Vegetable Images classification using customized Convolutional Neural Network (CNN) model.

Course Code: CSE-453
Course Title: Digital Image Processing
Date of Submission: 6th July, 2024

Submitted By-
Name: Tofayel Ahmmed Babu
ID: 1904005
Level: 4
Term: I
Section: A

Submitted To-
Saadman Sakib
Lecturer, Department of CSE, CUET

Abstract: In this report, we present a custom Convolutional Neural Network (CNN) model designed for the classification of vegetable images. The dataset comprises images of 15 different vegetable classes. This report outlines the dataset characteristics, the architecture of the CNN model, the training process, performance evaluation, and analysis of the results. We also explore various techniques such as dropout, and early stopping to enhance the model's performance and generalization. Detailed code snippets and their explanations are provided to facilitate understanding and reproducibility.

Objectives: Followings are the core objectives of this assignment-

1. Design and implement a custom CNN architecture tailored for classifying vegetable images into 15 distinct classes.
2. Utilize modern deep learning techniques to optimize model performance.
3. Use advanced training techniques, such as learning rate scheduling and early stopping, to optimize the training process and avoid overfitting
4. Assess the model's accuracy on the test dataset to determine its real-world performance.
5. Generate and analyze the confusion matrix to understand the model's classification performance across different vegetable classes.
6. Plot training and validation accuracy and loss over epochs to monitor the model's learning progress. Identify potential issues such as overfitting or underfitting by analyzing these plots.

Introduction: A Convolutional Neural Network (CNN) is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data. When it comes to Machine Learning, Artificial Neural Networks perform really well. Neural Networks are used in various datasets like images, audio, and text. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use Recurrent Neural Networks more precisely an LSTM, similarly for image classification we use Convolution Neural networks. Convolutional Neural Networks (CNNs) extract features from images using convolutional layers, followed by pooling and fully connected layers. In this way they excel in capturing spatial hierarchies and patterns, making them ideal for analyzing visual data. In a regular Neural Network there are three types of layers:

1. Input Layers: It's the layer in which we give input to our model. The number of neurons in this layer is equal to the total number of features in our data (number of pixels in the case of an image).
2. Hidden Layer: The input from the Input layer is then fed into the hidden layer. There can be many hidden layers depending on our model and data size. Each hidden layer can have different numbers of neurons which are generally greater than the number of features. The output from each layer is computed by matrix multiplication of the output of the previous layer with learnable weights of that layer and then by the addition of learnable biases followed by activation function which makes the network nonlinear.
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.

CNN model Architecture: There are two main parts to a CNN architecture

1. A convolution tool that separates and identifies the various features of the image for analysis in a process called as Feature Extraction. The network of feature extraction consists of many pairs of convolutional or pooling layers.
2. A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the features extracted in previous stage. This CNN model of feature extraction aims to reduce the number of features present in a dataset. It creates new features which summarizes the existing features contained in an original set of features.

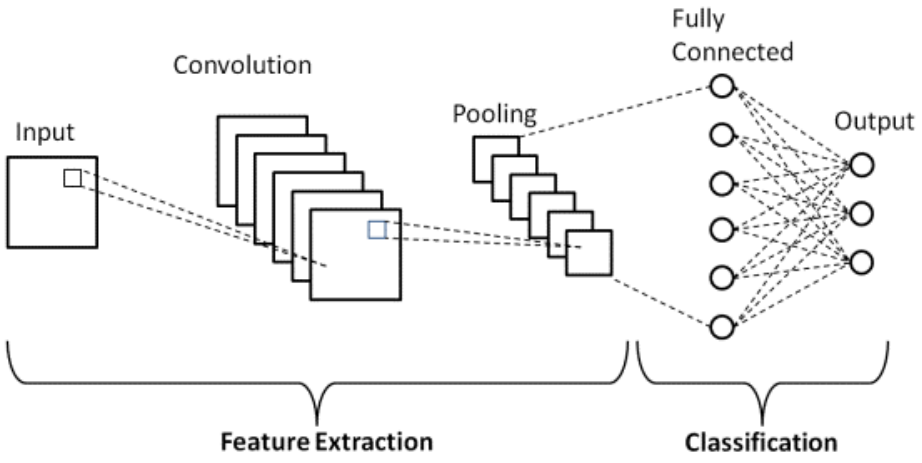

Figure-01: CNN model architecture

In addition to these convolutional, pooling and fully connected layers, there are two more important parameters which are the dropout layer and the activation function which are define below:

Convolutional Layer: This layer is the first layer that is used to extract the various features from the input images. In this layer, the mathematical operation of convolution is performed between the input image and a filter of a particular size MxM. By sliding the filter over the input image, the dot product is taken between the filter and the parts of the input image with respect to the size of the filter (MxM). The output is termed as the Feature map which gives us information about the image such as the corners and edges. Later, this feature map is fed to other layers to learn several other features of the input image. The convolution layer in CNN passes the result to the next layer once applying the convolution operation in the input. Convolutional layers in CNN benefit a lot as they ensure the spatial relationship between the pixels is intact.

Pooling Layer: In most cases, a Convolutional Layer is followed by a Pooling Layer. The primary aim of this layer is to decrease the size of the convolved feature map to reduce the computational costs. This is performed by decreasing the connections between layers and independently operates on each feature map. Depending upon method used, there are several types of pooling operations such as max-pooling, average-pooling. It basically summarizes the features generated by a convolution layer. In max-pooling, the largest element is taken from feature map. Average-pooling calculates the average of the elements in a predefined sized Image section. The Pooling Layer usually serves as a bridge between the Convolutional Layer and the FC Layer.

Fully Connected Layer: The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture. In this, the input image from the previous layers are flattened and fed to the FC layer. The flattened vector then undergoes few more FC layers where the mathematical functions operations usually take place. In this stage, the classification process begins to take place. The reason two layers are connected is that two fully connected layers will perform better than a single connected layer. These layers in CNN reduce the human supervision

Dropout: Usually, when all the features are connected to the FC layer, it can cause overfitting in the training dataset. Overfitting occurs when a particular model works so well on the training data causing a negative impact in the model's performance when used on a new data.

To overcome this problem, a dropout layer is utilized wherein a few neurons are dropped from the neural network during training process resulting in reduced size of the model. On passing a dropout of 0.5, 50% of the nodes are dropped out randomly from the neural network. Dropout

results in improving the performance of a machine learning model as it prevents overfitting by making the network simpler. It drops neurons from the neural networks during training.

Activation Functions: Finally, one of the most important parameters of the CNN model is the activation function. They are used to learn and approximate any kind of continuous and complex relationship between variables of the network. In simple words, it decides which information of the model should fire in the forward direction and which ones should not at the end of the network. It adds non-linearity to the network. There are several commonly used activation functions such as the ReLU, Softmax, tanH and the Sigmoid functions. Each of these functions have a specific usage. For a binary classification CNN model, sigmoid and softmax functions are preferred. For a multi-class classification, generally softmax is used. In simple terms, activation functions in a CNN model determine whether a neuron should be activated or not.

Dataset Analysis: The dataset used in this study is the Vegetable Image Dataset, which includes images of 15 different classes of vegetables. The dataset is divided into three subsets: training, validation, and test sets. Each subset contains images belonging to all 15 classes. The data is organized in the following directories:

- train/: Contains training images.
- validation/: Contains validation images.
- test/: Contains test images.

Each directory contains subdirectories corresponding to the 15 vegetable classes. There is total 21000 images and the train, validation and test data ratio are 5:1:1. Data generator snippet is included below-

```
Found 15000 images belonging to 15 classes.
Found 3000 images belonging to 15 classes.
Found 3000 images belonging to 15 classes.
```

To ensure our model is trained on a balanced dataset, we analyzed the distribution of images across different classes in our training, validation and test dataset. Number of images per class-

```
Training dataset:
Broccoli: 1000
Capsicum: 1000
Bottle_Gourd: 1000
Radish: 1000
Tomato: 1000
Brinjal: 1000
Pumpkin: 1000
Carrot: 1000
Papaya: 1000
Cabbage: 1000
Bitter_Gourd: 1000
Cauliflower: 1000
Bean: 1000
Cucumber: 1000
Potato: 1000
```

```
Validation dataset:
Broccoli: 200
Capsicum: 200
Bottle_Gourd: 200
Radish: 200
Tomato: 200
Brinjal: 200
Pumpkin: 200
Carrot: 200
Papaya: 200
Cabbage: 200
Bitter_Gourd: 200
Cauliflower: 200
Bean: 200
Cucumber: 200
Potato: 200
```

```
Test dataset:
Broccoli: 200
Capsicum: 200
Bottle_Gourd: 200
Radish: 200
Tomato: 200
Brinjal: 200
Pumpkin: 200
Carrot: 200
Papaya: 200
Cabbage: 200
Bitter_Gourd: 200
Cauliflower: 200
Bean: 200
Cucumber: 200
Potato: 200
```

Fig-02: Dataset split ratio into train test and validation

Figure-03 provide an overview of the frequency of images on the train, validation and test dataset.

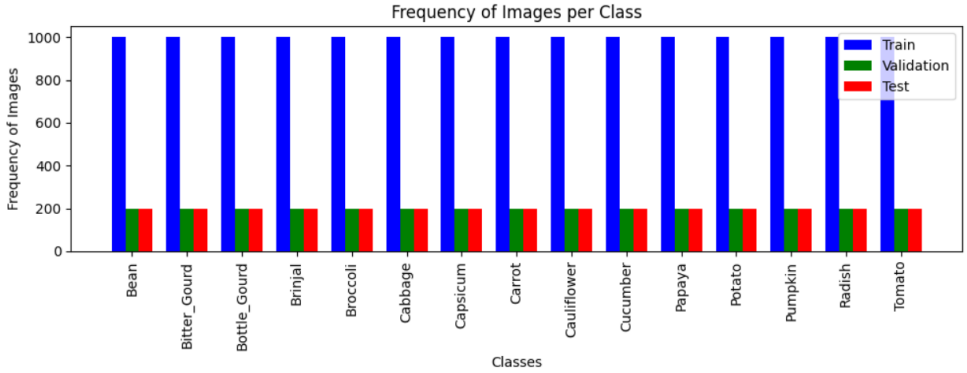

Figure-03: Frequency of images per class

Data visualization: For visualizing the data, we loaded and displayed images from the dataset using matplotlib.

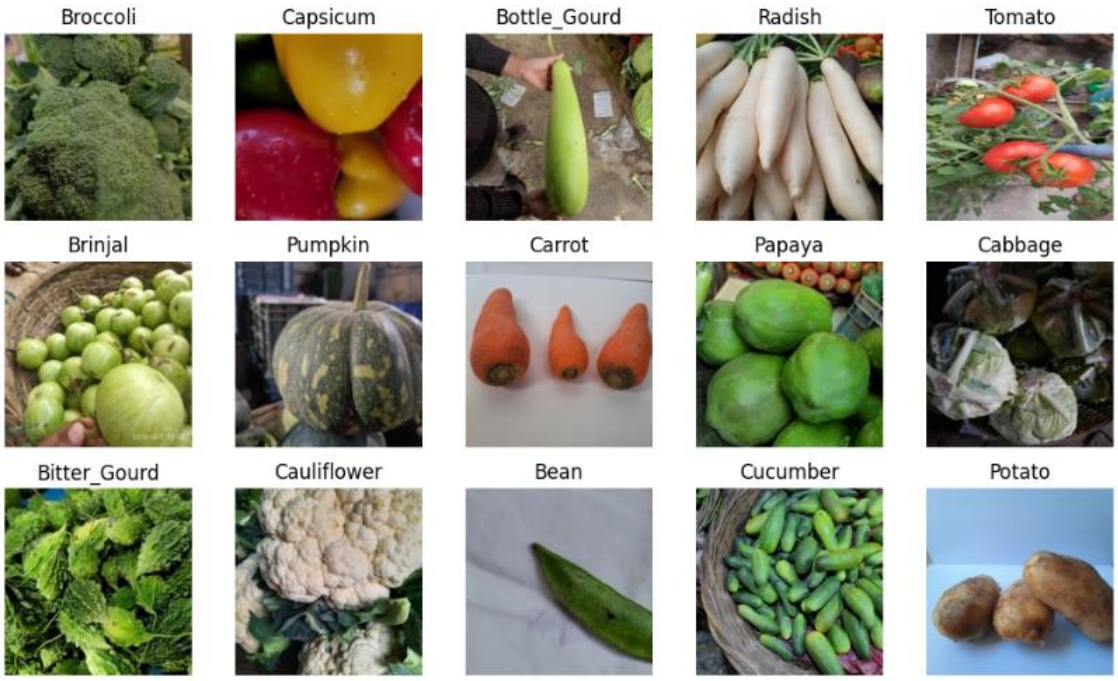

Figure-04: The first image of each class from the train dataset

In Figure-02, we have shown the first images of each class from the train dataset. Corresponding label of the class is added above of the images.

Code Explanation:

Model Architecture:

1. Input Layer:

- **Conv2D (32 filters, kernel size 5x5, ReLU activation):** Input Shape: (224, 224, 3). This layer convolves the input image with 32 filters (kernels) each of size 5x5. And relu activation function is applied to introduce non-linearity.
- **MaxPooling2D (pool size 2x2):** It reduces the spatial dimensions (width and height) by a factor of 2 using max pooling. Max-pooling Helps in reducing computational complexity and controlling overfitting.

2. Intermediate Layers:

- **Conv2D (64 filters, kernel size 5x5, ReLU activation):** Another convolutional layer to capture more complex patterns. Increases the depth of feature maps to 64.
- **MaxPooling2D (pool size 2x2):** Further reduces the spatial dimensions.
- **Conv2D (128 filters, kernel size 3x3, ReLU activation):** Continues to deepen the feature maps with 128 filters. Uses smaller kernel size for finer feature extraction.
- **MaxPooling2D (pool size 2x2):** Downsamples the feature maps.
- **Conv2D (256 filters, kernel size 3x3, ReLU activation):** Further increases the depth with 256 filters. Captures even more complex and abstract features.
- **MaxPooling2D (pool size 2x2):** Downsamples the feature maps to a manageable size.

3. Fully Connected Layers:

- **Flatten:** Converts the 3D feature maps into a 1D vector. It prepares the data for input into the fully connected layers.
- **Dense (512 neurons, ReLU activation):** Fully connected layer designed with 512 neurons. And Applied ReLU activation to introduce non-linearity.
- **Dropout (rate=0.5):** 50% dropouts of the neuron used in dense layer to prevents overfitting by forcing the network to learn redundant representations.
- **Dense (15 neurons, softmax activation):** Designed output layer for classification with 15 classes and used softmax activation to output probabilities for each class.

Code snippet:

```
model = Sequential([
    Conv2D(32, (5, 5), activation='relu', input_shape=(224, 224, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (5, 5), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(256, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(15, activation='softmax')
])
```

Following hyperparameters also used for the better performance of the model during training.

Optimizer (Adam): Learning Rate: 10^{-4} controls the step size during optimization. Adam optimizer adapts the learning rate during training, accelerating convergence.

Loss Function (categorical_crossentropy): Used for multi-class classification, computes the cross-entropy loss between predicted and actual class distributions.

Callbacks: EarlyStopping: Halts training if validation accuracy doesn't improve after a specified number of epochs (patience = 10).

ReduceLROnPlateau: Reduces learning rate by a factor of 0.2 if validation loss plateaus (patience = 5).

Epochs: 50 epochs are used to train the model.

model.summary():

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 220, 220, 32)	2,432
max_pooling2d (MaxPooling2D)	(None, 110, 110, 32)	0
conv2d_1 (Conv2D)	(None, 106, 106, 64)	51,264
max_pooling2d_1 (MaxPooling2D)	(None, 53, 53, 64)	0
conv2d_2 (Conv2D)	(None, 51, 51, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 128)	0
conv2d_3 (Conv2D)	(None, 23, 23, 256)	295,168
max_pooling2d_3 (MaxPooling2D)	(None, 11, 11, 256)	0
flatten (Flatten)	(None, 30976)	0
dense (Dense)	(None, 512)	15,860,224
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 15)	7,695

Total params: 16,290,639 (62.14 MB)
Trainable params: 16,290,639 (62.14 MB)
Non-trainable params: 0 (0.00 B)

Figure-05: Model summary (total number of trainable parameters)

Model Evaluation and Result Analysis: At-first we trained the model using the training data. Following is the code snippet.

```
# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    epochs=50,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size,
    callbacks=[checkpoint, early_stopping, reduce_lr, early_stopping_alert]
)
```

```
Epoch 15/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 33s 70ms/step - accuracy: 0.9561 - loss: 0.1365 - val_accuracy: 0.9677 - val_loss: 0.1354 - learning_rate: 1.0000e-04
Epoch 16/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 0s 76us/step - accuracy: 1.0000 - loss: 0.0783 - val_accuracy: 0.9583 - val_loss: 0.2457 - learning_rate: 1.0000e-04
Epoch 17/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 33s 70ms/step - accuracy: 0.9671 - loss: 0.1067 - val_accuracy: 0.9684 - val_loss: 0.1252 - learning_rate: 1.0000e-04
Epoch 18/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 0s 61us/step - accuracy: 0.9688 - loss: 0.1273 - val_accuracy: 0.9583 - val_loss: 0.0560 - learning_rate: 2.0000e-05
Epoch 19/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 34s 72ms/step - accuracy: 0.9835 - loss: 0.0570 - val_accuracy: 0.9778 - val_loss: 0.0990 - learning_rate: 2.0000e-05
Epoch 20/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 0s 69us/step - accuracy: 1.0000 - loss: 0.0088 - val_accuracy: 0.9583 - val_loss: 0.0763 - learning_rate: 2.0000e-05
Epoch 21/50
468/468 ━━━━━━━━━━━━━━━━━━━━ 34s 71ms/step - accuracy: 0.9888 - loss: 0.0414 - val_accuracy: 0.9782 - val_loss: 0.1008 - learning_rate: 2.0000e-05
Epoch 22/50
  1/468 ━━━━━━━━━━━━━━━━━━━━ 32s 69ms/step - accuracy: 1.0000 - loss: 0.0416
Early stopping triggered at epoch 22
468/468 ━━━━━━━━━━━━━━━━━━━━ 0s 69us/step - accuracy: 1.0000 - loss: 0.0416 - val_accuracy: 0.9583 - val_loss: 0.1957 - learning_rate: 2.0000e-05
```

Figure-06: Training History

Concern about overfitting of the model: If a model performs well on the training set but poorly on the validation set, it's likely overfitting. Learning curves i.e. plotting the model's performance on both the training and validation sets over time, one can determine the overfitting of the model. If the two curves start to diverge, it's a sign of overfitting. To remove overfitting problem, we used several techniques like early stopping and reduced learning rate.

```
Epoch 22/50
  1/468 ━━━━━━━━━━━━━━━━━━━━ 32s 69ms/step - accuracy: 1.0000 - loss: 0.0416
Early stopping triggered at epoch 22
468/468 ━━━━━━━━━━━━━━━━━━━━ 0s 69us/step - accuracy: 1.0000 - loss: 0.0416 - val_accuracy: 0.9583 - val_loss: 0.1957 - learning_rate: 2.0000e-05
```

Figure-07: Early stopping and unnecessary epochs skip

Here, we have seen among 50 epochs, after the 22 epochs the training stopped, because the validation accuracy did not increase significantly. If we visualize the training accuracy vs validation accuracy, we have seen that the two curves are shows similar pattern. Analyzing the curve we can say, we have tried to overcome overfitting problem and can expect a better prediction from the model.

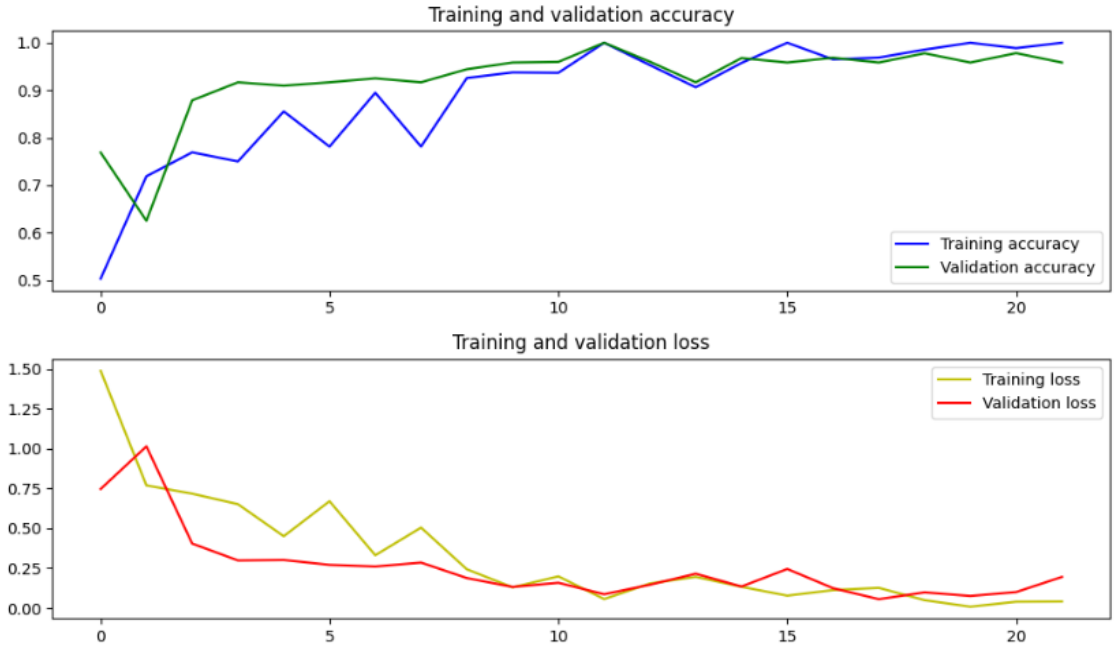

Figure-08: Training accuracy and loss vs validation accuracy and loss

Confusion matrix and classification report: Using the following code snippet we predict on the test data, and then print the confusion matrix and classification report.

```
# Predict on test data
predictions = model.predict(test_generator)
y_pred = np.argmax(predictions, axis=1)
y_true = test_generator.classes
```

```
# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_labels, yticklabels=class_labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix')
plt.show()
```

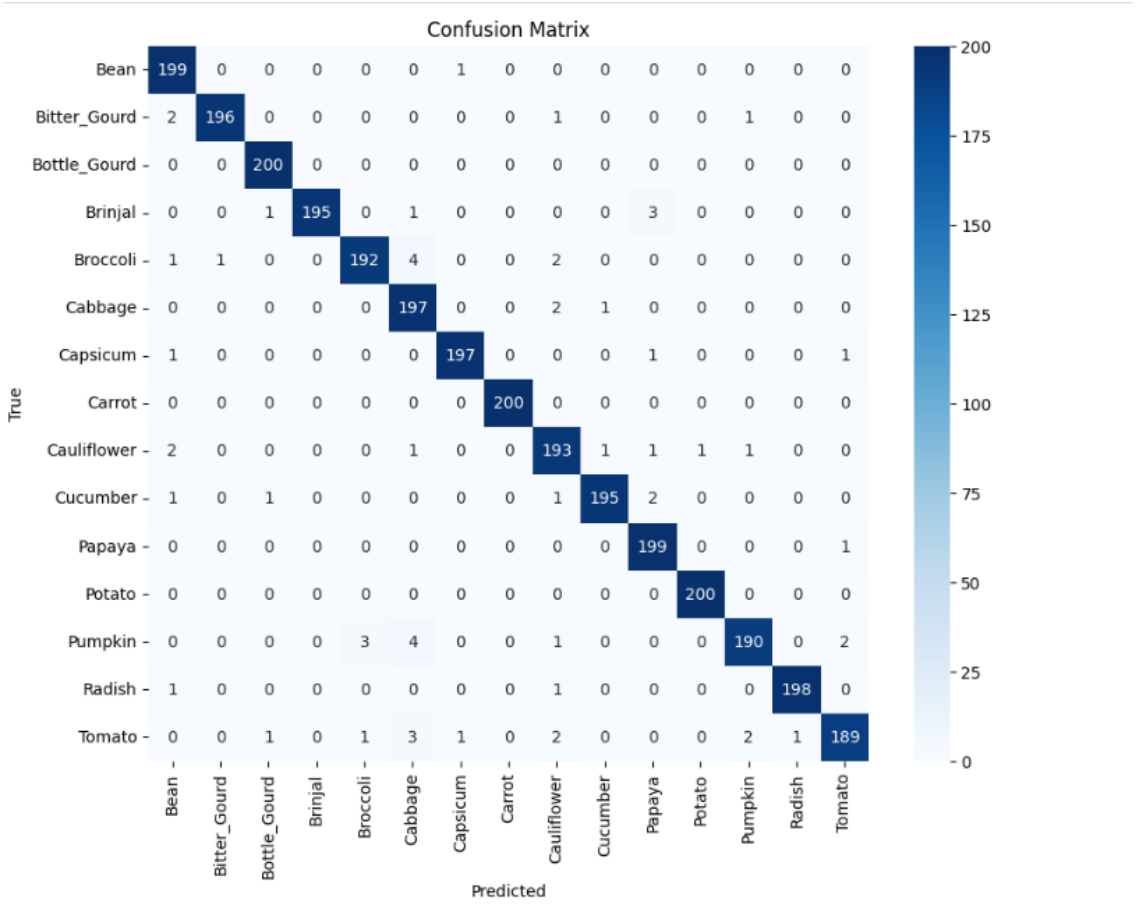

Figure-09: Confusion matrix

According to the confusion matrix analysis for 15 classes, the model's performance can be detailed as follows: The "Bean" class had 199 correct predictions (true positives), while the "Bitter Gourd" class saw 196 correct predictions. The model accurately identified 200 instances for the "Bottle Gourd" class and 195 instances for the "Brinjal" class. For "Broccoli," there were 192 correct predictions, and "Cabbage" had 197 true positives. Similarly, the model correctly predicted 197 instances of the "Capsicum" class and 200 instances of the "Carrot"

class. The "Cauliflower" class had 193 true positives, while the "Cucumber" class had 195. For the "Papaya" class, there were 199 correct predictions, and the "Potato" class had 200. The model accurately identified 192 instances of the "Pumpkin" class and 190 instances of the "Radish" class. Lastly, the "Tomato" class had 189 correct predictions. This breakdown highlights the model's effectiveness in predicting each class correctly. According to this heatmap we printed the classification report and got the following result-

```
# Classification report
report = classification_report(y_true, y_pred, target_names=class_labels)
print(report)
```

	precision	recall	f1-score	support
Bean	0.96	0.99	0.98	200
Bitter_Gourd	0.99	0.98	0.99	200
Bottle_Gourd	0.99	1.00	0.99	200
Brinjal	1.00	0.97	0.99	200
Broccoli	0.98	0.96	0.97	200
Cabbage	0.94	0.98	0.96	200
Capsicum	0.99	0.98	0.99	200
Carrot	1.00	1.00	1.00	200
Cauliflower	0.95	0.96	0.96	200
Cucumber	0.99	0.97	0.98	200
Papaya	0.97	0.99	0.98	200
Potato	1.00	1.00	1.00	200
Pumpkin	0.98	0.95	0.96	200
Radish	0.99	0.99	0.99	200
Tomato	0.98	0.94	0.96	200
accuracy			0.98	3000
macro avg	0.98	0.98	0.98	3000
weighted avg	0.98	0.98	0.98	3000

Figure-10: Classification report of our customized model

We have seen there, the model accuracy is 98%(approx.).

```
test_loss, test_acc = model.evaluate(test_generator, steps=test_generator.samples // test_generator.batch_size)
print(f'Test accuracy: {test_acc*100:.2f}%')
```

```
93/93 ━━━━━━━━━━━━━━━━━━━━ 5s 53ms/step - accuracy: 0.9856 - loss: 0.0692
Test accuracy: 97.98%
```

Figure-11: Model evaluation

Discussion: The model's architecture, with its sequential stacking of convolutional and pooling layers, enabled effective feature extraction and representation. Utilizing a diverse set of convolutional filters allowed the model to capture intricate patterns and details within the images. The Adam optimizer, with its adaptive learning rate, facilitated efficient and rapid convergence during training. Overall, this customized CNN model proved to be a reliable and

high-performing solution for the vegetable image classification task, achieving near-expert level accuracy(98% approx.).

References:

1. https://www.geeksforgeeks.org/introduction-convolution-neural-network/
2. https://www.upgrad.com/blog/basic-cnn-architecture/
3. https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset
4. https://poloclub.github.io/cnn-explainer/