SCHOOL OF COMPUTER SCIENCE

September 2024 Semester

Computer Vision and Natural Language Processing

(ITS69204)

Group Assignment (30%)

Due Date: 30 Nov 2024, 11.59 PM

Submission Channel: myTIMeS

Fruits and Vegetable Image Recognition

Dataset:

https://www.kaggle.com/datasets/kritikseth/fruit-and-vegetable-image-recognition/data

Member Names	Student ID
Hans Andre (Leader)	0354227
Mohamud Abdi	0360301
Sin Vee Young	0353411
Choi Chee Kin	0359936

Introduction

The accurate recognition of images is a crucial application of Computer Vision, with several applications in many industries. For our topic, we have chosen a Fruits and Vegetable Image Recognition program. This has many applications in sectors such as agriculture, retail, and nutrition. Produce can be identified and categorized by automated systems in agriculture to help with sorting and quality control procedures. Similarly, by precisely recognizing products without the use of barcodes or human input, these systems can enhance inventory management in the retail industry. In addition to improving operational effectiveness, these applications lay the foundation for developing more intelligent, interactive systems for everyday tasks. The development of reliable object recognition models has grown more possible with the widespread use of digital photos and developments in artificial intelligence.

Our goal in this project is to use Convolutional Neural Networks (CNNs) to categorize fruit and vegetable images from the Fruit and Vegetable Image Recognition dataset, which is made publicly available on Kaggle. This dataset offers plenty of data for deep learning model training because it includes a variety of fruit and vegetable photos taken in various settings and from various angles. The project uses this dataset to investigate CNNs' potential for correctly identifying, classifying and categorizing a wide range of produce.

The objective of this project is to develop a deep learning model that can accurately identify fruits and vegetables from images. In order to classify the images into the respective categories, this will involve preprocessing the dataset, designing and implementing a CNN architecture, and training the model. Additionally, we intend to assess the model's performance using common metrics like accuracy, precision, recall, and F1-score in order evaluate the model's effectiveness and potential limits.

Methodology

Architecture design:

Input Layer

- Input Shape: (64, 64, 3)
    - The model accepts RGB images resized to 64x64 pixels as input.

Convolutional and Pooling Layers

Block 1: This block extracts low-level features such as edges and corners.

- Conv2D Layer 1: 32 filters, 3x3 kernel, ReLU activation.
- Conv2D Layer 2: 32 filters, 3x3 kernel, ReLU activation.
- MaxPooling2D: Pool size of 2x2 with a stride of 2.

Block 2: This block captures more complex features and spatial patterns.

- Conv2D Layer 3: 64 filters, 3x3 kernel, ReLU activation.
- Conv2D Layer 4: 64 filters, 3x3 kernel, ReLU activation.
- MaxPooling2D: Pool size of 2x2 with a stride of 2.

Flattening Layer

Flatten Layer: This is necessary to pass data to fully connected layers.

- Converts the 2D feature maps from the convolutional layers into a 1D vector.

Fully Connected (Dense) Layers

Dense Layer 1:

- 512 units, ReLU activation.
- Captures high-level abstractions and relationships in the data.

Dense Layer 2:

- 256 units, ReLU activation.
- Further processes the features and prepares them for classification.

Dropout Layer:

- Dropout rate of 0.5.
- Randomly drops neurons during training to prevent overfitting.

Output Layer

Dense Layer 3 (Output Layer):

- 36 units (one for each class of fruits/vegetables).
- Softmax activation to produce probability distributions across the 36 classes.

Summary of CNN model :

Layer Type	Details
Input Layer	Input shape: (64, 64, 3)
Conv2D + Conv2D	32 filters, 3x3 kernel, ReLU activation (2 layers)
MaxPooling2D	Pool size: 2x2, stride: 2
Conv2D + Conv2D	64 filters, 3x3 kernel, ReLU activation (2 layers)
MaxPooling2D	Pool size: 2x2, stride: 2
Flatten	Converts 2D feature maps to 1D vector
Dense	512 units, ReLU activation
Dense	256 units, ReLU activation
Dropout	Dropout rate: 0.5
Output Dense	36 units, Softmax activation

Design Justification

Dataset Characteristics

Dataset Size:

- The dataset contains 36 distinct classes of fruits and vegetables, with approximately 10 images per class.
- While the dataset offers diversity, its small size poses challenges such as potential overfitting and difficulty in generalizing the model to unseen data.

Image Characteristics:

- Images are RGB and vary in background, lighting, and orientation, introducing complexity in recognizing objects consistently.
- The dataset is multi-class, requiring the model to classify each image into one of the 36 categories.

1. Input Shape: (64, 64, 3):
    - Images are resized to 64x64 pixels to balance computational efficiency and retain key features.
    - The RGB colour mode ensures all colour information is preserved, critical for distinguishing between similar-looking products.
    -
2. Convolutional Layers:
    - Two convolutional layers per block (with 32 and 64 filters) progressively extract low to high-level features such as edges, textures, and complex patterns.

    Reasoning: With only ~10 images per class, deeper networks risk overfitting due to limited data. By using 4 convolutional layers in total, the architecture is deep enough to extract key features while avoiding overparameterization.

3. Pooling Layers:
    - Max-pooling layers reduce spatial dimensions (by 2x2) and retain the most critical features while discarding redundant information.

    Reasoning: Pooling combats the limited dataset size by reducing computational requirements and helps prevent overfitting.

4. Fully Connected (Dense) Layers:
    - A dense layer with 512 units and another with 256 units process the extracted features into a form suitable for classification.

    Reasoning: These layers ensure the model captures complex relationships between features while progressively reducing dimensionality.

5. Dropout Regularization:
    - A dropout rate of 0.5 is applied before the output layer to randomly deactivate neurons during training.

    Reasoning: With a small dataset, overfitting is a major risk. Dropout mitigates this by forcing the model to learn more robust features.

6. Output Layer (Softmax Activation):
    - The output layer contains 36 units, corresponding to the number of classes, with softmax activation for multi class classification.

    Reasoning: Softmax is the ideal choice as it outputs probabilities across all 36 classes, ensuring the model identifies the most likely class.

7. Choice of Filters and Layers:
    - The model starts with 32 filters and progresses to 64 filters to handle the increasing complexity of feature extraction.

    Reasoning: A small dataset does not justify the use of very deep architectures with hundreds of filters, as they would lead to overfitting without significant performance improvement.

Implementation

The model used in the project is a convolutional neural network (CNN), modified based on the VGG16 algorithm for fruit and vegetable recognition. Some key aspects of deep learning implementation discussed in the work include proper preprocessing of data, fine tuning of the models,evaluation of performance metrics, and detailed visualization of results. Here is a breakdown of how the project was implemented, the framework and methodologies used.

Code Documentation

This file is used for developing a deep learning model that can accurately identify fruits and vegetables from images. It includes several major steps needed in order to make the model a success. For instance, data preprocessing, model creation, compiling, and training, and also the visualization of training results. This code includes a few key steps:

1. Setup:
    - Imported libraries such as TensorFlow, NumPy and Matplotlib
    - Mounting Google Drive to access the dataset
    - Unzipping the dataset containing images of fruits and vegetables

2. Data Preprocessing:
    - Use ImageDataGenerator to load and preprocess image
    - Apply data augmentation such as resizing to the training data
    - Create data generators for training, validation, and testing sets

3. Model Building:
    - Load a pre-trained VGG15 model as a base
    - Add custom layers on top of VGG16 to adapt it to the fruit and vegetable classification task
    - Freeze initial layers of VGG16 to leverage pre-trained knowledge

4. Model Training:
    - Compiling the model with an optimizer, loss function, and metrics
    - Train the model on the training data, using the validation data to monitor performance
    - Fine-tune the model by unfreezing some layers of VGG16 and training further

5. Model Evaluation:
    - Evaluate the model on the test data to assess its performance on unseen images
    - Generate a classification report and a confusion matrix to analyze the results

6. Visualization:
    - Plot training and validation accuracy and loss curves
    - Visualize the confusion matrix to see which class the model confuses
    - Display predictions on sample images to get a qualitative sense of the model's performance

Setup

We base our project on TensorFlow, an open-source software library for dataflow and different types of deep learning neural networks, as well as several tools like Matplotlib, Seaborn, Plotly for visualization and Scikit-learn for metric evaluations. Google Colab is used to execute the code, as we imported the code from google drive .

Code Execution Setup

The setup begins by installing essential libraries. Libraries such as TensorFlow and Matplotlib are installed in the environment.

```
[ ]  !pip install tensorflow

[ ]  !pip install matplotlib
```

TensorFlow: Used for deep learning model creation, training, and evaluation.

Matplotlib: Utilized for data visualization and analysis.

Importing Necessary Libraries

```
# Import libraries
import os
import cv2
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import classification_report, confusion_matrix
```

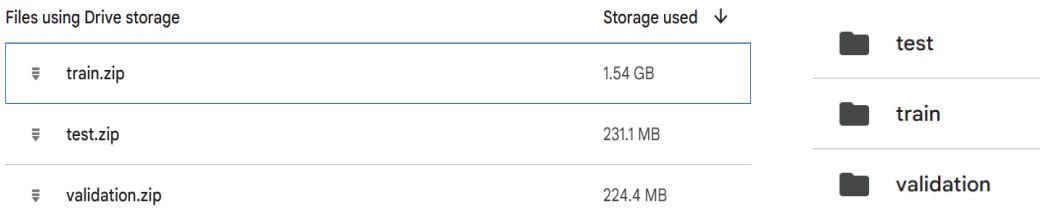
Imports essential python libraries for data preprocessing, model building, training, evaluation, and visualization.

- VGG16: Pre-trained deep learning model on ImageNet data, used for transfer learning to boost classification accuracy.
- TensorFlow: Used for building, training, and evaluating deep learning models.
- Matplotlib: For data visualization.
- Seaborn: For creating informative and attractive visualizations.
- Plotly: For creating interactive plots, particularly useful for plotting confusion matrices and performance metrics.
- OpenCV & Numpy: For image processing tasks.

Dataset Extraction

Datasets are stored in compressed .zip files on Google Drive. The files (train.zip , test.zip ,validation.zip) are extracted into corresponding directories for easier accessibility.

```
# Unzipping datasets
# Changed the extraction path to a directory
with zipfile.ZipFile(train_zip, 'r') as zip_ref:
    zip_ref.extractall('/content/drive/MyDrive/train') # Extracting to a directory named 'train'

with zipfile.ZipFile(test_zip, 'r') as zip_ref:
    zip_ref.extractall('/content/drive/MyDrive/test') # Extracting to a directory named 'test'

with zipfile.ZipFile(val_zip, 'r') as zip_ref:
    zip_ref.extractall('/content/drive/MyDrive/validation') # Extracting to a directory named 'validation'
```



This step organizes the datasets into train, test, and validation directories for streamlined processing.

Path Configuration

Paths are explicitly defined to access the datasets

```
# Updated paths to extracted directories
train_path = '/content/drive/MyDrive/train' # Path to the extracted 'train' directory
test_path = '/content/drive/MyDrive/test' # Path to the extracted 'test' directory
validation_path = '/content/drive/MyDrive/validation' # Path to the extracted 'validation' directory
```

Data Preprocessing

Image Data Augmentation

Data augmentation is implemented to increase the variability of training samples and improve model generalization, this is done to prevent overfitting and enhance model performance.

```
# Data preprocessing with ImageDataGenerator
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

test_datagen = ImageDataGenerator(rescale=1./255)
```

Data Generators

Image generators are defined for the training, validation, and test sets. Converts images into batches for feeding into the model.

```
# Create data generators
train_generator = train_datagen.flow_from_directory(
    train_path,
    target_size=(128, 128),
    batch_size=16,
    class_mode='categorical'
)

validation_generator = test_datagen.flow_from_directory(
    validation_path,
    target_size=(128, 128),
    batch_size=32,
    class_mode='categorical'
)

test_generator = test_datagen.flow_from_directory(
    test_path,
    target_size=(128, 128),
    batch_size=32,
    class_mode='categorical',
    shuffle=False  # Important for consistent evaluation
)
```

- train_generator: Prepares augmented training data.
- validation_generator: Prepares validation data for performance monitoring.
- test_generator: Prepares testing data for evaluation.
- target_size ensures all images are resized to 128x128 for compatibility with the model.
- class_mode specifies a categorical problem with multiple output classes.

```
Found 3115 images belonging to 36 classes.
Found 351 images belonging to 36 classes.
Found 359 images belonging to 36 classes.
```

Model Building

Base Model: VGG16

A pre-trained VGG16 model is used as the base

```
# Load VGG16 model (pre-trained on ImageNet) and fine-tune
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
```

Pre-trained Weights: The model leverages pre-trained ImageNet weights for feature extraction.

Frozen Layers: Initially, the layers of VGG16 are frozen to prevent updates during training

```
base_model.trainable = False  # Freeze base model layers
```

Custom Layers

Additional layers are added on top of VGG16 for classification

```
# Build the model
model = Sequential([
    base_model,
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(train_generator.num_classes, activation='softmax')  # Number of classes
])
```

- Flatten Layer: Converts the multidimensional output of the base model to a 1D vector.
- Dense Layers: Includes a hidden layer (512 neurons) and an output layer with a softmax activation for multiclass classification.softmax
- Dropout Layer: Reduces overfitting by randomly disabling 50% of neurons during training.

Model Training

The model is compiled using the Adam optimizer, categorical cross-entropy loss function and uses accuracy as the evaluation metric.

```
# Compile the model
model.compile(optimizer=Adam(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

- Optimizer: Adam with a learning rate of 1×10−41 \times 10^{-4}1×10−4.
- Loss Function: Categorical cross-entropy for multi-class classification.
- Metric: Accuracy to monitor model performance.

Initial Training

The model is trained for 25 epochs:

```
# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=50,
    epochs=25,
    validation_data=validation_generator
)
```

```
Epoch 25/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 297s 5s/step - accuracy: 0.5161 - loss: 1.7703 - val_accuracy: 0.7778 - val_loss: 0.9183
```

Output after 25 epochs:

- Accuracy: 51.61%
- Validation Accuracy: 77.78%

Fine-tuning

To enhance performance, the base model is fine-tuned by unfreezing some layers

```
# Fine-tune the model (unfreeze some layers in the base model)
base_model.trainable = True
for layer in base_model.layers[:15]:  # Freeze the first 15 layers
    layer.trainable = False
```

The model is retrained for 10 epochs with a lower learning rate:

```
model.compile(optimizer=Adam(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Output:

- Test Accuracy: 89.42%.

Model Evaluation

A detailed classification report provides precision, recall, and F1-score for each class:

```
# Evaluate the model
loss, accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {accuracy*100:.2f}%")

# Classification report and confusion matrix
Y_pred = model.predict(test_generator)
y_pred = np.argmax(Y_pred, axis=1)

print('Classification Report')
print(classification_report(test_generator.classes, y_pred, target_names=test_generator.class_indices.keys()))
```

Visualization

The confusion matrix visualizes classification performance:

```
# Confusion matrix
cm = confusion_matrix(test_generator.classes, y_pred)

# Plot confusion matrix with annotations
cm_fig = px.imshow(cm,
                   labels=dict(x="Predicted", y="True", color="Count"),
                   x=list(test_generator.class_indices.keys()),
                   y=list(test_generator.class_indices.keys()),
                   text_auto=True)
cm_fig.update_layout(title="Confusion Matrix")
cm_fig.show()
```

Training and Evaluation

A Convolutional Neural Network (CNN) was used to make this code to recognize fruits and vegetables. The code makes use of a pre-trained VGG16 model, which is fine-tuned on the provided dataset of fruits and vegetables.

Dataset Splitting

Dataset splitting is an important step in machine learning and deep learning, it divides the dataset into subsets for training, validation, and testing. This ensures that the model is tested on data it hasn't seen during training, which is for assessing the model's generalization ability.

When splitting the dataset, we need to maintain the distribution of key features and labels. An optimized dataset split ensures maximum utilization of data while minimizing the risk of overfitting or underfitting.

For this project we've split the dataset to a training, validation, and test split.

- Training: This set is used to train the model, allowing it to learn from the data in the dataset given.
- Testing: This set is used to evaluate the model after it's trained.
- Validation: Used to tune parameters and monitor the model's performance during training to avoid overfitting.

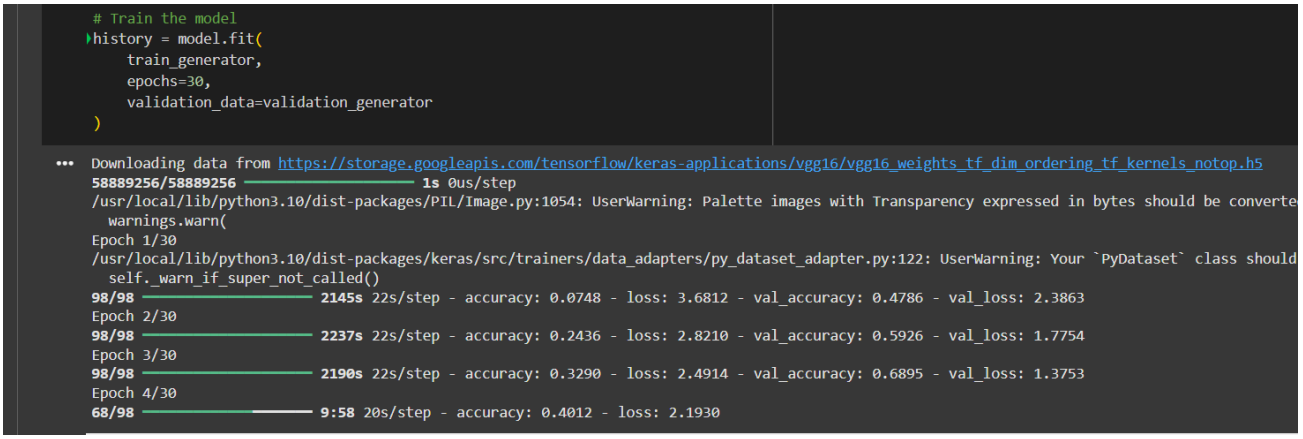
Usually, a split ratio of 70% for training, 15% for validation, and 15% for testing is used, which is what was used in this project and we believe is a balanced representation of the classes.

With the split above it ensures that the training set has enough data to learn patterns, the validation set helps fine-tune the model and prevent overfitting and the test set is used to evaluate the model generalization.

Training the CNN Model

We've had to design, build, and train a Convolutional Neural Network (CNN) to classify images based on the features it learns from the dataset we selected. CNNs are ideal for image classification because they automatically detect important features such as edges, textures, and shapes. Training CNN involves optimizing the network's parameters and allows the model to learn features from the input data through multiple layers of convolutions, activations, pooling, and fully connected layers.

When we were training the CNN we initially didn't put any hyperparameters when training the model and the image size we set it for was also too large with resulted in over 30 minutes of wait time for the training of each epoch which was too much, as seen below:



```
# Train the model
history = model.fit(
    train_generator,
    epochs=30,
    validation_data=validation_generator
)
```

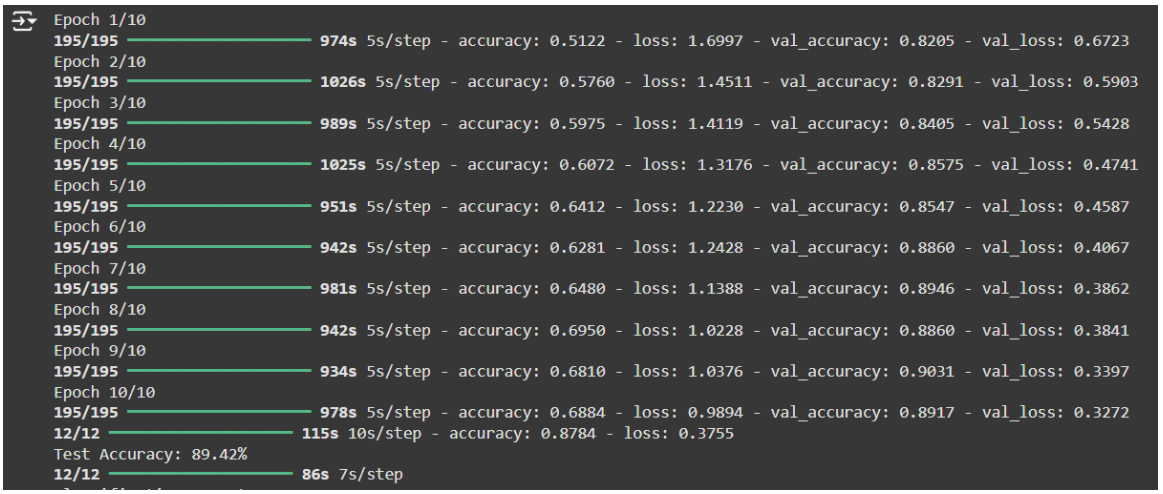
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
/usr/local/lib/python3.10/dist-packages/PIL/Image.py:1054: UserWarning: Palette images with Transparency expressed in bytes should be converte
  warnings.warn(
Epoch 1/30
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122: UserWarning: Your `PyDataset` class should
  self._warn_if_super_not_called()
98/98 ━━━━━━━━━━━━━━━━━━━━ 2145s 22s/step - accuracy: 0.0748 - loss: 3.6812 - val_accuracy: 0.4786 - val_loss: 2.3863
Epoch 2/30
98/98 ━━━━━━━━━━━━━━━━━━━━ 2237s 22s/step - accuracy: 0.2436 - loss: 2.8210 - val_accuracy: 0.5926 - val_loss: 1.7754
Epoch 3/30
98/98 ━━━━━━━━━━━━━━━━━━━━ 2190s 22s/step - accuracy: 0.3290 - loss: 2.4914 - val_accuracy: 0.6895 - val_loss: 1.3753
Epoch 4/30
68/98 ━━━━━━━━━━━━━━━━━━━━ 9:58 20s/step - accuracy: 0.4012 - loss: 2.1930
```

After that we reduced the image size a put in a hyperparameter to reduce the steps to 50 from the previous 98 and reduced the epochs from the previous 30 to 25 which greatly reduced our wait time for each epoch from over 30 minutes to about 5 minutes for each, as seen below:

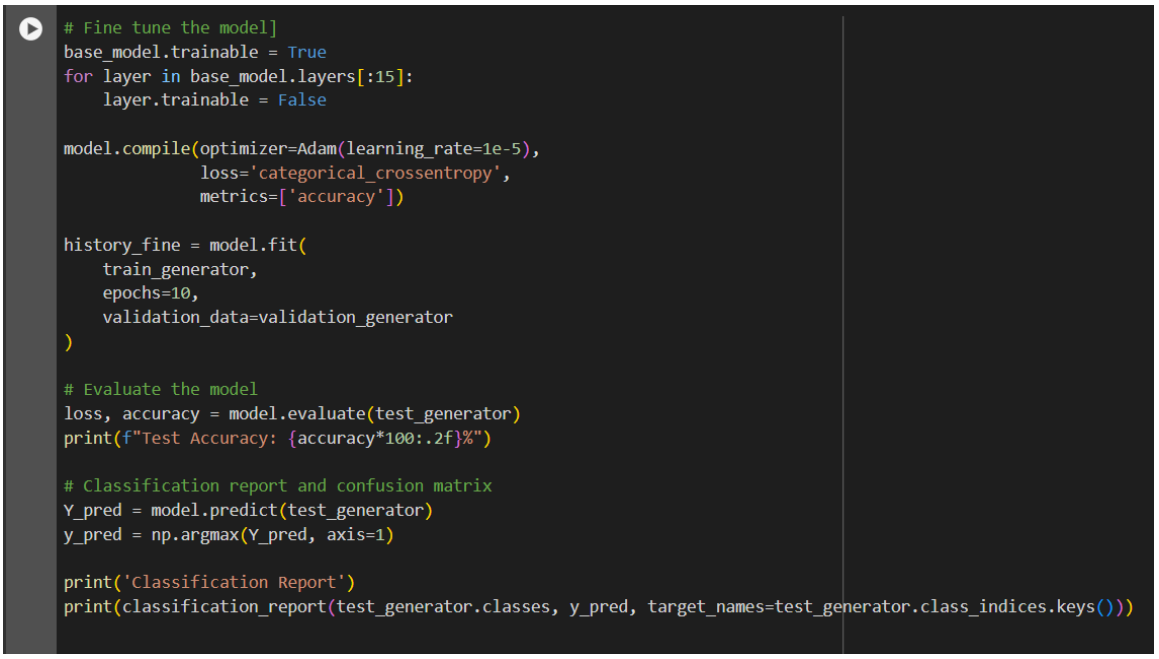
```
# Train the model
history = model.fit(
    train_generator,
    steps_per_epoch=50,
    epochs=25,
    validation_data=validation_generator
)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_ker
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/25
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122: UserWarning: Your `PyData
  self._warn_if_super_not_called()
/usr/local/lib/python3.10/dist-packages/PIL/Image.py:1054: UserWarning: Palette images with Transparency expressed in bytes s
  warnings.warn(
50/50 ━━━━━━━━━━━━━━━━━━━━ 380s 7s/step - accuracy: 0.0407 - loss: 3.9049 - val_accuracy: 0.1880 - val_loss: 3.1986
Epoch 2/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 341s 7s/step - accuracy: 0.0981 - loss: 3.4060 - val_accuracy: 0.2821 - val_loss: 2.8912
Epoch 3/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 288s 6s/step - accuracy: 0.1351 - loss: 3.2498 - val_accuracy: 0.3818 - val_loss: 2.6414
Epoch 4/25
45/50 ━━━━━━━━━━━━━━━━━━━━ 18s 4s/step - accuracy: 0.1906 - loss: 3.0502/usr/lib/python3.10/contextlib.py:153: UserWarning: Y
  self.gen.throw(typ, value, traceback)
50/50 ━━━━━━━━━━━━━━━━━━━━ 257s 5s/step - accuracy: 0.1920 - loss: 3.0434 - val_accuracy: 0.3960 - val_loss: 2.4306
Epoch 5/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 311s 6s/step - accuracy: 0.2260 - loss: 2.8813 - val_accuracy: 0.4387 - val_loss: 2.2120
Epoch 6/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 288s 6s/step - accuracy: 0.2528 - loss: 2.7954 - val_accuracy: 0.5128 - val_loss: 2.0557
Epoch 7/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 280s 6s/step - accuracy: 0.2757 - loss: 2.6299 - val_accuracy: 0.5926 - val_loss: 1.8522
Epoch 8/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 314s 6s/step - accuracy: 0.3145 - loss: 2.5146 - val_accuracy: 0.5926 - val_loss: 1.7758
Epoch 9/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 332s 6s/step - accuracy: 0.3280 - loss: 2.4751 - val_accuracy: 0.6097 - val_loss: 1.6890
Epoch 10/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 286s 6s/step - accuracy: 0.3494 - loss: 2.4035 - val_accuracy: 0.6182 - val_loss: 1.5926
Epoch 11/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 336s 7s/step - accuracy: 0.3472 - loss: 2.3500 - val_accuracy: 0.6439 - val_loss: 1.5236
Epoch 12/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 316s 6s/step - accuracy: 0.3808 - loss: 2.2599 - val_accuracy: 0.6353 - val_loss: 1.4803
Epoch 13/25
50/50 ━━━━━━━━━━━━━━━━━━━━ 344s 7s/step - accuracy: 0.3617 - loss: 2.3064 - val_accuracy: 0.6353 - val_loss: 1.4138
```

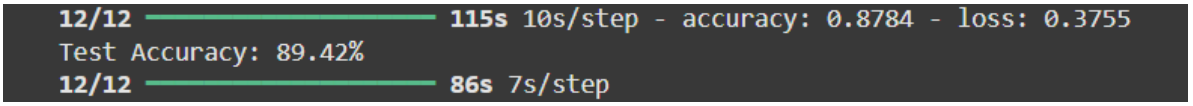
We then fine tune the training with 10 epochs and don't put a restriction on the steps and wait for over 10 minutes for each step, as seen below:



We also included the accuracy result in this cell and we see that the training results after the training and fine tuning gives us a 89.42% accuracy, as seen below:



```
# Fine tune the model]
base_model.trainable = True
for layer in base_model.layers[:15]:
    layer.trainable = False

model.compile(optimizer=Adam(learning_rate=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history_fine = model.fit(
    train_generator,
    epochs=10,
    validation_data=validation_generator
)

# Evaluate the model
loss, accuracy = model.evaluate(test_generator)
print(f"Test Accuracy: {accuracy*100:.2f}%")

# Classification report and confusion matrix
Y_pred = model.predict(test_generator)
y_pred = np.argmax(Y_pred, axis=1)

print('Classification Report')
print(classification_report(test_generator.classes, y_pred, target_names=test_generator.class_indices.keys()))
```



Evaluation and Metrics

After training the CNN model, it's important to evaluate its performance on a test set to see if the model works. The test dataset has data that the train dataset hasn't seen so we will see how the model evaluates new unseen data.

There are several metrics the we aimed to assess the performance of during the evaluation:

- Precision: Ratio of true positive predictions to total predicted positives.
- Recall: Ratio of true positive predictions to total actual positives.
- F1 Score: THarmonic mean of precision and recall.
- Confusion Matrix: Summarizes prediction results into true positives and false positives.
- Model Accuracy and Loss (before and after fine tuning): Proportion of correct predictions to the total predictions made.

Classification Result

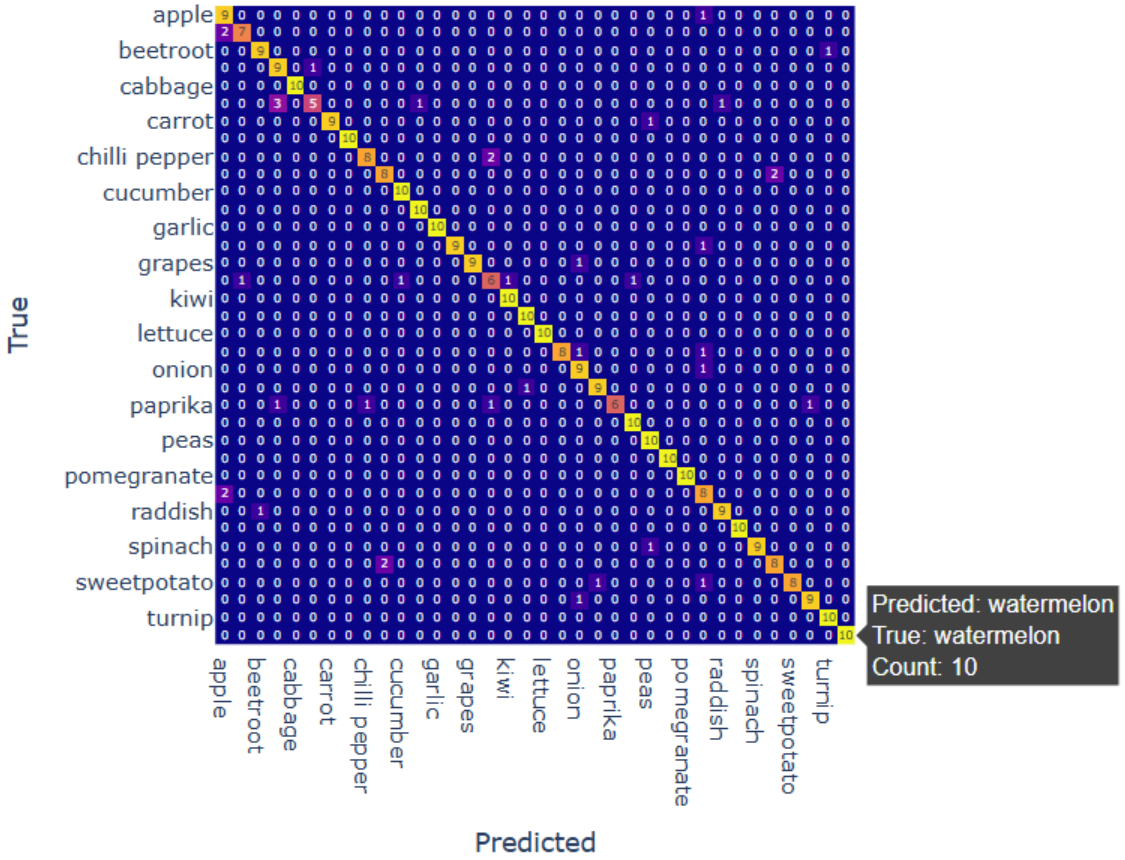
In the classification result below we see the evaluation results of the precision, recall, f1-score and an additional support.

From the classification result we see an 89% accuracy. The macro average and weighted average have a 90%, both metrics showed balanced performance for each class, with slight variations, which is expected for real-world datasets. In the training mentioned in the report the steps and epochs were reduced during training due to the time it took to train the model. However we believe that if the hyperparameters we applied during the training were removed and if the model was given more time to train it could result in a higher accuracy.

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 86s 7s/step
Classification Report
              precision    recall  f1-score   support

       apple       0.69      0.90      0.78        10
      banana       0.88      0.78      0.82         9
    beetroot       0.90      0.90      0.90        10
  bell pepper       0.69      0.90      0.78        10
     cabbage       1.00      1.00      1.00        10
    capsicum       0.83      0.50      0.62        10
      carrot       1.00      0.90      0.95        10
  cauliflower       1.00      1.00      1.00        10
chilli pepper       0.89      0.80      0.84        10
        corn       0.80      0.80      0.80        10
    cucumber       0.91      1.00      0.95        10
    eggplant       0.91      1.00      0.95        10
      garlic       1.00      1.00      1.00        10
      ginger       1.00      0.90      0.95        10
      grapes       1.00      0.90      0.95        10
    jalepeno       0.67      0.60      0.63        10
        kiwi       0.91      1.00      0.95        10
       lemon       0.91      1.00      0.95        10
     lettuce       1.00      1.00      1.00        10
       mango       1.00      0.80      0.89        10
       onion       0.75      0.90      0.82        10
      orange       0.90      0.90      0.90        10
     paprika       1.00      0.60      0.75        10
        pear       0.91      1.00      0.95        10
        peas       0.83      1.00      0.91        10
   pineapple       1.00      1.00      1.00        10
 pomegranate       1.00      1.00      1.00        10
      potato       0.62      0.80      0.70        10
     raddish       0.90      0.90      0.90        10
   soy beans       1.00      1.00      1.00        10
     spinach       1.00      0.90      0.95        10
   sweetcorn       0.80      0.80      0.80        10
 sweetpotato       1.00      0.80      0.89        10
      tomato       0.90      0.90      0.90        10
      turnip       0.91      1.00      0.95        10
  watermelon       1.00      1.00      1.00        10

    accuracy                           0.89       359
   macro avg       0.90      0.89      0.89       359
weighted avg       0.90      0.89      0.89       359
```

Confusion matrix

The confusion matrix is plotted to visualize the true positives, false positives, true negatives, and false negatives for each class, this is to identify where the model is making mistakes. In the confusion matrix below we see that the results are mostly positive outcomes as in the results we see that 15 were true positives scoring a 10 on the confusion matrix and the rest of the 21 were false positives however 11 had a score of 9 which also relates to the 89.42% accuracy that the test had. Overall most predictions were correct, with minimal false positives and false negatives.

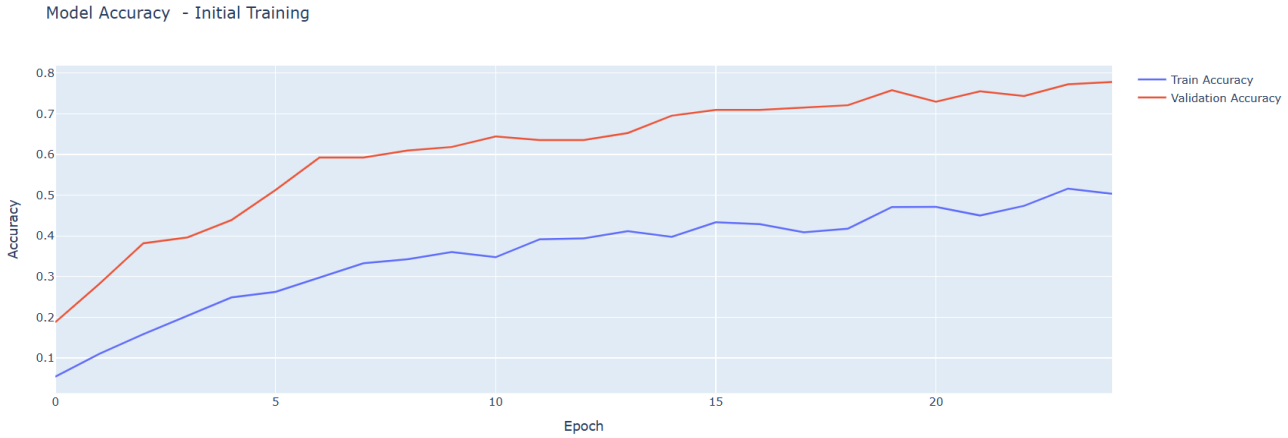


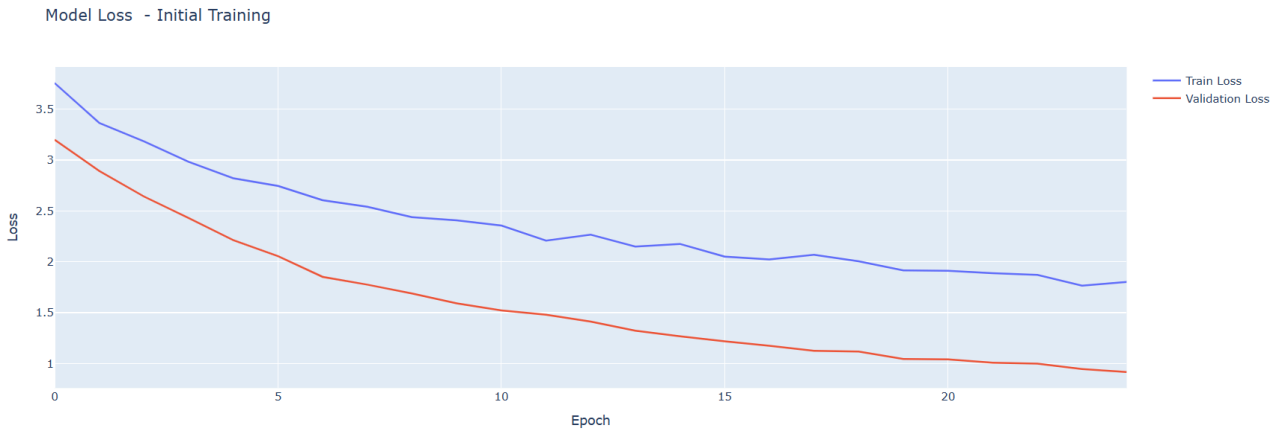
Model Accuracy and Loss (before and after fine tuning)

Last evaluation is to visualize the training and validation accuracy/loss over the course of the epochs before and after the fine tuning to assess how well the model has learned.

Before fine tuning

During the initial training of the model the amount of epochs that was used is 25 and 50 steps for each. This results in the graphs shown below, whereas the more training it gets from the epochs the higher the accuracy of the model and the loss also decreases the more training it gets.

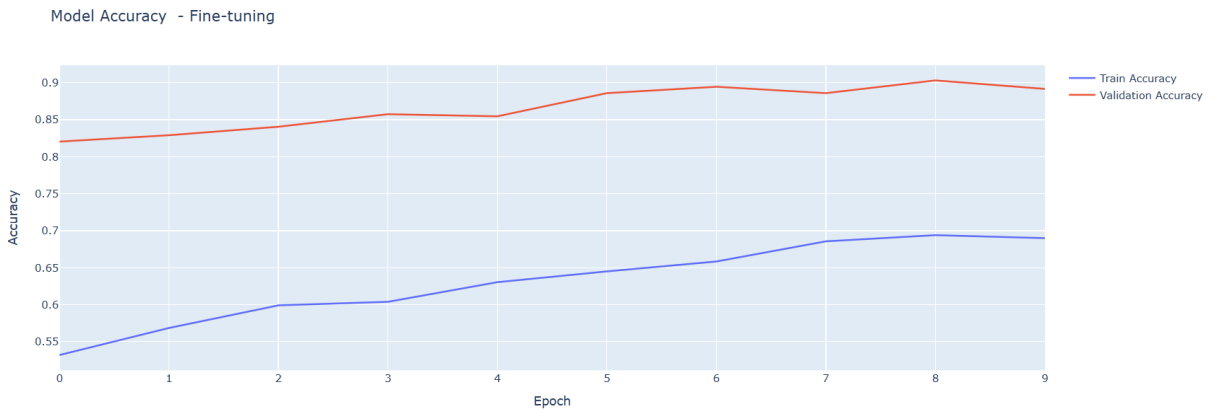


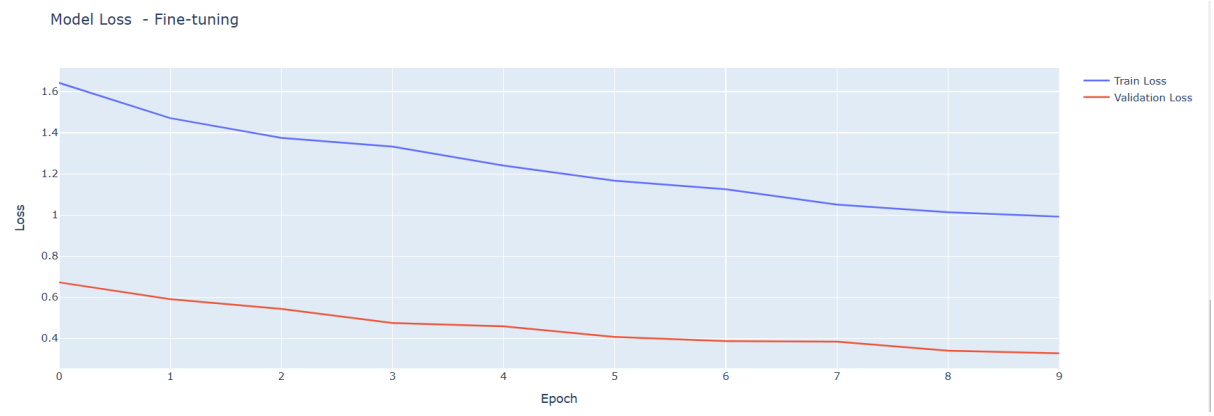


After fine tuning

In the fine tuning we used 10 epochs and it had 195 steps for each, this show a very high training accuracy at 70% and the validation increases to 90%. The loss also greatly decreases after the fine tuning with the training dropping to 1 and the validation dropping to about 0.3.

.



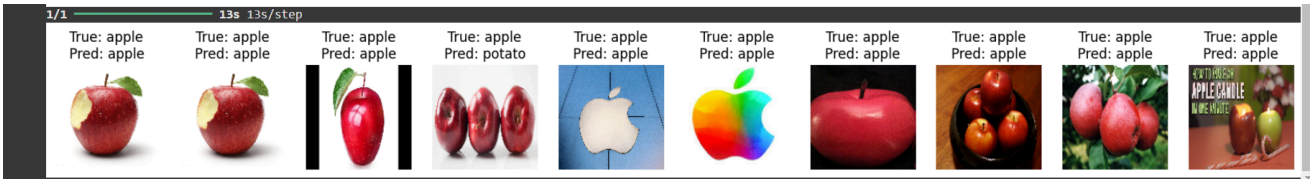


If the model got even more training and fine tuning it could achieve higher percentages of accuracy and validations and lower loss of the training and validations.

Testing the model with images

For the final testing we test the model to predict images from the dataset, using 10 images and viewing the predictions that the model gave based on the images it is shown that it has predicted 9 of the 10 images correctly as seen below:

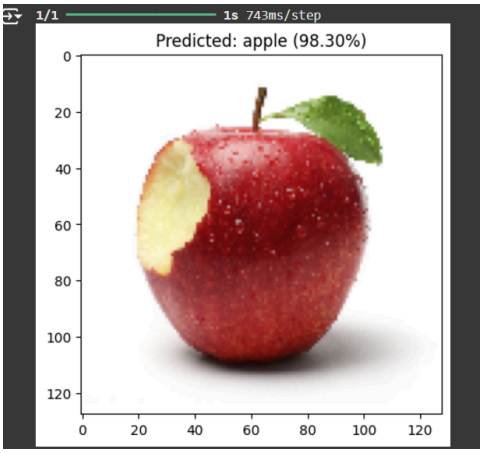
```python
# Display predictions
def display_predictions(generator, model, num_images=10):
    generator.reset()
    images, labels = next(generator)
    predictions = model.predict(images)

    fig, axes = plt.subplots(1, num_images, figsize=(20, 20))
    for i in range(num_images):
        ax = axes[i]
        ax.imshow(images[i])
        true_label = list(generator.class_indices.keys())[np.argmax(labels[i])]
        pred_label = list(generator.class_indices.keys())[np.argmax(predictions[i])]
        ax.set_title(f"True: {true_label}\nPred: {pred_label}")
        ax.axis('off')
    plt.show()

display_predictions(test_generator, model)
```

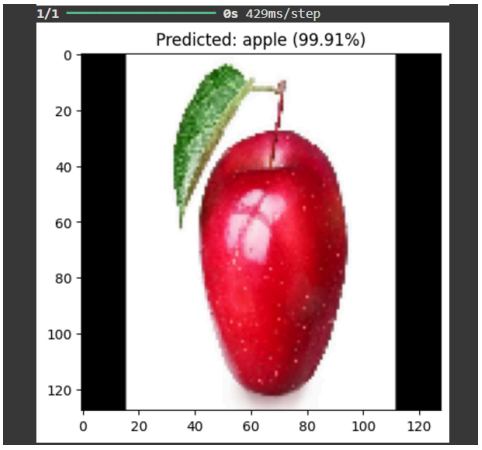


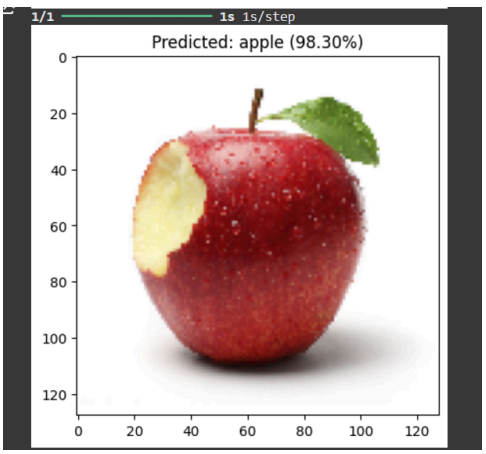
The next test is to see the percentage with images. The code below shows the function to predict the percentage of an image using 5 images from the validation set.

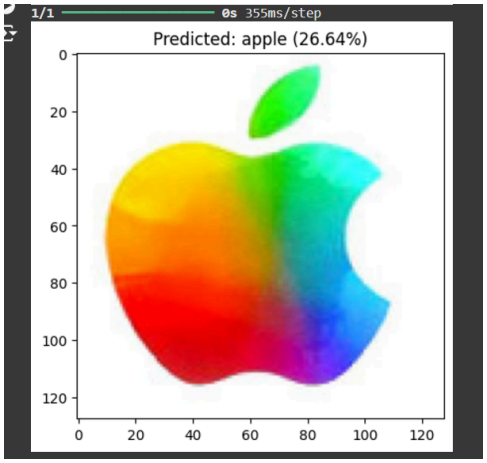
```python
#predict image
def predict_sample(image_path, model, class_indices):
    try:
        img = load_img(image_path, target_size=(128, 128))
        img_array = img_to_array(img) / 255.0
        img_array = np.expand_dims(img_array, axis=0)

        prediction = model.predict(img_array)
        predicted_class_index = np.argmax(prediction)
        predicted_class = class_indices[predicted_class_index]
        predicted_prob = prediction[0][predicted_class_index] * 100

        plt.imshow(img)
        plt.title(f'Predicted: {predicted_class} ({predicted_prob:.2f}%)')
        plt.show()
    except Exception as e:
        print(f"Error loading or predicting image {image_path}: {e}")

#images validation set
class_indices = validation_generator.class_indices
class_indices = dict((v, k) for k, v in class_indices.items())

validation_dir = validation_generator.directory

validation_images_dir = os.path.join(validation_dir, os.listdir(validation_dir)[0])
sample_images = os.listdir(validation_images_dir)[:5]

for img_name in sample_images:
    img_path = os.path.join(validation_images_dir, img_name)
    predict_sample(img_path, model, class_indices)
```

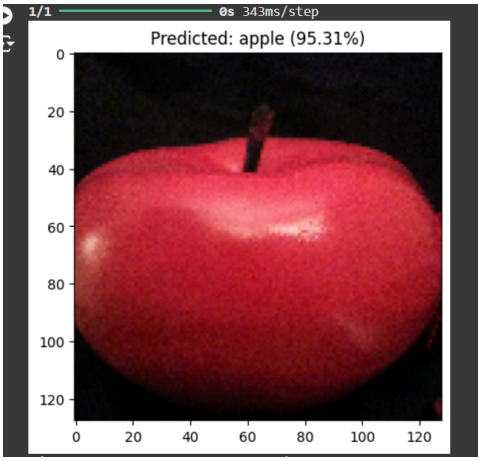
After the code is run the results are shown below. We can see that the model can predict a picture of an apple up to 99.91%. However there is one image of an apple that the model could predict was an apple but the percentage is not very high with 26.64%. Overall the testing is believed to be successful taking into account the 89.42% test accuracy that this model had.











Conclusion

Overall, this project showcases the ability of deep learning technologies such as Convolutional Neural Networks (CNNs) to solve multi-class image classification problems. By taking advantage of the publicly available dataset, we successfully created a CNN model that recognizes and classifies thousands of fruits and vegetables of 36 distinct classes.

We made sure that the architecture was designed optimally in choosing complexity as well as computational efficiency in dealing with the problem of too few images and classes present in the dataset. Activities such as image resizing and splitting the data into train and test sets were very effective ensuring the model was trained on a strong foundation. And with the use of dropout layers and max-pooling – overfitting was also reduced and this allowed the model to generalize better.

Among the main contributions to the development of the model were:

- Model Performance: The model was put in perspective within the use of the above mentioned metrics accuracy, precision, recall and F1-score, all served in giving a detailed evaluation of the model. The results showed that the model had a great ability in separating classes of produce that were similar; however if it could be further optimized then it would be able to perform stronger in real world scenarios.

- Data Challenges: We note the small-sized dataset, hence the need for augmenting the data or collecting a larger dataset in the future. The limitation defined in this way, nevertheless, did not prevent the model from performing well in classifying the objects.

- Practical Applications: As shown in this project, the image recognition models can be deployed in agriculture, retail and nutrition industries to facilitate processes including inventory, quality control and diet evaluation.

Future Work

To build upon the achievements of this project, the subsequent activities are recommended:

- Incorporating more images to each class in order to improve the diversity of backgrounds and orientations in the training samples.

- Using more sophisticated means including data augmentation and transfer learning to improve the outcome of the model.

- Tuning of hyper-parameters and adding further regularization mechanisms to control overfitting

To sum up, this project is able to demonstrate the potential of the CNNs in recognizing the images of fruits and vegetables, while acknowledging the shortcomings of small datasets. If such challenges are overcome, however, the model can develop into a more comprehensive and improved solution that can function effectively in any industrial setting, achieving substantial positive impact on the society