Chittagong University of Engineering and Technology

(Computer Science and Engineering)

Course Title: Digital Image Processing (Sessional)

Course code: CSE- 454

Assignment
on
Optimizing Convolutional Neural Network Architectures for Vegetable Image Classification: A Comprehensive Study

Submitted by

Md. Refaj Hossan

ID: 1904007

Submitted to

Dr. Kaushik Deb
Professor, Department of CSE, CUET

Saadman Sakib
Lecturer, Department of CSE, CUET

Md. Al-Mamun Provath
Lecturer, Department of CSE, CUET

Remarks

Objectives

Using a dataset of vegetable images, this report aims to present an overview of Convolutional Neural Networks (CNNs). Thus, the following categories best describe the primary objectives of this report-

- To comprehend the fundamental ideas behind Convolutional Neural Networks (CNN).
- To apply CNN architecture to a vegetable image dataset for the classification of various types of vegetables.
- To compare different CNN architectures and provide an analysis report on their performance.

Introduction

A Convolutional Neural Network (CNN) is a type of Deep Learning (DL) neural network architecture which can be defined as the extended version of Artificial Neural Network (ANN) and commonly used in Computer Vision (CV), which is a field of Artificial Intelligence (AI) that enables a computer to understand and interpret the images or visual data, consists of several types of layers i.e., Convolutional layer, Pooling layer, Fully Connected (FC) layer. However, in a regular neural network there are three types of layers as follows-

1. **Input Layers:** It is the layer in which the inputs have been given to a model to train. The number of neurons in this layer is same as the total number of features in data.
2. **Hidden Layers:** The input from the input layer is then fed into the hidden layer which can be many in numbers. Each hidden layer can have different numbers of neurons which are generally greater than the number of features. The output from each layer is calculated by matrix multiplication of the output of the previous layer with the learnable weights of that layer and then addition by learnable biases followed by activation functions to incorporate non-linearity functionality in the model.
3. **Output Layer:** The output from the hidden layer is then fed into a logistic function like sigmoid or softmax which converts the output of each class into the probability score of each class.

However, in CNN, the Convolutional layer applies filters to the input images to extract features, the Pooling layer downsamples the image to reduce computation, and the Fully Connected (FC) layer makes the final prediction by using back propagation and gradient descent. Therefore, a simple CNN architecture can be represented as following Fig. 1. Some important points in a Convolutional Neural Networks are as follows-

- Convolutional layers contain a set of learnable filters (kernels) with small widths and heights, matching the depth of the input volume (e.g., 3 for image input).
- For instance, if we need to apply convolution on an image with dimensions 64x64x3, the filters could be sized as 3x3x3 or 5x5x3 or something else, but must be smaller than the image dimensions.

- During the forward pass, each filter is slide across the entire input volume step by step, with each step referred to as a stride (which could be 2, 3, or even 4 for high-dimensional images), computing the dot product between the kernel weights and the corresponding input volume patch.
- As the filters slide, they produce a 2-D output for each filter, which are then stacked together to form an output volume with a depth equal to the number of filters. The network will learn all these filters.

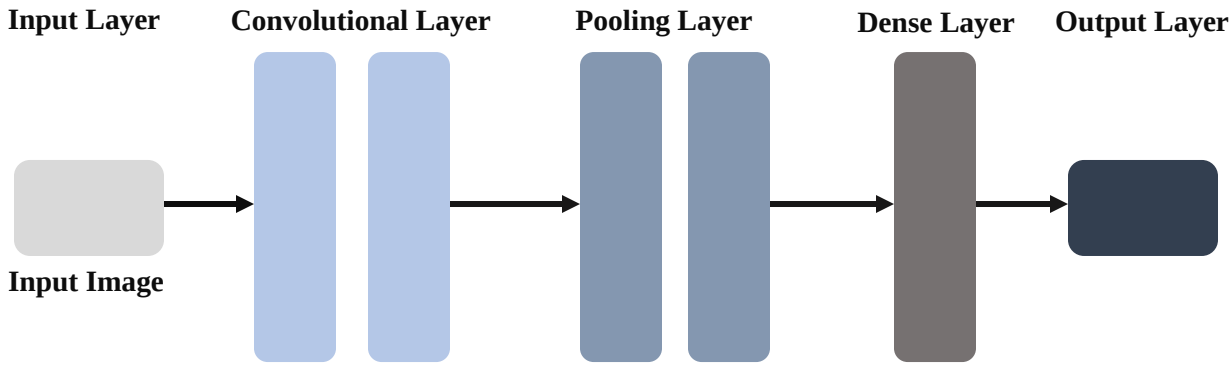

Fig. 1: A simple CNN architecture

Task Description

The main task is to classify 15 types of vegetables from total 21000 vegetables images dataset by using CNN architecture. At first, it is necessary to understand the dataset. A total of 21000 images from 15 classes are used where each class contains 1400 images of size 224×224 and in *.jpg format. The dataset split into 70% for training, 15% for validation, and 15% for testing purpose.

Table 1: Numbers of data in train, validation and test datasets

Class	Train	Validation	Test
Bean	1000	200	200
Bitter gourd	1000	200	200
Bottle gourd	1000	200	200
Brinjal	1000	200	200
Broccoli	1000	200	200
Cabbage	1000	200	200
Capsicum	1000	200	200
Carrot	1000	200	200
Cauliflower	1000	200	200
Cucumber	1000	200	200

Papaya	1000	200	200
Potato	1000	200	200
Pumpkin	1000	200	200
Radish	1000	200	200
Tomato	1000	200	200
Total	**15000**	**3000**	**3000**

Code Explanation

During the process of building a CNN architecture to classify these classes, multiple CNN architectures were tested through trial and error to achieve the best performance. Among these architectures, the best that have been achieved will be explained in this section.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import confusion_matrix, classification_report
import itertools
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Here, this code snippets are used to import all necessary libraries such as *numpy* for array related computation, *matplot* for plotting graphs, *cv2* for doing computer vision related task such as image read and others, *LabelBinarizer* for one hot encoding that is to transform categorical labels to a binary array where each column represents one class, *Conv2D*, *MaxPooling2D* for computing with convolutional and pooling layer respectively and so on.

```
IMG_SIZE = 224
DATA_DIR = '/kaggle/input/vegetable-image-dataset/Vegetable Images/'

def load_dataset(data_dir, img_size):
    images = []
    labels = []
    for folder in os.listdir(data_dir):
        folder_path = os.path.join(data_dir, folder)
        for file in os.listdir(folder_path):
```

```
            file_path = os.path.join(folder_path, file)
            image = cv2.imread(file_path)
            image = cv2.resize(image, (img_size, img_size))
            images.append(image)
            labels.append(folder)
    images = np.array(images, dtype='float32') / 255.0
    labels = np.array(labels)
    return images, labels
```

Here, at first, *IMG_SIZE* sets the size to which all images will be resized (224x224 pixels) and *DATA_DIR* specifies the root directory containing the dataset. Then, the following function *load_dataset* reads all images and their labels from the specified directory. It iterates through each folder (each representing a class) and each file (each representing an image) within those folders. Each image is loaded using OpenCV (cv2), resized to the defined size, and appended to a list. Labels (folder names) are also collected. The function returns two arrays i.e., one for images (normalized to range [0, 1]) and one for labels.

```
train_images, train_labels = load_dataset(os.path.join(DATA_DIR, 'train'),
IMG_SIZE)
val_images, val_labels = load_dataset(os.path.join(DATA_DIR, 'validation'),
IMG_SIZE)
test_images, test_labels = load_dataset(os.path.join(DATA_DIR, 'test'), IMG_SIZE)

# Convert labels to one-hot encoding
lb = LabelBinarizer()
train_labels = lb.fit_transform(train_labels)
val_labels = lb.transform(val_labels)
test_labels = lb.transform(test_labels)
print("Successfully loaded all files!!!")
```

The load_dataset function is called for training, validation, and test sets, specifying the respective subdirectories. Then, labels are converted from categorical string labels to one-hot encoded vectors using *LabelBinarizer*. *fit_transform* is applied to the training labels and transform to the validation and test labels to ensure consistent encoding. A success message is printed after loading and preprocessing all data to ensure that all things loaded successfully.

```
def plot_histogram(images, labels, lb):
    label_names = lb.classes_
    label_counts = np.sum(labels, axis=0)
    plt.figure(figsize=(10, 5))
    plt.bar(label_names, label_counts)
    plt.xlabel('Vegetable')
    plt.ylabel('Count')
    plt.title('Number of images per class')
```

```
    plt.xticks(rotation=90)
    plt.show()

plot_histogram(train_images, train_labels, lb)
```

The above code snippet defines and uses a function to create a histogram that visualizes the number of images per class in the training dataset. This helps in understanding the class distribution, which is important for ensuring that the dataset is balanced or identifying if any class is underrepresented though it has been depicted in Table 1.

```
def plot_radar_chart(labels, lb):
    label_names = lb.classes_
    label_counts = np.sum(labels, axis=0)
    angles = np.linspace(0, 2 * np.pi, len(label_names), endpoint=False).tolist()
    label_counts = np.concatenate((label_counts,[label_counts[0]]))
    angles += angles[:1]

    fig, ax = plt.subplots(figsize=(8, 8), subplot_kw=dict(polar=True))
    ax.fill(angles, label_counts, color='red', alpha=0.25)
    ax.plot(angles, label_counts, color='red', linewidth=2)
    ax.set_yticklabels([])
    ax.set_xticks(angles[:-1])
    ax.set_xticklabels(label_names, fontsize=12)
    plt.show()

plot_radar_chart(train_labels, lb)
```

The above code snippet defines and uses a function to create a radar chart that visualizes the number of images per class in the training dataset. The radar chart provides a clear and immediate visual representation of the data distribution in a circular layout, making it easier to compare the counts across different classes. Here,

- *angles* is used to create a list of angles (in radians) equally spaced around the circle for each class. *np.linspace* is used to generate these angles.
- *label_counts = np.concatenate((label_counts, [label_counts[0]]))* is used to append the first count to the end of the label_counts array to close the loop on the radar chart.
- *angles += angles[:1]*is used to append the first angle to the end of the angles list to close the loop on the radar chart.

The output of this code snippet is as following Fig. 2.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.3,
```

```
    height_shift_range=0.3,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
datagen.fit(train_images)
```

This code snippet sets up an *ImageDataGenerator* from *TensorFlow/Keras* for data augmentation. The class allows for real-time data augmentation during model training. Here, below parameters are used to augment the data during training phase.

1. *rotation_range:* Specifies the range (in degrees) for random rotations applied to the images.
2. *width_shift_range and height_shift_range:* Ranges for randomly shifting the width and height of the images, respectively, expressed as a fraction of total width or height.
3. *shear_range:* Defines the intensity of shearing transformation applied to the images.
4. *zoom_range*: Range for random zooming applied to the images.
5. *horizontal_flip:* Boolean indicating whether to randomly flip images horizontally.
6. *fill_mode:* Strategy for filling in newly created pixels during transformations ('nearest' fills with the nearest pixel value).

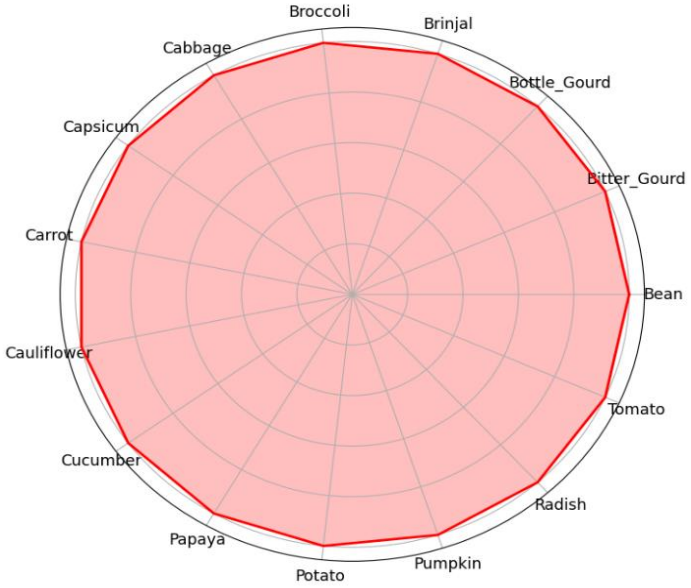

Fig. 2: Radar plot of vegetable image classes

```
def build_custom_cnn(input_shape, num_classes):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))
    return model
```

This code snippet is the core of the CNN architecture which defines necessary Convolutional layers, Pooling layers and Fully Connected layers. Here, *input_shape* specifies the shape of input images ((IMG_SIZE, IMG_SIZE, 3) for RGB images (i.e., image size is 224)), and *num_classes* indicates the number of output classes (i.e., 15).
At first, *model = Sequential()* is used to create a Sequential model. Then, the following steps are performed to build the full architecture.

- Addition of three convolutional layers (Conv2D) with increasing filter depths (32, 64, and 128) and a filter size of (3, 3). Each convolutional layer uses the ReLU activation function (activation='relu') to incorporate non-linear properties.
- After each convolutional layer, a MaxPooling2D layer is added to down-sample the feature maps, reducing spatial dimensions by half (2, 2).
- The Flatten layer converts the 2D matrix of features into a vector, preparing it for the fully connected layers.
- Two fully connected (Dense) layers are added: one with 128 neurons and ReLU activation, and another output layer with *num_classes* (here, it is 15) neurons and softmax activation function for multi-class classification.
- A Dropout layer with a dropout rate of 0.5 is added after the first fully connected layer to prevent overfitting by randomly setting a fraction of input units to zero during training.

```
IMG_SIZE = 224
input_shape = (IMG_SIZE, IMG_SIZE, 3)
num_classes = 15
model = build_custom_cnn(input_shape, num_classes)

model.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])
```

Here, the first portion of the above code snippet initialize the custom CNN model with necessary parameters. After defining this, *model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy', metrics=['accuracy'])* is used to compile configures it for training. Adam is an optimization algorithm often used for training neural networks. Here, it's configured with a learning rate of 0.001 which is the most common value in most of the

architectures. *categorical_crossentropy* is commonly used for multi-class classification problems to specify loss function.

```
callbacks = [
    EarlyStopping(patience=10, restore_best_weights=True),
    ReduceLROnPlateau(patience=5)
]

history = model.fit(
    datagen.flow(train_images, train_labels, batch_size=32),
    validation_data=(val_images, val_labels),
    epochs=50,
    callbacks=callbacks
)
```

Here, the first potion of the code snippet that is *callbacks* are defined and used to train the previously defined CNN model.

- **EarlyStopping:** This callback monitors the model's validation loss and stops training when no improvement is seen for a specified number of epochs (patience=10). The *restore_best_weights=True* argument ensures that the model weights from the epoch with the best monitored quantity (in this case, validation loss) are restored at the end of training.

- **ReduceLROnPlateau:** This callback reduces the learning rate when a metric has stopped improving. Here, patience=5 means that if no improvement is observed in the validation loss after 5 epochs, the learning rate will be reduced.

Later on, *model.fit()* starts the model training with 50 epochs. *datagen.flow(train_images, train_labels, batch_size=32)* is used to generate batches of augmented data during training phase. It's used here with flow method to provide batches of augmented with a batch size of 32. *validation_data=(val_images, val_labels)* specifies the validation data on which the model performance will be evaluated after each epoch. val_images and val_labels are the validation set images and corresponding labels.

```
def plot_confusion_matrix(cm, class_names):
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title('Confusion Matrix')
    plt.show()

predictions = model.predict(test_images)
```

```
pred_labels = np.argmax(predictions, axis=1)
true_labels = np.argmax(test_labels, axis=1)

cm = confusion_matrix(true_labels, pred_labels)
plot_confusion_matrix(cm, lb.classes_)
print(classification_report(true_labels, pred_labels, target_names=lb.classes_))
```

The code predicts labels for test images using a trained model and computes a confusion matrix to evaluate classification performance. It then visualizes the confusion matrix using *plot_confusion_matrix*. Finally, it prints a classification report summarizing precision, recall, F1-score, and support for each class based on the confusion matrix.

Result Analysis

In this section, the results between three different CNN architectures that have been performed will be analyzed. The result we have found for the mentioned code snippets is as following figure Fig. 3.

	precision	recall	f1-score	support
Bean	1.00	0.99	1.00	200
Bitter_Gourd	0.99	0.98	0.99	200
Bottle_Gourd	1.00	0.99	1.00	200
Brinjal	0.99	0.99	0.99	200
Broccoli	0.97	1.00	0.98	200
Cabbage	0.99	0.98	0.99	200
Capsicum	0.99	0.98	0.99	200
Carrot	1.00	1.00	1.00	200
Cauliflower	0.99	0.98	0.98	200
Cucumber	0.99	0.99	0.99	200
Papaya	0.99	0.99	0.99	200
Potato	1.00	1.00	1.00	200
Pumpkin	0.98	0.98	0.98	200
Radish	1.00	0.99	1.00	200
Tomato	0.99	0.99	0.99	200
accuracy			0.99	3000
macro avg	0.99	0.99	0.99	3000
weighted avg	0.99	0.99	0.99	3000

Fig. 3: Classification report of CNN architecture (Method 1)

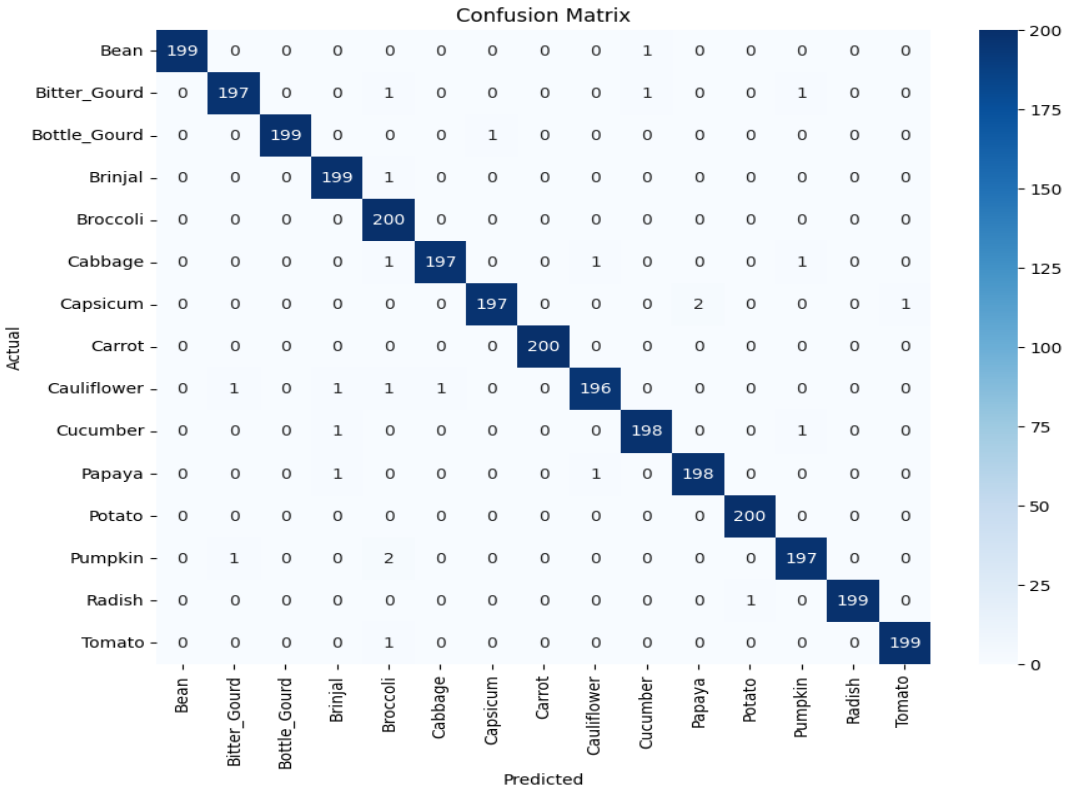

Fig. 4: Confusion matrix for CNN architecture (Method 1)

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 128)	0
flatten (Flatten)	(None, 86528)	0
dense (Dense)	(None, 128)	11,075,712
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 15)	1,935

Fig. 5: Parameters used in CNN architecture (Method 1)

Here, we have observed that approximately 99 percent accuracy has been obtained in this architecture with about 33,512,687 (127.84 MB) parameters where-

- Trainable params: 11,170,895 (42.61 MB)
- Non-trainable params: 0 (0.00 B)
- Optimizer params: 22,341,792 (85.23 MB)

However, instead of using ReLU activation function as activation function, by using Leaky ReLU, we have observed result as following which is much more satisfying and none of the results are overfitted because the training and validation accuracy of the last epoch was approximately same i.e., training accuracy 97.13 percentiles and validation accuracy was 98.97 percentiles.

	precision	recall	f1-score	support
Bean	1.00	0.99	1.00	200
Bitter_Gourd	1.00	0.99	1.00	200
Bottle_Gourd	1.00	1.00	1.00	200
Brinjal	1.00	1.00	1.00	200
Broccoli	1.00	1.00	1.00	200
Cabbage	1.00	1.00	1.00	200
Capsicum	1.00	0.99	1.00	200
Carrot	1.00	1.00	1.00	200
Cauliflower	1.00	0.98	0.99	200
Cucumber	0.99	1.00	1.00	200
Papaya	1.00	1.00	1.00	200
Potato	1.00	1.00	1.00	200
Pumpkin	1.00	1.00	1.00	200
Radish	1.00	1.00	1.00	200
Tomato	1.00	1.00	1.00	200
accuracy			1.00	3000
macro avg	1.00	1.00	1.00	3000
weighted avg	1.00	1.00	1.00	3000

Fig. 6: Classification report of CNN architecture (Method 3)

But, while we have used filters depth in decreasing order (128, 64, 32) (Method 2), the output result is much more dissatisfying which gave us only 7 percentiles accuracy on test set. It can be because of starting with lower filter depths (32) and gradually increasing (32 -> 64 -> 128) allows the network to learn fundamental features first and then build upon them with more complex representations with this dataset with this hyperparameters. But in others dataset, it can be vice versa also.

Discussion

This report offers a thorough examination of theoretical ideas as well as real-world applications of convolutional neural networks (CNNs) using a dataset of vegetable images. The report starts off by outlining the core ideas of CNNs and highlighting how convolutional and pooling layers enable CNNs to learn hierarchical representations of visual data. The implementation of a CNN architecture designed to classify several vegetable varieties from the given dataset follows, showing how convolutional layers collect information like textures and forms that are essential for precise classification. The comparison of several CNN architectures highlights how important architecture selection is for model performance, since different configurations affect computing efficiency and accuracy. Additionally, the process of determining which architecture is superior can be learned by trial and error. Moreover, this report also highlights that starting with lower filter

depths and progressively increasing them can lead to better accuracy by enabling the network to learn foundational features before tackling more complex patterns for this vegetable image dataset. This approach reflects the iterative nature of feature learning in CNNs. Practical issues including model improvement, hyperparameter tuning, and the effect of dataset properties on performance are also covered in this report. The most crucial thing to remember, though, is that the entire training process took a very long time, roughly **2.15 hours, to complete 50 epochs** utilizing a **GPU T4x2** and the required hardware. It is one of the most important insights of the entire experiment, and while it can be minimized by utilizing fewer epochs, the requisite accuracy may not be achieved. All things considered, the report offers a balanced viewpoint on CNNs, demonstrating their effectiveness in vegetable image classification assignments and representing the significance of careful architectural planning and optimization techniques to achieve peak performance.

References:

1. https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset
2. https://ieeexplore.ieee.org/document/9465499/
3. https://www.geeksforgeeks.org/introduction-convolution-neural-network/
4. https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network