# Tomato Leaf Disease Prediction Using Deep Learning Techniques CNN and Transfer Learning

**Project Report submitted in partial fulfillment of the requirements for the award of the degree of,**

**BACHELOR OF SCIENCE AND**

**STATISTICS**

Submitted by:

**Vallivedu Lohith - 321910305032**

**Under the guidance of**

**Dr. Magesh Chinnaswamy**

**Professor**

**GITAM SCHOOL OF TECHNOLOGY**

**GANDHI INSTITUTE OF TECHNOLOGY AND MANAGEMENT**

**(Deemed to be University)**

**Bengaluru Campus**

**DEPARTMENT BACHELOR OF SCIENCE AND STATISTICS**

**GITAM INSTITUTE OF TECHNOLOGY**

**GITAM**
**(Deemed to be University)**

## CERTIFICATE

This is to certify that the project report entitled **“Tomato Leaf Disease Prediction Using Deep Learning Techniques CNN and Transfer Learning ”** is a bonafide record of work carried out by **Vallivedu lohith,** submitted in partial fulfillment of the requirement for the award of the degree of **BSC**.

| | |
|---|---|
| **Project Guide.** | **Head of the Department.** |
| **SIGNATURE OF THE GUIDE** | **SIGNATURE OF THE HoD** |
| **Dr.Magesh chinnaswamy,** | **Dr. Venkateswarlu,** |
| professor | Professor. |

**DEPARTMENT BACHELOR OF SCIENCE AND STATISTICS**

**GITAM INSTITUTE OF TECHNOLOGY**

**GITAM**

**(Deemed to be University)**

## DECLARATION

We, hereby declare that the project report entitled **"Tomato Leaf Disease Prediction Using Deep Learning Techniques CNN and Transfer Learning"** is an original work done in the **Department of Bachelor of science and engineering, GITAM Institute of Technology, GITAM (Deemed to be University)** submitted in partial fulfillment of the requirements for the award of the degree of **BSC.** in Bachelor of science and Statistics. The work has not been submitted to any other college university for the award of any degree.

**Date:**

| Registration No(s). | Name(s) | Signature(s) |
|---|---|---|
| **1. 321910305032** | **Vallivedu lohith** | |

## ACKNOWLEDGEMENT

We were able to complete our project successfull. However, it would not have been possible without the kind support and help of many individuals. We would like to extend our sincere thanks to all of them.

Thanksgiving will not end without thanking the parents where even the word Thanks is not enough for the sacrifices they made. I would like to thank my parents for everything they have given and have made my path easier and to reach this point.

We are highly indebted to GITAM (Deemed to be University), Bangalore for their guidance and constant supervision as well as for providing necessary information regarding the project and also for their support in completing the project.

Would like to express our gratitude towards Prof. S Dinesh (Director of GST), Prof. Venkateswarlu(HOD of BSC, GST), Mrs. Balaji. V(Professor, BSC, GST) and also to all the other supporting faculty and staff for their kind cooperation and encouragement which helped us in the completion of this project.

# ABSTRACT

Crop diseases are a key danger for food security, but their speedy identification is still difficult in many portions of the world because of the lack of the essential infrastructure. The mixture of increasing worldwide smartphone dispersion and current advances in computer vision made conceivable by deep learning has cemented the way for smartphone-assisted disease identification. Using a public dataset of 9000 images of infected and healthy Tomato leaves collected under controlled conditions, we trained a convolutional neural network to identify 10 diseases. The trained model achieved an accuracy of 91% on a held-out test set, demonstrating the feasibility of this approach. Overall, the approach of training deep learning models on increasingly large and publicly available image datasets presents a clear path toward smartphone-assisted crop disease diagnosis on a massive global scale.

The high scale prevalence of diseases in crops affects the production quantity and quality. Solving the problem of early identification/diagnosis of diseases by exploiting a quick and consistent reliable method will benefit the farmers. In this context, work focuses on classification and identification of tomato leaf diseases using convolutional neural network (CNN) techniques. We consider four CNN architectures, namely, VGG-16, VGG-19, ResNet, and **Inception V3**, and use feature extraction and parameter-tuning to identify and classify tomato leaf diseases. We test the underlying models on two datasets, a laboratory-based dataset and self-collected data from the field. We observe that all architectures perform better on the laboratory-based dataset than on field-based data, with performance on various metrics showing variance in the range 10%–15%. Inception V3 is identified as the best performing algorithm on both datasets.

Here we use the requirements of tensorflow version of 2.8.0 version and numpy version of 1.21.5 and matplotlib as 3.2.2 version in the conclusion we will upload an image and it will preprocess the image and store that image in the local system uploads folder and then the data is preprocessed and we will get the disease name and what are the precocious need to take to reduce it and we have used some localization where the person can change the language to their own language and see the output

# TABLE OF CONTENTS

## Introduction

Tomatoes (biological name: Solanum lycopersicum) grows on mostly well drained soil  and Nine out of 10 farmers grow tomatoes in their field. Many gardeners also grow tomatoes in their garden to use fresh grown tomatoes in their kitchens and get a good taste of food. However, farmers and gardeners are sometimes unable to get proper progress in plant growth. The tomatoes may not sometimes appear on plants or sometimes the tomatoes may get bad looking and disease-ful black spots at the bottom part.

The identification of tomato plant disease may start from, to diagnose the portion having infection in plant then to note the differences such as brown or black patches and holes on the plant and then to look for the insects also.

Tomatoes and similar vegetables like potatoes or brinjal must not be planted on the same farm for more than one time in a period of three years . To maintain the fertility of soil we should ideally precede tomato planting by any member of the grass family e.g. wheat, corn, rice, sugarcane etc.

The tomato problems may be divided into two sections: bacteria or fungi or poor cultivation habits causing 16 diseases while insects causing 5 other types of diseases. Ralstonia solanacearum bacteria causes a serious form of Bacterial wilt. This bacteria can survive in soil for a long time period and enter roots through natural wounds made during secondary roots emergence or man made during cultivating or transplanting or even insects.

High moisture and high temperature favors disease development. The bacteria fills, water conducting tissue of plant, with slime by multiplying rapidly inside it. This results in affecting the vascular system of the plant, while the leaves may stay green. On a cross section view of an infected plant stem, it appears brown with yellowish material coming out of it.

In the research article, we have proposed a novel method to identify the disease in tomato crops after analyzing the images of leaves. The work will solve farmers' problems of plant's disease identification without running after plant scientists. It will thus help them cure the plant's disease in a timely fashion and will thus increase both quality and the quantity of food crops produce and therefore help in increasing farmer's profit.

## 1. Literature survey

| | |
|---|---|
| AuthorsName | Ding Jiang, Fudong Li, Yuequan Yang, Song Yu |
| Title | Tomato Leaf Disease Detection using Convolution Neural Network |
| Year | 2020 |
| Abstract | Tomato is the most popular crop in the world and in every kitchen, it is found in different forms irrespective of the cuisine. After potato and sweet potato, it is the crop which is cultivated worldwide. India ranked 2 in the production of tomatoes. However, the quality and quantity of tomato crops goes down due to the various kinds of diseases. So, to detect the disease a deep learning-based approach is discussed in the article. For the disease detection and classification, a Convolution Neural Network based approach is applied. In this model, there are 3 convolution and 3 max pooling layers followed by 2 fully connected layers. The experimental results shows the efficacy of the proposed model over pre-trained model i.e. VGG16, InceptionV3 and MobileNet. The classification accuracy varies from 76% to 100% with respect to classes and average accuracy of the proposed model is 91.2% for the 9 disease and 1 healthy class. |
| Methodology | The proposed CNN model has been executed on the NVIDIA DGX v100 machine. The machine is equipped with40600 CUDA cores, 5120 tensor cores, 128 GB RAM and 1000 TFLOPS speed. As in the data set images per classare different, so the class balance data augmentation technique has been applied. In the proposed CNN architecture there are three convolution and max pooling layers are used. In each layer various numbers of filters have been applied.The architecture of proposed CNN model is depicted. <br> Relu and Batch normalization are used in this building blocks of separable convolutions. <br> The proposed algorithm is also compared with the pre-trained model in terms of number of trainable and non-trainable parameters and it is observed that the proposed model is far better than the pre-trained models i.e.VGG16, MobileNetand InceptionV3. |

| | |
|---|---|
| Observation | Detecting the tomato leaf disease using CNN |

| | |
|---|---|
| Authors Name | Surampalli Ashok; Gemini Kishore; Velpula Rajesh |
| Title | Tomato Leaf Disease Detection Using Deep Learning Techniques |
| Year | 2018 |
| Abstract | Early Detection of Plant Leaf Detection is a major necessity in a growing agricultural economy like India. Not only as an agricultural economy but also with a large amount of population to feed, it is necessary that leaf diseases in plants are detected at a very early stage and predictive mechanisms to be adopted to make them safe and avoid losses to the agri-based economy. This paper proposes to identify the Tomato Plant Leaf disease using image processing techniques based on Image segmentation, clustering, and open-source algorithms, thus all contributing to a reliable, safe, and accurate system of leaf disease with the specialization to Tomato Plants. |
| Methodology | **Deep Learning Techniques**<br>● Discrete Wavelet Transform (DWT) is applied on the improved enhancement of tomato leaf image .<br>● The image is made out of pixels each with a luminous level and the GLCM is used to classify the leaf image or the segment of a leaf image depending upon various luminous levels.<br>● Segmentation is the process of categorizing the leaf image into smaller portions of texture, containing similar characteristics.<br>● A convolutional Neural Network algorithm used in this proposed method is a hierarchical feature extraction that maps the pixel values and evaluates the same with the trained dataset image. It is classified by several fully connected layers in the subsequent step and all adjustable parameters of the leaf portions are optimized by reducing the error over the training set. The compared image is classified into disease affected and normal leaf as the image classifier technique that has been deployed. The results of the same are stored into the database for further detection and analysis. |

| | |
|---|---|
| Observation | Early Detection of Plant Tomato Leaf diseases. |

| | |
|---|---|
| Authors Name | Tahmina Tashrif Mim; Md. Helal Sheik; Roksana Akter Shampa |
| Title | Leaves Diseases Detection of Tomato Using Image Processing |
| Year | 2019 |
| Abstract | This research paper tends to merge or combine a part of the agricultural sector with science and technology to reduce the loss caused by insect's attack and diseases of plant leaves. More specifically, this research happens to combine the agricultural sector with computer science. Since agriculture is a vast sector to work on, to simplify the work, we are detecting vegetable plant diseases using Artificial Intelligence and computer science. To implement this idea, we have chosen "Tomato" as the core vegetable which's leaf diseases are to be predicted by using the algorithms of Artificial Intelligence, CNN and computer science. |
| Methodology | CNN (convolutional neural networks) and image processing techniques have been used to create and train the system with tomato leaves dataset as input. After processing the given images, an accurate outcome is to be expected as output. This is the reason why rural farmers having little knowledge on disease detection and advanced technologies can also use this system without any difficulty and hustle. |
| Observation | Detecting the five types of tomato leaf diseases |

## 2. Software and Hardware Specifications

## 3.1 Specific Requirements

### 3.1.1Functional Requirements:

The functional requirements for a system describe what the system should do. Those requirements depend on the type of software being developed, the expected users of the software. These are a statement of services the system should provide, how the system should react to particular inputs and how the system should behave in a particular situation.

### 3.1.2 Non-Functional Requirements:

Non-functional requirements are requirements that are not directly concerned with the specified function delivered by the system. They may relate to emergent system properties such as reliability, response time, and store occupancy. Some of the non-functional requirements related to this system are hereby below: Reliability: Reliability-based on this system defines the evaluation result of the system, correct identification of the facial expressions, and maximum evaluation rate of the facial expression recognition of any input images.

Ease of Use:

The system is simple, user-friendly, graphical user interface implemented so anyone can use this system without any difficulties.

## 3.2 Hardware and Software Requirement

### 3.3.1 Hardware Requirement

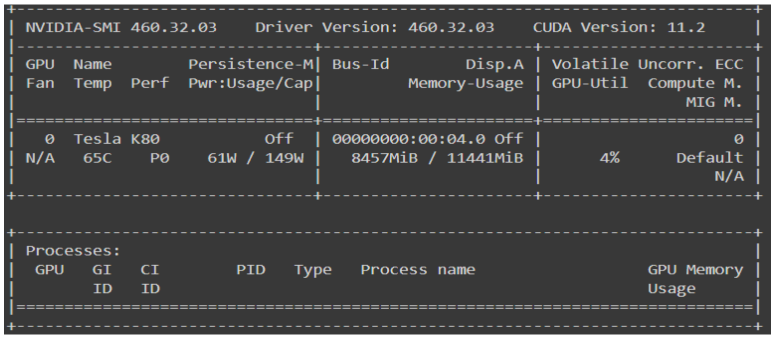
Processor- Intel i3 and above

- RAM minimum 4Gb and higher
- Hard disk- 500GB minimum

### 3.3.2 Software Requirements:

- Python IDE- Python 3.6 and above
- OS- Windows or linux
- Jupyter notebook- Setup tools and pip to be installed for 3.6 and above (or) google colab for free GPU'S
- TensorFlow version = 2.8.0
- numpy = 1.21.5

matplotlib = 3.2.2



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   65C    P0    61W / 149W |   8457MiB / 11441MiB |      4%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

## Numpy

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors.

The Python programming language was not initially designed for numerical computing, but attracted the attention of the scientific and engineering community early on, so that a special interest group called matrix- sig was founded in 1995 with the aim of defining an array computing package. Among its members was Python designer and maintainer Guido van Rossum, who implemented extensions to Python's syntax (in particular the indexing syntax) to make array computing easier.

An implementation of a matrix package was completed by Jim Fulton, then generalized by Jim Hugunin to become Numeric,[4] also variously called Numerical Python extensions or NumPy. Hugunin, a graduate student at Massachusetts Institute of Technology (MIT), joined the Corporation for National Research Initiatives (CNRI) to work on JPython in 1997 leaving Paul Dubois of Lawrence Livermore National Laboratory (LLNL) to take over as maintainer. Other early contributors include David Ascher, Konrad Hinsen and Travis Oliphant.

A new package called Numarray was written as a more flexible replacement for Numeric. Like Numeric, it is now deprecated. Numarray had faster operations for large arrays, but was slower than Numeric on small ones, so for a time both packages were used for different use cases. The last version of Numeric v24.2 was released on 11 November 2005 and numarray v1.5.2 was released on 24 August 2006.

There was a desire to get Numeric into the Python standard library, but Guido van Rossum decided that the code was not maintainable in its state then.

In early 2005, NumPy developer Travis Oliphant wanted to unify the community around a single array package and ported Numarray's features to Numeric, releasing the result as NumPy 1.0 in 2006.[7] This new project was part of SciPy. To avoid installing the large SciPy package just to get an array object, this new package was separated and called NumPy. Support for Python 3 was added in 2011 with NumPy version 1.5.0.

In 2011, PyPy started development on an implementation of the NumPy API for PyPy. It is not yet fully compatible with NumPy

NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays, requiring rewriting some code, mostly inner loops using NumPy.

Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted,[16] and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars. In comparison, MATLAB boasts a large number of additional toolboxes, notably Simulink, whereas NumPy is intrinsically integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; SciPy is a library that adds more MATLAB-like functionality and Matplotlib is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

Python bindings of the widely used computer vision library OpenCV utilize NumPy arrays to store and operate on data. Since images with multiple channels are simply represented as three-dimensional arrays, indexing, slicing or masking with other arrays are very efficient ways to access specific pixels of an image. The NumPy array as a universal data structure in OpenCV for images, extracted feature points, filter kernels and many more vastly simplifies the programming workflow and debugging.

## Matplotlib

- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits

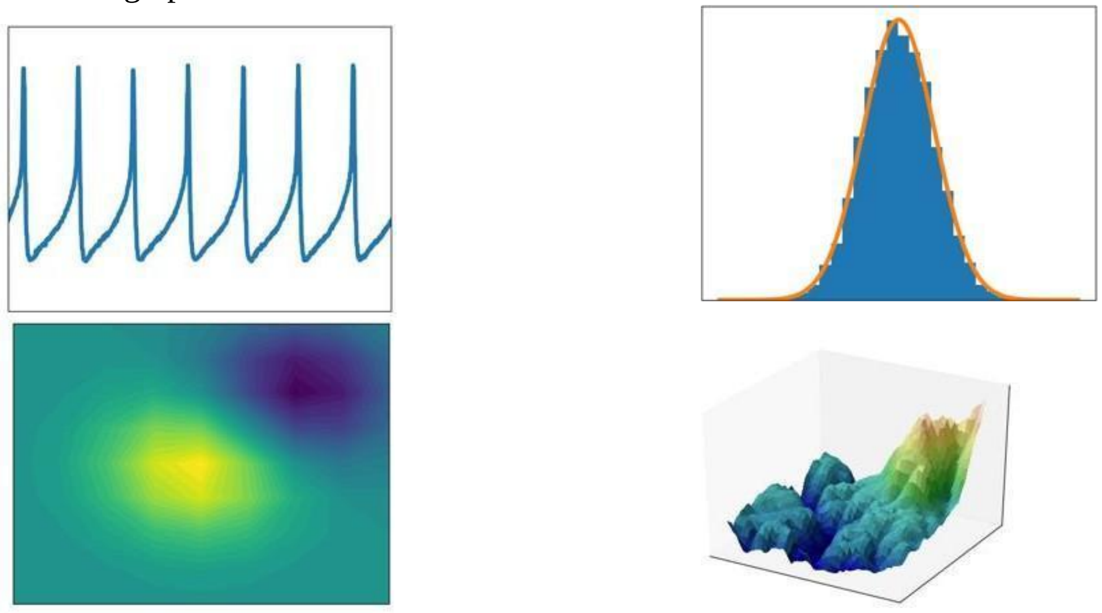


Fig 3.1: Sample graphs of matplotlib

- Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, error charts, scatterplots, etc., with just a few lines of code. For examples, see the sample plots and thumbnail gallery.

- For simple plotting the pyplot module provides a MATLAB-like interface, particularly when combined with IPython. For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

## Tensor Flow

TensorFlow is a free and open-source software library for data flow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks.

TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on February 11, 2017. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional CUDA and SYCL extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS.

Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as tensors. During the Google I/O Conference in June 2016, Jeff Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

**Training set and testing set:**

Machine learning is about learning some properties of a data set and then testing those properties against another data set. A common practice in machine learning and deep learning is to evaluate an algorithm by splitting a data set into two. We call one of those sets the within training set, on which we learn some properties; we call the other set the testing set, on which we test the learned properties.

## 3. Problem Statement

The tomato leaf disease prediction problem includes modeling images of leaf's with different types of disease with the knowledge of the ones it is trained . This model is then used to identify whether a new leaf is healthy or a diseased leaf and the precautions to reduce the disease.

### 3.1 Objective

The main aims are, firstly, to identify the disease of a leaf and, secondly, to what precautions need to be taken . Here have trained with different models out of all inception_v3 from transfer learning performs good when compare to other model like vgg19,vgg16,Resnet50 , in terms of accuracy and predicting and loss score

## 4. Designing

### 4.1 System Architecture

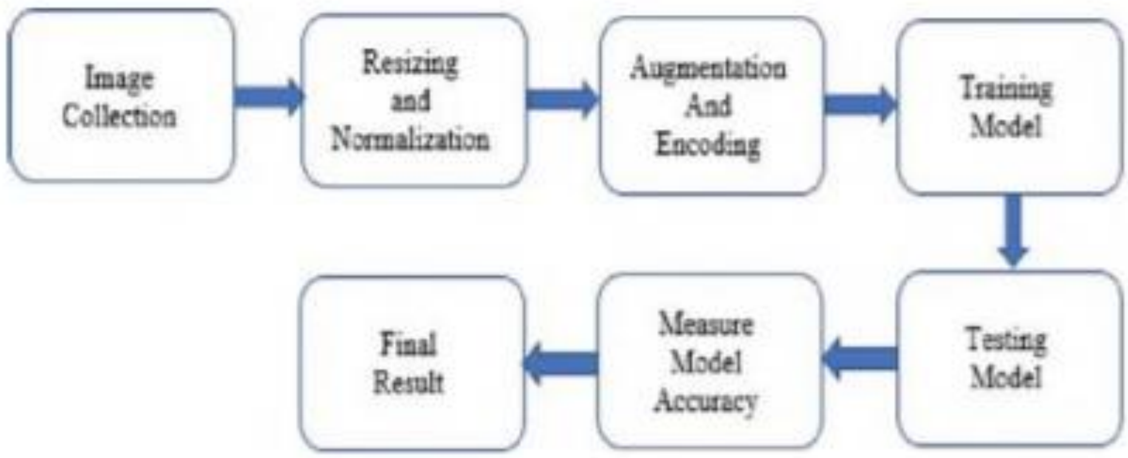


Fig.5.1: System architecture

**Image Collection**: We will collect the data of 10 different types of disease where 9 are disease and 1 is healthy plant leaf's. we have collected the data from kaggle from the training purpose

**Resizing and Normalization :** Normalization gives equal weights/importance to each variable so that no single variable steers model performance in one direction just because they are bigger numbers.

**Data Augmentation :** changing the images to different flippers , so that tomorrow if we get any new image it should be predict correctly by not telling this is not a flippered image

**Data Splitting:** creating the training set and testing set so that we can train the algorithm and understand the performance of that model. And test is used for the predicting the model

**Training:** Making the particular algorithm understand the train data and become intelligent in that concept.

**Testing**: process used to predict the outputs for the inputs in the test set.

**comparing the metrics**: Accuracy is the measure of metrics

The below given images are the disease

## 4.2 Methodology

### Transfer Learning?

Transfer learning is a machine learning technique in which a model trained for one task is reused for a second related task.
Transfer learning is an optimization that allows for rapid progress or performance gains when modeling a second task.
Transfer learning is related to problems such as multi-task learning and concept drift, and is not limited to deep learning research areas.
Nonetheless, transfer learning is popular in deep learning given the vast resources required to train deep learning models, or the large and rewarding datasets in which deep learning models are trained.
Transfer learning works in deep learning only if the model functions learned from the first task are common.

### How do I use Transfer Learning?

You can use the transfer channel with your own prediction modeling issue.
Two general approaches are:

Develop a model approach
Model approach

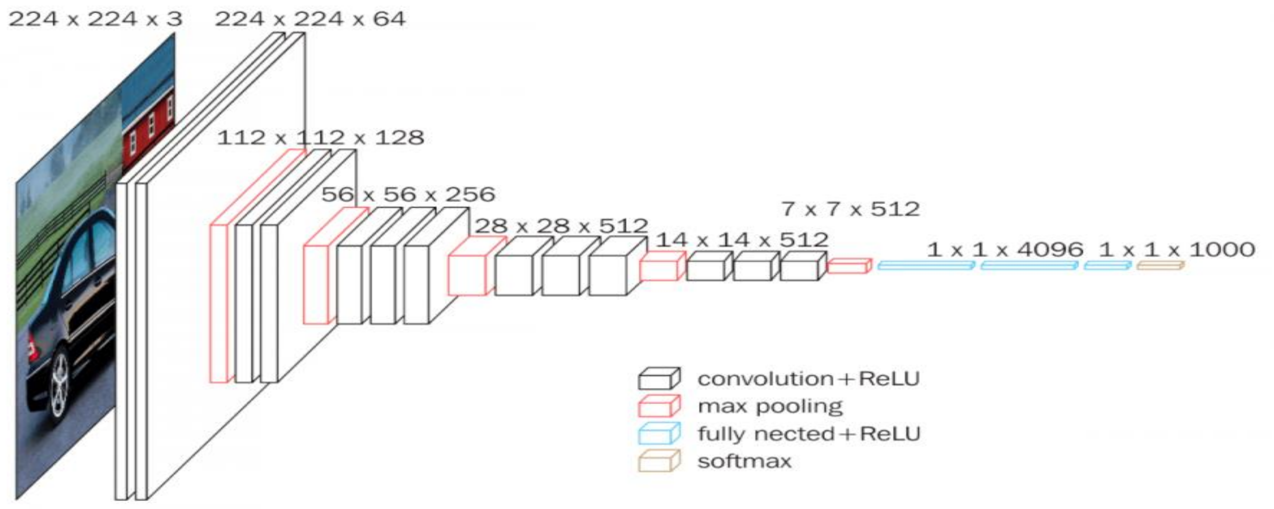
Develop model approach

Please select a source task. Input data, input data, input data, output data, and / or conceptual data, output data, and / or conceptual data, and / or conceptual data to output data, and rich data, which have relevant predictable modeling problems need to be selected. Develop a source model. Next, you need to develop a clever model of this first task. This model must be better than a naive model to ensure that feature learning has been performed. Model reuse models can be used as the starting point of the model of the second object of interest by fitting to the source task. This can be included in all parts of the model depending on the modeling technology used. Model Reuse Model Predefined models can be used as a starting point for models in the second task of interest. This can be included in all parts of the model depending on the modeling technology used. Model models Optionally, the model may need to adapt or refine input pairs data available for the task. This second type of transport learning is common in the field of deep learning.
Representation of deep learning transfer lights
This concrete can be concretely specific in two general examples of transfer learning using deep learning models.
Learn to Transfer Image Data
It is customary to execute transmitters with predictive modeling problems using image data as input image data. This may be a prediction task that shoots photos or video data as input.
These types of issues are common to use a deep learning model for large and sophisticated image classification tasks, such as I000Class Imagenet Competition.
A research group that develops models for this competition and often releases the final model under the allowable reuse permit. These models may take several days or several weeks to train modern hardware.
These models can be downloaded and recorded directly to the new model expected as input image data.

## What is VGG16? — Introduction to VGG16

VGG16 is a simple and widely used Convolutional Neural Network (CNN) Architecture used for ImageNet, a large visual database project used in visual object recognition software research. The VGG16 Architecture was developed and introduced by Karen Simonyan and Andrew Zisserman from the University of Oxford, in the year 2014, through their article "Very Deep Convolutional Networks for Large-Scale Image Recognition." 'VGG' is the abbreviation for Visual Geometry Group, which is a group of researchers at the University of Oxford who developed this architecture, and '16' implies that this architecture has 16 layers (explained later).
The VGG16 model achieved 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous models submitted to ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in the year 2014. It made improvements over AlexNet architecture by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple three × three kernel-sized filters one after another. VGG16 was trained for weeks using NVIDIA Titan Black GPUs.
VGG16 is used in many deep learning image classification techniques and is popular due to its ease of implementation. VGG16 is extensively used in learning applications due to the advantage that it has.
VGG16 is a CNN Architecture, which was used to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014. It is still one of the best vision architecture to date.

## VGG16 architecture



During training, the input to the convnets is a fixed-size 224 x 224 RGB image. Subtracting the mean RGB value computed on the training set from each pixel is the only pre-processing done here. The image is passed through a stack of convolutional (conv.) layers, where filters with a very small receptive field: 3 × 3 (which is the smallest size to capture the notion of left/right, up/down, center and has the same effective receptive field as one 7 x 7), is used. It is deeper, has more non-linearities, and has fewer parameters. In one of the configurations, 1 × 1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity), are also utilized. The convolution stride and the spatial padding of conv. layer input is fixed to 1 pixel for 3 x 3 convolutional layers, which ensures that the spatial resolution is preserved after convolution. Five max-pooling layers, which follow some of the convolutional layers, helps in spatial pooling. Max-pooling is performed over a 2×2 pixel window, with stride 2.
There are three Fully-Connected (FC) layers that follow a stack of convolutional layers (these have different depths in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.
The 16 layer VGG architecture was the best performing, and it achieved a top-5 error rate of 7.3% (92.7% accuracy) in ILSVRC — 2014, as mentioned above. VGG16 had significantly outperformed the previous generation of models ILSVRC — 2012 and ILSVRC — 2013 competitions.

**What is VGG19? — Introduction to VGG19**

VGG19 is a variant of the VGG model which in short consists of 19 layers (16 convolution layers, 3 Fully connected layers, 5 MaxPool layers and 1 SoftMax layer). There are other variants of VGG like VGG11, VGG16 and others. VGG19 has 19.6 billion FLOPs.

AlexNet came out in 2012 and it improved on the traditional Convolutional neural networks, So we can understand VGG as a successor of the AlexNet but it was created by a different group named as Visual Geometry Group at Oxford's and hence the name VGG, It carries and uses some ideas from its predecessors and improves on them and uses deep Convolutional neural layers to improve accuracy.

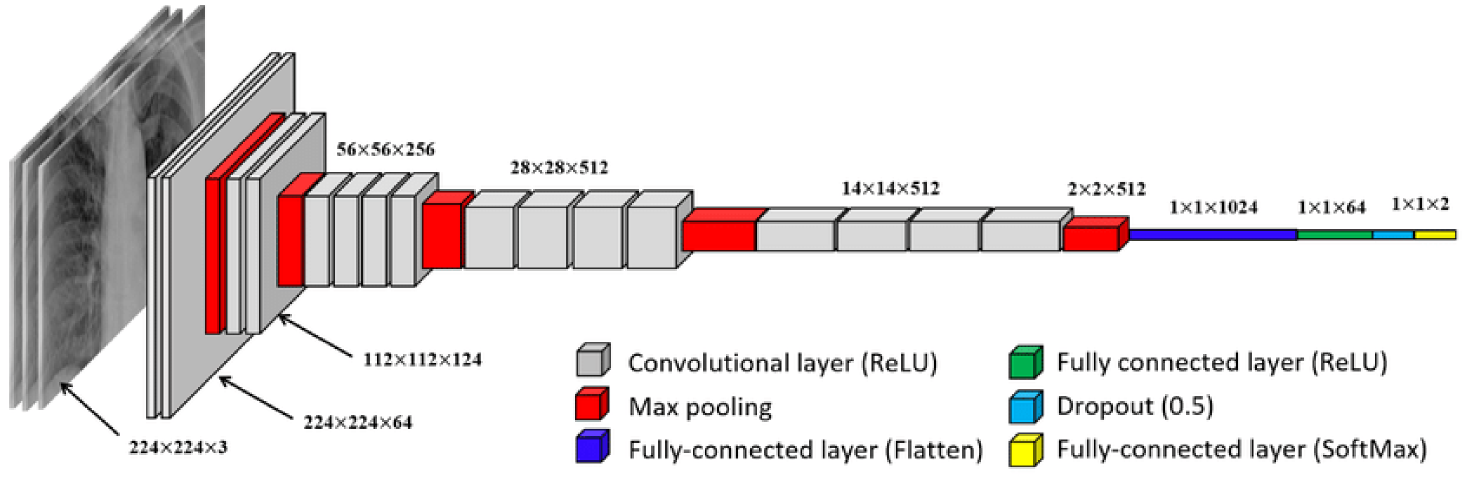
Let's explore what VGG19 is and compare it with some of other versions of the VGG architecture and also see some useful and practical applications of the VGG architecture.

Before diving in and looking at what VGG19 Architecture is, let's take a look at ImageNet and a basic knowledge of CNN.

Convolutional Neural Network(CNN)
First of all let's explore what ImageNet is. It is an Image database consisting of 14,197,122 images organized according to the WordNet hierarchy. This is an initiative to help researchers, students and others in the field of image and vision research.

ImageNet also hosts contests from which one was ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) which challenged researchers around the world to come up with solutions that yields the lowest top-1 and top-5 error rates (top-5 error rate would be the percent of images where the correct label is not one of the model's five most likely labels). The competition gives out a 1,000 class training set of 1.2 million images, a validation set of 50 thousand images and a test set of 150 thousand images

**VGG19 architecture**



56×56×256
28×28×512
14×14×512
2×2×512
1×1×1024
1×1×64
1×1×2
112×112×124
224×224×64
224×224×3
Convolutional layer (ReLU)
Max pooling
Fully-connected layer (Flatten)
Fully connected layer (ReLU)
Dropout (0.5)
Fully-connected layer (SoftMax)

A fixed size of (224 * 224) RGB image was given as input to this network which means that the matrix was of shape (224,224,3).
The only preprocessing that was done is that they subtracted the mean RGB value from each pixel, computed over the whole training set.
Used kernels of (3 * 3) size with a stride size of 1 pixel, this enabled them to cover the whole notion of the image.
spatial padding was used to preserve the spatial resolution of the image.
max pooling was performed over a 2 * 2 pixel windows with stride 2.
This was followed by Rectified linear unit(ReLu) to introduce non-linearity to make the model classify better and to improve computational time as the previous models used tanh or sigmoid functions that proved much better than those.

implemented three fully connected layers from which first two were of size 4096 and after that a layer with 1000 channels for 1000-way ILSVRC classification and the final layer is a softmax function.

**What is Resnet50? — Introduction to Resent50**

ResNet50 is a variant of the ResNet model which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer. It has 3.8 x 10^9 Floating points operations. It is a widely used ResNet model and we have explored ResNet50 architecture in depth.

In 2012 at the ILSVRC2012 classification contest AlexNet won the first prize. After that ResNet was the most interesting thing that happened to computer vision and the deep learning world.

Because of the framework that ResNets presented it was made possible to train ultra deep neural networks and by that I mean that the network can contain hundreds or thousands of layers and still achieve great performance.

The ResNets were initially applied to the image recognition task but as it is mentioned in the paper that the framework can also be used for non computer vision tasks also to achieve better accuracy.

Many of you may argue that simply stacking more layers also gives us better accuracy. Why was there a need for Residual learning for training ultra deep neural networks?

While the Resnet50 architecture is based on the above model, there is one major difference. In this case, the building block was modified into a bottleneck design due to concerns over the time taken to train the layers. This used a stack of 3 layers instead of the earlier 2.

Therefore, each of the 2-layer blocks in Resnet34 was replaced with a 3-layer bottleneck block, forming the Resnet 50 architecture. This has much higher accuracy than the 34-layer ResNet model. The 50-layer ResNet achieves a performance of 3.8 bn FLOPS.

**ResNet50 With Keras**

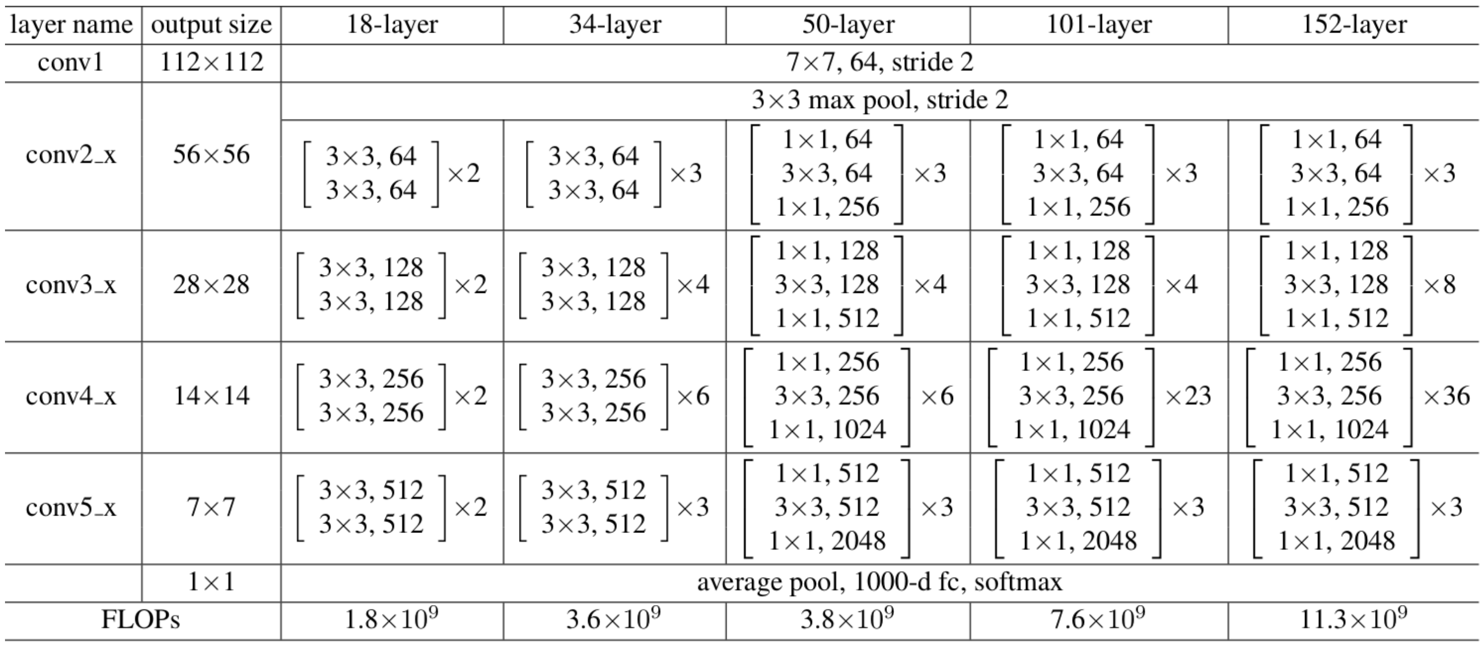
Keras is a deep learning API that is popular due to the simplicity of building models using it. Keras comes with several pre-trained models, including Resnet50, that anyone can use for their experiments.

Therefore, building a residual network in Keras for computer vision tasks like image classification is relatively simple. You only need to follow a few simple steps.

**How to use ResNet50 with Keras**

Step #1: Firstly, you need to run a code to define the identity blocks to transform the CNN into a residual network and build the convolution block.
Step #2: The next step is building the 50-layer Resnet model by combining both blocks.
Step #3: Finally, you need to train the model for the required task. Keras allows you to easily generate a detailed summary of the network architecture you built. This can be saved or printed for future use.

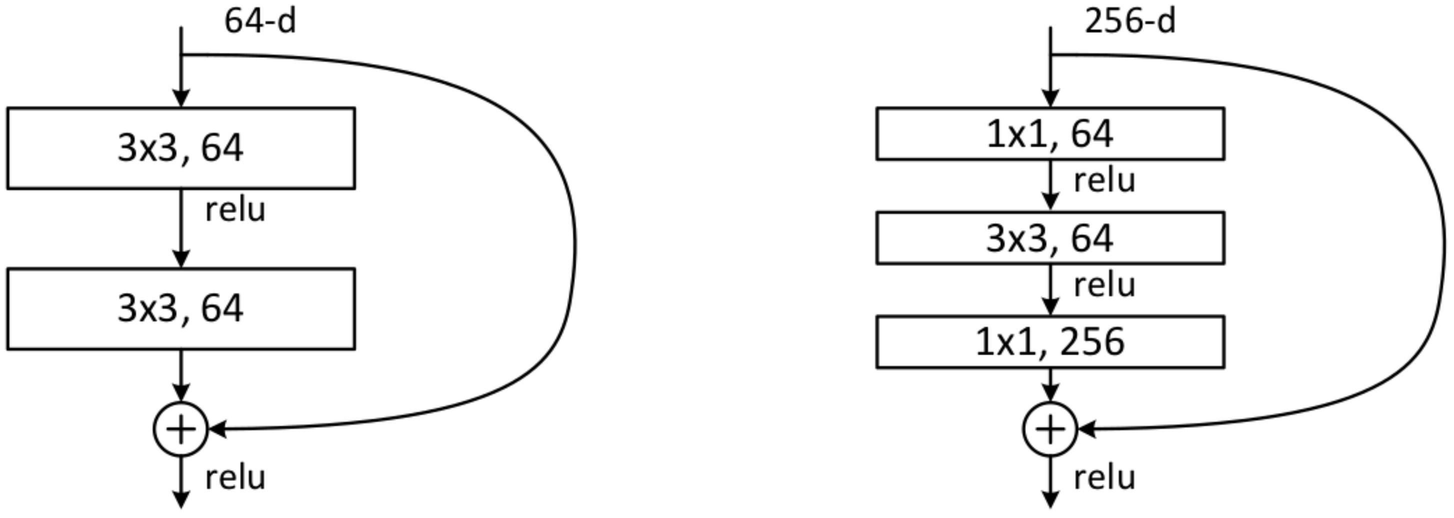
**ResNet50 architecture**



| layer name | output size | 18-layer | 34-layer | 50-layer | 101-layer | 152-layer |
|---|---|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride 2 | | | | |
| conv2_x | 56×56 | 3×3 max pool, stride 2 | | | | |
| | | $\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$ | $\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$ |
| conv3_x | 28×28 | $\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$ | $\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$ | $\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$ | $\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$ | $\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$ |
| conv4_x | 14×14 | $\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$ | $\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$ | $\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$ | $\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$ | $\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$ |
| conv5_x | 7×7 | $\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$ | $\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$ | $\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$ |
| | 1×1 | average pool, 1000-d fc, softmax | | | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ | $7.6\times10^9$ | $11.3\times10^9$ |

Now we are going to discuss Resnet 50 and also the architecture for the above talked 18 and 34 layers ResNet is also given residual mapping and not shown for simplicity.

There was a small change that was made for the ResNet 50 and above that, before this, the shortcut connections skipped two layers but now they skip three layers, and also there were 1 * 1 convolution layers added that we are going to see in detail with the ResNet 50 Architecture.



Convolution with a kernel size of 7 * 7 and 64 different kernels all with a stride of size 2 giving us 1 layer. Next, we see max pooling with also a stride size of 2.

In the next convolution, there is a 1 * 1,64 kernel following this a 3 * 3,64 kernel and at last, a 1 * 1,256 kernels, These three layers are repeated a total 3 times so giving us 9 layers in this step.
Next, we see the kernel of 1 * 1,128 after that a kernel of 3 * 3,128 and at last a kernel of 1 * 1,512 this step was repeated 4 times so giving us 12 layers in this step.
After that, there is a kernel of 1 * 1,256 and two more kernels with 3 * 3,256 and 1 * 1,1024 and this is repeated 6 times giving us a total of 18 layers.
And then again a 1 * 1,512 kernel with two more of 3 * 3,512 and 1 * 1,2048 and this was repeated 3 times giving us a total of 9 layers.
After that, we do an average pool and end it with a fully connected layer containing 1000 nodes and at the end a softmax function so this gives us 1 layer.
We don't actually count the activation functions and the max/ average pooling layers.

so totaling this it gives us a 1 + 9 + 12 + 18 + 9 + 1 = 50 layers Deep Convolutional network.

## Inception v3

### Introduction to Inception models

The Inception V3 is a deep learning model based on Convolutional Neural Networks, which is used for image classification. The inception V3 is a superior version of the basic model Inception V1 which was introduced as GoogLeNet in 2014. As the name suggests it was developed by a team at Google.

### Inception V1

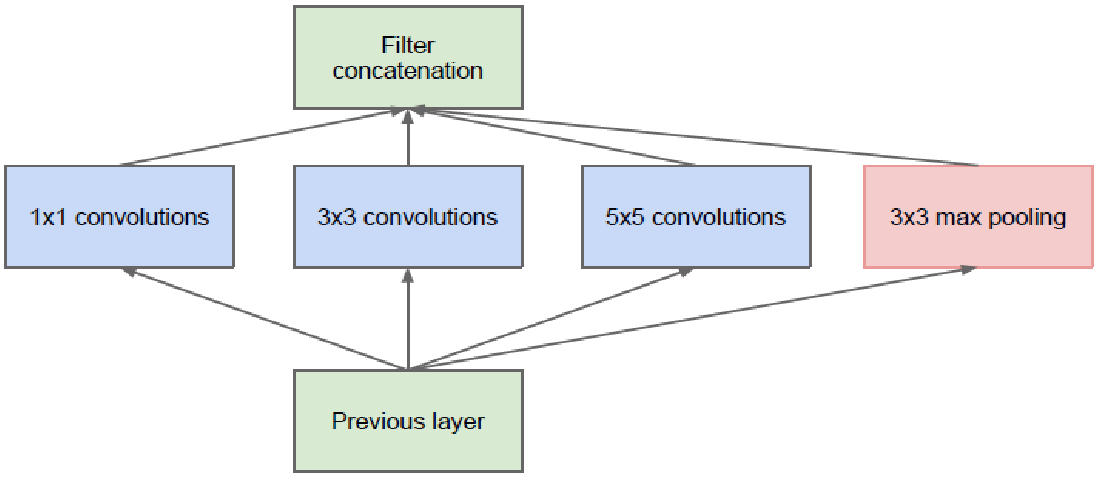
When multiple deep layers of convolutions were used in a model it resulted in the overfitting of the data. To avoid this from happening the inception V1 model uses the idea of using multiple filters of different sizes on the same level. Thus in the inception models instead of having deep layers, we have parallel layers thus making our model wider rather than making it deeper.

The Inception model is made up of multiple Inception modules.

The basic module of the Inception V1 model is made up of four parallel layers.

1×1 convolution
3×3 convolution
5×5 convolution
3×3 max-pooling
**Convolution -** The process of transforming an image by applying a kernel over each pixel and its local neighbors across the entire image.

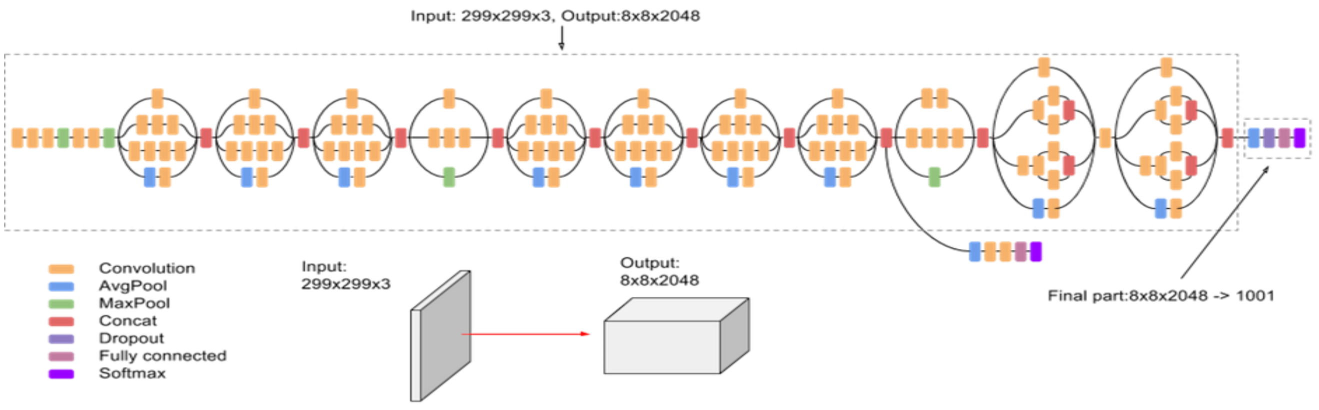
**Pooling -** Pooling is the process used to reduce the dimensions of the feature map. There are different types of pooling but the most common ones are max pooling and average pooling.



This module of the Inception V1 is called the Naive form. One of the drawbacks of this naive form is that even the 5×5 convolutional layer is computationally pretty expensive i.e. time-consuming and requires high computational power.

To overcome this the authors added a 1×1 convolutional layer before each convolutional layer, which results in reduced dimensions of the network and faster computations.

**Inception V3 Model Architecture:**



The inception v3 model was released in the year 2015, it has a total of 42 layers and a lower error rate than its predecessors. Let's look at what are the different optimizations that make the inception V3 model better.

1. The major modifications done on the Inception V3 model are
2. Factorization into Smaller Convolutions
3. Spatial Factorization into Asymmetric Convolutions
4. Utility of Auxiliary Classifiers
5. Efficient Grid Size Reduction

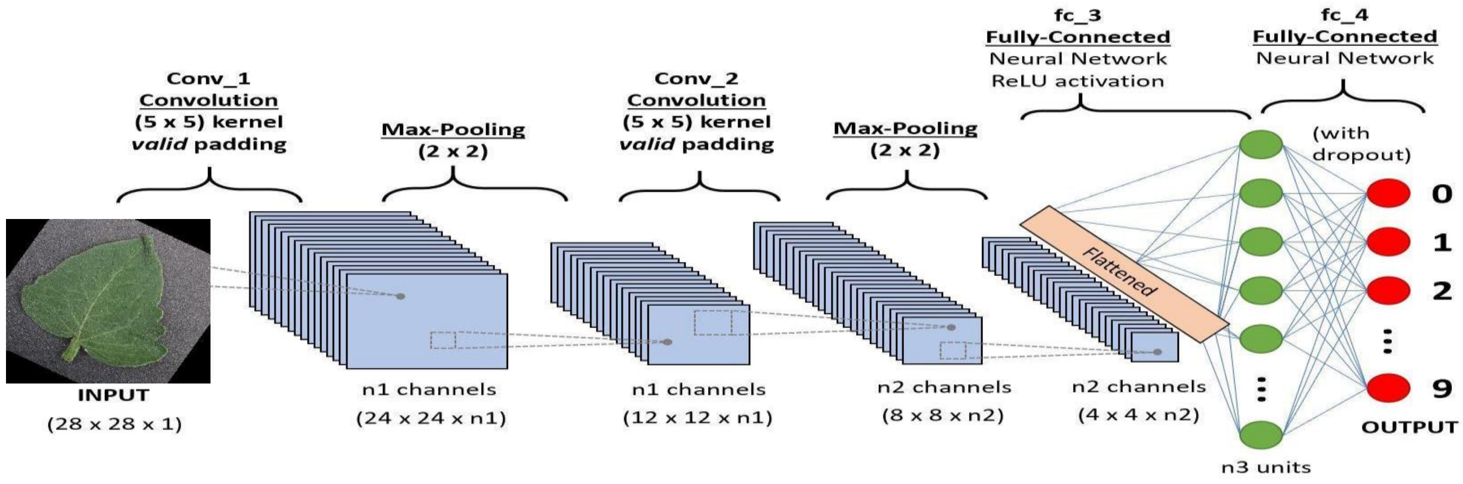
**Training Methodology** :

We have trained our networks with stochastic gradient utilizing the TensorFlow distributed machine learning system using 50 replicas running each on a NVidia Kepler GPU with batch size 32 for 100 epochs. Our earlier experiments used momentum with a decay of 0.9, while our best models were achieved using RMSProp with a decay of 0.9 and ϙ = 1.0. We used a learning rate of 0.045, and decayed every two epochs using an exponential rate of 0.94. In addition, gradient clipping with threshold 2.0 was found to be useful to stabilize the training. Model evaluations are performed using a running average of the parameters computed over time.

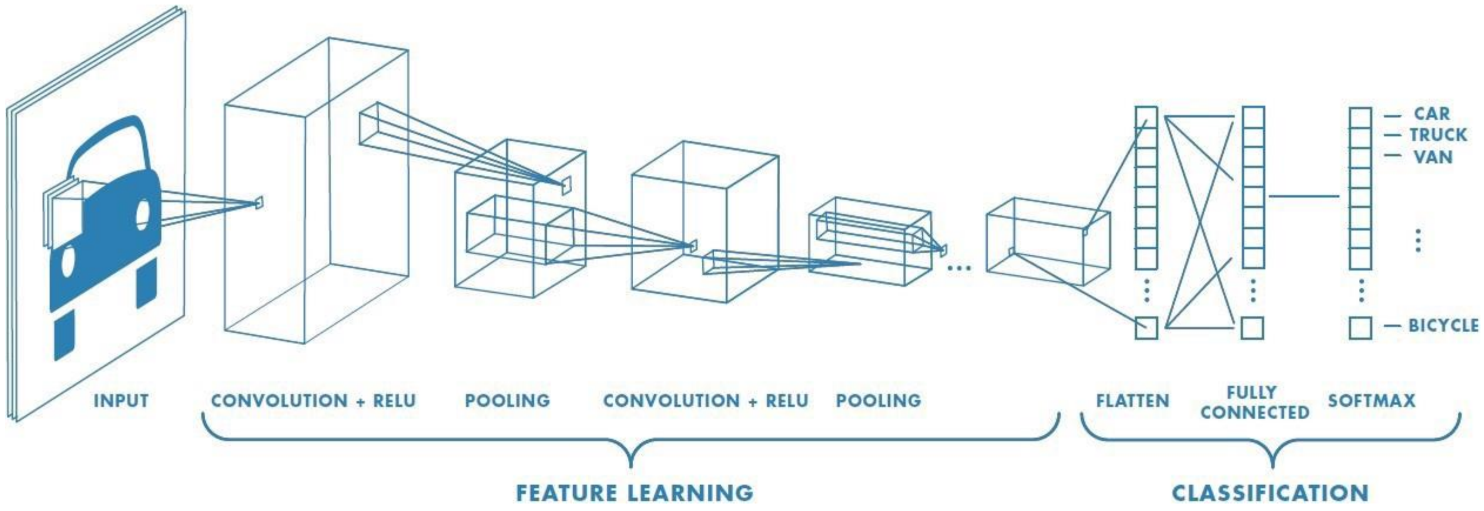
## 4.3 Convolution Neural Network:

Image classification is the task of taking an input image and outputting a class or a probability of classes that best describes the image. In CNN, we take an image as an input, assign importance to its various aspects/features in the image and be able to differentiate one from another. The preprocessing required in CNN is much lesser as compared to other classification algorithms.

## ARCHITECTURE:



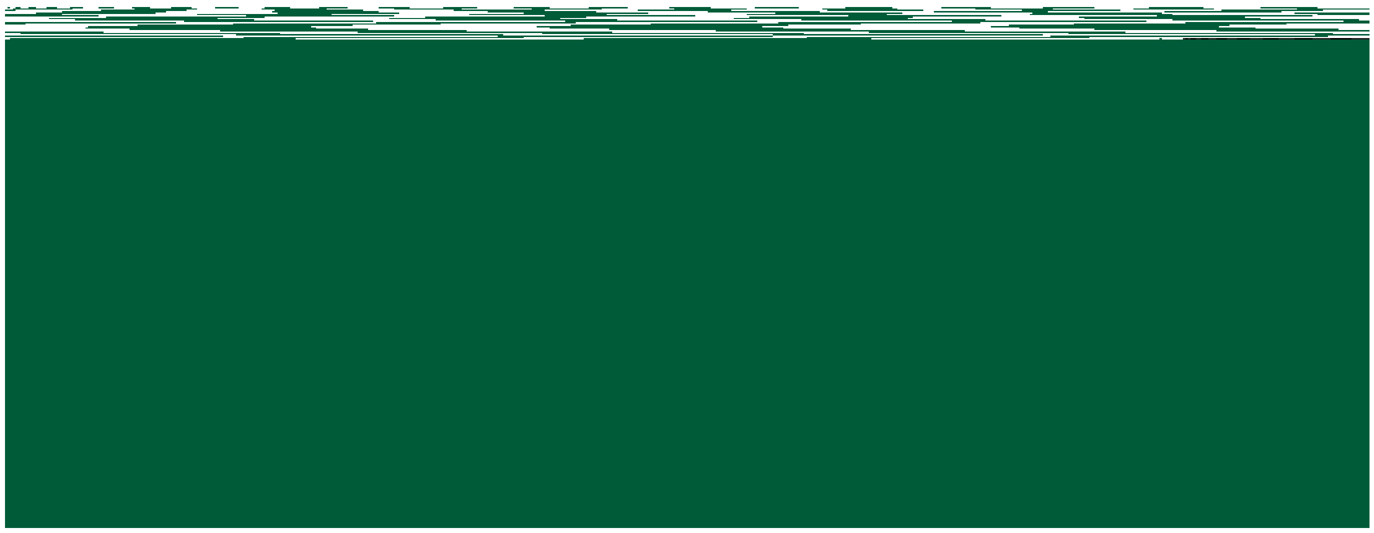
A CNN typically has three layers: a convolutional layer, pooling layer, and fully connected layer.



**Convolutional layer:**

I am pretty sure you have come across the word ‘convolution’ in your life before and here its meaning doesn’t change. Yes! you are right, this layer is all about convolving objects on one another. The convolution layer is the core building block of CNN. It carries the main portion of the network’s computational load.

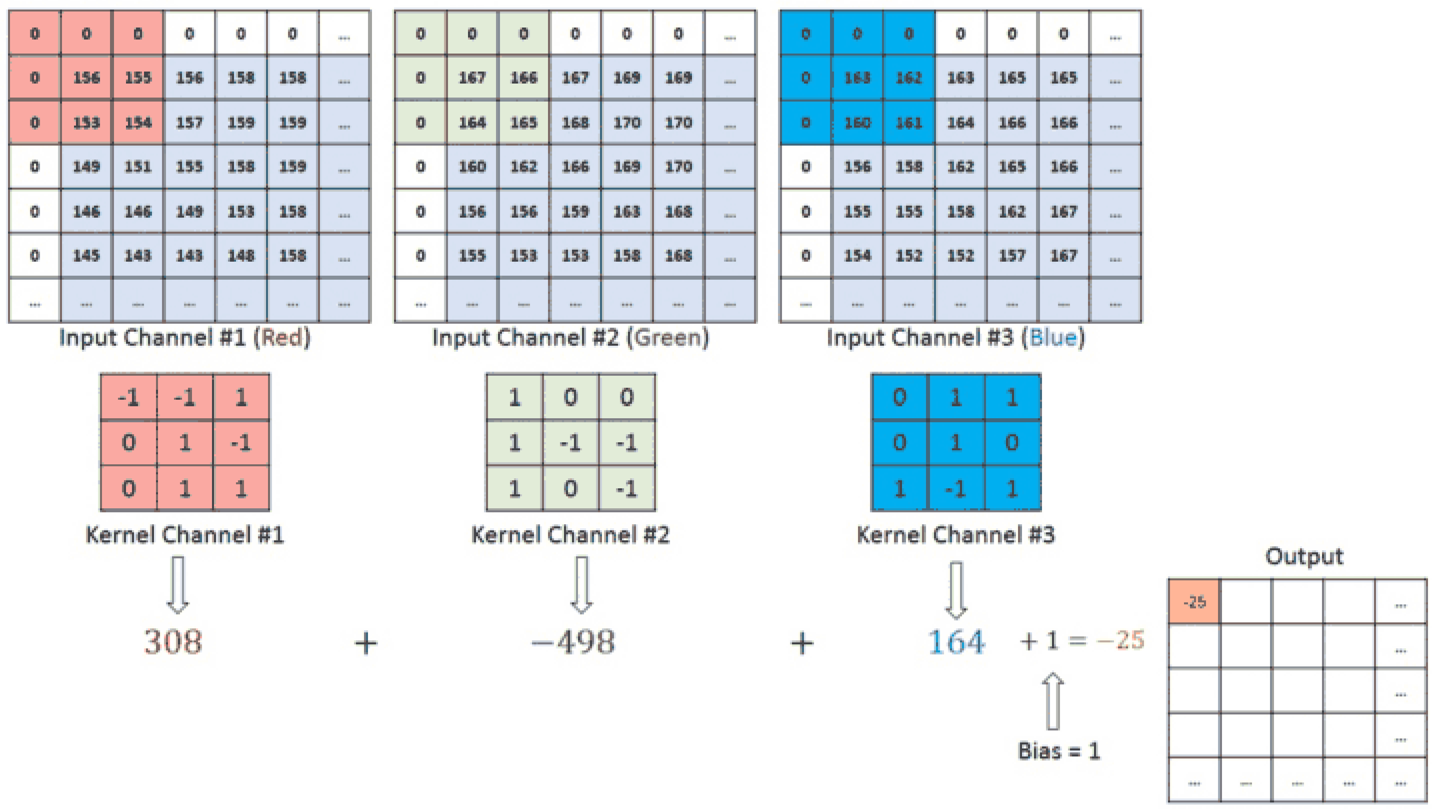
The main objective of convolution is to extract features such as edges, colors, and corners from the input. As we go deeper inside the network, the network starts identifying more complex features such as shapes,digits, face parts as well.



This layer performs a dot product between two matrices, where one matrix(known as filter/kernel)is the set of learnable parameters, and the other matrix is the restricted portion of the image.

If the image is RGB then the filter will have smaller height and width compared to the image but it will have the same depth(height x width x 3) as of the image.

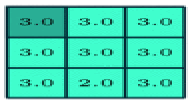
For RGB images, the convolving part can be visualized as follows:



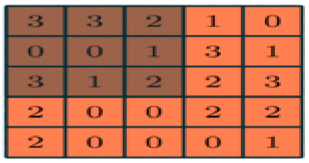
At the end of the convolution process, we have a featured matrix that has fewer parameters(dimensions) than the actual image as well as more clear features than the actual one. So, now we will work with our featured matrix from now on.

**Pooling Layer:**

This layer is solely to decrease the computational power required to process the data. It is done by decreasing the dimensions of the featured matrix even more. In this layer, we try to extract the dominant features from a restricted amount of neighborhood. Let us make it clear by taking an example.



| | | |
|---|---|---|
| 3.0 | 3.0 | 3.0 |
| 3.0 | 3.0 | 3.0 |
| 3.0 | 2.0 | 3.0 |



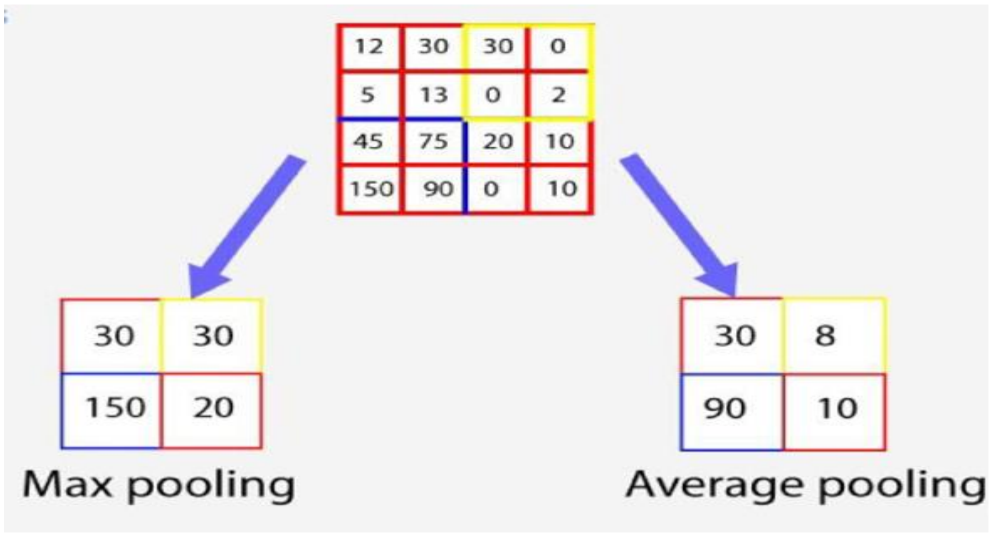
| | | | | |
|---|---|---|---|---|
| 3 | 3 | 2 | 1 | 0 |
| 0 | 0 | 1 | 3 | 1 |
| 3 | 1 | 2 | 2 | 3 |
| 2 | 0 | 0 | 2 | 2 |
| 2 | 0 | 0 | 0 | 1 |

The orange matrix is our featured matrix, the brown one is a pooling kernel and we get our blue matrix as output after pooling is done. So, here what we are doing is taking the maximum amongst all the numbers which are in the pooling region and shifting the pooling region each time to process another neighborhood of the matrix.

There are two types of pooling techniques: AVERAGE-pooling and MAX-pooling.

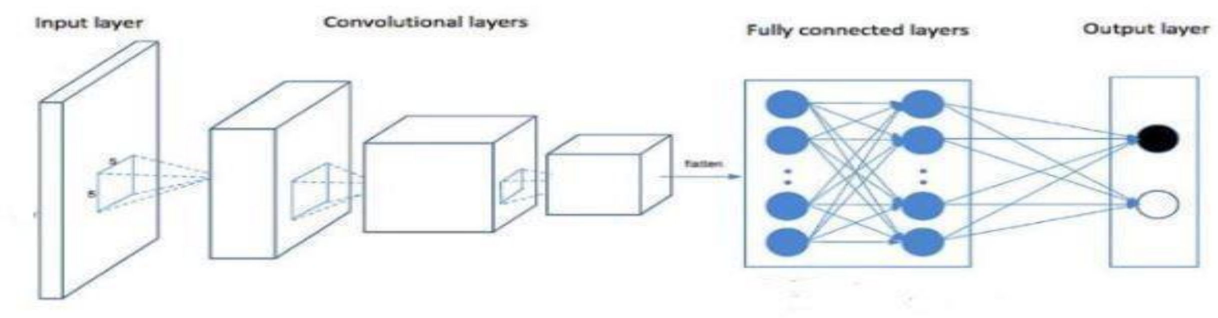
The difference between these two is, in AVERAGE-pooling, we take the average of all the values of the pooling region and in MAX-pooling, we just take the maximum amongst all the values lying inside the pooling region.

So, after the pooling layer, we have a matrix containing the main features of the image and this matrix has even lesser dimensions, which will help a lot in the next step.



**Fully connected layer:**

Till now we haven't done anything about classifying different images, what we have done is highlight some features in an image and reduce the dimensions of the image drastically.

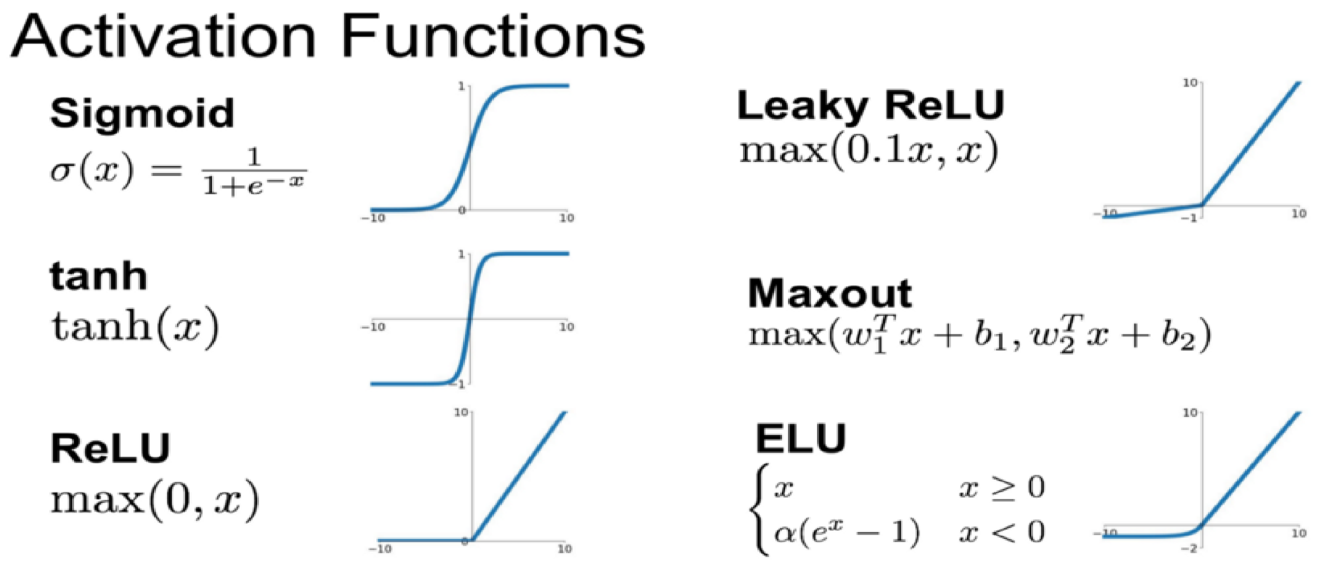


From here on, we are actually going to do the classification process.

Now that we have converted our input image into a suitable form for our Multi-Level fully connected architecture, we shall flatten the image into one column vector. The flattened output is fed to a feed-forward neural network and backpropagation is applied to every iteration of training. Over a series of epochs, the model can distinguish between dominating and certain low-level features in images and classify them.

**Activation Function**

The activation function is a node that is put at the end of or in between Neural Networks. They help to decide if the neuron would fire or not.



We have different types of activation functions just as the figure above, but for this post, my focus will be on Rectified Linear Unit (ReLU).

ReLU function is the most widely used activation function in neural networks today. One of the greatest advantages ReLU has over other activation functions is that it does not activate all neurons at the same time. From the image for ReLU function above, we'll notice that it converts all negative inputs to zero and the neuron does not get activated. This makes it very computational efficient as few neurons are activated per time. It does not saturate at the positive region. In practice, ReLU converges six times faster than tanh and sigmoid activation functions.

Some disadvantage ReLU presents is that it is saturated at the negative region, meaning that the gradient at that region is zero. With the gradient equal to zero, during backpropagation all the weights will not be updated, to fix this, we use Leaky ReLU. Also, ReLU functions are not zero-centered. This means that for it to get to its optimal point, it will have to use a zig-zag path which may be longer.

**What is the Sigmoid Function?**

A Sigmoid function is a mathematical function which has a characteristic S-shaped curve. There are a number of common sigmoid functions, such as the logistic function, the hyperbolic tangent, and the arctangent

. In machine learning, the term

sigmoid function is normally used to refer specifically to the logistic function, also called the logistic sigmoid function.

All sigmoid functions have the property that they map the entire number line into a small range such as between 0 and 1, or -1 and 1, so one use of a sigmoid function is to convert a real value into one that can be interpreted as a probability.

**What is ReLU Activation Function?**

ReLU stands for rectified linear activation unit and is considered one of the few milestones in the deep learning revolution. It is simple yet really better than its predecessor activation functions such as sigmoid or tanh.

ReLU activation function formula

Now how does ReLU transform its input? It uses this simple formula:

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

The ReLU function is its derivative and both are monotonic. The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. Thus it gives an output that has a range from 0 to infinity.

**Why is ReLU the best activation function?**

As we have seen above, the ReLU function is simple and it consists of no heavy computation as there is no complicated math. The model can, therefore, take less time to train or run. One more important property that we consider the advantage of using ReLU activation function is sparsity.

Usually, a matrix in which most entries are 0 is called a sparse matrix and similarly, we desire a property like this in our neural networks where some of the weights are zero. Sparsity results in concise models that often have better predictive power and less overfitting/noise. In a sparse network, it's more likely that neurons are actually processing meaningful aspects of the problem. For example, in a model detecting human faces in images, there may be a neuron that can identify ears, which obviously shouldn't be activated if the image is not of a face and is a ship or mountain.

Since ReLU gives output zero for all negative inputs, it's likely for any given unit to not activate at all which causes the network to be sparse. Now let us see how the ReLu activation function is better than previously famous activation functions such as sigmoid and tanh.

The activation functions that were used mostly before ReLU such as sigmoid or tanh activation function saturated. This means that large values snap to 1.0 and small values snap to -1 or 0 for tanh and sigmoid respectively. Further, the functions are only really sensitive to changes around the mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh. This caused them to have a problem called vanishing gradient problem. Let us briefly see what the vanishing gradient problem is.

Neural Networks are trained using the process gradient descent. The gradient descent consists of the backward propagation step which is basically chain rule to get the change in weights in order to reduce the loss after every epoch. It is important to note that the derivatives play an important role in updating weights. Now when we use activation functions such as sigmoid or tanh, whose derivatives have only decent values from a range of -2 to 2 and are flat elsewhere, the gradient keeps decreasing with the increasing number of layers.

This reduces the value of the gradient for the initial layers and those layers are not able to learn properly. In other words, their gradients tend to vanish because of the depth of the network and the activation shifting the value to zero. This is called the vanishing gradient problem.

ReLU, on the other hand, does not face this problem as its slope doesn't plateau, or "saturate," when the input gets large. Due to this reason models using ReLU activation function converge faster.

But there are some problems with ReLU activation functions such as exploding gradients. The exploding gradient is opposite of the vanishing gradient and occurs where large error gradients accumulate and result in very large updates to neural network model weights during training. Due to this, the model is unstable and unable to learn from your training data.

Also, there is a downside to being zero for all negative values and this problem is called "dying ReLU."A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0. Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover. Such neurons are not playing any role in discriminating the input and is essentially useless. Over time you may end up with a large part of your network doing nothing. The dying problem is likely to occur when the learning rate is too high or there is a large negative bias.

Lower learning rates often alleviate this problem. Alternatively, we can use Leaky ReLU which we will discuss next.

## What is the Softmax Function?

The softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

The softmax function is sometimes called the softmax function, or multi-class logistic regression. This is because the softmax is a generalization of logistic regression that can be used for multi-class classification, and its formula is very similar to the sigmoid function which is used for logistic regression. The softmax function can be used in a classifier only when the classes are mutually exclusive.

Many multi-layer neural networks end in a penultimate layer that outputs real-valued scores that are not conveniently scaled and which may be difficult to work with. Here the softmax is very useful because it converts the scores to a normalized probability distribution, which can be displayed to a user or used as input to other systems. For this reason, it is usual to append a softmax function as the final layer of the neural network.

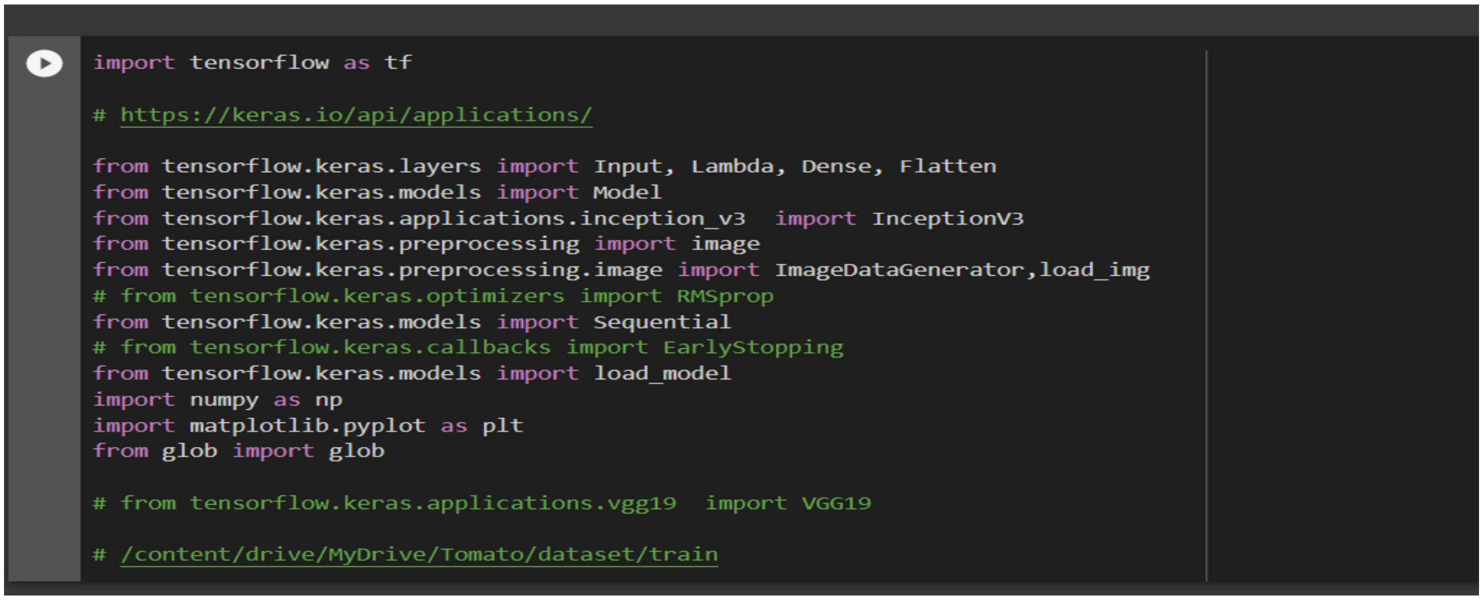
**IMPLEMENTATION:**

**Inception v3 performs well when we compare it to all other Transfer Learning algorithms**

**Step: 1**

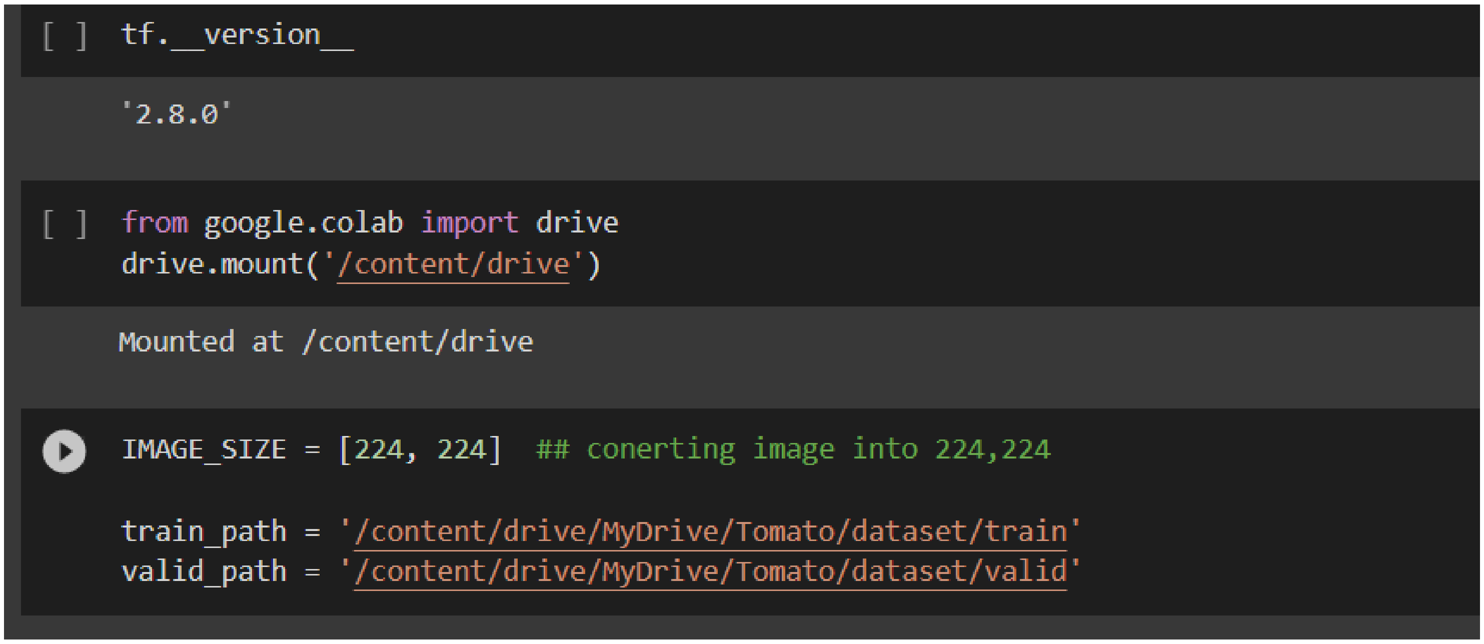
Import libraries

Here we use the tensorflow 2.8.0 version so inbuilt keras and importing all the necessary libraries



```
import tensorflow as tf

# https://keras.io/api/applications/

from tensorflow.keras.layers import Input, Lambda, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications.inception_v3  import InceptionV3
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator,load_img
# from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.models import Sequential
# from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.models import load_model
import numpy as np
import matplotlib.pyplot as plt
from glob import glob

# from tensorflow.keras.applications.vgg19  import VGG19

# /content/drive/MyDrive/Tomato/dataset/train
```

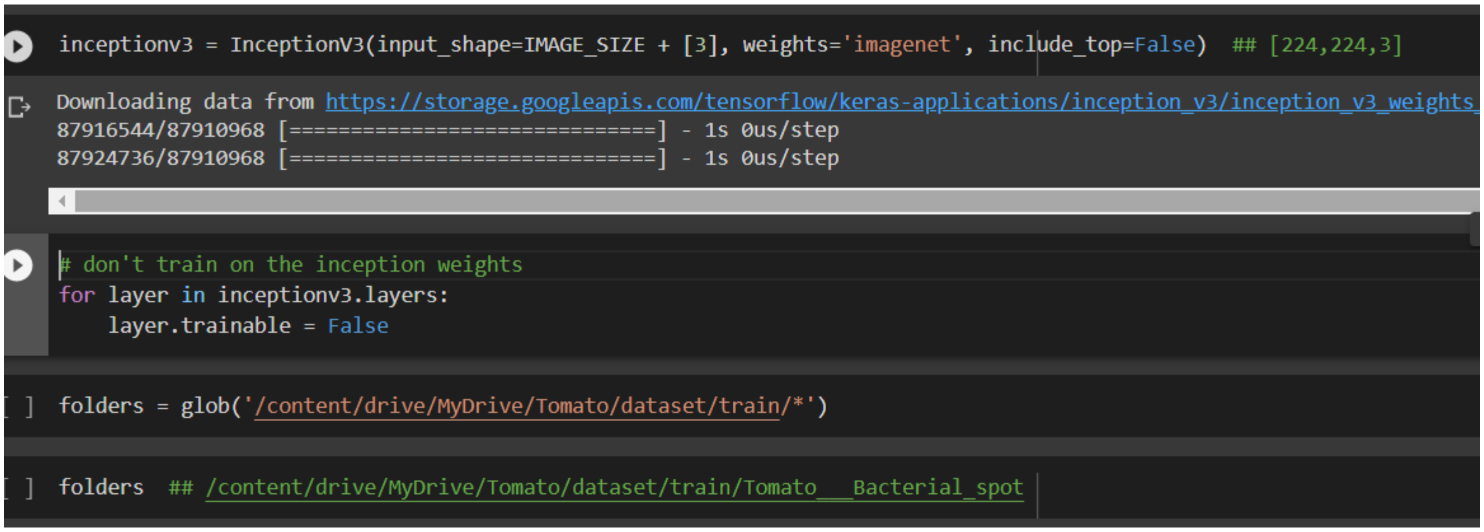
**Step: 2**



```
[ ] tf.__version__

    '2.8.0'

[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive

    IMAGE_SIZE = [224, 224]  ## conerting image into 224,224

    train_path = '/content/drive/MyDrive/Tomato/dataset/train'
    valid_path = '/content/drive/MyDrive/Tomato/dataset/valid'
```

1. Here we are using the TensorFlow version of the 2.8.0 version where there is an inbuilt Keras in it, if you are using the TensorFlow version of 2.2.0 then you have to import Keras separately
2. we have mounted to the google drive
3. Giving the image size as [224,224] where every input size will change to 224,224 and we have given the train and test(valid) paths in which the data is present in the drive

**Step: 3**



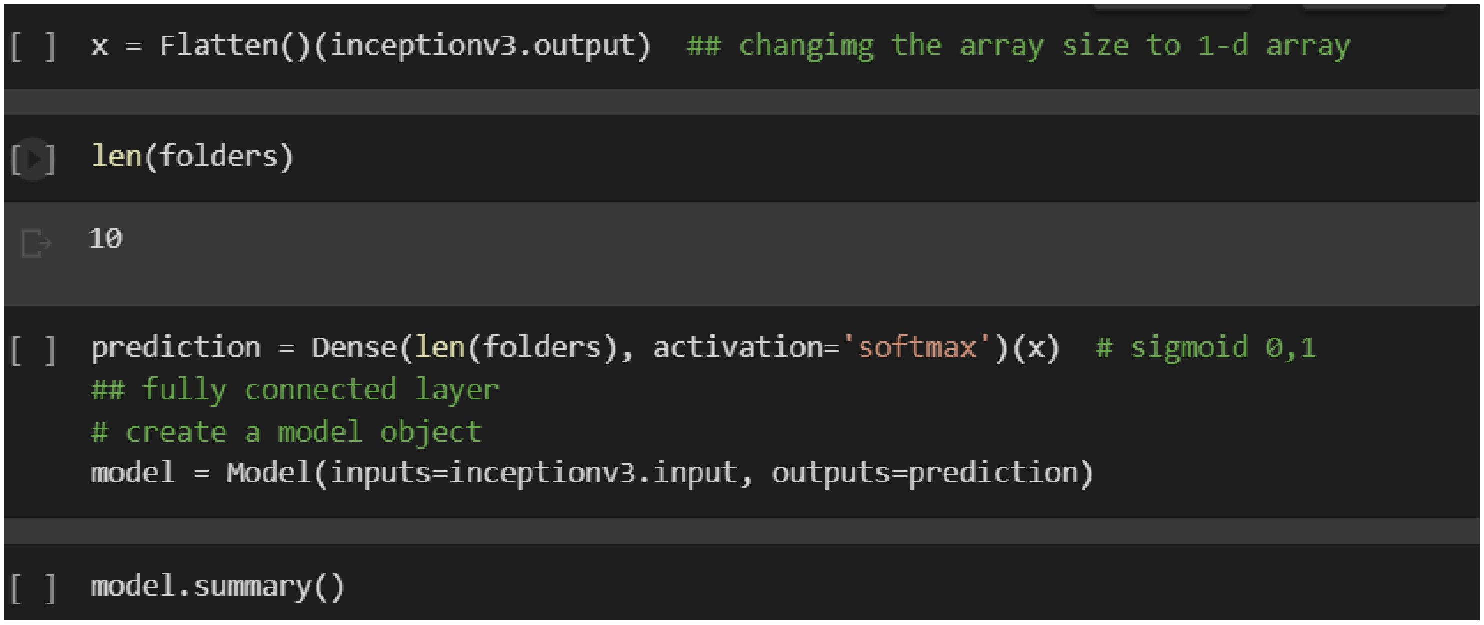
```
inceptionv3 = InceptionV3(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False)  ## [224,224,3]

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/inception_v3/inception_v3_weights_
87916544/87910968 [==============================] - 1s 0us/step
87924736/87910968 [==============================] - 1s 0us/step

# don't train on the inception weights
for layer in inceptionv3.layers:
    layer.trainable = False

] folders = glob('/content/drive/MyDrive/Tomato/dataset/train/*')

] folders  ## /content/drive/MyDrive/Tomato/dataset/train/Tomato___Bacterial_spot
```

1. we have run the model of InceptionV3 if you are running for the first time you need it will

download the model (it will take some time) and we take the weights of “image net” it is a default parameter and we will include_top = False we don’t need the top layer (input layer of “imagenet”)

2. we don’t need to train on the existing weights of the model we need to use new weights to update
3. we will connect to the glob to see all the folders in training data, if you see we have 10 classification or 10 different types of leaves where 1 is healthy and other 9 are disease
4. if you see the length of folders you can see 10 classes

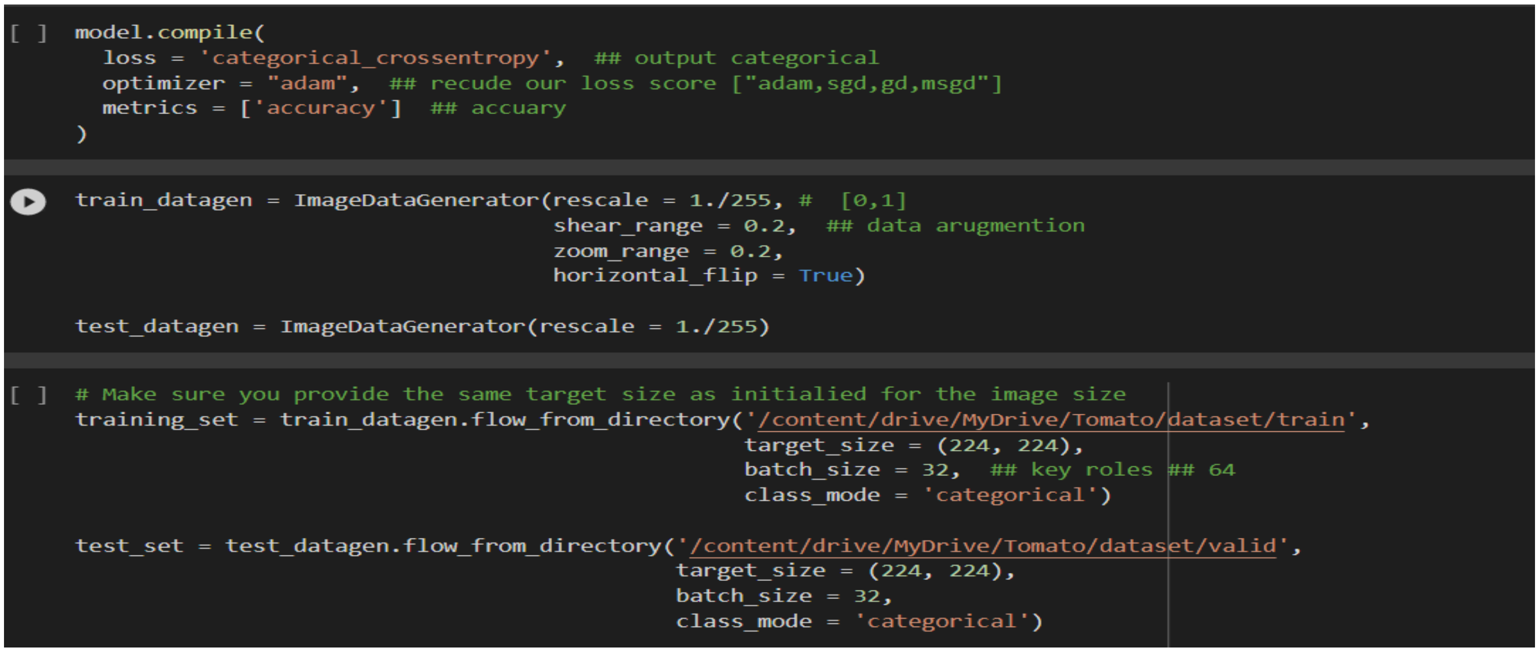
**Step: 4**



```
[ ]  x = Flatten()(inceptionv3.output)  ## changimg the array size to 1-d array

[ ]  len(folders)

     10

[ ]  prediction = Dense(len(folders), activation='softmax')(x)  # sigmoid 0,1
     ## fully connected layer
     # create a model object
     model = Model(inputs=inceptionv3.input, outputs=prediction)

[ ]  model.summary()
```

1. Adding flatten layer to it and output feature
2. Connecting the fully connected layer by Dense function and using “SIGMOID” activation function and putting the model inputs and outputs in model
3. If you look model.summary()
4. You can see the entire model

**Step: 5**



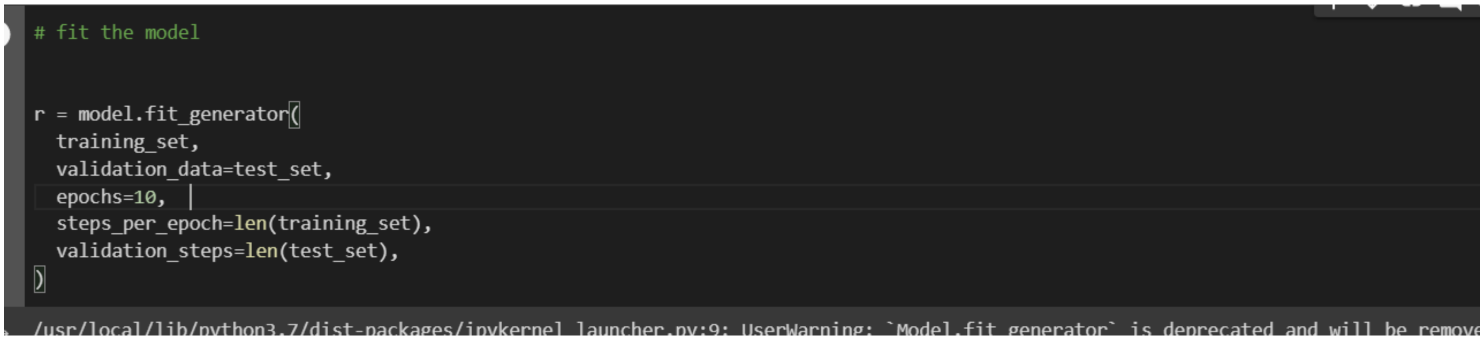
```
[ ] model.compile(
      loss = 'categorical_crossentropy',  ## output categorical
      optimizer = "adam",  ## recude our loss score ["adam,sgd,gd,msgd"]
      metrics = ['accuracy']  ## accuary
    )
```

```
    train_datagen = ImageDataGenerator(rescale = 1./255, #  [0,1]
                                       shear_range = 0.2,  ## data arugmention
                                       zoom_range = 0.2,
                                       horizontal_flip = True)

    test_datagen = ImageDataGenerator(rescale = 1./255)
```

```
[ ] # Make sure you provide the same target size as initialied for the image size
    training_set = train_datagen.flow_from_directory('/content/drive/MyDrive/Tomato/dataset/train',
                                                     target_size = (224, 224),
                                                     batch_size = 32,  ## key roles ## 64
                                                     class_mode = 'categorical')

    test_set = test_datagen.flow_from_directory('/content/drive/MyDrive/Tomato/dataset/valid',
                                                target_size = (224, 224),
                                                batch_size = 32,
                                                class_mode = 'categorical')
```

1. we will compile the model and use the loss function as categorical_crossentropy because our output features is categorical features
2. we use adam optimizers, we have a lot of optimizers. Adagard and SGD, Mini Batch SGD, and many but out of all adam performs good
3. we use "accuray" as our metrics to see the model accuracy
4. we have done the data argumentation this is the important step where the images are horizontally flipped,batch_size=32,class_mode is the output feature is of categorical features

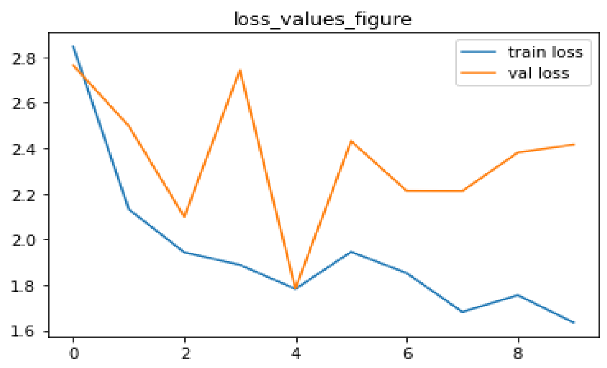
**Step: 6**

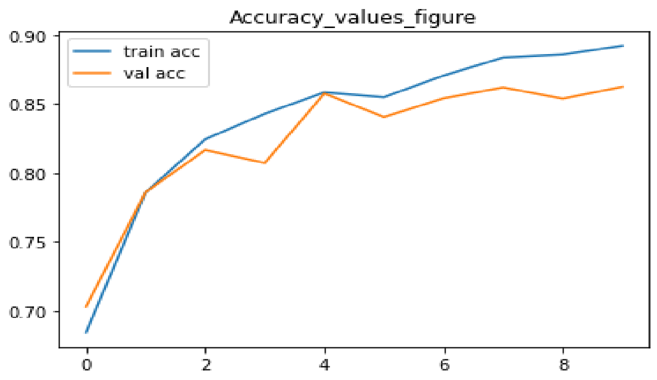


```
# fit the model


r = model.fit_generator(
  training_set,
  validation_data=test_set,
  epochs=10,
  steps_per_epoch=len(training_set),
  validation_steps=len(test_set),
)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:9: UserWarning: `Model.fit_generator` is deprecated and will be remove
```

We will fit the model by giving the parameters of train data, validation data as the test data and epochs of 10 based on the hyper meter tuning and other parameters (This step will take some time to execute)

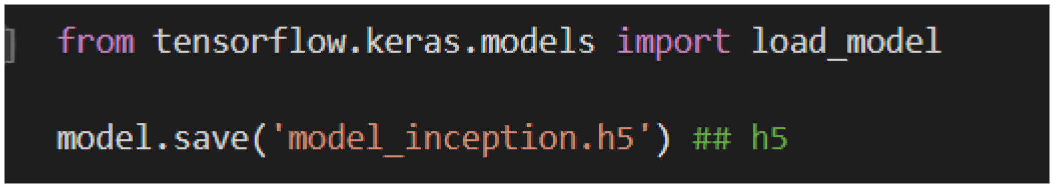
**Loss Score AND Accuracy Score:**





**Saving the model:**

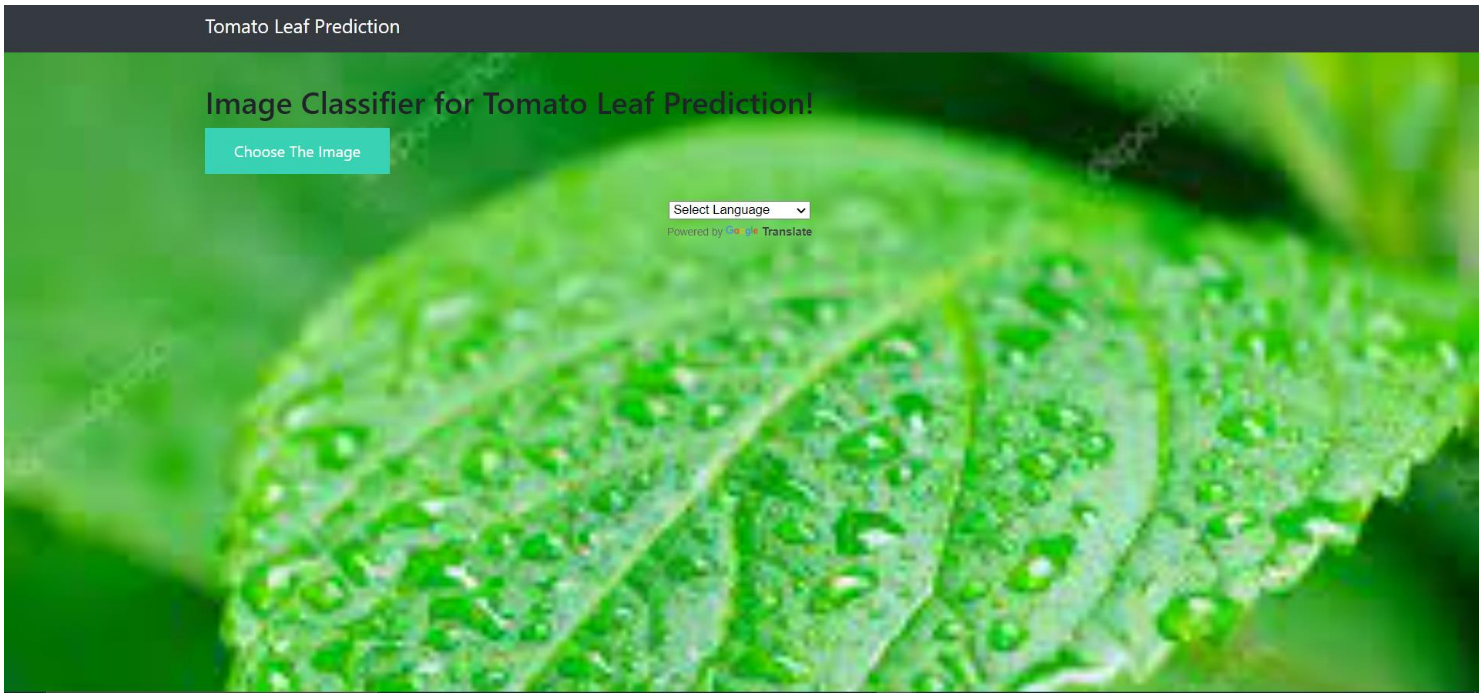
We will save the model to the h5 file where the h5 file is the large file



```
from tensorflow.keras.models import load_model

model.save('model_inception.h5') ## h5
```

And now we will write the front end code and link those both the model and the front end by using FLASK web framework in python

## Conclusion :


Tomato Leaf Prediction
Image Classifier for Tomato Leaf Prediction!
Choose The Image
Select Language
Powered by Google Translate

**The above image is the website page of this where you guys upload the image in the upload image and then the image is being in the uploads folders in my localhost my model will preprocess the image and predict**

**the disease of the leaf and what are the precocious need to take to make it as a healthy leaf and we have added a localization where the person can see change the output language to the local language so that they can easily understand and happy ending !**

**References:**

- Agarwal, Mohit & Singh, Abhishek & Arjaria, Siddhartha & Sinha, Amit & Gupta, Suneet. (2020). ToLeD: Tomato Leaf Disease Detection using Convolutional Neural Network. Procedia Computer Science. 167. 293-301. 10.1016/j.procs.2020.03.225

- P. Tm, A. Pranathi, K. SaiAshritha, N. B. Chittaragi and S. G. Koolagudi, "Tomato Leaf Disease Detection Using Convolutional Neural Networks," *2018 Eleventh International Conference on Contemporary Computing (IC3)*, 2018, pp. 1-5, doi: 10.1109/IC3.2018.8530532.

- T. T. Mim, M. H. Sheik, R. A. Shampa, M. S. Reza and M. S. Islam, "Leaves Diseases Detection of Tomato Using Image Processing," *2019 8th International Conference System Modeling and Advancement in Research Trends (SMART)*, 2019, pp. 244-249, doi: 10.1109/SMART46866.2019.9117437.