**Jamia Hamdard**

**Department of Computer Science & Engineering**

**School of Engineering Science & Technology,**

**New Delhi - 110062**



**MINI PROJECT REPORT**

**POTATO DISEASE CLASSIFICATION**

**USING CNN**

**(BTCSEAI-601)**

**B.Tech (CSE) A.I.**

**3rd Year, 6th Sem.**

**Submmitted by: Faraz Bin Tariq (2020-350-017) Submmitted to: BHAVYA ALANKAR SIR**

DECLARATION

I, **Mr. Faraz Bin Tariq** a student of **< Bachelor of Technology Computer Science Engineering (Artificial intelligence) > (B.Tech CSE AI),(Enrolment No: 2020-350-017)** hereby declare that the Project/Dissertation entitled**“ potato disease classification using convolutional neural network ”** which is being submitted by me to the Department of Computer Science, Jamia Hamdard , New Delhi in partial fulfillment of the requirement for the award of the degree of **< Bachelor of Technology Computer Science Engineering (Artificial intelligence)> (B.Tech CSE AI),** is my original work and has not been submitted anywhere else for the award of any Degree, Diploma, Associateship, Fellowship or other similar title or recognition.

**(Signature and Name of the Applicant)**

**Date:**

**Place:**

**INDEX**

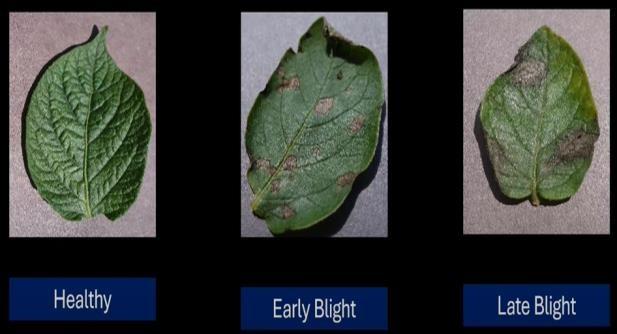
1. **INTRODUCTION**
2. **METHODOLOGY**
3. **DATA AUGMENTATION**
4. **Visualize some of the images from our dataset**
5. **Data Augmentation**
6. **Model Architecture**
7. **Compiling the Model**
8. **Plotting the Accuracy and Loss Curves**
9. **Now run inference on few sample images**

**10)Saving the Model**

**INTRODUCTION**

The term "potato disease classification using CNN" refers to the classification of various potato diseases using convolutional neural networks (CNN). Deep learning neural networks of the kind known as CNNs are frequently employed for image identification and classification applications.

If not properly detected and handled, potato infections can cause large crop losses, which is a serious worry for potato farmers. Late blight, early blight, black scurf, and powdery scab are a few examples of prevalent potato illnesses.



It is possible to create a computer-based system that can quickly and effectively detect the type of illness afflicting a specific crop of potatoes by utilising CNNs to classify potato diseases. This can assist farmers in taking the proper action to reduce crop losses and increase yields, such as using the proper fungicides or changing their cultivation techniques.

Overall, the classification of potato diseases using CNN has the potential to greatly enhance potato farming methods and contribute to more efficient and sustainable agriculture.

**Problem Statement for Potato Leaf Disease Prediction**

Each year, many diseases that infect potato plants create substantial financial losses for farmers that grow potatoes. The two most common illnesses are Early Blight and Late Blight. Both early and late blight are brought on by distinct microorganisms, but if farmers can identify the illness early and treat it well, they can avoid a lot of waste and financial loss. It's critical to correctly identify the type of disease present in that potato plant because there are certain differences between the treatments for early and late blight. Convolutional Neural Network - Deep Learning will be used in the background to detect plant illnesses.

## Potato Leaf Disease Prediction Project Description

## Here, we'll create a whole Deep Learning project for the agricultural industry.

## Using a straightforward and traditional convolutional neural network architecture, we will develop a straightforward image classification model that will classify Potato Leaf Disease.

## We’ll start with collecting the data, data cleaning and preprocessing, model building, and FastApi finally, we’ll use ReactJS to build a web-based application and deploy it on Google Cloud.

## Potato Leaf Disease Prediction Data Collection

## The process of gathering data is the first step in any data science project. We must first gather data. First, we can use pre-made data; second, we can purchase it from a third-party provider; or third, we can obtain it through Kaggle ,etc. The second approach is to assemble a group of data analysts whose task it is to obtain these photographs from farmers and label them as either showing healthy potato leaves or showing signs of early or late blight. So, this group of annotators collaborates with farmers; they visit the fields and either ask the farmers to take a photo of a leaf or they can take the photo themselves. The photos are then classified with the aid of specialists in agriculture. in order to manually gather the data. But, this procedure will take some time. The third choice is to create a web-scraping script that will browse various websites for potato photographs, collect those images, and then use other tools to annotate the data. I'm using pre-made data from Kaggle for this project.

## Dataset Link: <https://www.kaggle.com/datasets/arjuntejaswi/plant-village>

## 2. METHODOLOGY

## As can be seen from Fig. 1, this project's study has gone through several stages. in the shape of an analytical framework. There are four steps in the suggested research framework, which are as follows:

## 

## Potato Leaf Disease Prediction Data Loading

Our dataset must be in the following format.

Potato Leaf Dataset –> main folder

—-| train

—-| Potato\_Healthy

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Early\_Blight

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Late\_Blight

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| test

—-| Potato\_Healthy

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Early\_Blight

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Late\_Blight

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| valid

—-| Potato\_Healthy

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Early\_Blight

—-| img1.jpg

—-| img2.jpg

—-| img3.jpg

—-| Potato\_Late\_Blight

—-| img1.jpg

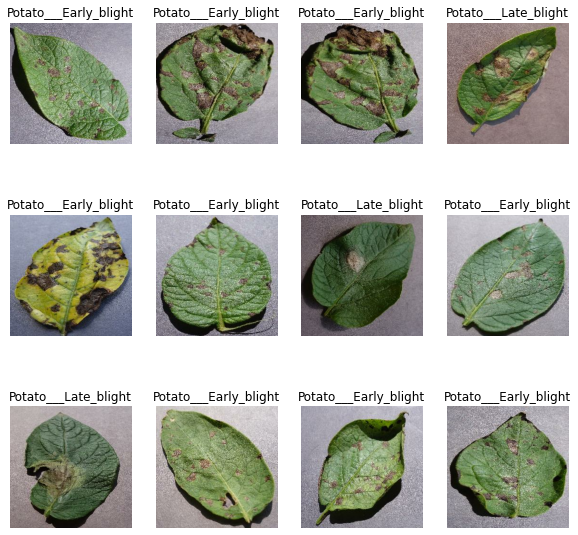
—-| img2.jpg

—-| img3.jpg

## In this study, we'll only utilise 2156 photos total. As we all know, a deep learning model needs a lot of data to be trained. We'll employ one of the most straightforward and efficient techniques, known as data augmentation, to solve this issue. First, let's define data augmentation.

## 3.Data Augmentation: Data Augmentation is a procedure that produces multiple plausible variations of each training sample to fictitiously increase the training dataset's size. Overfitting is reduced as a result. In data augmentation, each image in the training set will be gently shifted, rotated, and resized by various percentages, and all of the resulting images will be added to the training set. As a result, the model is able to accommodate variations in the object's size, orientation, and location in the image. The images' contrast and lighting settings are editable. The photos can be rotated both vertically and horizontally. Merging all of the modifications will allow us to increase the size of our training set.

**5.Visualize some of the images from our dataset**

Output **:**

**Building the Model**

**6.Creating a Layer for Resizing and Normalization**

We should resize our photographs to the correct size before sending them over the network.

Also, we should normalise the image pixel value to enhance model performance (keeping them in range 0 and 1 by dividing by 256).

Both of these things ought to occur during training and inference. In light of this, we can include that as a layer in our sequential model.

You might be wondering why we need to resize the image to (256,256) once more (256,256). You are correct that we don't need to, however this would be helpful after the model training is complete and predictions are being made using it. At that point, anyone can give a non-(256,256) image, and this layer will resize it.

**7.Data Augmentation**

Data Augmentation is needed when we have less data, this boosts the accuracy of our model by augmenting the data.

**Applying Data Augmentation to Train Dataset**

train\_ds = train\_ds.map(

    lambda x, y: (data\_augmentation(x, training=True), y)

).prefetch(buffer\_size=tf.data.AUTOTUNE)

**8.Model Architecture**

We use a CNN coupled with a Softmax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

input\_shape = (BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE, CHANNELS)

n\_classes = 3

model = models.Sequential([

    resize\_and\_rescale,

    layers.Conv2D(32, kernel\_size = (3,3), activation='relu', input\_shape=input\_shape),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64,  kernel\_size = (3,3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64,  kernel\_size = (3,3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

    layers.Dense(n\_classes, activation='softmax'),

])

model.build(input\_shape=input\_shape)

model.summary()

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

sequential (Sequential) (32, 256, 256, 3) 0

conv2d (Conv2D) (32, 254, 254, 32) 896

max\_pooling2d (MaxPooling2D (32, 127, 127, 32) 0

)

conv2d\_1 (Conv2D) (32, 125, 125, 64) 18496

max\_pooling2d\_1 (MaxPooling (32, 62, 62, 64) 0

2D)

conv2d\_2 (Conv2D) (32, 60, 60, 64) 36928

max\_pooling2d\_2 (MaxPooling (32, 30, 30, 64) 0

2D)

conv2d\_3 (Conv2D) (32, 28, 28, 64) 36928

max\_pooling2d\_3 (MaxPooling (32, 14, 14, 64) 0

2D)

conv2d\_4 (Conv2D) (32, 12, 12, 64) 36928

max\_pooling2d\_4 (MaxPooling (32, 6, 6, 64) 0

2D)

conv2d\_5 (Conv2D) (32, 4, 4, 64) 36928

max\_pooling2d\_5 (MaxPooling (32, 2, 2, 64) 0

2D)

flatten (Flatten) (32, 256) 0

dense (Dense) (32, 64) 16448

dense\_1 (Dense) (32, 3) 195

=================================================================

Total params: 183,747

Trainable params: 183,747

Non-trainable params: 0

**9.Compiling the Model**

We use `adam` Optimizer, `SparseCategoricalCrossentropy` for losses, `accuracy` as a metric

model.compile(

    optimizer='adam',

    loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=False),

    metrics=['accuracy']

)

history = model.fit(

    train\_ds,

    batch\_size=BATCH\_SIZE,

    validation\_data=val\_ds,

    verbose=1,

    epochs=50,

)

Epoch 1/50

54/54 [==============================] - 20s 255ms/step - loss: 0.8802 - accuracy: 0.5341 - val\_loss: 0.8462 - val\_accuracy: 0.5938

Epoch 2/50

54/54 [==============================] - 11s 196ms/step - loss: 0.6033 - accuracy: 0.7396 - val\_loss: 0.6225 - val\_accuracy: 0.6979

Epoch 3/50

54/54 [==============================] - 9s 172ms/step - loss: 0.3647 - accuracy: 0.8403 - val\_loss: 0.3065 - val\_accuracy: 0.8802

Epoch 4/50

54/54 [==============================] - 10s 176ms/step - loss: 0.2776 - accuracy: 0.8999 - val\_loss: 0.2702 - val\_accuracy: 0.8750

Epoch 5/50

54/54 [==============================] - 10s 179ms/step - loss: 0.2448 - accuracy: 0.8953 - val\_loss: 0.1857 - val\_accuracy: 0.9062

Epoch 6/50

54/54 [==============================] - 9s 174ms/step - loss: 0.2020 - accuracy: 0.9144 - val\_loss: 0.2987 - val\_accuracy: 0.9115

Epoch 7/50

54/54 [==============================] - 10s 185ms/step - loss: 0.1751 - accuracy: 0.9288 - val\_loss: 0.1854 - val\_accuracy: 0.9375

Epoch 8/50

54/54 [==============================] - 10s 180ms/step - loss: 0.1436 - accuracy: 0.9444 - val\_loss: 0.2273 - val\_accuracy: 0.9167

Epoch 9/50

54/54 [==============================] - 10s 175ms/step - loss: 0.1128 - accuracy: 0.9583 - val\_loss: 0.1425 - val\_accuracy: 0.9479

Epoch 10/50

54/54 [==============================] - 10s 179ms/step - loss: 0.1218 - accuracy: 0.9549 - val\_loss: 0.2310 - val\_accuracy: 0.9115

Epoch 11/50

54/54 [==============================] - 10s 179ms/step - loss: 0.1524 - accuracy: 0.9398 - val\_loss: 0.0774 - val\_accuracy: 0.9688

Epoch 12/50

54/54 [==============================] - 10s 186ms/step - loss: 0.1062 - accuracy: 0.9578 - val\_loss: 0.1787 - val\_accuracy: 0.9427

Epoch 13/50

54/54 [==============================] - 9s 172ms/step - loss: 0.1299 - accuracy: 0.9549 - val\_loss: 0.0929 - val\_accuracy: 0.9531

Epoch 14/50

54/54 [==============================] - 9s 169ms/step - loss: 0.0971 - accuracy: 0.9601 - val\_loss: 0.1230 - val\_accuracy: 0.9531

Epoch 15/50

54/54 [==============================] - 9s 171ms/step - loss: 0.0967 - accuracy: 0.9659 - val\_loss: 0.0804 - val\_accuracy: 0.9635

Epoch 16/50

54/54 [==============================] - 9s 172ms/step - loss: 0.0764 - accuracy: 0.9676 - val\_loss: 0.1225 - val\_accuracy: 0.9531

Epoch 17/50

54/54 [==============================] - 9s 174ms/step - loss: 0.1157 - accuracy: 0.9543 - val\_loss: 0.2200 - val\_accuracy: 0.9219

Epoch 18/50

54/54 [==============================] - 10s 175ms/step - loss: 0.0947 - accuracy: 0.9659 - val\_loss: 0.1852 - val\_accuracy: 0.9271

Epoch 19/50

54/54 [==============================] - 9s 174ms/step - loss: 0.0737 - accuracy: 0.9711 - val\_loss: 0.0923 - val\_accuracy: 0.9583

Epoch 20/50

54/54 [==============================] - 9s 173ms/step - loss: 0.0518 - accuracy: 0.9815 - val\_loss: 0.0678 - val\_accuracy: 0.9688

Epoch 21/50

54/54 [==============================] - 9s 172ms/step - loss: 0.0473 - accuracy: 0.9826 - val\_loss: 0.0516 - val\_accuracy: 0.9740

Epoch 22/50

54/54 [==============================] - 9s 173ms/step - loss: 0.0510 - accuracy: 0.9803 - val\_loss: 0.3043 - val\_accuracy: 0.8958

Epoch 23/50

54/54 [==============================] - 9s 175ms/step - loss: 0.0510 - accuracy: 0.9792 - val\_loss: 0.2573 - val\_accuracy: 0.9062

Epoch 24/50

54/54 [==============================] - 10s 176ms/step - loss: 0.0820 - accuracy: 0.9670 - val\_loss: 0.0828 - val\_accuracy: 0.9635

Epoch 25/50

54/54 [==============================] - 9s 175ms/step - loss: 0.0459 - accuracy: 0.9844 - val\_loss: 0.0912 - val\_accuracy: 0.9740

Epoch 26/50

54/54 [==============================] - 10s 176ms/step - loss: 0.0361 - accuracy: 0.9867 - val\_loss: 0.0354 - val\_accuracy: 0.9844

Epoch 27/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0461 - accuracy: 0.9838 - val\_loss: 0.0364 - val\_accuracy: 0.9844

Epoch 28/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0414 - accuracy: 0.9838 - val\_loss: 0.1192 - val\_accuracy: 0.9479

Epoch 29/50

54/54 [==============================] - 9s 168ms/step - loss: 0.0424 - accuracy: 0.9861 - val\_loss: 0.0509 - val\_accuracy: 0.9844

Epoch 30/50

54/54 [==============================] - 9s 167ms/step - loss: 0.0348 - accuracy: 0.9873 - val\_loss: 0.1987 - val\_accuracy: 0.9531

Epoch 31/50

54/54 [==============================] - 10s 178ms/step - loss: 0.0437 - accuracy: 0.9821 - val\_loss: 0.0371 - val\_accuracy: 0.9948

Epoch 32/50

54/54 [==============================] - 10s 193ms/step - loss: 0.0439 - accuracy: 0.9855 - val\_loss: 0.1708 - val\_accuracy: 0.9375

Epoch 33/50

54/54 [==============================] - 10s 187ms/step - loss: 0.0558 - accuracy: 0.9774 - val\_loss: 0.1559 - val\_accuracy: 0.9531

Epoch 34/50

54/54 [==============================] - 9s 171ms/step - loss: 0.0412 - accuracy: 0.9821 - val\_loss: 0.1024 - val\_accuracy: 0.9583

Epoch 35/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0312 - accuracy: 0.9902 - val\_loss: 0.0919 - val\_accuracy: 0.9583

Epoch 36/50

54/54 [==============================] - 10s 178ms/step - loss: 0.0431 - accuracy: 0.9844 - val\_loss: 0.0217 - val\_accuracy: 0.9948

Epoch 37/50

54/54 [==============================] - 10s 178ms/step - loss: 0.0353 - accuracy: 0.9896 - val\_loss: 0.0092 - val\_accuracy: 1.0000

Epoch 38/50

54/54 [==============================] - 9s 173ms/step - loss: 0.0206 - accuracy: 0.9936 - val\_loss: 0.0079 - val\_accuracy: 1.0000

Epoch 39/50

54/54 [==============================] - 9s 171ms/step - loss: 0.0307 - accuracy: 0.9913 - val\_loss: 0.0209 - val\_accuracy: 0.9896

Epoch 40/50

54/54 [==============================] - 9s 175ms/step - loss: 0.0143 - accuracy: 0.9948 - val\_loss: 0.0240 - val\_accuracy: 0.9896

Epoch 41/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0196 - accuracy: 0.9936 - val\_loss: 0.0441 - val\_accuracy: 0.9844

Epoch 42/50

54/54 [==============================] - 9s 173ms/step - loss: 0.0382 - accuracy: 0.9832 - val\_loss: 0.2912 - val\_accuracy: 0.9271

Epoch 43/50

54/54 [==============================] - 9s 172ms/step - loss: 0.0416 - accuracy: 0.9832 - val\_loss: 0.0425 - val\_accuracy: 0.9896

Epoch 44/50

54/54 [==============================] - 9s 171ms/step - loss: 0.0162 - accuracy: 0.9948 - val\_loss: 0.0567 - val\_accuracy: 0.9792

Epoch 45/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0990 - accuracy: 0.9653 - val\_loss: 0.0892 - val\_accuracy: 0.9688

Epoch 46/50

54/54 [==============================] - 9s 171ms/step - loss: 0.0243 - accuracy: 0.9919 - val\_loss: 0.0174 - val\_accuracy: 0.9948

Epoch 47/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0476 - accuracy: 0.9844 - val\_loss: 0.0217 - val\_accuracy: 0.9896

Epoch 48/50

54/54 [==============================] - 9s 170ms/step - loss: 0.0184 - accuracy: 0.9931 - val\_loss: 0.1227 - val\_accuracy: 0.9635

Epoch 49/50

54/54 [==============================] - 10s 184ms/step - loss: 0.0298 - accuracy: 0.9884 - val\_loss: 0.0528 - val\_accuracy: 0.9844

Epoch 50/50

54/54 [==============================] - 11s 196ms/step - loss: 0.0189 - accuracy: 0.9948 - val\_loss: 0.0064 - val\_accuracy: 1.0000

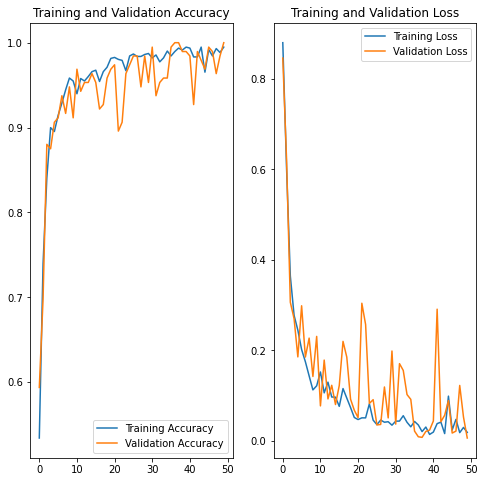
scores = model.evaluate(test\_ds)

8/8 [==============================] - 1s 14ms/step - loss: 0.0063 - accuracy: 1.0000

scores

[0.006251859944313765, 1.0]

**10.Plotting the Accuracy and Loss Curves**



**11.Run prediction on a sample image**



**12.Now run inference on few sample images**



**13.Saving the Model**

We append the model to the list of models as a new version

import os

model\_version=max([int(i) for i in os.listdir("../models") + [0]])+1

model.save(f"../models/{model\_version}")

model.save("../potatoes.h5")

**summary:-**

In order to help with early disease detection and management in potato crops, using CNNs for potato disease plant classification entails gathering and preprocessing labelled image data, designing and training a CNN model, evaluating its performance, and using the trained model for classification of new images.

**References : -**

1. <https://www.analyticsvidhya.com/blog/2021/12/end-to-end-potato-leaf-disease-prediction-project-a-complete-guide/>
2. <https://www.cambridge.org/core/journals/advances-in-animal-biosciences/article/abs/potato-disease-classification-using-convolution-neural-networks/E9303F667377BD763C3054CB8488D36C>