## Abstract

When plants and crops are affected by pests it affects the agricultural production of the country. Usually farmers or experts observe the plants with naked eye for detection and identification of disease. But this method can be time processing, expensive and inaccurate. Automatic detection using image processing techniques provide fast and accurate results. This paper is concerned with a new approach to the development of plant disease recognition model, based on leaf image classification, by the use of deep convolutional networks. Advances in computer vision present an opportunity to expand and enhance the practice of precise plant protection and extend the market of computer vision applications in the field of precision agriculture. Novel way of training and the methodology used facilitate a quick and easy system implementation in practice. All essential steps required for implementing this disease recognition model are fully described throughout the paper, starting from gathering images in order to create a database, assessed by agricultural experts, a deep learning framework to perform the deep CNN training. This method paper is a new approach in detecting plant diseases using the deep convolutional neural network trained and fine-tuned to fit accurately to the database of a plant’s leaves that was gathered independently for diverse plant diseases. The advance and novelty of the developed model lie in its simplicity; healthy leaves and background images are in line with other classes, enabling the model to distinguish between diseased leaves and healthy ones or from the environment by using CNN.

***INTRODUCTION***

The problem of efficient plant disease protection is closely related to the problems of sustainable agriculture Inexperienced pesticide usage can cause the development of long-term resistance of the pathogens, severely reducing the ability to fight back. Timely and accurate diagnosis of plant diseases is one of the pillars of precision agriculture. It is crucial to prevent unnecessary waste of financial and other resources, thus achieving healthier production in this changing environment, appropriate and timely disease identification including early prevention has never been more important. There are several ways to detect plant pathologies. Some diseases do not have any visible symptoms, or the effect becomes noticeable too late to act, and in those situations, a sophisticated analysis is obligatory. However, most diseases generate some kind of manifestation in the visible spectrum, so the naked eye examination of a trained professional is the prime technique adopted in practice for plant disease detection. In order to achieve accurate plant disease diagnostics a plant pathologist should possess good observation skills so that one can identify characteristic symptoms. Variations in symptoms indicated by diseased plants may lead to an improper diagnosis since amateur gardeners and hobbyists could have more difficulties determining it than a professional plant pathologist. An automated system designed to help identify plant diseases by the plant’s appearance and visual symptoms could be of great help to amateurs in the gardening process and also trained professionals as a verification system in disease diagnostics. Advances in computer vision present an opportunity to expand and enhance the practice of precise plant protection and extend the market of computer vision applications in the field of precision agriculture. Exploiting common digital image processing techniques such as colour analysis and thresholding were used with the aim of detection and classification of plant diseases. In machine learning and cognitive science, ANN is an information-processing paradigm that was inspired by the way biological nervous systems, such as the brain, process information. Neural networks or connectionist systems are a computational approach used in computer science and other research disciplines, which is based on a large collection of neural units (artificial neurons), loosely mimicking the way a biological brain solves problems with large clusters of biological neurons connected by axons. Each neural unit is connected with many others, and links can be enforcing or inhibitory in their effect on the activation state of connected neural units. Each individual neural unit may have a summation function which combines the values of all its inputs together. There may be a threshold function or limiting function on each connection and on the unit itself, such that the signal must surpass the limit before propagating to other neurons. These systems are self-learning and trained, rather than explicitly programmed, and excel in areas where the solution or feature detection is difficult to express in a traditional computer program. Neural networks typically consist of multiple layers or a cube design, and the signal path traverses from front to back. Back propagation is the use of forward stimulation to reset weights on the "front" neural units and this is sometimes done in combination with training where the correct result is known. More modern networks are a bit more free flowing in terms of stimulation and inhibition with connections interacting in a much more chaotic and complex fashion. Dynamic neural networks are the most advanced, in that they dynamically can, based on rules, form new connections and even new neural units while disabling others.

The goal of the neural network is to solve problems in the same way that the human brain would, although several neural networks are more abstract. Modern neural network projects typically work with a few thousand to a few million neural units and millions of connections, which are still several orders of magnitude less complex than the human brain and closer to the computing power of a worm. New brain research often stimulates new patterns in neural networks. One new approach is using connections which span much further and link processing layers rather than always being localized to adjacent neurons. Other research being explored with the different types of signal over time that axons propagate, such as Deep Learning, interpolates greater complexity than a set of Boolean variables being simply on or off. Their inputs can also take on any value between 0 and 1. Also, the neuron has weights for each input and an overall bias. The weights are real numbers expressing importance of the respective inputs to the output. The bias is used for controlling how easy the neuron is getting to output 1. For a neuron with really big bias it is easy to output 1, but when the bias is very negative then it is difficult to output 1.

# Materials and Methods:-

# The Dataset:-

# The Dataset was taken from kaggle of PlantVillage dataset present online as such the code was also written on the online kernel of Kaggle for better computation and analysis of training loss and validation.

# Image Preprocessing and Labelling:

# Preprocessing images commonly involves removing low-frequency background noise, normalizing the intensity of the individual particles images, removing reflections, and masking portions of images. Image preprocessing is the technique of enhancing data Furthermore, procedure of image preprocessing involved cropping of all the images manually, making the square around the leaves, in order to highlight the region of interest (plant leaves). During the phase of collecting the images for the dataset, images with smaller resolution and dimension less than 500 pixels were not considered as valid images for the dataset. In addition, only the images where the region of interest was in higher resolution were marked as eligible candidates for the dataset. In that way, it was ensured that images contain all the needed information for feature learning. Many resources can be found by searching across the Internet, but their relevance is often unreliable. In the interest of confirming the accuracy of classes in the dataset, initially grouped by a keywords search, agricultural experts examined leaf images and labeled all the images with appropriate disease acronym. As it is known, it is important to use accurately classified images for the training and validation dataset. Only in that way may an appropriate and reliable detecting model be developed. In this stage, duplicated images that were left after the initial iteration of gathering and grouping images into classes were removed from the dataset.

# Neural Network Training :-

Training the deep convolutional neural network for making an image classification model from a dataset was proposed. Tensor Flow is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. Tensor Flow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well. In machine learning, a convolutional neural network is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli in a restricted region of space known as the receptive field. The receptive fields of different neurons partially overlap such that they tile the visual field. The response of an individual neuron to stimuli within its receptive field can be approximated mathematically by a convolution operation. Convolutional networks were inspired by biological processes and are variations of multilayer perceptron designed to use minimal amounts of pre-processing. They have wide applications in image and video recognition, recommender systems and natural language processing. Convolutional neural networks (CNNs) consist of multiple layers of receptive fields. These are small neuron collections which process portions of the input image. The outputs of these collections are then tiled so that their input regions overlap, to obtain a higher-resolution representation of the original image; this is repeated for every such layer. Tiling allows CNNs to tolerate translation of the input image. Convolutional networks may include local or global pooling layers, which combine the outputs of neuron clusters. They also consist of various combinations of convolutional and fully connected layers, with point wise nonlinearity applied at the end of or after each layer. A convolution operation on small regions of input is introduced to reduce the number of free parameters and improve generalization .One major advantage of convolutional networks is the use of shared weight in convolutional layers, which means that the same filter (weights bank) is used for each pixel in the layer; this both reduces memory footprint and improves performance. The layer’s parameters are comprised of a set of learnable kernels which possess a small receptive field but extend through the full depth of the input volume. Rectified Linear Units (Re LU) are used as substitute for saturating nonlinearities. This activation function adaptively learns the parameters of rectifiers and improves accuracy at negligible extra computational cost. In the context of artificial neural networks, the rectifier is an activation function defined as:

f (x)=max(0,x)

,where x is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering. This activation function was first introduced to a dynamical network by Hahn loser et al. in a 2000 paper in Nature with strong biological motivations and mathematical justifications. It has been used in convolutional networks more effectively than the widely used logistic sigmoid (which is inspired by probability theory; see logistic regression) and its more practical counterpart, the hyperbolic tangent. The rectifier is, as of 2015, the most popular activation function for deep neural networks. Deep CNN with ReLUs trains several times faster. This method is applied to the output of every convolutional and fully connected layer. Despite the output, the input normalization is not required; it is applied after ReLU nonlinearity after the first and second convolutional layer because it reduces top-1 and top-5 error rates. In CNN, neurons within a hidden layer are segmented into “feature maps.” The neurons within a feature map share the same weight and bias. The neurons within the feature map search for the same feature. These neurons are unique since they are connected to different neurons in the lower layer. So for the first hidden layer, neurons within a feature map will be connected to different regions of the input image. The hidden layer is segmented into feature maps where each neuron in a feature map looks for the same feature but at different positions of the input image. Basically, the feature map is the result of applying convolution across an image. The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input. Stacking the activation maps for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map. When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The extent of this connectivity is a hyper parameter called the receptive field of the neuron. The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern. Three hyper parameters control the size of the output volume of the convolutional layer: the depth, stride and zero-padding.

1. **Depth** of the output volume controls the number of neurons in the layer that connect to the same region of the input volume. All of these neurons will learn to activate for different features in the input. For example, if the first Convolutional Layer takes the raw image as input, then different neurons along the depth dimension may activate in the presence of various oriented edges, or blobs of color.

2. **Stride** controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1, a new depth column of neurons is allocated to spatial positions only 1 spatial unit apart. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. Conversely, if higher strides are used then the receptive fields will overlap less and the resulting output volume will have smaller dimensions spatially.

3. **Stride controls** how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1, a new depth column of neurons is allocated to spatial positions only 1 spatial unit apart. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. Conversely, if higher strides are used then the receptive fields will overlap less and the resulting output volume will have smaller dimensions spatially.

Parameter sharing scheme is used in convolutional layers to control the number of free parameters. It relies on one reasonable assumption: That if one patch feature is useful to compute at some spatial position, then it should also be useful to compute at a different position. In other words, denoting a single 2-dimensional slice of depth as a depth slice, we constrain the neurons in each depth slice to use the same weights and bias. Since all neurons in a single depth slice are sharing the same parameterization, then the forward pass in each depth slice of the CONV layer can be computed as a convolution of the neuron's weights with the input volume (hence the name: convolutional layer). Therefore, it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input. The result of this convolution is an activation map, and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter Sharing contributes to the translation invariance of the CNN architecture. It is important to notice that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a CNN have some specific centred structure, in which we expect completely different features to be learned on different spatial locations. One practical example is when the input is faces that have been centred in the image: we might expect different eye-specific or hair-specific features to be learned in different parts of the image. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a locally connected layer. Another important layer of CNNs is the pooling layer, which is a form of nonlinear down sampling.

Pooling operation gives the form of translation invariance; it operates independently on every depth slice of the input and resizes it spatially. Overlapping pooling is beneficially applied to lessen over fitting. Also in favour of reducing over fitting, a dropout layer is used in the first two fully connected layers. But the shortcoming of dropout is that it increases training time 2-3 times comparing to a standard neural network of the exact architecture. Bayesian optimization experiments also proved that ReLUs and dropout have synergy effects, which means that it is advantageous when they are used together. The advance of CNNs refers to their ability to learn rich mid-level image representations as opposed to hand-designed low-level features used in other image classification methods.

METHODOLOGY:

1. **Image Acquisition**

The pictures of the leaves are caught using the high definition camera having RGB components not the gray scale. Shading change segments of the leaf picture are distinguished, and after that, forwarded to a device capable of autonomous shading change.

1. **Image Pre-processing**

To expel commotion in a picture or elective article expulsion, entirely unexpected pre-preparing procedures are considered. Picture cutting, for instance, editing of the leaf picture to instigate the intrigued picture locale. Picture smoothing is done through the smoothing channel. Picture improving is managed for expanding the qualification. The RGB pictures into the dark pictures abuse shading change utilizing condition i.e.

f(x)=0.299\*R + 0.58\*G + 0.114\*B

At that point, the visual diagram accomplishment that appropriates the powers of the photos is connected to the picture to support the sickness pictures. The added substance appropriation perform is utilized to disperse power esteems.

1. **Image Segmentation**

The third phase is segmentation, means that image partitioning into numerous portions and segments of similar intensities and similarities. The process of segmentation is often done in many ways using algorithms for instance Otsu methodology, HIS model, k-means algorithm, etc.

* The segmentation mistreatment boundary edge detection: The image from the acquisition phase is taken into account and forwarded to the HIS model. The program for edge and spot detection runs the main program and detects the infected diseased part of the plant.

• K-means algorithm for clustering: This K-means algorithm is employed for clustering and classification of objects supported by a group of options into K variety of categories. The grouping of objects is done by minimizing the gap between them and therefore forming the desired group.

The algorithmic rule is: => deduce the centroid of K clusters arbitrarily.

=> Then after, assign every component from within the input image to new cluster that closely resembles a particular cluster.

=> Then once more, work out through all the derived cluster’s center. So that every segment has some cluster. Loop back and repeat the steps two and three till all the portions are traversed.

1. **Feature Extraction**

The process of features extraction plays a vital portrayal in disease symptoms identification. In several applications of image processing, feature extraction is employed. The components such as, color, segments, texture, edges, shades etc. are various distinguishable options which are then utilized in detection of any symptoms. According to Monica jhuria et al. these components are crucial and the detection of any sickness in the plants becomes very easy using their results. According to her, morphological results from these parameters provide higher accuracy as compared to the opposite options. Texture of leaves is however the color only which is distributed over the image completely.

5.Classification:

The fifth phase after feature extraction is classification that means the educational information pictures are classified using a neural network. The segmented features extracted from the output as image are taken as the inputs for this classification phase. The algorithms such as, convolution neural network are used for the optimum classification. It matches the information from the classified portions with the databases previously stored and helps in exact detection of the disease.

**Software Description:**

PYTHON:

PYTHON 3.7:

Python is an interpreter, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object- oriented programming. Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in manya reason most platforms and

may be freely distributed. The same site also contains distributions of and pointers to many free third party Python modules, programs and tools, and additional documentation. The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications. This tutorial introduces the reader informally to the basic concepts and features of the Python language and system. It helps to have a Python interpreter handy for hands-on experience, but all examples are self-contained, so the tutorial can be read off- line as well. For a description of standard objects and modules, see library-index. Reference-index gives a more formal definition of the language. To write extensions in C or C++, read extending-index and c-api-index. There are also several books covering Python in depth. This tutorial does not attempt to be comprehensive and cover every single feature, or even every commonly used feature. Instead, it introduces many of Python’s most notes worthy features, and will give you a good idea of the language’s flavor and style. After reading it, you will be able to read and write Python modules and programs, and you will be ready to learn more about the various Python library modules described in library-index. If you do much work on computers, eventually you find that there’s some task you’d like

to automate. For example, you may wish to perform a search-and-replace over a large number of text files, or rename and rearrange a bunch of photo files in a complicated way. Perhaps you’d like to write a small custom database, or a specialized

GUI application or a simple game. If you’re a professional software developer, you may have to work with several C/C++/Java libraries but find the usual write/compile/test/re-compile cycle is too slow. Perhaps you’re writing a test suite for such a library and find writing the testing code a tedious task. Or maybe you’ve written a program that could use an extension language, and you don’t want to design and implement a whole new language for your application.

Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn’t work, you can exit the interpreter by typing the following command: quit(). The interpreter’s line-editing features include interactive editing, history substitution and code completion on systems that support read line. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. Ifnothing appears to happen, or if ^P is echoed, command line editing isn’t available; you’ll only be able to use backspace to remove characters from the current line. The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a script from that file. A second way of starting the interpreter is python -c command [arg] ..., which executes the statement(s) in command, analogous to the shell’s -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote commands in its entiretywithsinglequotes.SomePythonmodulesarealsousefulasscripts. These can be invoked using python-m module [arg]...,which executes the source file for the module as if you had spelled out its full name on the command line. When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script.

There are tools which use doc strings to automatically produce online or printed documentation or to let the user interactively browse through code; it’s good practice to include doc strings in code that you write, so make a habit of it. The execution of a function introduces a new symbol table usedfor the local variables of the function. More precisely, all variable assignments in a functions to read the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a global statement), although they may be referenced. The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object).1 When a function calls another function, a new local symbol table is created for that call. A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function.

Annotations are stored in the annotations attribute of the function as a dictionary and haven o effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal ->, followed by an expression, between the parameter list and the colon denoting the end of the def statement.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operator is and does not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators. Comparisons can be chained. For example,a<b==c tests whether a is less than band more over b equals c. Comparisons may be combined using the Boolean operators and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition. The Boolean operators and are so-called short-circuit operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but Bis false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state. Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation. In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and all member functions are virtual. A sin Modula-3, there are no short hands for referencing the object’s members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. A sin Small talk, classes themselves are objects. This provides Semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, sub scripting etc.) can be redefined for class instances.(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object- oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples).However, aliasing has a possibly surprising effect on these mantic of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

A namespace is a mapping from names to objects. Most name spaces are currently implemented as Python dictionaries, but that’s normally not noticeable in any way (except for performance), and it may change in the future. Examples of name spaces are: these to f built-in names (containing functions such as abs(), and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function maximize without confusion — users of the modules must prefix it with the module name. By the way, I use the word attribute for any name following a dot — for example, in the expression z. real, real is an attribute of the object z. Strictly speaking, references to names in modules are attribute references: in the expression modname.funcname, modname is a module object and funcname is an attribute of it. In this case there happens to be a straight forward mapping between the module’s attributes and the global names defined in the module: they share the same namespace!1 Attributes may be read-only or writable. In the latter case, assignment to attributes is

possible. Module attributes are writable: you can

write modname.the\_answer = 42. Writable attributes may also be deleted with the del statement. For example, del mod name .the\_ answer will remove the attribute the\_answer from the object named by mod name. Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits.The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called main, so they have their own global namespace.(The built-in names actually also live in a module; this is called built ins.) The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

To speed uploading modules, Python caches the compiled version

of each module in the pycache directory under the name

module.version.pyc, where the version encodes the format of the compiled

file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of spam.py would be cached as

pycache/spam.cpython-33.pyc. This naming

convention allows compiled modules from different releases and different versions of Python to coexist. Python checks the modification date of the source against the compiled version to see if it’s out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures. Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that’s loaded directly from the command line. Second, it does not check the cache if there is no source module. To support anon-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module. Some tips for experts:

* You can use the -O or -OO switches on the Python command to reduce the size of a compiled module. The -O switch removes assert statements, the -OO switch removes both assert statements and doc

strings. Since some programs may rely on having these available, you should only use this option if you know what you’re doing. “Optimized”

modules have an opt- tag and are usually smaller. Future releases may change the effects of optimization.

* A program doesn’t run any faster when it is read from a .pyc file than when it is read from a .py file; the only thing that’s faster about .pyc files is the speed with which they are loaded.
* The module compile all can create .pyc files for all modules in a directory.
* There is more detail on this process, including a flow chart of the decisions

***PYTHON PACKAGES WORKING:***

There are two types of models available in Keras: [the Sequential model](https://keras.io/models/sequential) and [the Model class used with functional API](https://keras.io/models/model).

These models have a number of methods in common:

* model.summary(): prints a summary representation of your model. Shortcut for [utils.print\_summary](https://keras.io/utils/" \l "print_summary)
* model.get\_config(): returns a dictionary containing the configuration of the model. The model can be reinstantiated from its config via:

config = model.get\_config()

model = Model.from\_config(config)

*# or, for Sequential:*

model = Sequential.from\_config(config)

* model.get\_weights(): returns a list of all weight tensors in the model, as Numpy arrays.
* model.set\_weights(weights): sets the values of the weights of the model, from a list of Numpy arrays. The arrays in the list should have the same shape as those returned by get\_weights().
* model.to\_json(): returns a representation of the model as a JSON string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the JSON string via:

**from** keras.models **import** model\_from\_json

json\_string = model.to\_json()

model = model\_from\_json(json\_string)

* model.to\_yaml(): returns a representation of the model as a YAML string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the YAML string via:

**from** keras.models **import** model\_from\_yaml

yaml\_string = model.to\_yaml()

model = model\_from\_yaml(yaml\_string)

* model.save\_weights(filepath): saves the weights of the model as a HDF5 file.
* model.load\_weights(filepath, by\_name=False): loads the weights of the model from a HDF5 file (created by save\_weights). By default, the architecture is expected to be unchanged. To load weights into a different architecture (with some layers in common), use by\_name=True to load only those layers with the same name.

**Sequential:**

## Useful attributes of Model

* model.layers is a list of the layers added to the model.

## Sequential model methods

### compile

compile(self, optimizer, loss, metrics=**None**, sample\_weight\_mode=**None**, weighted\_metrics=**None**, target\_tensors=**None**)

Configures the model for training.

**Arguments**

* **optimizer**: String (name of optimizer) or optimizer object. See [optimizers](https://keras.io/optimizers).
* **loss**: String (name of objective function) or objective function. See [losses](https://keras.io/losses). If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.
* **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use metrics=['accuracy']. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as metrics={'output\_a': 'accuracy'}.
* **sample\_weight\_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to "temporal". Nonedefaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different sample\_weight\_mode on each output by passing a dictionary or a list of modes.
* **weighted\_metrics**: List of metrics to be evaluated and weighted by sample\_weight or class\_weight during training and testing.
* **target\_tensors**: By default, Keras will create a placeholder for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensor (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the target\_tensors argument. It should be a single tensor (for a single-output Sequential model).
* **\*\*kwargs**: When using the Theano/CNTK backends, these arguments are passed into K.function. When using the TensorFlow backend, these arguments are passed into tf.Session.run.

**Raises**

* **ValueError**: In case of invalid arguments for optimizer, loss, metrics or sample\_weight\_mode.

**Example**

model = Sequential()

model.add(Dense(32, input\_shape=(500,)))

model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',

loss='categorical\_crossentropy',

metrics=['accuracy'])

### fit

fit(self, x=**None**, y=**None**, batch\_size=**None**, epochs=1, verbose=1, callbacks=**None**, validation\_split=0.0, validation\_data=**None**, shuffle=**True**, class\_weight=**None**, sample\_weight=**None**, initial\_epoch=0, steps\_per\_epoch=**None**, validation\_steps=**None**)

Trains the model for a fixed number of epochs (iterations on a dataset).

**Arguments**

* **x**: Numpy array of training data. If the input layer in the model is named, you can also pass a dictionary mapping the input name to a Numpy array. x can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **y**: Numpy array of target (label) data. If the output layer in the model is named, you can also pass a dictionary mapping the output name to a Numpy array. y can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **batch\_size**: Integer or None. Number of samples per gradient update. If unspecified, it will default to 32.
* **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire x and y data provided. Note that in conjunction with initial\_epoch, epochs is to be understood as "final epoch". The model is not trained for a number of iterations given by epochs, but merely until the epoch of index epochs is reached.
* **verbose**: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
* **callbacks**: List of keras.callbacks.Callback instances. List of callbacks to apply during training. See [callbacks](https://keras.io/callbacks).
* **validation\_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the x and y data provided, before shuffling.
* **validation\_data**: tuple (x\_val, y\_val) or tuple (x\_val, y\_val, val\_sample\_weights) on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. This will override validation\_split.
* **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when steps\_per\_epoch is not None.
* **class\_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
* **sample\_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape (samples, sequence\_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specifysample\_weight\_mode="temporal" in compile().
* **initial\_epoch**: Epoch at which to start training (useful for resuming a previous training run).
* **steps\_per\_epoch**: Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default None is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.
* **validation\_steps**: Only relevant if steps\_per\_epoch is specified. Total number of steps (batches of samples) to validate before stopping.

**Returns**

A History object. Its History.history attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

**Raises**

* **RuntimeError**: If the model was never compiled.
* **ValueError**: In case of mismatch between the provided input data and what the model expects.

### evaluate

evaluate(self, x=**None**, y=**None**, batch\_size=**None**, verbose=1, sample\_weight=**None**, steps=**None**)

Computes the loss on some input data, batch by batch.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs). x can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **y**: labels, as a Numpy array. y can be None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
* **batch\_size**: Integer. If unspecified, it will default to 32.
* **verbose**: verbosity mode, 0 or 1.
* **sample\_weight**: sample weights, as a Numpy array.
* **steps**: Integer or None. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of None.

**Returns**

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### predict

predict(self, x, batch\_size=**None**, verbose=0, steps=**None**)

Generates output predictions for the input samples.

The input samples are processed batch by batch.

**Arguments**

* **x**: the input data, as a Numpy array.
* **batch\_size**: Integer. If unspecified, it will default to 32.
* **verbose**: verbosity mode, 0 or 1.
* **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of None.

**Returns**

A Numpy array of predictions.

### train\_on\_batch

train\_on\_batch(self, x, y, class\_weight=**None**, sample\_weight=**None**)

Single gradient update over one batch of samples.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
* **y**: labels, as a Numpy array.
* **class\_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
* **sample\_weight**: sample weights, as a Numpy array.

**Returns**

Scalar training loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

**Raises**

* **RuntimeError**: if the model was never compiled.

### test\_on\_batch

test\_on\_batch(self, x, y, sample\_weight=**None**)

Evaluates the model over a single batch of samples.

**Arguments**

* **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
* **y**: labels, as a Numpy array.
* **sample\_weight**: sample weights, as a Numpy array.

**Returns**

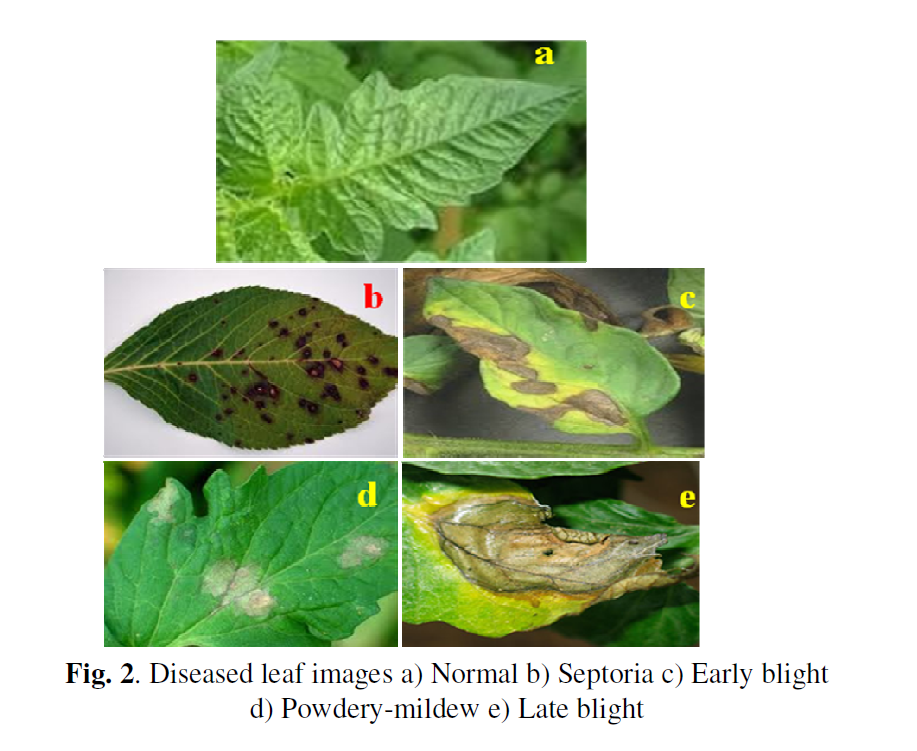
Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other metrics). The attribute model.metrics\_names will give you the display labels for the scalar outputs.

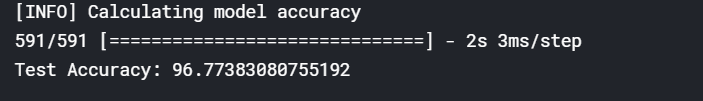
**Raises**

* **RuntimeError**: if the model was never compiled.

***Results And Conclusion***

The Results presented in this section are related to training with the whole database containing both original and augmented images. As it is known that convolutional networks are able to learn features when trained on larger datasets, results achieved when trained with only original images will not be explored. After fine-tuning the parameters of the network, an overall accuracy of 96.77% was achieved. Furthermore, the trained model was tested on each class individually. Test was performed on every image from the validation set. As suggested by good practice principles, achieved results should be compared with some other results. In addition, there are still no commercial solutions on the market, except those dealing with plant species recognition based on the leaves images. In this paper, a approach of using deep learning method was explored in order to automatically classify and detect plant dise ases from leaf images. The complete procedure was described, respectively, from collecting the images used for training and validation to image pre-processing and augmentation and finally the procedure of training the deep CNN and fine-tuning. Different tests were performed in order to check the performance of newly created model. As the presented method has not been exploited, as far as we know, in the field of plant disease recognition, there was no comparison with related results, using the exact technique.





***References And Bibliography***

[1]. Aakanksha Rastogi, Ritika Arora, Shanu Sharma, “Leaf Disease Detection and Grading using Computer Vision Technology &Fuzzy Logic,” presented at the 2nd International Conference on Signal Processing and Integrated Networks (SPIN), IEEE, 2015, pp. 500–505.

[2]. Garima Tripathi, Jagruti Save, “AN IMAGE PROCESSING AND NEURAL NETWORK BASED APPROACH FOR DETECTION AND CLASSIFICATION OF PLANT LEAF DISEASES,” Int. J. Comput. Eng. Technol. IJCET, vol. 6, no. 4, pp. 14–20, Apr. 2015.

[3]. S. Arivazhagan, R. Newlin Shebiah, S. Ananthi, S. Vishnu Varthini, “Detection of unhealthy region of plant leaves and classification of plant leaf diseases using texture features,” Agric Eng Int CIGR J., vol. 15, no. 1, pp. 211–217, Mar. 2013.

[4]. Prof. Sanjay B. Dhaygude, Mr.NitinP.Kumbhar, “Agricultural plant Leaf Disease Detection Using Image Processing”IJAREEIE, vol. 2(1),pp. 599-602, January 2013.

[5]. K. Muthukannan, P. Latha, R. PonSelvi and P. Nisha, “CLASSIFICATION OF DISEASED PLANT LEAVES USING NEURAL NETWORK ALGORITHMS,” ARPN J. Eng. Appl. Sci., vol. 10, no. 4, pp. 1913–1918, Mar. 2015.

[6]. Md. Nazrul Islam, M.A. Kashem, MahmudaAkter and Md. JamilurRahman, “An Approach to Evaluate Classifiers for Automatic Disease Detection and Classification of Plant Leaf,” presented at the International Conference on Electrical, Computer and Telecommunication Engineering, RUET, Rajshahi-6204, Bangladesh, 2012, pp. 626–629.

[7]. Hrishikesh Kanjalkar P. and Prof. Lokhande S. 2013. Detection and Classification of Plant Leaf Diseases using ANN. International Journal of Scientific & Engineering Research. ISSN: 2229 5518.

[8]. Suhaili Beeran kutty. and Noor ezan Abdullah. 2013. ‘Classification of Watermelon Leaf Diseases Using Neural Network Analysis’, IEEE Business Engineering and Industrial Applications Colloquium (BEIAC). **International Journal of Engineering Science and Computing, March 2017** 5328 http://ijesc.org/