (Conv2D)	(None, 8, 8, 64)	8256
max_pooling2d_3 (MaxPooling2	(None, 4, 4, 64)	0
dropout_1 (Dropout)	(None, 4, 4, 64)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 500)	512500
dropout_2 (Dropout)	(None, 500)	0
dense_2 (Dense)	(None, 10)	5010

Total params: 528,054
Trainable params: 528,054

It is interesting note that in the first convolutional layer the shape changes from (32,32,3) as stated earlier to (32,32,16) with an increase in parameters to 208. Right after that there is a max pooling layer which reduced the size from (32X32) to (16X16) without changing any parameters. This follows the second convolutional layer with an increase in parameters to 8256. An interesting layer to note is drop out. The dropout layer takes in a float value between 0 and 1. The function of the drop out layer is to cancel some of the input weights. This doesn't means that the same weights are canceled throughout the training. In each epoch a random sets of weights are canceled or assigned 0. This is mainly used to ensure a fair chance for all the weights to be considered during training as sometimes a higher weighted node may dominate the entire training process.

At the end of the architecture we end up with 528054 trainable parameters. Imagine running it for 100 epochs and in the end getting an accuracy of only 68%. That is why deep learning has become such an important topic nowadays. Most of the time is consumed in parameter tuning and training. But even improving the algorithm by 1% is a great feat on its own. We will see this in our experiment which we train from scratch in a CPU and not a GPU.

5. Evaluation of a model

Once the model is created and fitted there are lot of ways in which it can be evaluated to check how the model is performing. The simplest method of checking the model is taking out the errors. In most machine learning algorithm the Root mean square error is calculated or the absolute mean error is taken as a criteria to evaluate the model performance. We need model evaluation because of the following reason.

5.1 Over and Under Fitting

There are lot of problems associated with training a model some of the common problems include over fitting and under fitting a model. In over fitting the model has a high training accuracy but the accuracy in the testing set is very high compared to the validation set. In this the model performs very well during the testing stage but when it comes to working on unseen data it performs really badly. An analogy can be made as this model to be a student who by hearts everything that is taught in class and even be the topper but when it comes to applying what he learnt in the real world he fails miserably.

In under fitting the model is trained less and both the training and validation accuracy are low. In such a case the model is believed to be naïve or too general. This performs badly in both the training and testing set. An analogy can be made as person learning a language. A person learning one language cannot expect to communicate with the rest of the world. May perform well in certain regions only.

5.2 Confusion Matrix

This is perhaps the most important criteria for the evaluation of a model it gives an idea of how the model is predicting the result. The following is the structure for the matrix-

	Predicted True	**Predicted False**
Is True	True positive	False negative
Is False	False positive	True negative

Based on these values a model may be an accuracy, precision or recall based model.

5.3 Accuracy based model

It is the ratio of correctly classified point divided by total number of points. From the confusion matrix the accuracy can be written as-

$$accuracy = \frac{True\ positive + True\ negative}{Total\ number\ of\ points}$$

Accuracy is not always helpful especially if the dataset only few occurrence of a different event. Therefore there are different methods to evaluate a model

5.4 Precision

The precision model is used when we are allowed to make false positive predictions. For example if we are developing an algorithm to detect sick patient it is no harm to predict a healthy patient sick and make him/her take the test compared to the predicting sick patient healthy. Thus, the model has to be as precise with its prediction as possible [11]. The precision is given by-

$$precision = \frac{True\ positive}{True\ positive + False\ positive}$$

5.5 Recall

The recall model is used when we are not allowed to make any false positives. For example in an E-mail classifying algorithm it would be okay if we misclassified some of the mails as not spam compared to classifying some important mail as spam [22].

$$recall = \frac{True\ positive}{True\ positive + False\ negative}$$

5.6 F1 score

A model can have both precession a recall in one model. But we need only one criteria to determine the model performance instead of two. For this F1 score comes into play. It is a harmonic mean between the precision and recall value[22].

$$F1 = \frac{2 * P * R}{P + R}$$

P=Precision, R=recall

5.7 F_β Score

The F1 score is more generic and gives us if a model is more towards a recall model or precession model. The F_β score is used to measure the specific value as how much of it is a precise model or a recall model.

$$F_\beta = (1 + \beta^2) * \frac{P * R}{\beta XP + R}$$

P=Precision, R=recall

An interesting thing to note here is that as β=0, we have a precision model and as β increases we have more recall. The model shifts from precision to recall with increase in β.

6. Experiment

In this experiment we try to classify 3 species of plants. These species were randomly chosen in the wild with different backgrounds and different stages of maturity. A minimum of 200 pictures were taken of each species. These were then preprocessed and separated into train test and validation set. A convolutional neural network was built and its performance was evaluated thoroughly.

6.1 Aim

The goal of the experiment was to try to make a convolution neural network from scratch from the theory that were discussed in the earlier sections*.

6.1.1 Sub-Goals

- To collect the data-set that is pictures of 3-species of plants in the wild.
- To apply pre-processing and sort the train test and validation dataset
- To construct a CNN
- To evaluate its performance

6.2 Equipment and Software used

- Nikon DSLR camera.
- Desktop(CPU) with intel Xeon Processor
- Sciebo- For dataset transfer.
- Python-Programming language
- Jupyter Notebook as IDE

6.3 Procedure

1. Three plants found in the wild were chosen. These species were labeled ‘SpeciesA’, ‘SpeciesB’ and ‘SpciesC’ respectively.
2. The pictures were collected using a Nikkon DSLR.
3. Once all the pictures were collected they were sorted into three folders marked as the labels above.
4. The pictures were manually sorted for duplicates or pictures with low lighting or too much noise.
5. All the libraries are installed following the instruction in chapter 1 and the IDE is started with python 3 kernel.
6. Then three sub-folders were created namely train, test and valid in each of the folder. The division is shown in figure 14 below

```
In [11]: print('There are %d total plant categories.' % len(plant_names))
         print('There are %s total plant images.\n' % len(np.hstack([train_files, valid_files, test_files])))
         print('There are %d training plant images.' % len(train_files))
         print('There are %d validation plant images.' % len(valid_files))

         There are 3 total plant categories.
         There are 684 total plant images.

         There are 414 training plant images.
         There are 123 validation plant images.
```

Figure 14: Summary of data set

7. The image was normalized using the following line of codes as shown in Figure 15

```
In [10]: from keras.preprocessing import image
         from tqdm import tqdm

         def path_to_tensor(img_path):
             # loads RGB image as PIL.Image.Image type
             img = image.load_img(img_path, target_size=(380, 380))
             # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
             x = image.img_to_array(img)
             # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
             return np.expand_dims(x, axis=0)

         def paths_to_tensor(img_paths):
             list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
             return np.vstack(list_of_tensors)

In [11]: from PIL import ImageFile
         train_tensors = paths_to_tensor(train_files).astype('float32')/255
         valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
         test_tensors = paths_to_tensor(test_files).astype('float32')/255

         print(train_tensors.shape)

         100%|██████████| 422/422 [01:17<00:00,  5.38it/s]
         100%|██████████| 197/197 [00:35<00:00,  5.81it/s]
         100%|██████████| 62/62 [00:11<00:00,  5.46it/s]
```

Figure 15: Image pre-processing

8. The architecture of the CNN was then made as shown in Table 3 below.
9. The model was then trained using the following line of code.

```
checkpoints='weight.best.my.hdf5'
my_check=ModelCheckPoint(filepath=checkpoints,verbose=1,
save_best_only=True)
history=model.fit(train,
train_targets,validation_data=(valid,valid_target),
epochs=5,batch_size=5, callbacks=[my_check], verbose=1)
```

Table 3: Model summary

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 378, 378, 64)	1792
max_pooling2d_6 (MaxPooling2	(None, 189, 189, 64)	0
dropout_9 (Dropout)	(None, 189, 189, 64)	0
conv2d_7 (Conv2D)	(None, 187, 187, 64)	36928
max_pooling2d_7 (MaxPooling2	(None, 93, 93, 64)	0
dropout_10 (Dropout)	(None, 93, 93, 64)	0
conv2d_8 (Conv2D)	(None, 91, 91, 128)	73856
max_pooling2d_8 (MaxPooling2	(None, 45, 45, 128)	0
dropout_11 (Dropout)	(None, 45, 45, 128)	0
conv2d_9 (Conv2D)	(None, 43, 43, 512)	590336
max_pooling2d_9 (MaxPooling2	(None, 21, 21, 512)	0
dropout_12 (Dropout)	(None, 21, 21, 512)	0
conv2d_10 (Conv2D)	(None, 19, 19, 1024)	4719616
max_pooling2d_10 (MaxPooling	(None, 9, 9, 1024)	0
dropout_13 (Dropout)	(None, 9, 9, 1024)	0
global_average_pooling2d_2 (	(None, 1024)	0
dense_5 (Dense)	(None, 128)	131200
dropout_14 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 128)	16512
dropout_15 (Dropout)	(None, 128)	0
dense_7 (Dense)	(None, 64)	8256
dropout_16 (Dropout)	(None, 64)	0
dense_8 (Dense)	(None, 3)	195

Total params: 5,578,691

6.5 Model Summary

There were 5 convolution layers chosen along with 2 dense layers in the fully connected part of the network. The Global Average was used instead of Flattening to be the input layer. The output layer consist of 3 nodes each of one category. The total parameters to train was 5,578,691. List of parameters include-

- The number of train and validation data. The more we have of each the better our model performs.
- The filters to be convolved in each layer. Depends, having more not necessarily means better results.
- The weights in the dense layers in the fully connected layer.

These are parameters that cannot directly be controlled and has to be dealt with intuitively or by experience. Important to note is the number of train and validation data can easily be argued as controllable factor as one could easily go out and take more pictures. Although that is true but in real practice the dataset which is available is really noisy and sparse. This was encountered even while collecting this dataset as it was found out that number of available species in a particular region was really limited. Thus, limiting the dataset to only 684 total plants. This has a direct effect on the model as can be seen in the discussion section.

6.6 Model Limitations

After the architecture was made the model was trained. The architecture that is seen here was one of the best accuracy that was obtained from a number of architectures and parameter tuning. This was not at all an easy task in a CPU as a number of problems were encountered which are discussed.

1. Computational power- Since CPU was used instead of GPU the main limitation was computational power. To run a single epoch with the original size of the image is impossible even in a GPU but to run in it on a CPU the size of the data was reduced considerably as seen in figure 15 to 380X380 while the original were 3456X2304. This means a lot of the information are being compressed and limiting our trained model to be less accurate.[18]

The size of the pictures have been tested using various dimensions one being 512X512 doing so we had to reduce the batch size parameters in the above code lines which effected our loss significantly but increased the accuracy. This also meant that the algorithm took a lot of time to run and most of the time ran out of memory and froze. Thus, making this size impossible to train upon. Even with one slight change in parameters it took a long time with initial epoch size of 50 and batch_size of 25. Thus for getting a rough estimate how the algorithm performed a less epoch and batch size was used. This limited the accuracy significantly as can be seen in the result section. [18]

The model that was evaluated was one of the many models that were tried. It took more than two days to come up with this model. More than 12hrs a day were spent for three days continuously.

2. Data-Set- As the dataset only contained 684 images with training images about 400 and the validation about 130 it limited the model accuracy. Also, there were very less plants so we had to compromise with the angles and orientation. That's why we could not use image augmentation as it may result in duplicate images which will in turn result in over fitting. [19]

6.7 Results

On training the data a lot difficulties were encountered as discussed earlier this limited the accuracy as shown in figure 16.

```
In [13]: checkpoint_filepath = 'weights.best.my.hdf5'

         my_checkpointer = ModelCheckpoint(filepath=checkpoint_filepath,
                                           verbose=1, save_best_only=True)

         history=model.fit(train_tensors, train_targets,
                   validation_data=(valid_tensors, valid_targets),
                   epochs=5, batch_size=5, callbacks=[my_checkpointer], verbose=1)

WARNING:tensorflow:From /home/abir/anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.py:3066: to_int3
2 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 422 samples, validate on 197 samples
Epoch 1/5
422/422 [==============================] - 123s 291ms/step - loss: 10.6917 - acc: 0.3270 - val_loss: 10.5545 - val_acc:
0.3452

Epoch 00001: val_loss improved from inf to 10.55449, saving model to weights.best.my.hdf5
Epoch 2/5
422/422 [==============================] - 121s 287ms/step - loss: 11.1952 - acc: 0.3033 - val_loss: 10.5545 - val_acc:
0.3452

Epoch 00002: val_loss did not improve from 10.55449
Epoch 3/5
422/422 [==============================] - 121s 287ms/step - loss: 10.5054 - acc: 0.3460 - val_loss: 10.5545 - val_acc:
0.3452

Epoch 00003: val_loss did not improve from 10.55449
Epoch 4/5
422/422 [==============================] - 122s 289ms/step - loss: 10.6945 - acc: 0.3365 - val_loss: 10.5545 - val_acc:
0.3452

Epoch 00004: val_loss did not improve from 10.55449
Epoch 5/5
422/422 [==============================] - 121s 287ms/step - loss: 10.7709 - acc: 0.3318 - val_loss: 10.5545 - val_acc:
0.3452

Epoch 00005: val_loss did not improve from 10.55449
```

Figure 16: Training evaluation

As it can be seen a very less accuracy of .345 was obtained with this model architecture this is better than the previous models as they had a very poor accuracy even less than 10. Parameter tuning and better model architecture are still being done and the report will be update accordingly. For now it has an accuracy of 35%.

6.8 Model Improvement

After the first run parameters were modified and the model was run again the following are the observations from that run

6.8.1 Observations of Improved Model

1. The files are checked if they have the right labels. The image below shows the visualization of the first few images.

```
In [19]: def plots(ims,figsize=(10,6),rows=1,interp=False,titles=None):
             if type(ims[0]) is np.ndarray:
                 ins=np.array(ims).astype(np.uint8)
                 if (ims.shape[-1] !=3):
                     ims=ims.transpose((0,2,3,1))
             f=plt.figure(figsize=figsize)
             cols=len(ims)//rows if len(ims)%2 ==0 else len(ims)//rows
             for i in range(len(ims)):
                 sp=f.add_subplot(rows,cols,i+1)
                 sp.axis('off')
                 if titles is not None:
                     sp.set_title(titles[i], fontsize=10)
                 plt.imshow(ims[i],interpolation=None)

In [20]: imgs,labels=next(train_batches)

In [21]: plots(imgs,titles=labels)
```

Figure 17: Labeled Images

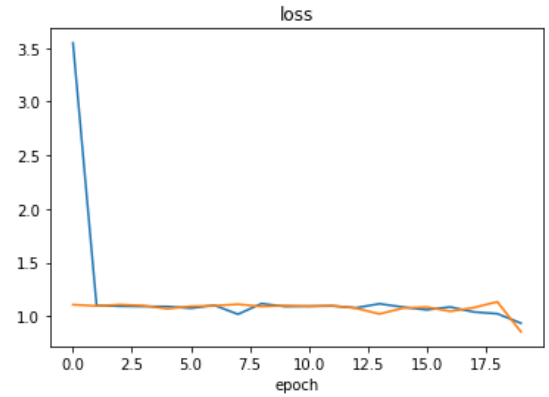
2. Reduced the batch sizes and running more number of times. The following image 18 shows the same. At the end the model improved to near about 60% this is a remarkable improvement from the previous model. This shows how sensitive it is to parameters as tuning a little resulted in such a dramatic improvement.

3. Plotted the curved to check for over or under fitting figure 19 shows the side by side comparison of the validation and training loss vs epochs and the other one shows the accuracy per epoch.

```
Epoch 1/20
 - 61s - loss: 3.5469 - acc: 0.2875 - val_loss: 1.1079 - val_acc: 0.3333
Epoch 2/20
 - 41s - loss: 1.1011 - acc: 0.3625 - val_loss: 1.0966 - val_acc: 0.3333
Epoch 3/20
 - 41s - loss: 1.0939 - acc: 0.3500 - val_loss: 1.1088 - val_acc: 0.4167
Epoch 4/20
 - 43s - loss: 1.0916 - acc: 0.4375 - val_loss: 1.0992 - val_acc: 0.2500
Epoch 5/20
 - 37s - loss: 1.0907 - acc: 0.3625 - val_loss: 1.0700 - val_acc: 0.5833
Epoch 6/20
 - 32s - loss: 1.0757 - acc: 0.4735 - val_loss: 1.0926 - val_acc: 0.3333
Epoch 7/20
 - 37s - loss: 1.1024 - acc: 0.3000 - val_loss: 1.0986 - val_acc: 0.3333
Epoch 8/20
 - 35s - loss: 1.0191 - acc: 0.4750 - val_loss: 1.1121 - val_acc: 0.3333
Epoch 9/20
 - 38s - loss: 1.1183 - acc: 0.3625 - val_loss: 1.0912 - val_acc: 0.4167
Epoch 10/20
 - 37s - loss: 1.0908 - acc: 0.3625 - val_loss: 1.0998 - val_acc: 0.4167
Epoch 11/20
 - 32s - loss: 1.0925 - acc: 0.5265 - val_loss: 1.0942 - val_acc: 0.3333
Epoch 12/20
 - 37s - loss: 1.0959 - acc: 0.3750 - val_loss: 1.0990 - val_acc: 0.4167
Epoch 13/20
 - 37s - loss: 1.0786 - acc: 0.4750 - val_loss: 1.0753 - val_acc: 0.4167
Epoch 14/20
 - 35s - loss: 1.1163 - acc: 0.2500 - val_loss: 1.0216 - val_acc: 0.6667
Epoch 15/20
 - 37s - loss: 1.0857 - acc: 0.3750 - val_loss: 1.0760 - val_acc: 0.3333
Epoch 16/20
 - 37s - loss: 1.0601 - acc: 0.3875 - val_loss: 1.0856 - val_acc: 0.5000
Epoch 17/20
 - 32s - loss: 1.0955 - acc: 0.2513 - val_loss: 1.0462 - val_acc: 0.5455
Epoch 18/20
 - 37s - loss: 1.0402 - acc: 0.4250 - val_loss: 1.0805 - val_acc: 0.3333
Epoch 19/20
 - 35s - loss: 1.0240 - acc: 0.4000 - val_loss: 1.1349 - val_acc: 0.0833
Epoch 20/20
 - 37s - loss: 0.9357 - acc: 0.5500 - val_loss: 0.8560 - val_acc: 0.5833
```

Figure 18: Training the second model

```
In [13]: plt.plot(history.history['loss'])
         plt.plot(history.history['val_loss'])
         plt.title('loss')
         plt.xlabel('epoch')
Out[13]: <matplotlib.text.Text at 0x7f6eaff1c630>
```




```
In [14]: plt.plot(history.history['acc'])
         plt.plot(history.history['val_acc'])
         plt.title('Accuracy')
         plt.xlabel('epoch')
Out[14]: <matplotlib.text.Text at 0x7f6eaf8a6be0>
```

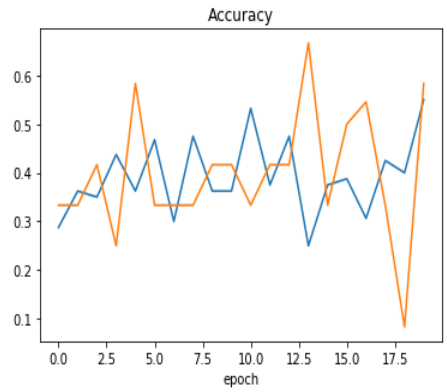


Figure 19: Loss and Accuracy plots for second model

6.8.2 Improved Model Discussion

The model shows how changing and tweaking few parameters can drastically change the outcome just from 35% we have an increase in validation accuracy to near about 60% and reaching an highest of 66.67%. This is a really good achievement given the fact of training in a sparse data set. Further, improvement can still be done using different image pre-processing but to get a good accuracy more number of images are necessary. The predictions can be seen which proves and encompasses the whole discussion in the previous section.

Looking at the graph it can be seen that the loss is similar for both the training and validation set usually in most models the validation loss starts increasing while the training loss decreases. This proves that the model is not over fitting.

The accuracy is however really drastic. As can be seen in the right side of the figure 18. The yellow line represents the validation in both the plots. The fluctuation proves that the model can be made to run for more epochs until it reaches a stable value. Here we have only made it run for 20 epochs.

6.8.3 Prediction

A prediction was made on this model on a random set of images and the result obtained is presented in figure 19.

```
In [19]: test_imgs,test_labels=next(test_batches)
         plots(test_imgs,titles=test_labels)
```


```
In [20]: pred=model.predict_generator(test_batches,steps=1,verbose=0)

In [21]: pred

Out[21]: array([[ 0.33217505,  0.35110179,  0.31672323],
                [ 0.04623965,  0.74546492,  0.20829545],
                [ 0.05537702,  0.72116643,  0.22345655],
                [ 0.1720916 ,  0.52886593,  0.29904243]], dtype=float32)
```

Figure 20: Prediction of second model

Here we can see that the model classifies species B in fact it classifies species B in all the cases with an higher accuracy upon investigating the cause the following can be given as few hypothesis-

1. The training images of species B were greater than the images for the other two categories.

2. The background of the plants played some role. In many plants there were green background the model might have trained them as well as species class. Thus, classifying anything that is green in species B.[22]

3. The train data may contain duplicates. Not duplicate images but as multiple images were taken from a single plant it may contain the same information. Thus showing such a drastic change in accuracy.[22]

6.9 Model Comparison

In the final model a few modifications were made to decrease the random fluctuations that the previous model was having. To do so the batch size was increased while the learning rate of the optimizer was decreased to take small steps towards finding the global minima. The model was then also run for 100 epochs. A really good and stable accuracy over 61% was obtained as shown in figure 20.

```
Epoch 92/100
 - 71s - loss: 0.7112 - acc: 0.6906 - val_loss: 0.8243 - val_acc: 0.5833
Epoch 93/100
 - 71s - loss: 0.7914 - acc: 0.6417 - val_loss: 0.8005 - val_acc: 0.5981
Epoch 94/100
 - 77s - loss: 0.7632 - acc: 0.6250 - val_loss: 0.8291 - val_acc: 0.5917
Epoch 95/100
 - 75s - loss: 0.7262 - acc: 0.6500 - val_loss: 0.7614 - val_acc: 0.6729
Epoch 96/100
 - 68s - loss: 0.7775 - acc: 0.6326 - val_loss: 0.8309 - val_acc: 0.5750

Epoch 97/100
 - 75s - loss: 0.7432 - acc: 0.6833 - val_loss: 0.8221 - val_acc: 0.5701
Epoch 98/100
 - 75s - loss: 0.8397 - acc: 0.6000 - val_loss: 0.8841 - val_acc: 0.5607
Epoch 99/100
 - 77s - loss: 0.7295 - acc: 0.6250 - val_loss: 0.8297 - val_acc: 0.5583
Epoch 100/100
 - 67s - loss: 0.7318 - acc: 0.6591 - val_loss: 0.7981 - val_acc: 0.6168
```

Figure 21: Third model increased val_accuracy

Furthermore, the quality of the predictions increased as well. Instead of predicting just one plant. The model has started predicting over the other species as well as can be seen in figure 22 below.


```
In [47]: pred=model.predict_generator(test_batches,steps=1)

In [48]: pred

Out[48]: array([[ 0.09904362,  0.61555582,  0.28540063],
                [ 0.06275547,  0.69191378,  0.24533074],
                [ 0.02978287,  0.78011554,  0.19010165],
                [ 0.41124058,  0.15998216,  0.42877722]], dtype=float32)
```

Figure 22: Prediction of the model

As it is seen this is the most reliable model it is more coherent with its predictions. A maximum of 75 % percent can be obtained as it was achieved in one of the random models that were tried as shown in figure 23.

```
Epoch 27/100
 - 40s - loss: 1.1028 - acc: 0.3174 - val_loss: 1.1326 - val_acc: 0.0833
Epoch 28/100
 - 47s - loss: 1.0914 - acc: 0.3750 - val_loss: 1.0803 - val_acc: 0.5000
Epoch 29/100
 - 47s - loss: 1.0828 - acc: 0.4000 - val_loss: 1.0737 - val_acc: 0.4167
Epoch 30/100
 - 46s - loss: 1.0993 - acc: 0.3375 - val_loss: 1.1135 - val_acc: 0.2500
Epoch 31/100
 - 48s - loss: 1.1025 - acc: 0.4125 - val_loss: 1.0424 - val_acc: 0.5833
Epoch 32/100
 - 42s - loss: 1.0795 - acc: 0.4842 - val_loss: 1.0632 - val_acc: 0.5000
Epoch 33/100
 - 49s - loss: 1.0678 - acc: 0.5125 - val_loss: 1.0142 - val_acc: 0.7500
```

Figure 23: Random model fluctuating predictions

Although, a very high accuracy of 75% can be seen in this model but it also noteworthy too look how much fluctuation this model is having. From an accuracy of .083 in the 27th epoch it increased to 50% in just one and then again decreased to 25%. This model was like a coin toss just giving us random predictions at each epochs. This was due to the following reasons stated in stack overflow-

1. The batch size was low which resulted in random training images to be taken in different epochs.

2. The number of epochs were low. Being a random model the number of epochs do not really play that big a role.

3. The learning rate of the optimizer was too big making it fluctuate drastically in finding the global minima.

Even in the final model there is a further scope of improvement as it can be made more stable by further decreasing the learning rate or increasing the batch size. A batch size of 30 for both test and validation were chosen and a learning rate of .0001 with Adam optimizer was taken. The following shows the two graphs that were obtained-

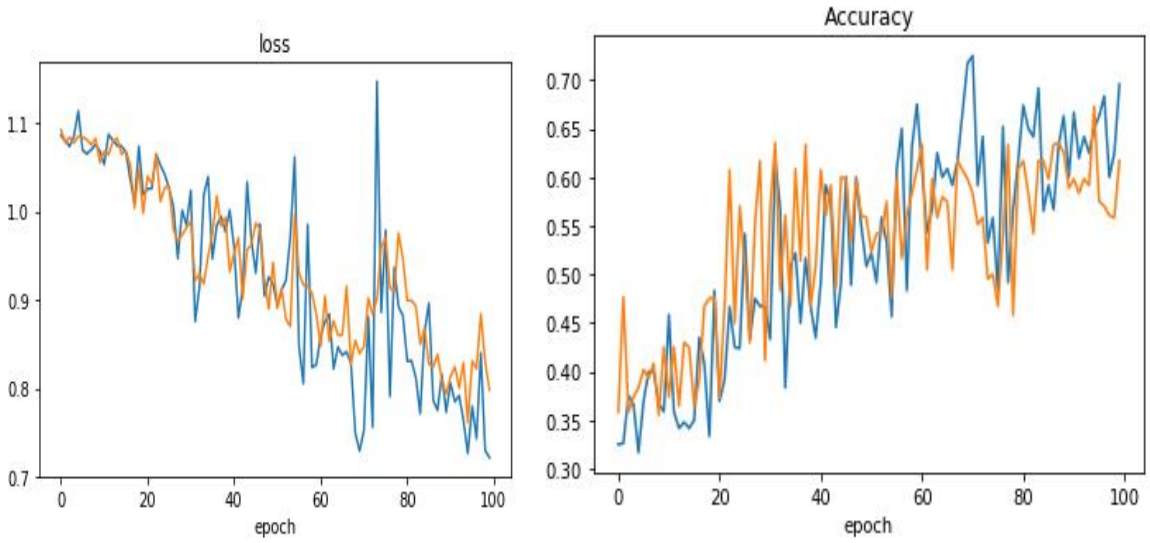


Figure 24: Loss and Validation accuracy curve for the third model

This is very promising to look at as it shows the validation loss increases over epochs on the left while the accuracy increase to the right. The fluctuations also decreases as the epochs progresses. This also shows that the model can be made to run for more number of epochs 120,150 or 200 even with further increase in the batch size. But as we are training in CPU and not GPU this limits the computation. To train the third model for 100 epochs took about 4 hours!

Conclusion

As it can be seen this started as a very poor model and then improved to get an accuracy of over 60% with further scope of improvement. For me personally this was really fulfilling to build a neural network from scratch and then improving upon it epoch by epoch, pun intended, was really something. The model has further scope of improvement so work will be still going on once a reasonable accuracy is obtained. The data set can be further increased and real time detection can be tried upon.

From here, the goal is to drive a robot autonomously and classify plants in real time. It can be used to detect whether a plant is harmful for a particular plantation or ecosystem either report it to human or try to remove the plant on its own. The next part of the project is to try to make the bot try to drive autonomously using reinforcement learning techniques if possible.

Appendix

I. Binary Bitwise Operation

As we know computer understands only binary a number system with base 2. That is it consists of only two digits 0 and 1 a binary bitwise and is a logical operation which follows the expression [20]

$$a * b = c$$

Yielding us the following table-

Table I: Bitwise And

A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

This is what happens when we do bit wise and operation in pixels as well. This means that in two image those pixels which align as 1 in both the picture will be passed in the output while the rest will be perceived as dark

II. Hough Transform

This is a very important concept in computer vision. This is usually used to check edges and connect lines between two points. The principle of hough transform is simple a line in the x-y plane will be a point in the corresponding hough plane. Similarly, a line in hough-space represents a family of lines that passes through a single point. Two points lying in the same line will intersect at a particular point in the hough-plane[8] [21]. An illustration can be seen in figure I

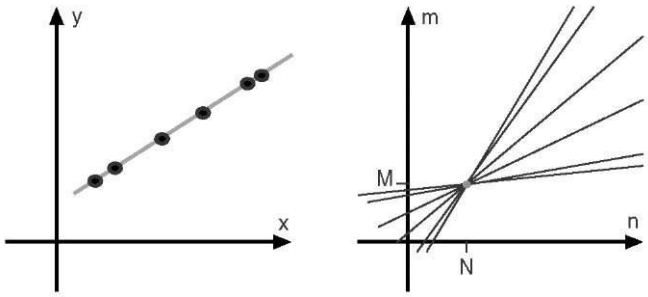


Figure I: Hough Transform (Source: biomedical image analysis)

An important thing to note here is that perpendicular lines with slope infinite cannot be represented in the hough plane that's why the line in the x-y is usually converted to polar coordinates and then the hough transform is applied [8][21]. That is instead of m-b the co-ordinate changes to rho and theta. The application is not defined here as it is not applied in this project but since the project extends beyond what is described it is a very important concept to carry along for future endeavors.

References

[1] NumPy — NumPy. (2018). Retrieved from http://www.numpy.org/

[2] Python Data Analysis Library — pandas: Python Data Analysis Library. (2019). Retrieved from https://pandas.pydata.org/

[3] OpenCV library. (2019). Retrieved from https://opencv.org/

[4] scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation. (2019). Retrieved from https://scikit-learn.org/stable/

[5] Matplotlib: Python plotting — Matplotlib 3.0.3 documentation. (2019). Retrieved from https://matplotlib.org/

[6] Home - Keras Documentation. (2019). Retrieved from https://keras.io/

[7] PyTorch. (2019). Retrieved from https://pytorch.org/

[8] Szeliski, R. (2010). *Computer vision: algorithms and applications*. Springer Science & Business Media.

[9] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

[10] Downey, A. B. (2011). *Think stats*. " O'Reilly Media, Inc.".

[11] Ng, A. (2011). Advice for applying machine learning. In *Machine learning*.

[12] Bottou, L. (2012). Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade* (pp. 421-436). Springer, Berlin, Heidelberg.

[13] Yuille, A., & Liu, C. (2019). Limitations of Deep Learning for Vision, and How We Might Fix Them. Retrieved from https://thegradient.pub/the-limitations-of-visual-deep-learning-and-how-we-might-fix-them/

[14] LeCun, Y., & Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, *3361*(10), 1995.

[15] Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[16] Bhattacharya, U., Vajda, S., Mallick, A., Chaudhuri, B. B., & Belaïd, A. (2004, October). On the choice of training set, architecture and combination rule of multiple MLP classifiers for multiresolution recognition of handwritten characters. In *Ninth International Workshop on Frontiers in Handwriting Recognition* (pp. 419-424). IEEE.

[17] Cook, A. (2017). Global Average Pooling Layers for Object Localization. Retrieved from https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/

[18] Sontag, E. D. (1998). VC dimension of neural networks. *NATO ASI Series F Computer and Systems Sciences*, *168*, 69-96.

[19] Beam, A. (2017). You can probably use deep learning even if your data isn't that big. Retrieved from https://beamandrew.github.io/deeplearning/2017/06/04/deep_learning_works.html

[20] Seshadri, V., Hsieh, K., Boroum, A., Lee, D., Kozuch, M. A., Mutlu, O., ... & Mowry, T. C. (2015). Fast bulk bitwise AND and OR in DRAM. *IEEE Computer Architecture Letters*, *14*(2), 127-131.

[21] Illingworth, J., & Kittler, J. (1988). A survey of the Hough transform. *Computer vision, graphics, and image processing*, *44*(1), 87-116.

[22] Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., & Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, *542*(7639), 115.