CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

Chapter 5 explains the implementation and testing of projects development about Potato and Tomato Diseases Detection Using Convolutional Neural Network Algorithm. Below is the code of the Convolutional Neural Network algorithms used to obtain results from the project developed.

```
import numpy as np
import pickle
import cv2
import tensorflow
import seaborn as sn
from os import listdir
from sklearn.preprocessing import LabelBinarizer
from keras.applications import MobileNet
from keras.models import Model
from tensorflow.keras.layers import GlobalAveragePooling2D
from keras.layers.core import Dropout, Dense
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers.legacy import Adam
from keras.preprocessing import image
from tensorflow.keras.utils import img_to_array
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, precision_score, recall_score
```

Figure 5.1 : Import libraries and packages

1. **NumPy** : a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
2. **Sklearn**: a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, *k*-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.
3. **Keras** : Keras is an open source neural network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, or Theano. Designed to enable

fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible.

4. **Matplotlib** : a plotting library for the Python programming language and its numerical mathematics extension.

```
def convert_image_to_array(image_dir):
    try:
        image = cv2.imread(image_dir)
        if image is not None :
            image = cv2.resize(image, default_image_size)
            return img_to_array(image)
        else :
            return np.array([])
    except Exception as e:
        print(f"Error : {e}")
        return None
```

Figure 5.2 : Convert image to array

Converted each image to an array using the function above. After converting each image to an array using this same function, the author did something similar to what the author has in the image.

```
image_list, label_list = [], []
try:
    print("[INFO] Loading images ...")
    root_dir = listdir(directory_root)
    for directory in root_dir :
        # remove .DS_Store from list
        if directory == ".DS_Store" :
            root_dir.remove(directory)

    for plant_folder in root_dir :
        plant_disease_folder_list = listdir(f"{directory_root}/{plant_folder}")

        for disease_folder in plant_disease_folder_list :
            # remove .DS_Store from list
            if disease_folder == ".DS_Store" :
                plant_disease_folder_list.remove(disease_folder)

        for plant_disease_folder in plant_disease_folder_list:
            print(f"[INFO] Processing {plant_disease_folder} ...")
            plant_disease_image_list = listdir(f"{directory_root}/{plant_folder}/{plant_disease_folder}/")


            for single_plant_disease_image in plant_disease_image_list :
                if single_plant_disease_image == ".DS_Store" :
                    plant_disease_image_list.remove(single_plant_disease_image)

            for image in plant_disease_image_list[:200]:
                image_directory = f"{directory_root}/{plant_folder}/{plant_disease_folder}/{image}"
                if image_directory.endswith(".jpg") == True or image_directory.endswith(".JPG") == True:
                    image_list.append(convert_image_to_array(image_directory))
                    label_list.append(plant_disease_folder)
    print("[INFO] Image loading completed")
except Exception as e:
    print(f"Error : {e}")
```

Figure 5.3 : Load the dataset

Next thing was to load the dataset , from the image above the author picked just 200 images from each folder.

```
label_binarizer = LabelBinarizer()
image_labels = label_binarizer.fit_transform(label_list)
pickle.dump(label_binarizer,open('label_transform.pkl', 'wb'))
n_classes = len(label_binarizer.classes_)
```

Figure 5.4 : Convert the image labels to binary

Using Scikit-learn's Label Binarizer , the author converted each image label to binary levels. Then the author saved the label binarizer instance using pickle after which the author printed the classes from the label binarizer.

```
np_image_list = np.array(image_list, dtype=np.float32) / 225.0
```

Figure 5.5 : Pre-process input data

In the image above, the author further pre-process the input data by scaling the data points from [0, 255] (the minimum and maximum RGB values of the image) to the range [0, 1].

```
print("[INFO] Splitting data to train, test")
X_train, X_test, Y_train, Y_test = train_test_split(np_image_list, image_labels, test_size=0.2, random_state = 42)
```

Figure 5.6 : Split data to train, test

In the image above, the author performed a training/testing split on the data using 80% of the images for training and 20% for testing.

```
aug = ImageDataGenerator(
    rotation_range=25,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest")
```

Figure 5.7 : Create image generator

In the image above, the author created an image generator object which performs random rotations, shifts, flips, crops, and sheers on our image dataset. This allows us to use a bigger dataset and still achieve high results.

```
kfold = KFold(n_splits=num_folds, shuffle=True, random_state=42)
fold_no = 1
for train, val in kfold.split(X_train, Y_train):
    print(f'Training for fold {fold_no} ...')
    X_train_cv, X_val_cv = X_train[train], X_train[val]
    Y_train_cv, Y_val_cv = Y_train[train], Y_train[val]

    # Define your model here and compile it
    # Create the base model of MobileNet
    model = MobileNet(weights='imagenet', include_top=False, input_shape=(height, width, depth))

    # Freeze the layers
    for layer in model.layers:
        layer.trainable = False

    # Add new layers
    x = model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.5)(x)
    predictions = Dense(n_classes, activation='softmax')(x)

    # Create the final model
    model = Model(inputs=model.input, outputs=predictions)

    opt = Adam(learning_rate=INIT_LR, decay=INIT_LR / EPOCHS)
    # distribution
    model.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])

    early_stopping = EarlyStopping(monitor='val_accuracy', patience=10)
```

Figure 5.8 : Create the MobileNet Model

First, KFold divides the dataset into k folds. If shuffle is set to true, then the splitting will be random. With cross validation, dataset is divided into n splits. N-1 split is used for training and the remaining split is used for testing. The model runs through the entire dataset n times and at each time, a different split is used for testing. Thus, it use all of data points for both training and testing.

Next step is create a MobileNet model where the classification layers will depend on the very last layer before the flatten operation. Then, freeze the convolutional layers to use the base model as a feature extractor. To convert the feature vectors into actual predictions, the author apply a Global Average Pooling 2D layer to convert the feature vector into a 1280 element vector. Then the author push this through a Dense layer to obtain the final prediction. The author used Keras Adam Optimizer for the model to compile.

```
    # Train the model
    history = model.fit(
        aug.flow(X_train_cv, Y_train_cv, batch_size=BS),
        validation_data=(X_val_cv, Y_val_cv),
        steps_per_epoch=len(X_train_cv) // BS,
        epochs=EPOCHS, verbose=1,
        callbacks=[early_stopping])

    # Generate generalization metrics
    scores = model.evaluate(X_val_cv, Y_val_cv, verbose=0)
    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {scores[0]}; {model.metrics_names[1]} of {scores[1]*100}%')
    print('------------------------------------------------------------------------')
    acc_per_fold.append(scores[1] * 100)
    loss_per_fold.append(scores[0])


    fold_no += 1

# == Provide average scores ==
print('------------------------------------------------------------------------')
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print('------------------------------------------------------------------------')
  print(f'> Fold {i+1} - Validation Loss: {loss_per_fold[i]} - Validation Accuracy: {acc_per_fold[i]}%')
print('------------------------------------------------------------------------')
print('Average scores for all folds:')
print(f'> Validation Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'> Validation Loss: {np.mean(loss_per_fold)}')
print('------------------------------------------------------------------------')
```

Figure 5.9 : Training the model

Training the model is initiated in Figure 5.9 where call model.fit , supplying our data augmentation object, training/testing data, and the number of epochs we wish to train for. The author used an epochs value of 25 for this project.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
#Train and validation accuracy
plt.plot(epochs, acc, 'b', label='Training accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
plt.title('Training & Validation accuracy')
plt.legend()

plt.figure()
#Train and validation loss
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training & Validation loss')
plt.legend()
plt.show()
```

Figure 5.10 : Plotted a graph

Using matplotlib, the author plotted a graph for Training and Validation accuracy and Training and Validation loss in Figure 5.10

```
# evaluate the model on unseen test data
test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=0)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_acc*100}%')
```

Figure 5.11 : Calculated the accuracy

Next in Figure 5.11 the author calculated the accuracy of the model using the test data (X_test and Y_test) created earlier. The author got an accuracy score of 91.34 %.

```
result=model.predict(npp_image)
itemindex = np.where(result==np.max(result))
print("probability:"+str(np.max(result))+"\n"+label_binarizer.classes_[itemindex[1][0]])
```

Figure 5.12 : Predict the results

Next in the Figure 5.12, the author gets results and score related to predictions that match the dataset that has been inputted according to the class that has been divided.

```
#Confusion Matrix
results=model.predict(X_test)
predicted_classes = np.argmax(results, axis=1)
true_classes = np.argmax(Y_test, axis=1)
confusion = confusion_matrix(true_classes, predicted_classes)

# create heatmap
sn.heatmap(confusion, annot=True, cmap='Blues', fmt='d')

# add x and y labels
plt.xlabel('Predicted')
plt.ylabel('True')

# show plot
plt.show()
```

Figure 5.13 : Confusion Matrix

Confusion matrix can be created by predictions made from a logistic regression. Once metrics is imported the author use the confusion matrix function on his actual and predicted values. To create a more interpretable visual display the author convert the table into a confusion matrix

display. Then, vizualizing the display requires that we import pyplot from matplotlib. Finally to display the plot use the functions plot() and show() from pyplot.

```
#Performance Metrics
acc = accuracy_score(true_classes, predicted_classes)
print(f'Test Accuracy: {acc*100}%')

f1 = f1_score(true_classes, predicted_classes, average='weighted')
print(f'F1 score: {f1}')

precision = precision_score(true_classes, predicted_classes, average='weighted')
print(f'Precision: {precision}')

recall = recall_score(true_classes, predicted_classes, average='weighted')
print(f'Recall: {recall}')
```

Figure 5.14 : Performance Metrics

The matrix provides us with many useful metrics that help us to evaluate out classification model. The different measures include: Accuracy, Precision, Recall, and the F-score, explained below.

1. **Accuracy** : accuracy measures how often the model is correct.
2. **Precision** : the amount of information that is conveyed by a value.
3. **Recall** : measures how good the model is at predicting positives.
4. **F-Score** : the "harmonic mean" of precision and recall. It considers both false positive and false negative cases and is good for imbalanced datasets.

5.2. Results

5.2.1. Training and Testing With Diverse Split Data Experiment

In this sub-chapter, we discuss the experiment regarding the parameter trial with predetermined split data. The experiment was carried out 5 times by looking for the best values from the parameters and split data described in the table below.

Table 5.1. First Experiments

Data Split	Batch Size	Learning Rate	Validation Acc	Test Acc
50% data training 50% data testing	64	0.1	80.70%	77.60%
		0.01	96.49%	92.36%
		0.001	98.24%	92.36%
	32	0.1	84.21%	80.55%
		0.01	92.98%	90.10%
		0.001	92.98%	88.88%
	16	0.1	80.70%	79.51%
		0.01	89.47%	87.84%
		0.001	96.49%	90.79%

Table 5.2. Second Experiments

Data Split	Batch Size	Learning Rate	Validation Acc	Test Acc
60% data training 40% data testing	64	0.1	76.81%	85.46%
		0.01	94.20%	92.62%
		0.001	92.75%	92.84%
	32	0.1	39.13%	30.58%
		0.01	92.75%	91.32%
		0.001	94.20%	92.84%
	16	0.1	56.52%	58.78%
		0.01	86.95%	90.88%
		0.001	92.75%	91.75%

Table 5.3. Third Experiments

Data Split	Batch Size	Learning Rate	Validation Acc	Test Acc
	64	0.1	83.74%	81.50%

70% data training 30% data testing		0.01	93.42%	93.35%
		0.001	92.80%	92.77%
	32	0.1	63.61%	21.96%
		0.01	93.18%	93.64%
		0.001	91.81%	94.79%
	16	0.1	60.41%	72.54%
		0.01	93.75%	92.77%
		0.001	93.75%	92.48%

Table 5.4. Fourth Experiments

Data Split	Batch Size	Learning Rate	Validation Acc	Test Acc
80% data training 20% data testing	64	0.1	41.30%	42.85%
		0.01	90.21%	95.67%
		0.001	91.30%	95.23%
	32	0.1	18.47%	19.91%
		0.01	90.21%	93.07%
		0.001	93.47%	93.93%
	16	0.1	71.73%	77.48%
		0.01	89.13%	93.50%
		0.001	91.30%	94.37%

Table 5.5. Fifth Experiments

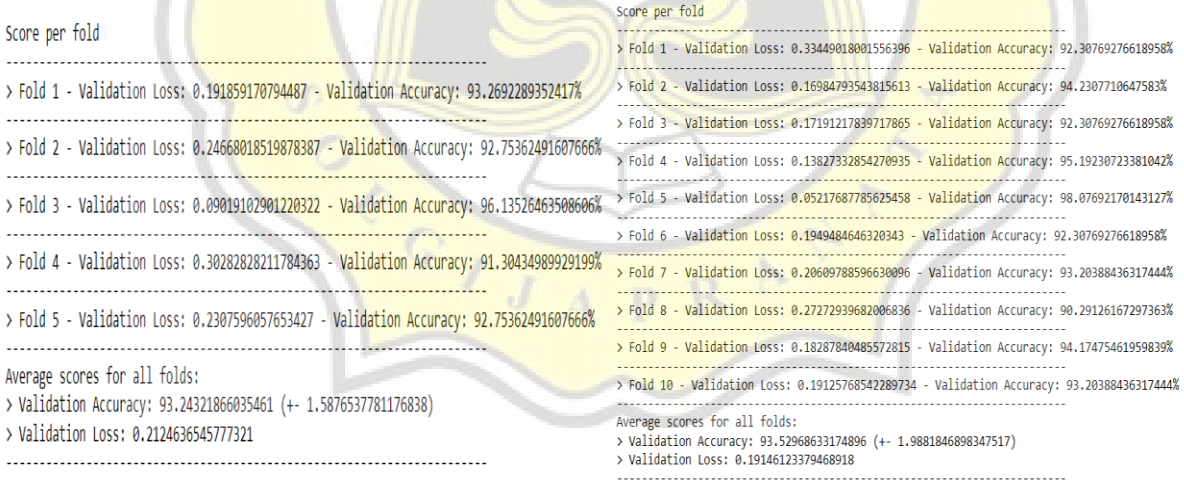
Data Split	Batch Size	Learning Rate	Validation Acc	Test Acc
90% data training 10% data testing	64	0.1	30.09%	34.48%
		0.01	90.29%	95.68%
		0.001	93.20%	94.82%
	32	0.1	77.66%	79.31%
		0.01	85.43%	90.51%

		0.001	93.20%	95.68%
	16	0.1	86.40%	86.20%
		0.01	84.46%	93.10%
		0.001	92.23%	93.10%

In the trial test table, it can be seen that from the five experiments the best results from the parameters used were in batch size 32, learning rate 0.001, with training data of 90% and testing data of 10%. It can be seen from the table above that validation accuracy reaches 93.20% and test accuracy reaches 95.68% which is the highest accuracy compared to other parameters. After this first trial stage and producing the best value, the best results will be used as parameters to conduct other trials so that maximum results are obtained and in accordance with predictions.

5.2.2. K-Fold Cross Validation Experiment

In this sub-chapter, the author cross-validation with numbers of folds, namely 5 and 10 with datasets divided into 10% testing data and 90% training data. Here are the results of KFold Cross Validation with k=5 and k=10.



(a) (b)

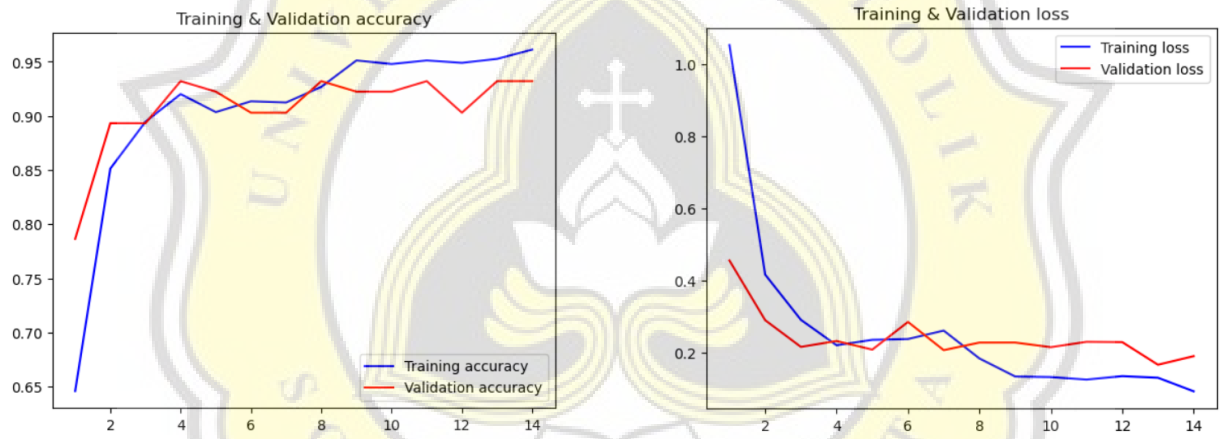
Figure 5.1 : (a) Cross Validation with k=5 and (b) Cross Validation with k=10

In the picture above, the author cross-validates with k = 5 and k = 10, from the results of the experiment above, with k = 5, it produced an average validation accuracy of 93.24% with a

deviation standard of 1.58%, while in k = 10 it produced an average validation accuracy of 93.52% with a deviation standard of 1.98% which means that the greater the deviation standard produced, the distribution of the middle value is also large, and vice versa. The purpose of the deviation standard is to see the distance between the average accuracy and the accuracy of each experiment (iteration). By using K-Fold cross validation also reduces overfitting where when the model overmatches the training data, and the loss continues to decrease while the validation loss does not change, or increases.

5.2.3. Graph of Validation Accuracy and Validation Loss Results

This sub-chapter displays a graph of validation accuracy and validation loss on split data with 90% training data and 10% testing data.



(a) (b)

Figure 5.1 : (a) Graph of Validation Accuracy and (b) Graph of Validation Loss

The pictures above show the graphic results of validation accuracy and validation loss with split data, namely 90% training data and 10% testing data. From the picture above, it can be said that the implemented model produces maximum results because the validation graph shows an increase while on the validation loss graph it decreases. Thus producing a model with good performance.

5.2.4. Confusion Matrix and Performance Metrics

Confusion Matrix is used to represent the predicted results with the actual conditions of the dataset has been trained using the Mobile Net architecture.

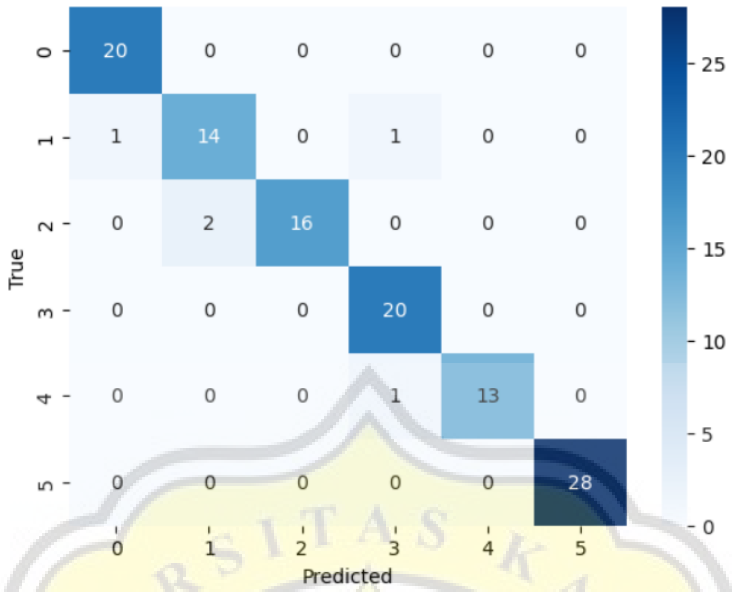


Figure 5.1 : Confusion Matrix Results

From the picture above, it can be seen that on the True Label there is an index 0 to 6 which states that it is a class on the dataset. index 0 is potato early blight, index 1 is potato healthy, index 2 is potato late blight, index 3 is tomato early blight, index 4 is tomato healthy, index 5 is tomato late blight. Based on the confusion matrix in the picture above, it can be said that in index 5, namely tomato late blight, it has the highest prediction of 28 samples, so the prediction results show the appropriate prediction.

The matrix provides us with many useful metrics that help us to evaluate out classification model. The different measures include: Accuracy, Precision, Recall, and the F-score, explained below.

1. **Accuracy** : Accuracy describes how accurately a model can correctly classify. Thus, accuracy is the ratio of correct predictions (positive and negative) to the overall data. In other words, accuracy is the degree of proximity of the predicted value to the actual (actual) value. The accuracy value can be obtained by the equation.

$$\text{Accuracy} \approx \frac{\text{True Positive+True Negative}}{\textit{True Positive+False Positive+True Negative+False Negative}} \quad (1)$$

2. **Precision** : Precision describes the degree of accuracy between the requested data and the predicted results provided by the model. Thus, precision is the ratio of positive true predictions compared to the overall positive predicted results. Of all the positive classes that have been predicted correctly, how much data is really positive. The precision value can be obtained by the equation.

$$\text{Precision} \approx \frac{\text{True Positive}}{\textit{True Positive+False Positive}} \quad (2)$$

3. **Recall** : Recall describes the model's success in reinventing information. Thus, recall is the ratio of positive correct predictions compared to the overall positive true data. The recall value can be obtained by the equation.

$$\text{Recall} \approx \frac{\text{True Positive}}{\textit{True Positive+False Positive}} \quad (3)$$

4. **F1-Score** : this is the harmonic mean of precision and recall and gives a better measure of the incorrectly classified cases than the accuracy metric.

$$\text{F1} - \text{Score} \approx \left(\frac{\textit{Recall}^{-1}+\textit{Precision}^{-1}}{2}\right)^{-1} \approx 2 * \frac{(\text{Precision}*\text{Recall})}{(\textit{Precision+Recall})} \quad (4)$$

With the basis of the Confusion Matrix table, it can then be calculated the Accuracy, Precision, Recall, and F1-Score. The four metrics are very useful for measuring the performance of the classifier or algorithm used to make predictions. Here are the results of the four metrics.

```
Test Accuracy: 95.6896551724138%
F1 score: 0.9567454493244827
Precision: 0.9588744588744589
Recall: 0.9568965517241379
```

Figure 5.2 : Performance Metrics Results