Evaluating Deep Learning models for Plant Leaf Diseases

Introduction

The agricultural industry is crucial in providing high-quality food and contributes most to growing economies and populations. Plant diseases have the potential to significantly reduce the production of food and wipe out species diversity [1]. Utilising precise or automatic detection methods and early diagnosis of plant diseases can improve food output quality and reduce financial losses. In recent years, deep learning has significantly improved the recognition accuracy of image classification and object detection algorithms. In this research, convolutional neural network models were created using deep learning techniques to identify and diagnose plant diseases using explicit images of healthy and diseased leaves. An **open database** comprising 55129 images including both healthy and diseased plants and was used to train the models. The best model architecture out of the ones that were trained was identified with a success rate of 90%. The model is a useful early warning tool and could be extended to support an integrated plant disease identification system operating in real cultivation conditions [2].

Convolutional Neural Networks

Convolutional neural networks (CNNs) is primarily used for image recognition and tasks that involve the processing of pixel database. The main characteristic of the algorithm is its ability to be trained through the process of supervised learning. CNN can automatically extract features from images, such as texture, colour, and shape, which are then used to classify the image as a healthy plant or diseased plant. Convolution is a mathematical operation that allows the merging of two sets of information. In the case of CNN, convolution is applied to the input data, the images, to filter out the information and produce a feature map. A filter is a small matrix of numbers that is used to detect patterns in images. Filters are also known as kernels or convolution matrices. A convolutional layer is responsible for recognising features in pixels, while the pooling layer is used for making these features more abstract. Finally, the fully-connected layers are responsible for using the acquired features for prediction [2, 3].

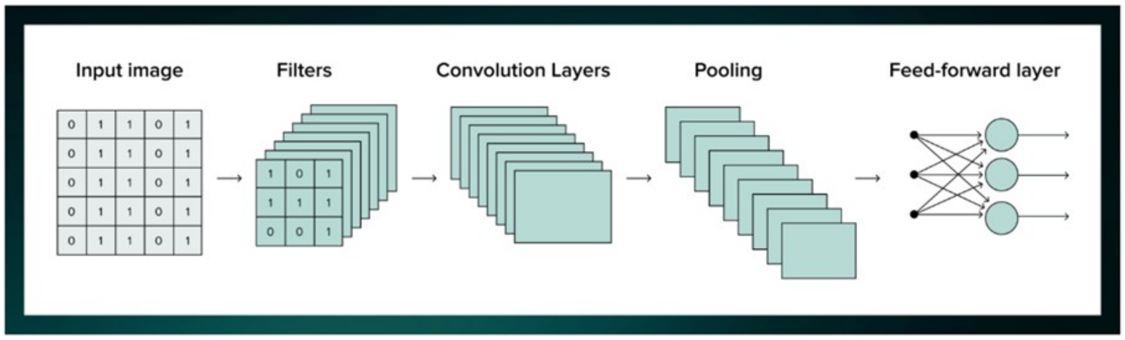


Fig:1 CNN Architecture

VGG

According to the research, the highest successful classification percentage of 99.53% was achieved by the Visual Geometry Group (VGG) model [2]. VGG convolutional neural network is one of the most popular image recognition architectures. The most crucial characteristics of convolutional neural networks are included in the VGG design. The algorithm is known for its robustness and easy–to–understand architecture. The key concept of VGG is that its network consists of small convolution filters.

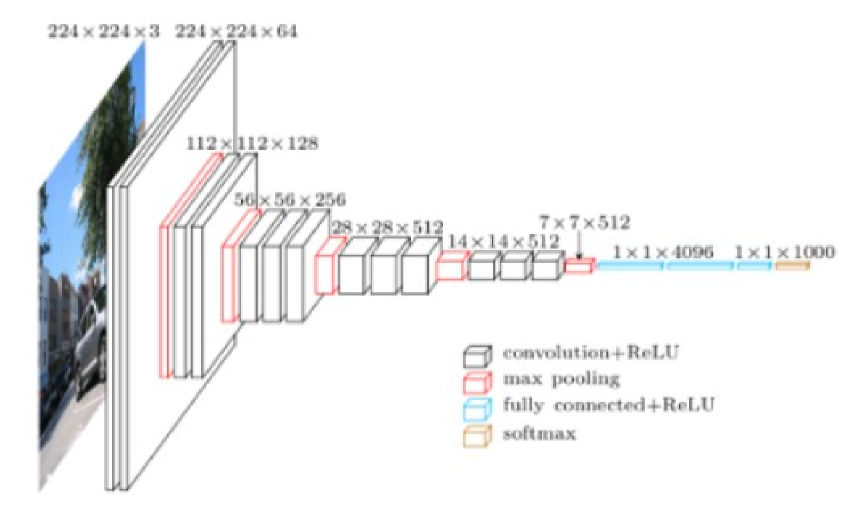


Fig: VGG16 architecture

An overview of the VGG design is provided below:

- Input — VGG receives an image input. The input images are cropped by the model, which is fixed for the complete database, taking the centre of each image and cropping it to a maximum size of 256×256.

- Convolutional layers — The convolutional filters of VGG use the smallest possible receptive field of 3×3. VGG also uses a 1×1 convolution filter as the input's linear transformation.
- ReLU activation — Rectified Linear Unit Activation Function (ReLU) is a linear function that outputs zero for negative inputs and a matching outcome for positive inputs. To maintain spatial resolution after convolution, VGG has a predetermined convolution stride of 1 pixel (the stride number represents how many pixels the filter "moves" to cover the entire space of the image).
- Hidden layers — All the VGG network's hidden layers use ReLU. The latter increases training time and memory consumption with little improvement in overall accuracy.
- Pooling layers – The number of parameters and dimensionality of the feature maps produced by each convolution phase are decreased by adding a pooling layer after a series of convolutional layers. Pooling is essential because the number of available filters quickly increases from 64 to 128, 256, and finally 512 in the final levels.
- Fully connected layers — VGG includes three fully connected layers. The first two layers have 4096 channels, and the third layer has 1000 channels, one for every class.

Implementation

In this section, we will implement and check the performance of the VGG16 model. We can use the pre-built VGG16 model from keras. The data we are going to use is similar to the one using in the paper, but this have only 37 classes of available. Let's begin by importing required modules and loading data.

```
from keras.applications.vgg16 import VGG16

import keras
from keras.layers import Dense
from keras.layers import Flatten
from keras.preprocessing.image import ImageDataGenerator

import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

The size of the **dataset** is about 1GB. So, we uploaded the zipped dataset to Google Drive and using it in Google Colab via mounting the drive into the notebook. If you have the dataset or want to execute on your local machine, you can skip this step.

```
# Path to the dataset
!unzip '/content/drive/MyDrive/Colab
Notebooks/Plant_leaf_diseases_dataset.zip'
```

Now that we have the data, we will split the data into 2 sets, i.e., one for training the model and another for testing to check the performance. For performing test train split, we will use the split-folders package.

```
!pip install split-folders
import splitfolders
```

```
Looking in indexes: https://pypi.org/simple, https://us-
python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: split-folders in
/usr/local/lib/python3.9/dist-packages (0.5.1)
```

We will split the data in 70:30 ratio for train and validation(test).

```
input_folder = 'Plant_leave_diseases_dataset_with_augmentation/'
output = 'data_splits/'
splitfolders.ratio(input_folder, output=output, seed=42, ratio=(.7,
.3))
```

```
Copying files: 55129 files [00:10, 5473.82 files/s]
```

Now that we have the data ready to use, let's start loading the training and validation sets.

```
dataset_path = 'data_splits/'

train_dir = dataset_path + "train"
valid_dir = dataset_path + "val"

train_gen = ImageDataGenerator(rescale=1./255, shear_range=0.2,
zoom_range=0.2, width_shift_range=0.2, height_shift_range=0.2,
fill_mode='nearest')

valid_gen = ImageDataGenerator(rescale=1./255)

batch_size = 32

train_set = train_gen.flow_from_directory(train_dir, target_size=(224,
224), batch_size=batch_size, class_mode='categorical')

valid_set = valid_gen.flow_from_directory(valid_dir, target_size=(224,
224), batch_size=batch_size, class_mode='categorical')
```

```
Found 38581 images belonging to 37 classes.
Found 16548 images belonging to 37 classes.
```

Data

As you can see, we have a total of 37 different classes of plant leave images, including healthy and diseased plants.

```
class_dict = train_set.class_indices
print(pd.Series(class_dict))
```

```
Apple___Apple_scab                                0
Apple___Black_rot                                 1
Apple___Cedar_apple_rust                          2
Apple___healthy                                   3
Background_without_leaves                         4
Blueberry___healthy                               5
Cherry___Powdery_mildew                           6
Cherry___healthy                                  7
Corn___Cercospora_leaf_spot Gray_leaf_spot        8
Corn___Common_rust                                9
Corn___Northern_Leaf_Blight                      10
Corn___healthy                                   11
Grape___Black_rot                                12
Grape___Esca_(Black_Measles)                     13
Grape___Leaf_blight_(Isariopsis_Leaf_Spot)       14
Grape___healthy                                  15
Orange___Haunglongbing_(Citrus_greening)         16
Peach___Bacterial_spot                           17
Peach___healthy                                  18
Pepper,_bell___Bacterial_spot                    19
Pepper,_bell___healthy                           20
Potato___Early_blight                            21
Potato___Late_blight                             22
Potato___healthy                                 23
Raspberry___healthy                              24
Soybean___healthy                                25
Squash___Powdery_mildew                          26
Strawberry___Leaf_scorch                         27
Strawberry___healthy                             28
Tomato___Bacterial_spot                          29
Tomato___Early_blight                            30
Tomato___Late_blight                             31
Tomato___Leaf_Mold                               32
Tomato___Septoria_leaf_spot                      33
Tomato___Spider_mites Two-spotted_spider_mite    34
Tomato___Target_Spot                             35
Tomato___healthy                                 36
dtype: int64
```

VGG16 Model

Let's define the VGG16 base model and initialize the classifier accordingly.

```
base_model = VGG16(include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False

classifier=keras.models.Sequential()
classifier.add(base_model)
classifier.add(Flatten())
classifier.add(Dense(37, activation='softmax'))
classifier.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 3s 0us/step
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688  
                                                                 
 flatten (Flatten)           (None, 25088)             0         
                                                                 
 dense (Dense)               (None, 37)                928293    
                                                                 
=================================================================
Total params: 15,642,981
Trainable params: 928,293
Non-trainable params: 14,714,688
_________________________________________________________________
```

We can run this code to see the architecture of our VGG model as seen in the previous section.

```
from keras.utils import plot_model
plot_model(base_model, show_shapes=True, to_file='vgg_block.png')
```

Let's compile the model with accuracy as the metric and cross entropy as the measure of loss.

```
classifier.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

Since the model is ready, we can get the training and testing samples and start the training process. We already defined batch size as 32 while loading the data.

```
train_num = train_set.samples
valid_num = valid_set.samples

history = classifier.fit(train_set,
                         steps_per_epoch=train_num//batch_size,
                         validation_data=valid_set,
                         epochs=10,
                         validation_steps=valid_num//batch_size,
                         )
```

```
Epoch 1/10
1205/1205 [==============================] - 606s 488ms/step - loss:
0.8596 - accuracy: 0.7612 - val_loss: 0.4505 - val_accuracy: 0.8760
Epoch 2/10
1205/1205 [==============================] - 582s 483ms/step - loss:
0.5504 - accuracy: 0.8561 - val_loss: 0.4991 - val_accuracy: 0.8757
Epoch 3/10
```

```
1205/1205 [==============================] - 575s 477ms/step - loss:
0.5331 - accuracy: 0.8711 - val_loss: 0.4149 - val_accuracy: 0.9011
Epoch 4/10
1205/1205 [==============================] - 596s 495ms/step - loss:
0.4900 - accuracy: 0.8838 - val_loss: 0.5022 - val_accuracy: 0.8995
Epoch 5/10
1205/1205 [==============================] - 599s 497ms/step - loss:
0.4710 - accuracy: 0.8944 - val_loss: 0.4717 - val_accuracy: 0.9096
Epoch 6/10
1205/1205 [==============================] - 597s 496ms/step - loss:
0.4639 - accuracy: 0.8976 - val_loss: 0.4456 - val_accuracy: 0.9168
Epoch 7/10
1205/1205 [==============================] - 576s 478ms/step - loss:
0.4332 - accuracy: 0.9071 - val_loss: 0.3949 - val_accuracy: 0.9236
Epoch 8/10
1205/1205 [==============================] - 592s 492ms/step - loss:
0.4382 - accuracy: 0.9084 - val_loss: 0.3243 - val_accuracy: 0.9380
Epoch 9/10
1205/1205 [==============================] - 594s 493ms/step - loss:
0.4187 - accuracy: 0.9156 - val_loss: 0.3234 - val_accuracy: 0.9384
Epoch 10/10
1205/1205 [==============================] - 593s 492ms/step - loss:
0.3863 - accuracy: 0.9212 - val_loss: 0.5245 - val_accuracy: 0.9119
```

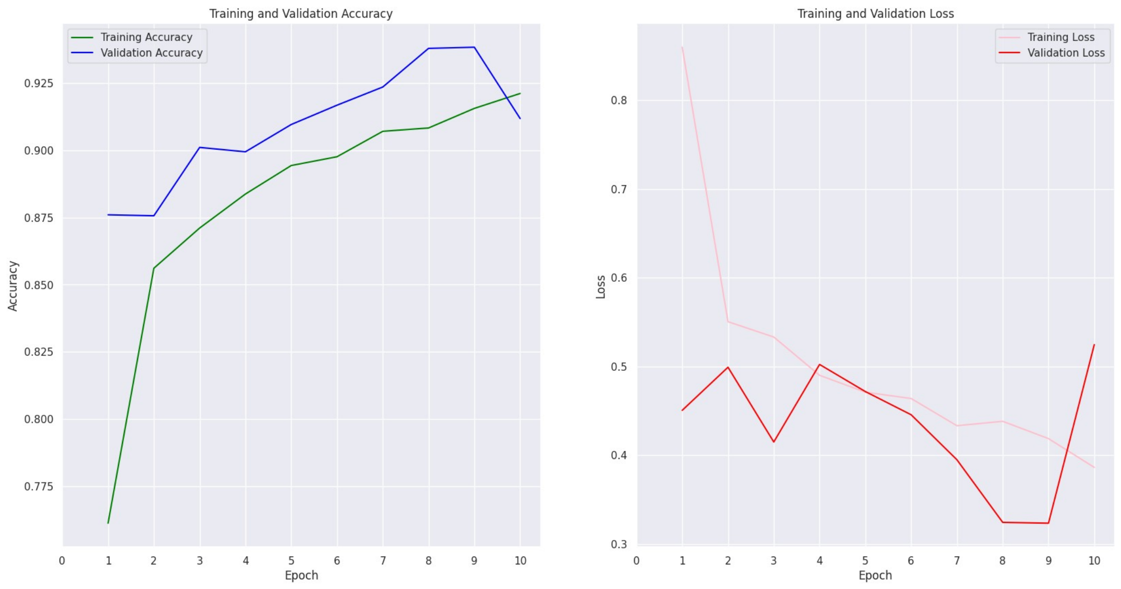
We have done only 10 epochs, and it took about 100 mins on NVIDIA Tesla T4 GPU that was available in Google Colab. So, on average 600s (10 mins) per Epoch. By end of this process, we have achieved about 92% accuracy and 91% validation accuracy. Let's look at the metrics on graph.

```
vgg_acc_scores = history.history['accuracy']
vgg_val_acc_scores = history.history['val_accuracy']
vgg_loss = history.history['loss']
vgg_val_loss = history.history['val_loss']
epochs = range(1, len(vgg_acc_scores) + 1)


plt.subplots(1, 2, figsize=(20,10))
plt.subplot(1, 2, 1)
plt.plot(epochs, vgg_acc_scores, color='green', label='Training
Accuracy')
plt.plot(epochs, vgg_val_acc_scores, color='blue', label='Validation
Accuracy')
plt.title('Training and Validation Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(range(0,11))
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs, vgg_loss, color='pink', label='Training Loss')
plt.plot(epochs, vgg_val_loss, color='red', label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.xticks(range(0,11))
plt.legend()

plt.show()
```



As you can see, the performance of the model increased gradually per cycle. When we have tested for similar dataset, we have achieved about 95% accuracy by 20 epochs.

```
print('Average accuracy of VGG16 on validation set: ',
round(sum(vgg_val_acc_scores) * 100/len(vgg_val_acc_scores), 2))
```

```
Average accuracy of VGG16 on validation set:  90.91
```

```
print('Average loss of VGG16 on validation set: ',
round(sum(vgg_val_loss)/len(vgg_val_loss), 2))
```

```
Average loss of VGG16 on validation set:  0.44
```

On average the VGG16 model has 90.91% accuracy and 0.44 loss. Now, we can build a AlexNet model and compare the performance of the model with VGG.

AlexNet

Let's build the AlexNet model basing on the information from the base paper. As per the paper, it has 5 Convolutional layers with 96, 256, 384, 384, 256 kernels respectively. The last 3 layers are connected without any max pooling. The filter sizes for the layers are 11 x 11 for the first Convolutional layers , 5 x 5 for the 2nd and 3 x 3 for the remaining layers.

```python
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D,
MaxPooling2D

AlexNet = Sequential()

# 1st Convolutional Layer
AlexNet.add(Conv2D(filters=96, input_shape=(224,224,3),
kernel_size=(11,11), strides=(4,4), padding="valid"))
AlexNet.add(Activation('relu'))

# Max Pooling
AlexNet.add(MaxPooling2D(pool_size=(3,3), strides=(2,2),
padding='valid'))

# 2nd Convolutional Layer
AlexNet.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1),
padding='valid'))
AlexNet.add(Activation('relu'))

# Max Pooling
AlexNet.add(MaxPooling2D(pool_size=(3,3), strides=(2,2),
padding='valid'))

# 3rd Convolutional Layer
AlexNet.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
padding='valid'))
AlexNet.add(Activation('relu'))

# 4th Convolutional Layer
AlexNet.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
padding='valid'))
AlexNet.add(Activation('relu'))

# 5th Convolutional Layer
AlexNet.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
padding='valid'))
AlexNet.add(Activation('relu'))

# Max Pooling
AlexNet.add(MaxPooling2D(pool_size=(2,2), strides=(2,2),
padding='valid'))

# Passing to a Fully Connected layer
AlexNet.add(Flatten())

# 1st Fully Connected Layer
AlexNet.add(Dense(4096, input_shape=(224*224*3,)))
AlexNet.add(Activation('relu'))
AlexNet.add(Dropout(0.5)) # Dropout to prevent over-fitting

# 2nd Fully Connected Layer
AlexNet.add(Dense(4096))
AlexNet.add(Activation('relu'))
AlexNet.add(Dropout(0.5))

# 3rd Fully Connected Layer
AlexNet.add(Dense(1000))
AlexNet.add(Activation('relu'))
AlexNet.add(Dropout(0.5))

# Output Layer
AlexNet.add(Dense(37))
AlexNet.add(Activation('softmax'))

plot_model(AlexNet, show_shapes=True, to_file='alex_block.png')
```

Similar to VGG, let's compile the model and fit with the same data.

```
# Compile the model
AlexNet.compile(loss=keras.losses.categorical_crossentropy,
optimizer='adam', metrics=['accuracy'])

alex_hist = AlexNet.fit(train_set,
                      steps_per_epoch=train_num//batch_size,
                      validation_data=valid_set,
                      epochs=10,
                      validation_steps=valid_num//batch_size
                      )
```

```
Epoch 1/10
1205/1205 [==============================] - 530s 434ms/step - loss:
3.3940 - accuracy: 0.1412 - val_loss: 3.0005 - val_accuracy: 0.2251
Epoch 2/10
1205/1205 [==============================] - 525s 435ms/step - loss:
2.9392 - accuracy: 0.2361 - val_loss: 2.6788 - val_accuracy: 0.2787
Epoch 3/10
1205/1205 [==============================] - 512s 425ms/step - loss:
2.7472 - accuracy: 0.2646 - val_loss: 2.5561 - val_accuracy: 0.2983
Epoch 4/10
1205/1205 [==============================] - 516s 428ms/step - loss:
2.6583 - accuracy: 0.2796 - val_loss: 2.5424 - val_accuracy: 0.2975
Epoch 5/10
1205/1205 [==============================] - 520s 432ms/step - loss:
2.6195 - accuracy: 0.2902 - val_loss: 2.3684 - val_accuracy: 0.3456
Epoch 6/10
1205/1205 [==============================] - 517s 429ms/step - loss:
2.5364 - accuracy: 0.3063 - val_loss: 2.2484 - val_accuracy: 0.3699
Epoch 7/10
```

```
1205/1205 [==============================] - 512s 425ms/step - loss:
2.4651 - accuracy: 0.3237 - val_loss: 2.1769 - val_accuracy: 0.3894
Epoch 8/10
1205/1205 [==============================] - 515s 428ms/step - loss:
2.4047 - accuracy: 0.3402 - val_loss: 2.0965 - val_accuracy: 0.4067
Epoch 9/10
1205/1205 [==============================] - 514s 426ms/step - loss:
2.3281 - accuracy: 0.3516 - val_loss: 2.1217 - val_accuracy: 0.4006
Epoch 10/10
1205/1205 [==============================] - 512s 425ms/step - loss:
2.2700 - accuracy: 0.3652 - val_loss: 2.0625 - val_accuracy: 0.4078
```

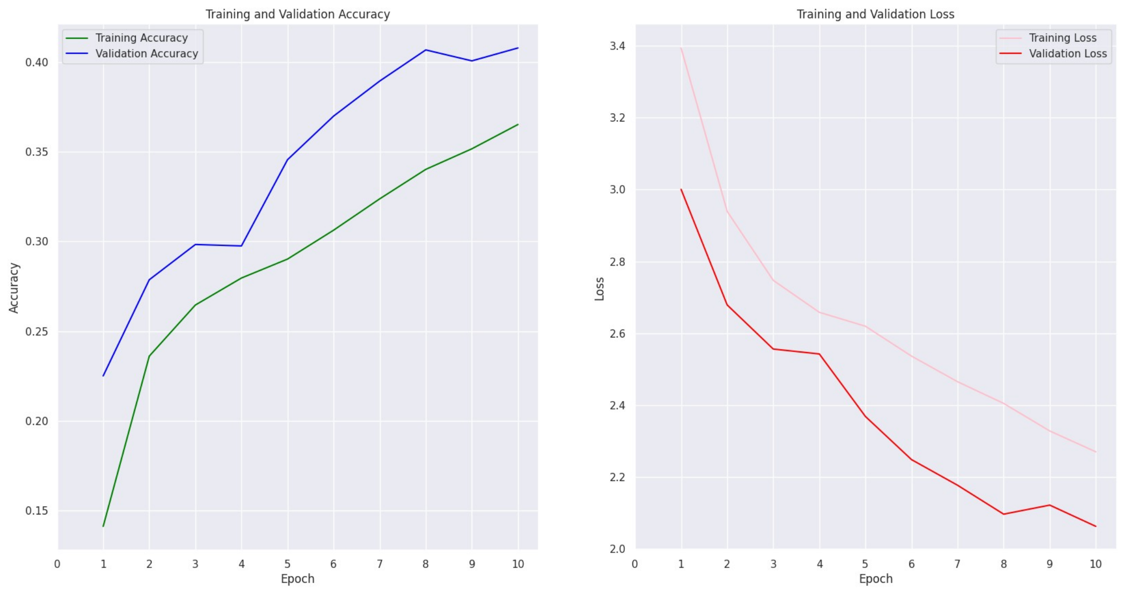
Similar to VGG, it took about 500s (8-9 mins) per Epoch and about 100 mins in total. By end of this process, we have achieved about 36% accuracy and 40% validation accuracy. Let's plot graphs to check the performance of AlexNet.

```
alex_acc_scores = alex_hist.history['accuracy']
alex_val_acc_scores = alex_hist.history['val_accuracy']
alex_loss = alex_hist.history['loss']
alex_val_loss = alex_hist.history['val_loss']
alex_epochs = range(1, len(alex_acc_scores) + 1)


plt.subplots(1, 2, figsize=(20,10))
plt.subplot(1, 2, 1)
plt.plot(alex_epochs, alex_acc_scores, color='green', label='Training
Accuracy')
plt.plot(alex_epochs, alex_val_acc_scores, color='blue',
label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(range(0,11))
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(alex_epochs, alex_loss, color='pink', label='Training Loss')
plt.plot(alex_epochs, alex_val_loss, color='red', label='Validation
Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.xticks(range(0,11))
plt.legend()

plt.show()
```



```
print('Average accuracy of AlexNet on validation set: ',
round(sum(alex_val_acc_scores) * 100/len(alex_val_acc_scores), 2))

print('Average loss of AlexNet on validation set: ',
round(sum(alex_val_loss)/len(alex_val_loss), 2))
```

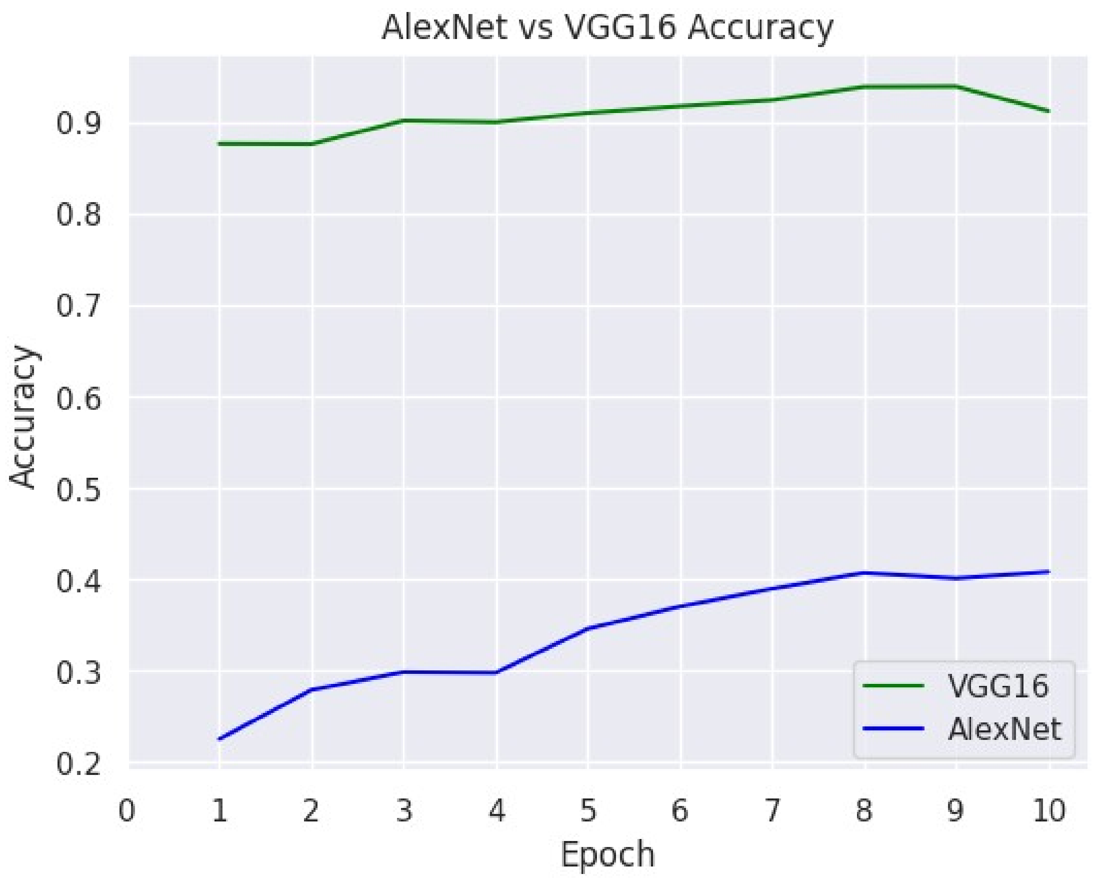
```
Average accuracy of AlexNet on validation set:  34.2
Average loss of AlexNet on validation set:  2.39
```

It looks like AlexNet model has an average of 34.2% accuracy after 10 epochs. When tested with just 5 epochs it was even lower than 30%. If we increase the epochs, the accuracy tends to increase. In the paper, they have run AlexNet for 150 epoch and got over 90% accuracy. Same is the case with VGG.

AlexNet vs VGG16 Accuracy comparsion

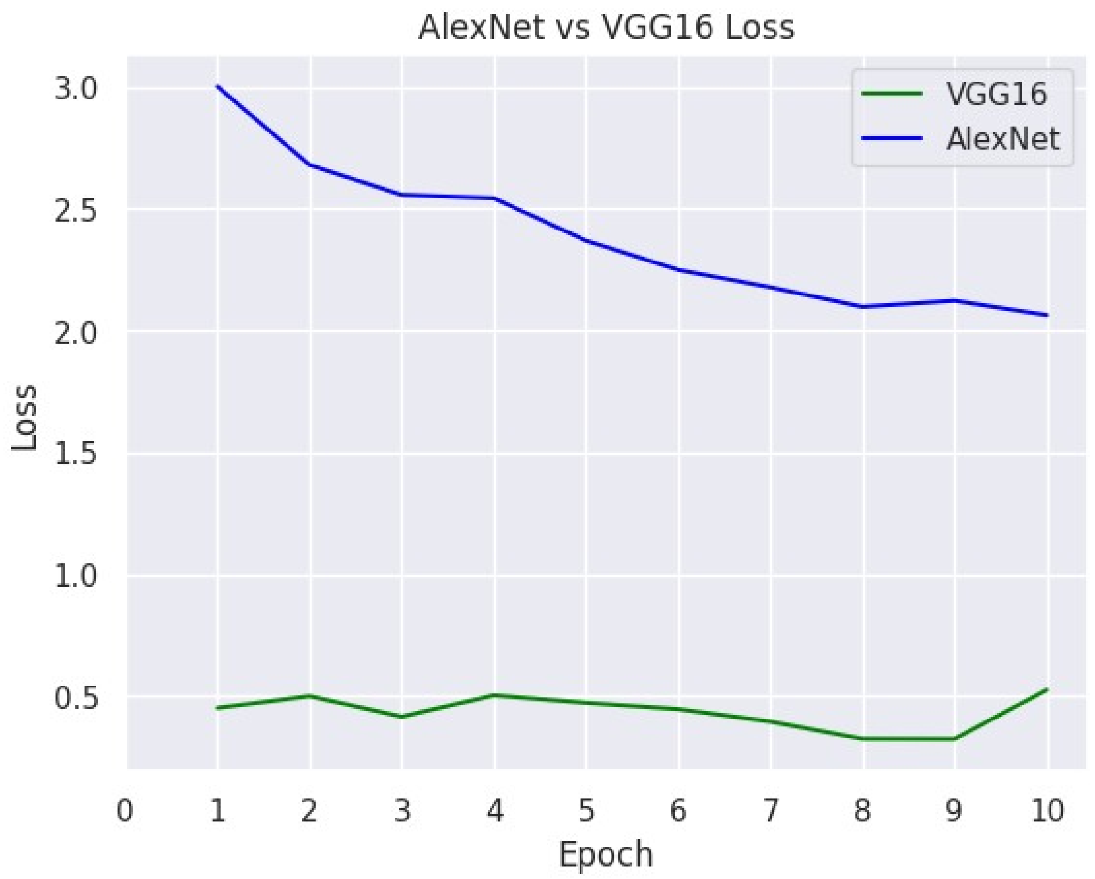
```
plt.plot(alex_epochs, vgg_val_acc_scores, color='green',
label='VGG16')
plt.plot(alex_epochs, alex_val_acc_scores, color='blue',
label='AlexNet')
plt.title('AlexNet vs VGG16 Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.xticks(range(0,11))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f122c97f880>
```



```
plt.plot(alex_epochs, vgg_val_loss, color='green', label='VGG16')
plt.plot(alex_epochs, alex_val_loss, color='blue', label='AlexNet')
plt.title('AlexNet vs VGG16 Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.xticks(range(0,11))
plt.legend()
```

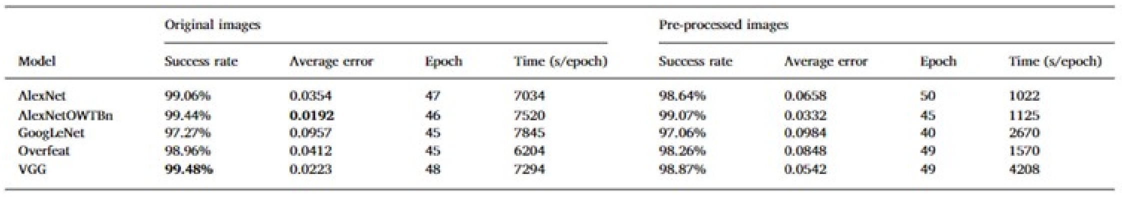
```
<matplotlib.legend.Legend at 0x7f122c39b550>
```



Looking at the above 2 graphs, we can conclude within the limits of our experiment that VGG has better performance compared to the AlexNet model.

Results

Apart from our experiment, from the paper, we can see that the lowest success rate was 97.06%, and the highest success rate was 99.48%. The results indicate that all CNN model designs had good accuracy in classifying plant diseases. The best model for plant disease identification and diagnosis out of the five was VGG with original images, which had the highest success rate. The second-highest success rate was attained by AlexNetOWTBn using pre-processed photos [2]. We can see similar performance variations for the original images as well.



	Original images				Pre-processed images			
Model	Success rate	Average error	Epoch	Time (s/epoch)	Success rate	Average error	Epoch	Time (s/epoch)
AlexNet	99.06%	0.0354	47	7034	98.64%	0.0658	50	1022
AlexNetOWTBn	99.44%	**0.0192**	46	7520	99.07%	0.0332	45	1125
GoogLeNet	97.27%	0.0957	45	7845	97.06%	0.0984	40	2670
Overfeat	98.96%	0.0412	45	6204	98.26%	0.0848	49	1570
VGG	**99.48%**	0.0223	48	7294	98.87%	0.0542	49	4208

Conclusion

Ferentinos' study is a remarkable contribution to the field of agricultural research since it gives a well-written and thorough examination of deep learning models for plant disease detection and diagnosis. The author explains the significance of this subject professionally and dives into the benefits and cons of various techniques. Compared to previous studies, this work stands out for its thorough examination of these models and its emphasis on future research goals to solve present obstacles and stimulate further developments [2]. The paper highlights the potential of deep learning models for plant disease detection and diagnosis, which can achieve high accuracy levels and potentially improve the efficiency of disease diagnosis. The study's results are insightful and well-supported, highlighting the significant advances made in plant disease diagnosis using deep learning algorithms.

Authors:

Bhageeradh Paleti (220138268)

Udit Garud (220270339)

Chetana Vandavasi (220126863)

References

[1] A. J., J. Eunice, D. E. Popescu, M. K. Chowdary, and J. Hemanth, "Deep Learning-Based Leaf Disease Detection in Crops Using Images for Agricultural Applications," Agronomy, vol. 12, no. 10, p. 2395, Oct. 2022, doi: https://doi.org/10.3390/agronomy12102395.

[2] K. P. Ferentinos, "Deep learning models for plant disease detection and diagnosis," Computers and Electronics in Agriculture, vol. 145, pp. 311–318, Feb. 2018, doi: https://doi.org/10.1016/j.compag.2018.01.009.

[3] K. Simonyan and A. Zisserman, "Published as a conference paper at ICLR 2015 VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION," 2015. Available: https://arxiv.org/pdf/1409.1556.pdf.

[4] R. Kumar, A. Chug, A. P. Singh, and D. Singh, "A Systematic Analysis of Machine Learning and Deep Learning Based Approaches for Plant Leaf Disease Classification: A Review," Journal of Sensors, vol. 2022, pp. 1–13, Jul. 2022, doi: https://doi.org/10.1155/2022/3287561.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Communications of the ACM, vol. 60, no. 6, pp. 84–90, May 2012, doi: https://doi.org/10.1145/3065386.

[6] A. P. J and G. Gopal, "Data for: Identification of Plant Leaf Diseases Using a 9-layer Deep Convolutional Neural Network," data.mendeley.com, vol. 1, Apr. 2019, doi: https://doi.org/10.17632/tywbtsjrjv.1. Data Source: https://data.mendeley.com/datasets/tywbtsjrjv/1

[7] “Alexnet Architecture | Introduction to Architecture of Alexnet,” Analytics Vidhya, Mar. 19, 2021. https://www.analyticsvidhya.com/blog/2021/03/introduction-to-the-architecture-of-alexnet/