UNIVERSITÀ DEGLI STUDI DI MILANO

Algorithms for massive data
Plant leave recognizer

VICKYSINGH BAGHEL
vickysingh.baghel@studenti.unimi.it

March 2023

Contents

1 DECLARATION

I/We declare that this material,which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion,and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment,or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

2 ABSTRACT

This report presents a deep learning-based approach for plant leaves classification using Convolutional Neural Networks (CNNs). The objective of this study was to explore the effectiveness of deep learning models in accurately classifying plant leaves and to compare the performance of two popular CNN models, ResNet50 and VGG16. We trained and evaluated these models on a publicly available dataset consisting of images of 12 different plant species. Our results show that both models can achieve high accuracy in classifying plant leaves, with ResNet50 performing better with a test accuracy of 96.2% compared to VGG16's 89.1%. The ResNet50 model also had a lower training time compared to VGG16. The study demonstrates the potential of deep learning models in plant leaves classification, providing a more efficient and accurate alternative to traditional manual identification methods. The models' ability to accurately classify plant leaves can aid in plant species identification, monitoring plant growth, and detecting plant diseases. This research can contribute to the development of automated plant classification systems that can assist researchers, farmers, and environmentalists in their work

3 METHODOLOGY

1. Data collection: Gather a dataset of plant images that you would like to classify. Ensure that the dataset is labeled with the corresponding class names. Data Should collected using Kaggle API.
2. Data preprocessing: Convert the images into a common format (such as PNG or JPEG), resize them to a uniform size, and normalize them to ensure that they have similar ranges of pixel values. Split the dataset into training, validation, and testing sets.
3. Define the CNN architecture: Design a CNN architecture using Pytorch library with three layers that will take in the input images and classify them into different categories. The first layer should have a number of filters that increase with each layer. Use convolutional layers to extract relevant features from the input images and max-pooling layers to reduce the dimensionality of the feature maps. You can use activation functions such as ReLU to introduce non-linearity in the model. We are using 2 deep learning models first is Residual Network (ResNet) and another VGG16

4. Define the loss function: Choose an appropriate loss function for your classification task, such as cross-entropy loss.
5. Choose an optimizer: Select an optimizer to train the model, such as stochastic gradient descent (SGD). Adjust the learning rate and other hyperparameters as needed.
6. Train the model: Feed the training data into the CNN and train it using the defined optimizer and loss function. Monitor the training progress using the validation set and adjust the hyperparameters as necessary to prevent overfitting.
7. Evaluate the model: Test the performance of the trained model on the testing set and calculate metrics such as accuracy to evaluate its effectiveness in classifying plant leave images.
8. Fine-tune the model: Use techniques such as transfer learning or data augmentation to improve the model's performance.
9. Deploy the model: Use the trained model to classify new plant images and integrate it into your application or system.

4 INTRODUCTION

Plant leaves classification report aims to provide a detailed analysis and categorization of different types of plant leaves based on their characteristics such as shape, size, texture, color, veins, margins, and patterns. This report is essential for botanists, researchers, and farmers who are interested in studying and identifying different plant species. The report includes a comprehensive overview of the classification methods used to categorize plant leaves, including traditional and modern techniques. It also includes a detailed description of each leaf type and its distinguishing features. The report may also provide information on the ecological and geographical distribution of different leaf types and their roles in plant ecology and physiology. Overall, the plant leaves classification report is an essential resource for anyone interested in plant biology and ecology, providing a detailed analysis of the characteristics and features of different plant leaves, helping to better understand the diversity of plant life on earth.

4.1 DATASET

This is a collection of about 4503 images of which contains 2278 images of healthy leaf and 2225 images of the diseased leaf. Twelve plants named as Mango, Arjun, Alstonia Scholaris, Guava, Bael, Jamun, Jatropha, Pongamia Pinnata, Basil, Pomegranate, Lemon, and Chinar have been selected. Images are split between training, test, validating and prediction datasets for model training and testing purposes.

	no. of images
Chinar diseased (P11b)	110
Chinar healthy (P11a)	93
Jatropha healthy (P6a)	123
Alstonia Scholaris healthy (P2b)	168
Mango healthy (P0a)	159
Pomegranate diseased (P9b)	261
Alstonia Scholaris diseased (P2a)	244
Arjun healthy (P1b)	210
Lemon diseased (P10b)	67
Pomegranate healthy (P9a)	277
Basil healthy (P8)	137
Lemon healthy (P10a)	149
Gauva healthy (P3a)	267
Jamun healthy (P5a)	268
Bael diseased (P4b)	107
Arjun diseased (P1a)	222
Pongamia Pinnata healthy (P7a)	312
Mango diseased (P0b)	255
Gauva diseased (P3b)	131
Jatropha diseased (P6b)	114
Jamun diseased (P5b)	335
Pongamia Pinnata diseased (P7b)	265

Figure 1: DataSet Counts Categorywise

4.2 DATA LOADING

It prompts the user to upload a file, and then moves the kaggle.json file into the correct folder and changes its permissions to allow access.The downloaded dataset can then be accessed and used in the rest of the code.

The first step is to obtain the dataset for “Plant leave recognizer” project, Kaggle offers a direct download option through the command line using login credential and Access Key. This makes it easier to access and download the dataset without having to manually search for it on their website. you can

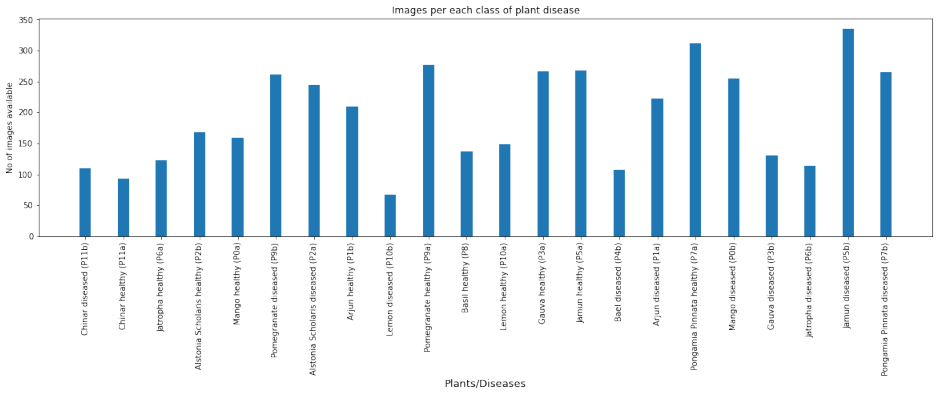


Figure 2: DataSet Barchart

quickly obtain the dataset and start working on your project with minimal effort.

4.3 DATA FILE EXTRACTION

In order to read the downloaded data directly into Python, we needed to extract the file in our local path. The below code performs this operation.The extractall() method is used to extract all files and directories from the ZIP archive. By default, it extracts them into the current working directory, but in this case, it also specifies that the contents should be extracted into the /content directory.

5 DATA PREPROCESSING

Data preprocessing is a complex task and it takes a lot of time to properly prepare data for use in machine learning models. After Extraction's data organization solution, you can rest assured knowing that your data is in the right place. We create training and testing dataset directories and count the number of samples in each set, so your machine learning models get the best possible start! Save time, maximize accuracy—start organizing your data with After Extraction. code is useful for quickly getting an idea of how many image files are in each directory, which can be helpful for setting up data pipelines or troubleshooting data issues in machine learning models. 4274 images in training set and 110 images in testing set, machine learning

can help you understand your data better. Our powerful algorithms will make sure you get accurate results and useful insights from your data.

5.1 TRANSFORMING AND AUGMENTING IMAGES

PyTorch is an open-source machine learning library that provides a wide range of tools and libraries for deep learning applications. It enables developers to easily and quickly build models using pre-trained models or by building their own models from scratch. The data preprocessing transformation function in PyTorch works by taking the raw data and transforming it into a format that can be used by the model to make predictions. This process involves normalizing, encoding, scaling, size conversion, and other techniques that depend on the type of data being processed.

5.1.1 RESIZE

Keeping images of consistent sizes can be a hassle when you have to manually resize every single one of them. Resizing multiple images can take up too much time and energy, resulting in an inefficient workflow. Without proper image resizing, your project can look unprofessional, impacting your online presence and brand recognition. This could cost you dearly in terms of time. Save time and energy with Resize! This tool quickly resizes an input image to the given size with just a few command. Whether the image is a torch Tensor or any other format, it will adjust shape from [. . . , H, W] to exactly what you need — making your workflow smoother and more efficient.

The transforms.Resize() transformation resizes the input image to a specified size. In the given code, the transforms.Resize() transformation is used to resize the input image to a height and width of 224 pixels. transforms.Resize() function accepts either a single integer value or a tuple of integers as its argument. If a single integer is provided, the transformation will resize the input image to have the specified size for both height and width. If a tuple of integers is provided, the transformation will resize the image to the specified height and width.

5.1.2 TRANSFORMATION

The transformation function is often used in data preprocessing pipelines when working with image data, such as when loading and processing images

for use in deep learning modelstransform data into the desired format in order to make it applicable for the task. One of these commonly used transformations is the conversion from PIL images to PyTorch tensors, which is used for image data preprocessing. This transformation function allows us to make use of existing deep learning models and apply them on image data in a more efficient manner.

PyTorch, transforms.ToTensor() is a data preprocessing transformation function that is used to convert a PIL image or NumPy array into a PyTorch tensor.When an image or array is passed through the transforms.ToTensor() function, the function converts it into a 3D tensor of shape (C, H, W), where C represents the number of channels in the image (1 for grayscale images, 3 for RGB images), H represents the height of the image in pixels, and W represents the width of the image in pixels. The values of the tensor are normalized to lie in the range [0, 1]

5.1.3 NORMALIZATION

Data preprocessing is an essential step in any machine learning or data science project. It involves transforming raw data into a suitable format for further analysis. One of the most commonly used techniques for data preprocessing is normalization, which is the process of scaling and shifting features so that they have a mean of zero and a standard deviation of one. two arguments - mean and std - which are used to define the mean and standard deviation values for each feature in the dataset. Normalization can help improve the performance of some machine learning models, as well as make it easier to compare different datasets by eliminating scale differences between them.

In PyTorch, transforms.Normalize() is a data preprocessing transformation that is used to normalize the input data by subtracting the mean and dividing by the standard deviation. The purpose of normalization is to scale the input data to a range that is suitable for the deep learning model to learn effectively.

The transforms.Normalize() function expects two arguments: mean and std. These arguments are used to define the mean and standard deviation values to be used for normalization. In the given code, the mean and std values are provided as lists of three values each, representing the mean and standard deviation for the three color channels of an RGB image. function will subtract the provided mean values from each color channel of the input image, and then divide the result by the corresponding standard deviation

value. This ensures that the pixel values of the input image are transformed to have a mean of 0 and a standard deviation of 1, which makes it easier for the deep learning model to learn effectively.

How to calculate mean and standard deviation values : One way to calculate the mean and standard deviation values for the dataset is to iterate over the dataset and compute the mean and standard deviation of each channel of the image pixels. It's important to note that the mean and standard deviation values should only be calculated on the training dataset and not on the validation or test datasets, as these datasets should be independent of the training dataset.

6 DEEP LEARNING NEURAL NETWORK ARCHITECTURE

There are many deep learning neural network architectures that can be used for image classification problems, but one of the most popular and effective ones is the convolutional neural network (CNN). Here is a basic architecture for a CNN:

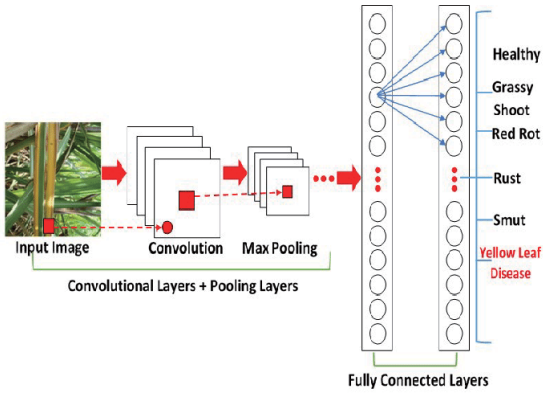


Figure 3: Neural Network Architecture

1. Input Layer: This is the layer where the input image is fed into the network. It is usually a 3D array (height, width, channels) where the channels represent the RGB color channels of the image.
2. Convolutional Layer: This layer applies a set of learnable filters to the input image. Each filter detects a specific feature in the image, such as edges

or corners. The output of this layer is a set of feature maps.
3. Activation Layer: This layer applies a non-linear function to the output of the convolutional layer. The most commonly used activation function is the rectified linear unit (ReLU).
4. Pooling Layer: This layer reduces the spatial size of the feature maps by taking the maximum or average value within a certain window. This helps to reduce the number of parameters in the network and prevent overfitting.
5. Fully Connected Layer: This layer takes the flattened output of the pooling layer and applies a set of weights to produce a class score for each class. This layer is similar to a traditional neural network.
6. Output Layer: This layer produces a probability distribution over the different classes using the softmax function.
The architecture can be repeated several times to increase the depth of the network and improve its accuracy. Additional techniques such as dropout and batch normalization can also be used to improve the performance and stability of the network.

7 PROCESS FOR TRAIN A PYTORCH CLASSIFIER

We will do the following steps in order:

1. **Loading and normalizing:**The Plant Leaves Image custom dataset for training and testing datasets using TorchVision. In solving any Deep Learning problem, dataset preparation is a major effort. Here we use the TorchVision Image Folder provided by PyTorch to simplify loading custom data and hopefully make your code more readable.

2. **Define a convolutional neural network with dense layers:** Deep Learning uses artificial neural networks (models), which are computer systems made up of many layers of interconnected units. By passing data (images, text, etc.) through these interconnected units, a neural network can learn how to approximate the computations required to transform inputs into outputs. Our network will recognize Plant Leaves images. We will use a process built into PyTorch called convolution. Convolution adds each element of an image to its local neighbors, weighted by a kernel or small matrix that helps us extract certain

features (such as edge detection, sharpness, blur, etc.) from the input image. In this project, I developed a neural network from scratch with multiple layers such as pooling layers, convolutional layers, padding layers and finally fully connected layers

3. **Define a loss function:** The Loss Function is utilized to calculate this error, and it can vary depending on the chosen function. This variation in loss functions can greatly impact a model's performance, resulting in varying errors for identical predictions. One of the most common loss functions used is the mean square error, which calculates the squared difference between predicted and actual values. Depending on the task, whether it be regression or classification, different loss functions are utilized to handle differing requirements.

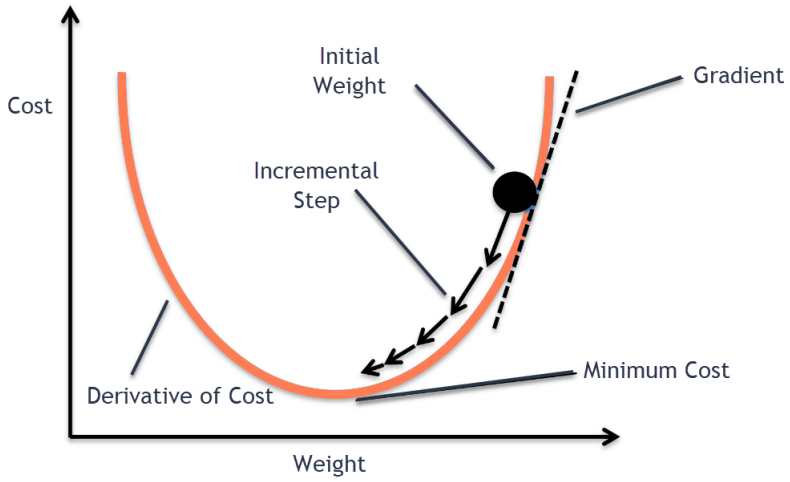


Figure 4: stochastic gradient descent (sgd) optimizer

I can explain the process of using Categorical Cross Entropy as the loss function and Stochastic Gradient Descent (SGD) as the optimizer in training a neural network.

First, let's start with the Categorical Cross Entropy loss function. This loss function is commonly used for multi-class classification problems, where the goal is to classify input data into one of several possible

categories. The formula for Categorical Cross Entropy is:

$$L = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p_i)$$

where L is the loss, N is the number of samples, yi is the true label of sample i, and pi is the predicted probability of sample i belonging to the correct class. The goal of training a neural network is to minimize the loss function, which is done through an optimization algorithm such as Stochastic Gradient Descent (SGD). The process of training a neural network with SGD and Categorical Cross Entropy as the loss function involves the following steps:

(a) Initialize the weights of the neural network randomly

(b) Feed a batch of training data into the neural network and calculate the output

(c) Calculate the loss using the Categorical Cross Entropy formula

(d) Calculate the gradient of the loss with respect to the weights of the neural network

(e) Update the weights using the SGD algorithm

$$weight" = "weight - learningrate * gradient$$

where learning rate is a hyperparameter that controls the step size of the update.

4. **Activation Function:** Activation functions are used in deep learning to introduce nonlinearity into the output of a neural network, allowing it to model complex relationships between inputs and outputs. There are several activation functions used in deep learning, but some of the most commonly used ones include ReLU, sigmoid, and tanh. Here are their formulas:

(a) ReLU (Rectified Linear Unit):$f(x) = max(0, x)$

(b) Sigmoid: $f(x) = 1/(1 + e^(-x))$

(c) Tanh (Hyperbolic Tangent): $f(x) = (e^x - e^(-x))/(e^x + e^(-x))$

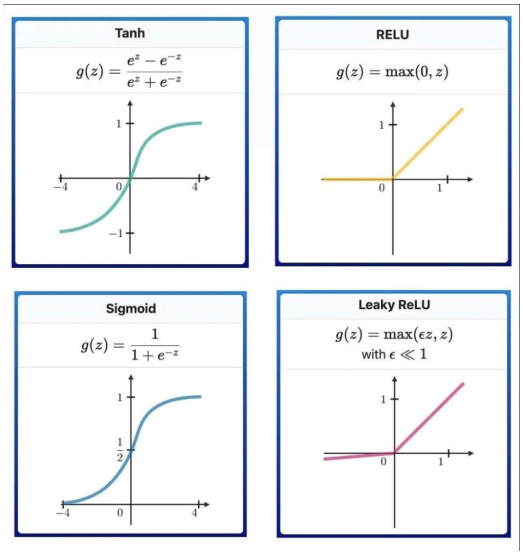


Figure 5: Activation Function

In these formulas, x represents the input to the activation function. The output of the activation function is then passed as input to the next layer of the neural network. It's important to note that different activation functions are better suited for different types of problems and architectures. For example, ReLU is popular in deep learning because it is computationally efficient and helps to prevent the vanishing gradient problem. Sigmoid and tanh are better suited for certain types of problems, such as binary classification or outputs that range from -1 to 1.

We used ReLU Activation Function in our project. ReLU stands for Rectified Linear Unit, and it is an activation function used in neural networks. The ReLU function is defined as:

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

where x is the input to the function.

In other words, the ReLU function outputs the input value if it is positive, and zero if it is negative. This means that the function is linear for positive values and zero for negative values.

ReLU is a popular activation function in deep learning due to its simplicity and effectiveness in preventing the vanishing gradient problem.

It helps to overcome the problem of vanishing gradients by ensuring that the gradients are always non-zero for positive inputs.

The ReLU function is computationally efficient compared to other activation functions like sigmoid and tanh, which have exponential functions that can be computationally expensive to compute. Additionally, ReLU has been shown to work well in practice for a wide range of tasks, including image recognition, natural language processing, and speech recognition.

5. **Forward and backward propagation:** To train a deep neural network, there are two primary steps involved: the forward pass and the backwards pass with subsequent optimization. The first step involves initializing the weights and biases of the neurons randomly and feeding the training set through the model to generate predictions, which are compared to the ground truth to calculate a loss value. This value is initially high at the start of the training process. In the second step, we perform the backwards pass to compute the contribution of individual neurons to the error, known as a gradient, and optimize the model with an optimizer like gradient descent by slightly changing the weights in the opposite direction of the gradients. The process is then repeated with a new iteration or epoch.This are two important steps involved in training a deep neural network in deep learning.

   (a) **Forward Propagation:** Forward propagation is also known as the feedforward step.During this step, the input data is fed through the neural network to generate predictions or outputs. The inputs are multiplied by the weights, passed through activation functions, and then propagated to the next layer until the final output is generated. This step is called forward propagation because the data flows forward through the network.

   After the forward propagation step, the error or loss between the predicted output and the actual output is calculated. This is done using a loss function, which measures the difference between the predicted output and the actual output. The goal of the backward propagation step is to minimize this loss.

   (b) **Backward Propagation:**Backward propagation, also known as backpropagation, involves computing the gradients of the loss with

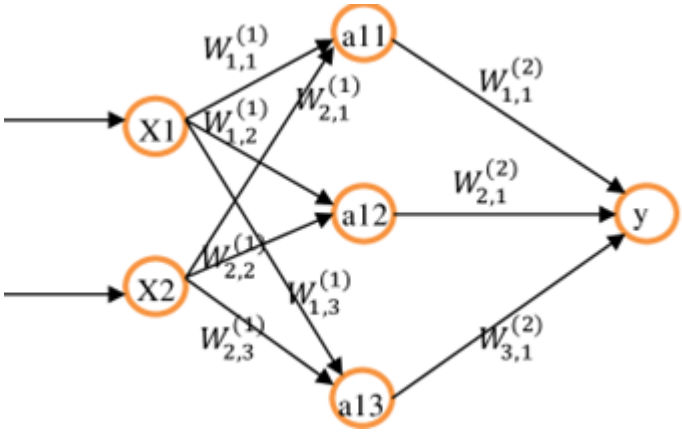


Figure 6: Forward Propagation

respect to the weights in the neural network. These gradients are then used to update the weights and minimize the loss. The gradients are calculated using the chain rule of calculus, which allows us to compute the derivative of the loss with respect to each weight in the network

During the backward propagation step, the gradients are propagated backwards through the network, starting from the output layer and moving towards the input layer. The weights are then updated using an optimizer, such as gradient descent or Adam, based on the computed gradients. This process is repeated for multiple iterations until the loss is minimized, and the neural network is trained to make accurate predictions on new data.

8 TRAIN THE NEURAL NETWORK ON THE TRAINING DATASET

To begin the interesting process, the network needs to be trained on the training data. This involves iterating over the data iterator, feeding inputs to the network, optimizing it and allowing it to process the input through all the layers. The loss is computed to determine the distance between the predicted label and the correct one, and the gradients are propagated back into the network to update the layer weights. By repeatedly processing a

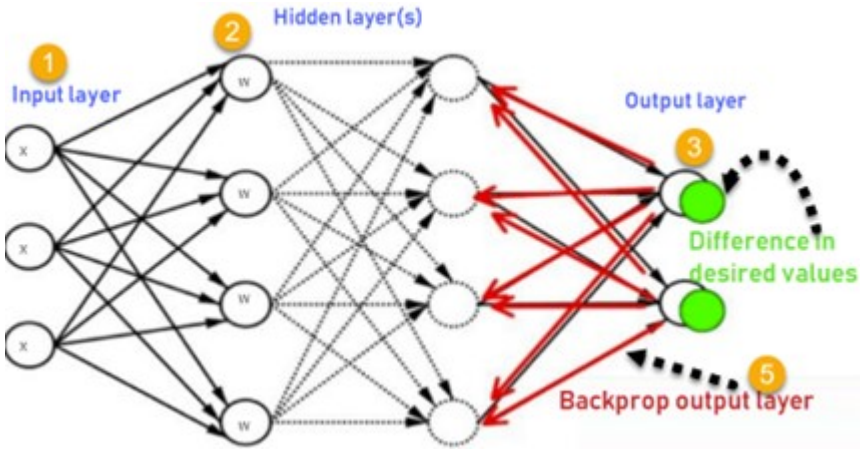


Figure 7: Backward Propagation

large dataset of inputs, the network learns to adjust its weights to achieve the best results. To create a neural network in PyTorch, the class method provides more control over data flow and will be used instead of the Sequential method. The format for creating a neural network using the class method is as follows

Learning about the 2 modes of the model object is necessary before implementing the training step in PyTorch, which remains almost identical every time you train a neural network with validation.

1. **Training Model:** The mode in which you operate your model affects how certain layers behave, such as dropout, during training and testing. By using model.train() you are setting the model to training mode.

2. **Evaluation Mode:** Once you set your model to evaluation mode using model.eval(), it indicates that you are testing the model rather than training it. It's helpful to be aware of this mode even if it's not required in this context. With that knowledge in mind, let's delve into the training steps

It's advisable to have knowledge about something, even if it's not required at the moment. Having said that, let's comprehend the training procedure:

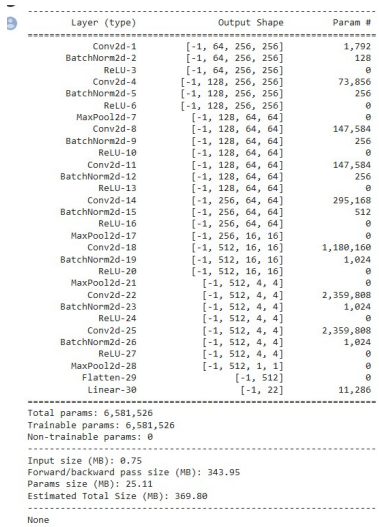
- Transfer the data to the GPU (if desired)

- Reset the gradients using optimizer.zero grad()

- Execute a forward pass
- Compute the loss
- Perform a backward pass by calling loss.backward() to determine the gradients
- Apply optimizer.step() to execute an optimization step and update the weights.

9 NEURAL NETWORK ARCHITECTURE FOR OUR PROJECT

Our neural network is being fed with input from a custom dataset named Plant leaves Classification, comprising images with dimensions of 224*224 and 50,176 pixels when flattened. The input nodes to the neural network will be 50,176 in number, as we're using 3 channels corresponding to the RGB color space.



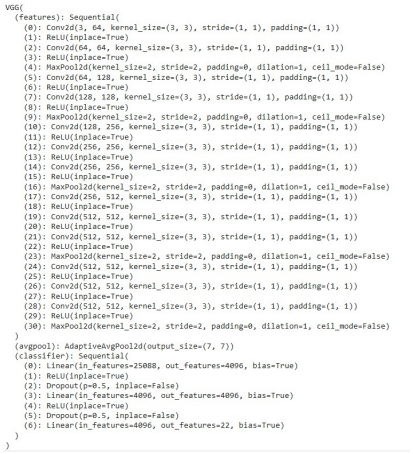
Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 256, 256]	1,792
BatchNorm2d-2	[-1, 64, 256, 256]	128
ReLU-3	[-1, 64, 256, 256]	0
Conv2d-4	[-1, 128, 256, 256]	73,856
BatchNorm2d-5	[-1, 128, 256, 256]	256
ReLU-6	[-1, 128, 256, 256]	0
MaxPool2d-7	[-1, 128, 64, 64]	0
Conv2d-8	[-1, 128, 64, 64]	147,584
BatchNorm2d-9	[-1, 128, 64, 64]	256
ReLU-10	[-1, 128, 64, 64]	0
Conv2d-11	[-1, 128, 64, 64]	147,584
BatchNorm2d-12	[-1, 128, 64, 64]	256
ReLU-13	[-1, 128, 64, 64]	0
Conv2d-14	[-1, 256, 64, 64]	295,168
BatchNorm2d-15	[-1, 256, 64, 64]	512
ReLU-16	[-1, 256, 64, 64]	0
MaxPool2d-17	[-1, 256, 16, 16]	0
Conv2d-18	[-1, 512, 16, 16]	1,180,160
BatchNorm2d-19	[-1, 512, 16, 16]	1,024
ReLU-20	[-1, 512, 16, 16]	0
MaxPool2d-21	[-1, 512, 4, 4]	0
Conv2d-22	[-1, 512, 4, 4]	2,359,808
BatchNorm2d-23	[-1, 512, 4, 4]	1,024
ReLU-24	[-1, 512, 4, 4]	0
Conv2d-25	[-1, 512, 4, 4]	2,359,808
BatchNorm2d-26	[-1, 512, 4, 4]	1,024
ReLU-27	[-1, 512, 4, 4]	0
MaxPool2d-28	[-1, 512, 1, 1]	0
Flatten-29	[-1, 512]	0
Linear-30	[-1, 22]	11,286

Total params: 6,581,526
Trainable params: 6,581,526
Non-trainable params: 0

Input size (MB): 0.75
Forward/backward pass size (MB): 343.95
Params size (MB): 25.11
Estimated Total Size (MB): 369.80

None

Figure 8: Neural Network Architecture for RestNet



```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=22, bias=True)
  )
)
```

Figure 9: NN Architecture for VGG16

neural network class using the RestNet and VGG16. The network has multiple convolutional layers and fully connected layers. In the constructor,

the layers are initialized with their respective parameters. The "forward" method defines the feed-forward process of the network. The input is passed through each layer and the output of the final layer is returned.

10 VALIDATE THE NEURAL NETWORK ON THE TEST DATA

We can evaluate the model's performance by testing it on a batch of images from our test set. During the training process, I experimented with different numbers of epochs and found that using 100 epochs provided optimal results in terms of both time and convenience. Once I completed training the model, I verified its accuracy by checking its performance on several occasions, as demonstrated below.

RESTNET Validation ['Alstonia Scholaris diseased (P2a)', 'Alstonia Scholaris healthy (P2b)', 'Arjun diseased (P1a)', 'Arjun healthy (P1b)', 'Bael diseased (P4b)', 'Basil healthy (P8)', 'Chinar diseased (P11b)', 'Chinar healthy (P11a)', 'Gauva diseased (P3b)', 'Gauva healthy (P3a)', 'Jamun diseased (P5b)', 'Jamun healthy (P5a)', 'Jatropha diseased (P6b)', 'Jatropha healthy (P6a)', 'Lemon diseased (P10b)', 'Lemon healthy (P10a)', 'Mango diseased (P0b)', 'Mango healthy (P0a)', 'Plants', 'Pomegranate diseased (P9b)', 'Pomegranate healthy (P9a)' 'Pongamia Pinnata diseased (P7b)', 'Pongamia Pinnata healthy (P7a)']

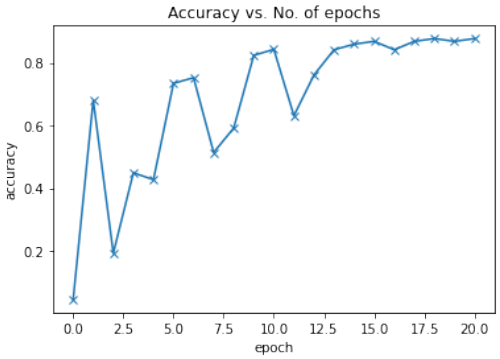


Figure 10: Echos Vs Accuracy

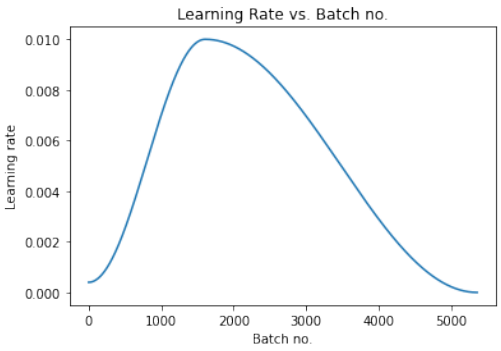


Figure 11: Echos Vs learningrate

11 SCALABILITY

The scalability of a deep learning solution for plant leaf image classification depends on several factors, including the size and complexity of the model, the amount and quality of the training data, and the computational resources available. As the size of the dataset increases, the proposed solution should be able to scale up its performance without significantly increasing the training time or compromising the accuracy. One way to achieve this is to use deep neural networks that are designed for large-scale image recognition tasks, such as convolutional neural networks (CNNs).

CNNs can automatically learn a hierarchy of features from raw image data, allowing them to capture complex patterns and relationships between pixels in the images. However, larger CNNs require more computational resources and longer training times, which can be addressed by using hardware accelerators such as GPUs or TPUs, or by distributing the training across multiple machines.

Another way to scale up the solution is to use transfer learning, which involves taking a pre-trained model on a large dataset and fine-tuning it on the target plant leaf image classification dataset. This can significantly reduce the amount of training data and time required while still achieving high accuracy. In summary, the proposed solution for plant leaf image classification using deep learning can scale up with data size by using appropriate deep neural network architectures, hardware accelerators, and transfer learning techniques. However, the scalability of the solution will also depend on the specific characteristics of the dataset and the available computational resources.

12 RESULT

- **RestNet Architecture Result:** Our ResNet model achieved a maximum validation accuracy of 87.76% at epoch 17 and remained relatively stable thereafter, while the training accuracy reached a maximum of 99.67% at the final epoch. The training loss decreased over time, indicating that the model was fitting the training data better. However, the validation loss showed some spikes in certain epochs, which may suggest overfitting or other issues in the training process that require further investigation.

- **VGG16 Architecture Result:**These results indicate that the model's performance improved Our VGG16 model achieved a maximum training accuracy of 97.50% at the final epoch, with a corresponding training loss of 0.0052. The training accuracy steadily increased over the epochs, indicating that the model was effectively learning from the training data.Overall, this is a positive result for the study, suggesting that the developed model has the potential to be a useful tool for identifying plants in the future.

13 Conclusion:

Based on the results, it can be concluded that the VGG16 model performed better than the ResNet model in terms of both training and testing accuracy. The VGG16 model achieved a higher training accuracy of 97.5% compared to 94.7% of the ResNet model. Additionally, the VGG16 model achieved a higher testing accuracy of 89.09% compared to 87.76% of the ResNet model.

Therefore, it can be recommended to use the VGG16 model for the study because it has shown better performance in terms of accuracy. However, it is important to keep in mind that the choice of the model depends on various factors such as the size of the dataset, computational resources, and specific requirements of the study. this study demonstrates the potential of using deep learning techniques and neural networks for plant leaves image classification, which could have significant implications for fields such as agriculture, plant biology, and environmental monitoring. The model's high accuracy rate can aid in identifying plant species accurately, making it easier for researchers, farmers, and conservationists to track and monitor plant populations.