07MIAR - Redes Neuronales y Deep Learning: Proyecto de programación "*Deep Vision in classification tasks*"

Estudiantes:

- Alfredo Osiglia Rodríguez
- Laura Elena Betancourt Leal
- Luis Angel Motta Valero

URL: https://colab.research.google.com/drive/10MT79DZ1qeFn0tQWoXRFrw251gCkOAaz?usp=sharing

Descripción previa

La siguiente información es extraída del repositorio donde se almacena el dataset original (https://www.kaggle.com/datasets/misrakahmed/vegetable-image-dataset).

Nombre: Vegetable Image Dataset

Clasificación y reconocimiento de vegetales

Contexto

El experimento inicial se realiza con 15 tipos de hortalizas comunes en todo el mundo. Las verduras elegidas para el experimento son: judía, calabaza amarga, calabaza de botella, berenjena, brécol, col, pimiento, zanahoria, coliflor, pepino, papaya, patata, calabaza, rábano y tomate. Se utiliza un total de **21000** imágenes de 15 clases, cada una de las cuales contiene 1400 imágenes de tamaño 224×224 y en formato *$.jpg$. El conjunto de datos se divide en un 70% para la formación, un 15% para la validación y un 15% para las pruebas.

Contenido

Este dataset contiene tres folders:

- train (15000 imágenes)
- test (3000 imágenes)
- validation (3000 imágenes)

Cada una de las carpetas anteriores contiene subcarpetas para distintos vegetales en las que están presentes las imágenes de los vegetales/hortalizas respectivos.

De acuerdo con lo anterior, las 15 clases son las siguientes:

$$Y = [\,'Bean','BitterGourd','BottleGourd','Brinjal','Broccoli','Cabbage','Capsicum','Carrot','Cauliflower','Cucumber','Papaya','Potato','Pump$$

1. Carga del conjunto de datos

En primer lugar, instalamos la API de Kaggle y procedemos a descargar el fichero vegetable-image-dataset.zip que incluye todas las imágenes que se utilizarán para entrenar a la red. Este fichero se descomprime en la carpeta my_dataset.

In [ ]:
```
%%capture
# Instalación de la última versión de la API de Kaggle en Colab
!pip install --upgrade --force-reinstall --no-deps kaggle
```

In [ ]:
```
# Seleccionar el API Token personal previamente descargado (fichero kaggle.json)
from google.colab import files
files.upload()
```

Choose Files No file selected — Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

Out[ ]:
```
{'kaggle.json': b'{"username":"angelm97","key":"b2d97666b7674148aa715f3cf203f73e"}'}
```

In [ ]:
```
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
```

In [ ]:
```
import os
```

```
os.environ['KAGGLE_USERNAME'] = "angelm97"
os.environ['KAGGLE_KEY'] = "b2d97666b7674148aa715f3cf203f73e"
```

In [ ]:
```
# Descargamos el dataset proveniente de Kaggle
!kaggle datasets download -d misrakahmed/vegetable-image-dataset
```

```
Downloading vegetable-image-dataset.zip to /content
 98% 524M/534M [00:04<00:00, 182MB/s]
100% 534M/534M [00:04<00:00, 127MB/s]
```

In [ ]:
```
# Creamos un directorio para descomprimir los datos
!mkdir my_dataset
```

In [ ]:
```
%%capture
# Descomprimimos los datos y los dejamos listos para trabajar
!unzip vegetable-image-dataset.zip -d my_dataset
```

2. Inspección del conjunto de datos

En este apartado hacemos una exploración inicial de los datos para confirmar que tienen la estructura adecuada.

In [ ]:
```
#Verificamos si todas las imágenes tienen el mismo tamaño
import os
from PIL import Image

def check_image_sizes_recursive(folder_path):
    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if file.lower().endswith('.jpg'):
                file_path = os.path.join(root, file)
                with Image.open(file_path) as img:
                    width, height = img.size
                    if width != 224 or height != 224:
                        print(f"File: {file_path} - Size: {width}x{height}")

#Mostramos los datos y ubicación de las imágenes que no cumplen con el tamaño de 224x224 píxeles
folder_path = './drive/MyDrive/my_dataset'
check_image_sizes_recursive(folder_path)
```

Se observa que algunas imágenes de la clase 'Papaya' y otras de la clase 'Bitter_Gourd' tienen unas un tamaño distinto a 224 x 224. Las imágenes en cuestión son las siguientes:

Clase 'Bitter_Gourd':

Carpeta train: 3 imágenes.

- 0526.jpg - Size: 224x205
- 0430.jpg - Size: 224x193
- 0609.jpg - Size: 224x200

Clase 'Papaya':

Carpeta est: 1 imagen.

- 1246.jpg - Size: 224x207

Carpeta train: 3 imágenes

- 0741.jpg - Size: 224x210
- 0126.jpg - Size: 224x211
- 0176.jpg - Size: 224x198

Carpeta validation: 2 imágenes

- 1138.jpg - Size: 224x187
- 1150.jpg - Size: 224x223

Todas las imágenes tienen un width the 224px pero no todas tienen la misma altura. Al asignar el atributo "padding=same" a las primeras capas convolucionales, Tensorflow añadirá un padding de ceros (píxeles de color negro) a estas imágenes para que cumplan con la altura de 224px del resto de imágenes. Esto no debería suponer un inconveniente para la red debido a que son muy pocas imágenes a las que se hará esta padding con respecto al tamaño del dataset.

A continuación, se separan los datos en conjuntos de train, test y validation de acuerdo a como están estructurados en el fichero de Kaggle y se hace visualizan imágenes aleatorias para confirmar que se han cargado de manera correcta.

In [ ]:
```
# Creamos una función para guardar las imágenes en tensores
```

```
import tensorflow as tf
from pathlib import Path
path = '/content/my_dataset/Vegetable Images'

#Se ha fijado el batch_size que viene por defecto en la documentación
def load_dataset(subfolder, batch_size=32):
  dataset_path = path + subfolder
  data_dir = Path(dataset_path)
  params = {
    'directory': data_dir,
    'seed': 0,
    'batch_size': batch_size,
    'image_size': (224,224)
    }
  dataset_params = {**params}
  dataset = tf.keras.utils.image_dataset_from_directory(**dataset_params)
  return dataset
```

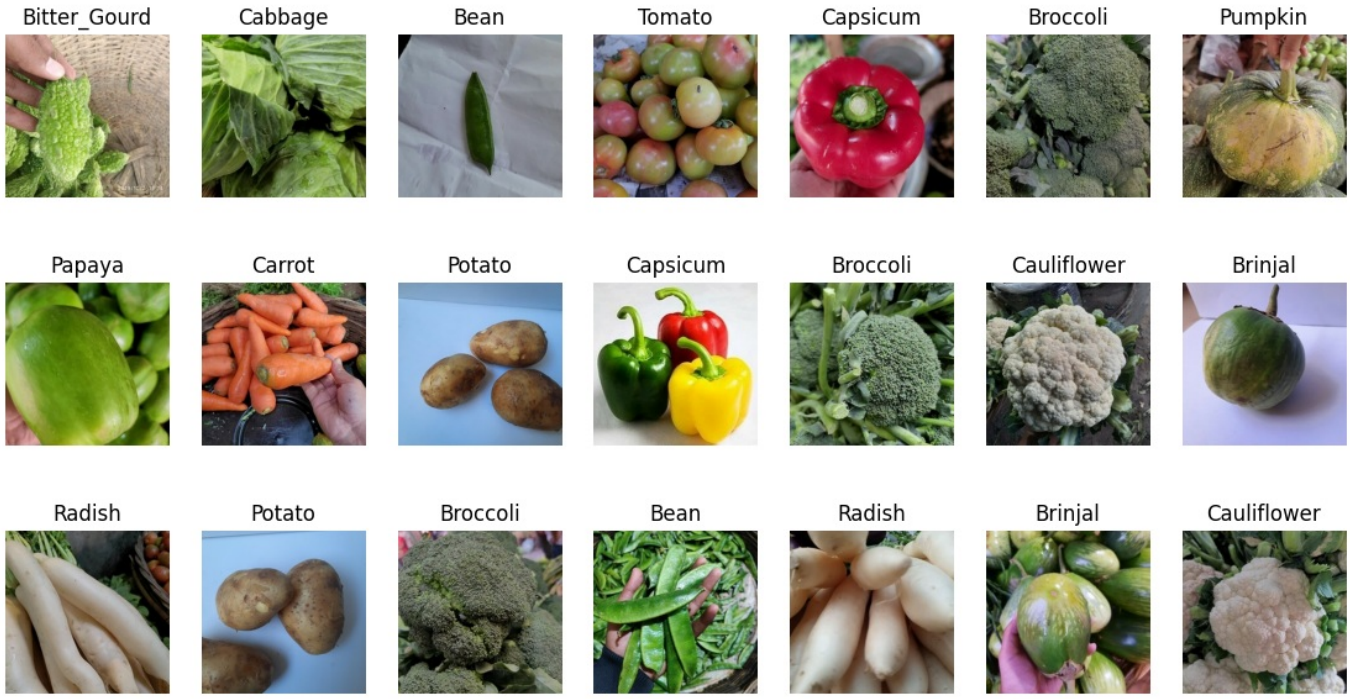
In [ ]:
```
# Guardamos los diferentes conjuntos de imágenes en variables
print('>> Training Set:')
train_ds = load_dataset('/train')
print('>> Testing Set:')
test_ds = load_dataset('/test', None)
print('>> Validation Set:')
val_ds = load_dataset('/validation')
```

```
>> Training Set:
Found 15000 files belonging to 15 classes.
>> Testing Set:
Found 3000 files belonging to 15 classes.
>> Validation Set:
Found 3000 files belonging to 15 classes.
```

In [ ]:
```
# Definimos las clases
class_names = test_ds.class_names
class_names
```

Out[ ]:
```
['Bean',
 'Bitter_Gourd',
 'Bottle_Gourd',
 'Brinjal',
 'Broccoli',
 'Cabbage',
 'Capsicum',
 'Carrot',
 'Cauliflower',
 'Cucumber',
 'Papaya',
 'Potato',
 'Pumpkin',
 'Radish',
 'Tomato']
```

In [ ]:
```
# Visualización de los datos de train
import matplotlib.pyplot as plt

plt.figure(figsize=(14,10))
for images, labels in train_ds.take(1):
  for i in range(21):
    ax = plt.subplot(4, 7, i+1)
    plt.imshow(images[i].numpy().astype('uint8'))
    plt.title(class_names[labels[i]])
    plt.axis('off')
```



3. Acondicionamiento de los datos

Teniendo ya las imágenes cargadas en el entorno y subdivididas en conjuntos de train, validation y test; se procede a normalizar las imágenes para todos los conjuntos y a codificar las etiquetas usando One-Hot Encoding únicamente para el train y validation.

In [ ]:
```
# Normalización y aplicación de One-hot Encoding

from tensorflow.keras.layers import Rescaling, CategoryEncoding

# Creación de una capa de re-escalado (normalización de datos)
rescaling_layer = Rescaling(1.0/255)
# Conversión a one-hot encoding
OHE = CategoryEncoding(num_tokens=15, output_mode="one_hot")

# Aplicamos normalización a cada imagen del dataset y OHE a las etiquetas
norm_train = train_ds.map(lambda x,y: (rescaling_layer(x), OHE(y)))
norm_val = val_ds.map(lambda x,y: (rescaling_layer(x), OHE(y)))
```

In [ ]:
```
# Separamos el conjunto de test entre imágenes y sus etiquetas
# Se utilizarán para evaluar la precisión del modelo
import numpy as np
x_test = []
y_test = []
norm_test = test_ds.map(lambda x,y: (rescaling_layer(x), y)) # dejamos intactas las etiquetas (y)
for image, label in norm_test.take(len(norm_test)):
  x_test.append(image)
  y_test.append(label)

x_test = np.array(x_test)
y_test = np.array(y_test)
print("Test set shape:")
print(f'Images: {x_test.shape}')
print(f'Labels: {y_test.shape}')
```

```
Test set shape:
Images: (3000, 224, 224, 3)
Labels: (3000,)
```

4.1 Desarrollo y entrenamiento de un modelo desde cero o from scratch

En este apartado se procede a entrenar un modelo "from scratch" definiendo la arquitectura desde cero, observando cómo se comporta en el entrenamiento y aplicando técnicas de regularización para mejorar su rendimiento. Partimos de una arquitectura con tres bloques convolucionales sobre los que únicamente se aplica max pooling antes de pasar al siguiente bloque y una capa MLP sin ningún tipo de regularización. Luego aplicamos las técnicas de batch normalization y dropout para observar cómo cambia el accuracy. En el tercer modelo se aplicará la técnica de Early Stopping. Finalmente, aplicamos la técnica de Data Augmentation para intentar reducir aún más el overfitting.

4.1.1 Modelo from scratch únicamente con max pooling

Definición del modelo

In [ ]:
```
# Importamos las librerías necesarias
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense, Dropout, BatchNormalization, MaxI
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt

########################################
###### Definimos la arquitectura ########
########################################
#BASE MODEL
# Definimos entradas
inputs = Input(shape=(224, 224, 3))

# Primer set de capas CONV
x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(inputs)
x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(x1)
x1 = MaxPooling2D(pool_size=(2, 2))(x1)
x1 = Dropout(0.25)(x1)

# Segundo set de capas CONV
x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x1) #(X)
x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x2) #(X)
x2 = MaxPooling2D(pool_size=(2, 2))(x2) #(X)


# Tercer set de capas CONV
x3 = Conv2D(256, (3, 3), padding="same", activation="relu")(x2) #(X)
x3 = Conv2D(256, (3, 3), padding="same", activation="relu")(x3) #(X)
x3 = MaxPooling2D(pool_size=(2, 2))(x3) #(X)


# TOP MODEL
# Primer (y único) set de capas FC => RELU
xfc = Flatten()(x3) #(X)
xfc = Dense(512, activation="relu")(xfc)
predictions = Dense(15, activation="softmax")(xfc) #(X)
```

Compilación del modelo

In [ ]:
```
# Unimos las entradas y el modelo mediante la función Model con parámetros inputs y outputs (Consultar la docume
model_cnn = Model(inputs=inputs, outputs=predictions) #(X)

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
model_cnn.compile(loss="categorical_crossentropy",
                  optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08),
                  metrics=["accuracy"]) #(X)

n_epochs=20

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
# IMPORTANTE (Consultar la documentación)
H = model_cnn.fit(norm_train, validation_data=norm_val, batch_size=128, epochs=n_epochs, verbose=1) #(X)
```

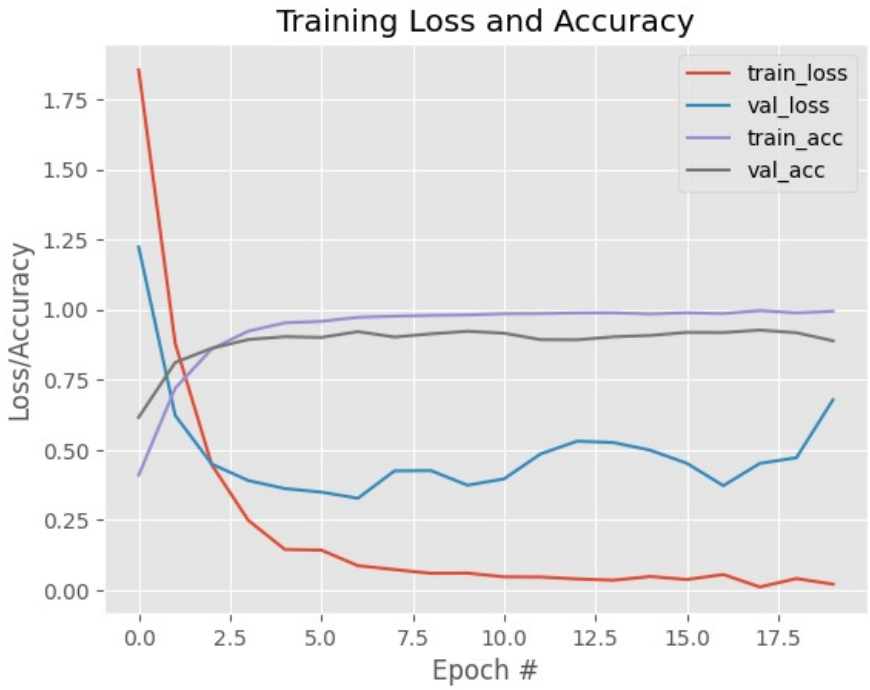
```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
469/469 [==============================] - 122s 237ms/step - loss: 1.8554 - accuracy: 0.4095 - val_loss: 1.2235
- val_accuracy: 0.6163
Epoch 2/20
469/469 [==============================] - 109s 231ms/step - loss: 0.8799 - accuracy: 0.7200 - val_loss: 0.6228
- val_accuracy: 0.8117
Epoch 3/20
469/469 [==============================] - 108s 230ms/step - loss: 0.4469 - accuracy: 0.8599 - val_loss: 0.4495
- val_accuracy: 0.8623
Epoch 4/20
469/469 [==============================] - 109s 232ms/step - loss: 0.2487 - accuracy: 0.9233 - val_loss: 0.3905
- val_accuracy: 0.8933
Epoch 5/20
469/469 [==============================] - 107s 228ms/step - loss: 0.1447 - accuracy: 0.9531 - val_loss: 0.3618
- val_accuracy: 0.9037
Epoch 6/20
469/469 [==============================] - 109s 231ms/step - loss: 0.1425 - accuracy: 0.9582 - val_loss: 0.3492
- val_accuracy: 0.9010
Epoch 7/20
469/469 [==============================] - 108s 230ms/step - loss: 0.0868 - accuracy: 0.9727 - val_loss: 0.3273
- val_accuracy: 0.9217
Epoch 8/20
469/469 [==============================] - 107s 227ms/step - loss: 0.0729 - accuracy: 0.9766 - val_loss: 0.4250
- val_accuracy: 0.9023
Epoch 9/20
469/469 [==============================] - 108s 230ms/step - loss: 0.0596 - accuracy: 0.9797 - val_loss: 0.4259
- val_accuracy: 0.9137
Epoch 10/20
469/469 [==============================] - 107s 228ms/step - loss: 0.0602 - accuracy: 0.9814 - val_loss: 0.3738
- val_accuracy: 0.9230
Epoch 11/20
469/469 [==============================] - 107s 227ms/step - loss: 0.0470 - accuracy: 0.9854 - val_loss: 0.3964
- val_accuracy: 0.9160
Epoch 12/20
469/469 [==============================] - 108s 229ms/step - loss: 0.0465 - accuracy: 0.9861 - val_loss: 0.4854
- val_accuracy: 0.8930
Epoch 13/20
469/469 [==============================] - 107s 228ms/step - loss: 0.0394 - accuracy: 0.9883 - val_loss: 0.5310
- val_accuracy: 0.8923
Epoch 14/20
469/469 [==============================] - 106s 226ms/step - loss: 0.0351 - accuracy: 0.9888 - val_loss: 0.5259
- val_accuracy: 0.9033
Epoch 15/20
469/469 [==============================] - 107s 229ms/step - loss: 0.0482 - accuracy: 0.9847 - val_loss: 0.4987
- val_accuracy: 0.9080
Epoch 16/20
469/469 [==============================] - 106s 226ms/step - loss: 0.0374 - accuracy: 0.9889 - val_loss: 0.4523
- val_accuracy: 0.9190
Epoch 17/20
469/469 [==============================] - 107s 229ms/step - loss: 0.0554 - accuracy: 0.9860 - val_loss: 0.3723
- val_accuracy: 0.9183
Epoch 18/20
469/469 [==============================] - 106s 226ms/step - loss: 0.0109 - accuracy: 0.9967 - val_loss: 0.4517
- val_accuracy: 0.9270
Epoch 19/20
469/469 [==============================] - 107s 228ms/step - loss: 0.0408 - accuracy: 0.9886 - val_loss: 0.4717
- val_accuracy: 0.9177
Epoch 20/20
469/469 [==============================] - 107s 228ms/step - loss: 0.0210 - accuracy: 0.9941 - val_loss: 0.6783
- val_accuracy: 0.8890
```

Evaluación del modelo

Como era de esperar, el modelo presenta una cantidad importante de overfitting debido a que no se ha aplicado ninguna técnica de regularización más allá del max pooling. Se observa que en la última época el modelo no solo no mejora si no que parece tener tendencia a divergir.

In [ ]:
```python
from sklearn.metrics import classification_report

# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

```
plt.show()
```



In [ ]:
```
# Evaluación del modelo
print("[INFO]: Evaluando el modelo de red neuronal...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_cnn.predict(x_test, batch_size=128) #(X)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo de red neuronal...
24/24 [==============================] - 11s 247ms/step
              precision    recall  f1-score   support

        Bean       0.70      0.84      0.77       200
Bitter_Gourd       0.94      0.88      0.90       200
Bottle_Gourd       0.93      0.95      0.94       200
     Brinjal       0.88      0.75      0.81       200
    Broccoli       0.87      0.87      0.87       200
     Cabbage       0.76      0.86      0.81       200
    Capsicum       0.98      0.97      0.97       200
      Carrot       0.96      0.99      0.98       200
 Cauliflower       0.81      0.80      0.80       200
    Cucumber       0.99      0.80      0.88       200
      Papaya       0.96      0.93      0.94       200
      Potato       0.98      0.96      0.97       200
     Pumpkin       0.84      0.92      0.88       200
      Radish       0.96      0.93      0.94       200
      Tomato       0.80      0.83      0.82       200

    accuracy                           0.88      3000
   macro avg       0.89      0.88      0.89      3000
weighted avg       0.89      0.88      0.89      3000
```

4.1.2 Modelo from scratch aplicando max pooling, batch normalization y droput

Definición del modelo

In [ ]:
```
# Importamos las librerías necesarias
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense, Dropout, BatchNormalization, MaxP
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt

########################################
###### Definimos la arquitectura ########
########################################
#BASE MODEL
# Definimos entradas
inputs = Input(shape=(224, 224, 3))
```

```
# Primer set de capas CONV => RELU => CONV => RELU => POOL
x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(inputs)
x1 = BatchNormalization()(x1)
x1 = Conv2D(32, (3, 3), padding="same", activation="relu")(x1)
x1 = BatchNormalization()(x1)
x1 = MaxPooling2D(pool_size=(2, 2))(x1)
x1 = Dropout(0.25)(x1)

# Segundo set de capas CONV => RELU => CONV => RELU => POOL
x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x1) #(X)
x2 = BatchNormalization()(x2) #(X)
x2 = Conv2D(64, (3, 3), padding="same", activation="relu")(x2) #(X)
x2 = BatchNormalization()(x2) #(X)
x2 = MaxPooling2D(pool_size=(2, 2))(x2) #(X)
x2 = Dropout(0.25)(x2) #(X)

# Tercer set de capas CONV => RELU => CONV => RELU => POOL
x3 = Conv2D(256, (3, 3), padding="same", activation="relu")(x2) #(X)
x3 = BatchNormalization()(x3) #(X)
x3 = Conv2D(256, (3, 3), padding="same", activation="relu")(x3) #(X)
x3 = BatchNormalization()(x3) #(X)
x3 = MaxPooling2D(pool_size=(2, 2))(x3) #(X)
x3 = Dropout(0.25)(x3) #(X)

# TOP MODEL
# Primer (y único) set de capas FC => RELU
xfc = Flatten()(x3) #(X)
xfc = Dense(512, activation="relu")(xfc) #(X)
xfc = BatchNormalization()(xfc) #(X)
xfc = Dropout(0.5)(xfc) #(X)
# Clasificador softmax
predictions = Dense(15, activation="softmax")(xfc) #(X)
```

Compilación del modelo

In [ ]:

```
# Unimos las entradas y el modelo mediante la función Model con parámetros inputs y ouputs (Consultar la docume
model_cnn = Model(inputs=inputs, outputs=predictions) #(X)

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
model_cnn.compile(loss="categorical_crossentropy",
#                 loss='sparse_categorical_crossentropy',
                  optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08),
                  metrics=["accuracy"]) #(X)

n_epochs=20

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
# IMPORTANTE (Consultar la documentación)
H = model_cnn.fit(norm_train, validation_data=norm_val, batch_size=128, epochs=n_epochs, verbose=1) #(X)
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
469/469 [==============================] - 178s 338ms/step - loss: 0.9029 - accuracy: 0.7320 - val_loss: 1.9501
- val_accuracy: 0.4670
Epoch 2/20
469/469 [==============================] - 157s 335ms/step - loss: 0.2626 - accuracy: 0.9175 - val_loss: 0.2112
- val_accuracy: 0.9283
Epoch 3/20
469/469 [==============================] - 156s 333ms/step - loss: 0.1347 - accuracy: 0.9586 - val_loss: 0.4888
- val_accuracy: 0.8603
Epoch 4/20
469/469 [==============================] - 161s 344ms/step - loss: 0.0984 - accuracy: 0.9687 - val_loss: 0.1278
- val_accuracy: 0.9597
Epoch 5/20
469/469 [==============================] - 156s 332ms/step - loss: 0.1015 - accuracy: 0.9678 - val_loss: 0.7386
- val_accuracy: 0.8440
Epoch 6/20
469/469 [==============================] - 156s 333ms/step - loss: 0.0583 - accuracy: 0.9818 - val_loss: 0.3975
- val_accuracy: 0.9030
Epoch 7/20
469/469 [==============================] - 156s 331ms/step - loss: 0.0505 - accuracy: 0.9842 - val_loss: 0.2120
- val_accuracy: 0.9397
Epoch 8/20
469/469 [==============================] - 156s 331ms/step - loss: 0.0573 - accuracy: 0.9815 - val_loss: 4.7413
- val_accuracy: 0.4270
Epoch 9/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0705 - accuracy: 0.9777 - val_loss: 0.3782
- val_accuracy: 0.9057
Epoch 10/20
469/469 [==============================] - 157s 334ms/step - loss: 0.0374 - accuracy: 0.9884 - val_loss: 0.1737
- val_accuracy: 0.9547
Epoch 11/20
469/469 [==============================] - 156s 331ms/step - loss: 0.0499 - accuracy: 0.9841 - val_loss: 0.0791
- val_accuracy: 0.9780
Epoch 12/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0481 - accuracy: 0.9845 - val_loss: 0.3780
- val_accuracy: 0.9147
Epoch 13/20
469/469 [==============================] - 155s 331ms/step - loss: 0.0447 - accuracy: 0.9867 - val_loss: 0.6211
- val_accuracy: 0.8853
Epoch 14/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0251 - accuracy: 0.9918 - val_loss: 0.1319
- val_accuracy: 0.9673
Epoch 15/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0231 - accuracy: 0.9924 - val_loss: 0.3108
- val_accuracy: 0.9243
Epoch 16/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0300 - accuracy: 0.9900 - val_loss: 0.0859
- val_accuracy: 0.9807
Epoch 17/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0333 - accuracy: 0.9899 - val_loss: 0.3023
- val_accuracy: 0.9350
Epoch 18/20
469/469 [==============================] - 155s 330ms/step - loss: 0.0392 - accuracy: 0.9875 - val_loss: 0.1152
- val_accuracy: 0.9733
Epoch 19/20
469/469 [==============================] - 155s 331ms/step - loss: 0.0233 - accuracy: 0.9921 - val_loss: 0.0681
- val_accuracy: 0.9853
Epoch 20/20
469/469 [==============================] - 155s 331ms/step - loss: 0.0272 - accuracy: 0.9905 - val_loss: 0.0925
- val_accuracy: 0.9787
```

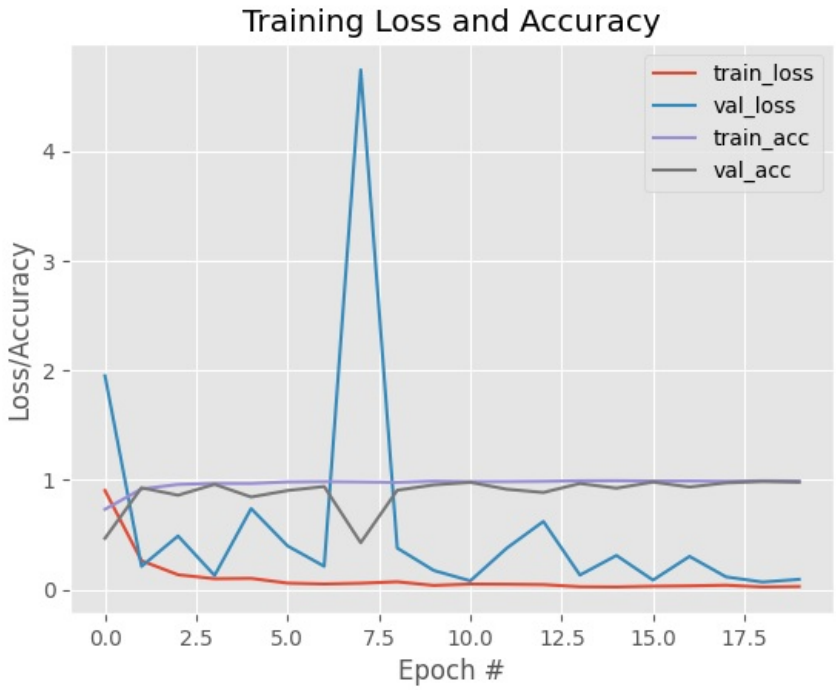
Evaluación del modelo

Aquí se puede apreciar una mejora considerable del overfitting visto en el modelo anterior, donde en la última epoch la diferencia entre validation loss y training loss es menor a 0.1. Es interesante destacar el pico validation loss en la epoch, que puede ser causa de diversos factores, como sensibilidad del modelo durante la inicialización o una variabilidad marcada en el batch de validación para esa epoch.

En este modelo se ha alcanzado un f1-score perfecto para la detección de capsicum (pimiento). En el modelo anterior el valor de este vegetal también fue el más alto con un valor de 0.97.

In [ ]:
```
# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
```

```
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```



In [ ]:
```
from sklearn.metrics import classification_report

# Evaluación del modelo
print("[INFO]: Evaluando el modelo de red neuronal...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_cnn.predict(x_test, batch_size=128) #(X)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

```
[INFO]: Evaluando el modelo de red neuronal...
24/24 [==============================] - 5s 211ms/step
              precision    recall  f1-score   support

        Bean       0.98      0.97      0.98       200
Bitter_Gourd       0.95      0.98      0.97       200
Bottle_Gourd       0.97      1.00      0.99       200
     Brinjal       0.98      0.97      0.97       200
    Broccoli       1.00      0.94      0.97       200
     Cabbage       0.98      0.99      0.99       200
    Capsicum       1.00      1.00      1.00       200
      Carrot       0.99      0.99      0.99       200
 Cauliflower       0.99      0.95      0.97       200
    Cucumber       0.99      0.99      0.99       200
      Papaya       0.98      0.97      0.98       200
      Potato       0.99      1.00      0.99       200
     Pumpkin       0.97      0.99      0.98       200
      Radish       1.00      1.00      1.00       200
      Tomato       0.98      0.97      0.98       200

    accuracy                           0.98      3000
   macro avg       0.98      0.98      0.98      3000
weighted avg       0.98      0.98      0.98      3000
```

4.1.3 Modelo from scratch aplicando Early Stopping

Definición de callbacks para Early Stopping y compilación

Hasta ahora se ha realizado el entrenamiento de la red con un número de epochs igual a 20. Ahora se aplica un Early Stopping con patience=3 y se espera tener un número de epochs distinto al finalizar el entrenamiento.

In [ ]:
```
# Unimos las entradas y el modelo mediante la función Model con parámetros inputs y ouputs (Consultar la docume
model_cnn = Model(inputs=inputs, outputs=predictions) #(X)

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
model_cnn.compile(loss="categorical_crossentropy",
#                 loss='sparse_categorical_crossentropy',
```

```
                  optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08),
                  metrics=["accuracy"]) #(X)

my_callbacks = [
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     patience=3, # Valor de patience con mejor resultado entre 2-5
                                     mode='min'),
    tf.keras.callbacks.ModelCheckpoint(filepath='/content/',
                                       monitor='val_accuracy',
                                       save_weights_only=True,
                                       mode='max'),
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

n_epochs=50

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
# IMPORTANTE (Consultar la documentación)
H = model_cnn.fit(norm_train, validation_data=norm_val, batch_size=128, epochs=n_epochs, verbose=1, callbacks=my
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/50
469/469 [==============================] - 187s 344ms/step - loss: 0.9696 - accuracy: 0.7077 - val_loss: 2.3668
- val_accuracy: 0.4303
Epoch 2/50
469/469 [==============================] - 167s 355ms/step - loss: 0.2769 - accuracy: 0.9138 - val_loss: 0.2281
- val_accuracy: 0.9283
Epoch 3/50
469/469 [==============================] - 162s 345ms/step - loss: 0.1432 - accuracy: 0.9549 - val_loss: 0.1376
- val_accuracy: 0.9550
Epoch 4/50
469/469 [==============================] - 162s 345ms/step - loss: 0.0992 - accuracy: 0.9690 - val_loss: 0.1139
- val_accuracy: 0.9583
Epoch 5/50
469/469 [==============================] - 169s 361ms/step - loss: 0.0898 - accuracy: 0.9720 - val_loss: 0.5827
- val_accuracy: 0.8633
Epoch 6/50
469/469 [==============================] - 168s 358ms/step - loss: 0.0722 - accuracy: 0.9773 - val_loss: 0.0716
- val_accuracy: 0.9803
Epoch 7/50
469/469 [==============================] - 181s 387ms/step - loss: 0.0537 - accuracy: 0.9828 - val_loss: 0.2802
- val_accuracy: 0.9187
Epoch 8/50
469/469 [==============================] - 176s 376ms/step - loss: 0.0684 - accuracy: 0.9789 - val_loss: 0.1418
- val_accuracy: 0.9590
Epoch 9/50
469/469 [==============================] - 176s 375ms/step - loss: 0.0549 - accuracy: 0.9816 - val_loss: 0.3444
- val_accuracy: 0.9203
```

Evaluación del modelo

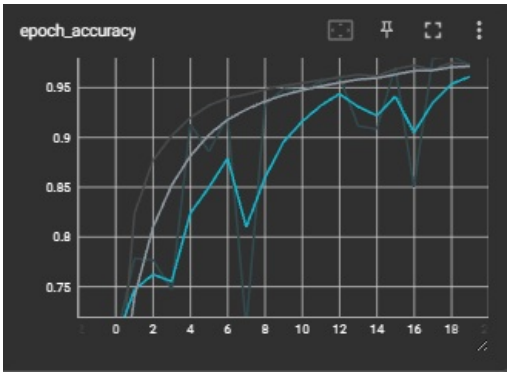
El modelo se detiene en la época 9. Aunque no hay mejora en el overfitting con respecto al modelo anterior, el tiempo de entrenamiento se ha reducido a la mitad. Los resultados de test son parecidos a los obtenidos en el modelo 4.1.1.

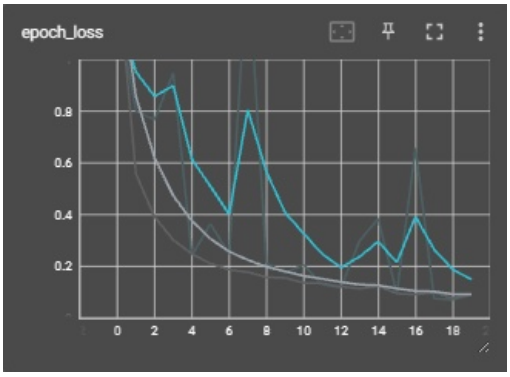
In [ ]:
```
# Muestro gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 9), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 9), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 9), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 9), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

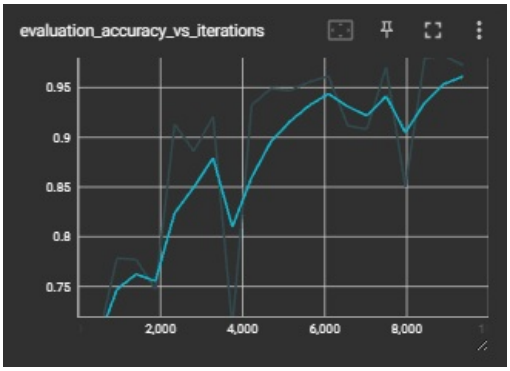
In [ ]:
```
# Evaluación del modelo
print("[INFO]: Evaluando el modelo de red neuronal...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_cnn.predict(x_test, batch_size=128) #(X)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names)) #(X)
```

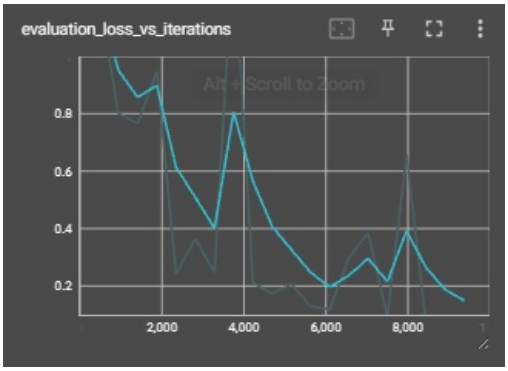
```
[INFO]: Evaluando el modelo de red neuronal...
24/24 [==============================] - 12s 290ms/step
              precision    recall  f1-score   support

        Bean       0.99      0.86      0.92       200
Bitter_Gourd       0.95      0.96      0.96       200
Bottle_Gourd       0.95      1.00      0.98       200
     Brinjal       0.91      0.91      0.91       200
    Broccoli       0.98      0.88      0.93       200
     Cabbage       0.96      0.91      0.93       200
    Capsicum       0.97      0.95      0.96       200
      Carrot       0.98      0.83      0.90       200
 Cauliflower       0.82      0.98      0.89       200
    Cucumber       0.95      0.98      0.97       200
      Papaya       0.89      0.97      0.93       200
      Potato       0.66      0.89      0.76       200
     Pumpkin       0.92      0.98      0.95       200
      Radish       0.95      0.82      0.88       200
      Tomato       0.92      0.71      0.80       200

    accuracy                           0.91      3000
   macro avg       0.92      0.91      0.91      3000
weighted avg       0.92      0.91      0.91      3000
```

> Gráficas de Tensorboard mostrando métricas de desempeño del modelo en función de las épocas e iteraciones









4.1.4 Modelo from scratch aplicando Data Augmentation

> Generación de contenedor DataGenerator para el aumento automático de muestras

In [ ]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=12, # grados de rotacion aleatoria
    width_shift_range=0.2, # fraccion del total (1) para mover la imagen
    height_shift_range=0.2, # fraccion del total (1) para mover la imagen
    horizontal_flip=True, # girar las imagenes horizontalmente (eje vertical)
    # shear_range=0, # deslizamiento
    zoom_range=0.2, # rango de zoom
    # fill_mode='nearest', # como rellenar posibles nuevos pixeles
    # channel_shift_range=0.2 # cambios aleatorios en los canales de la imagen
)
```

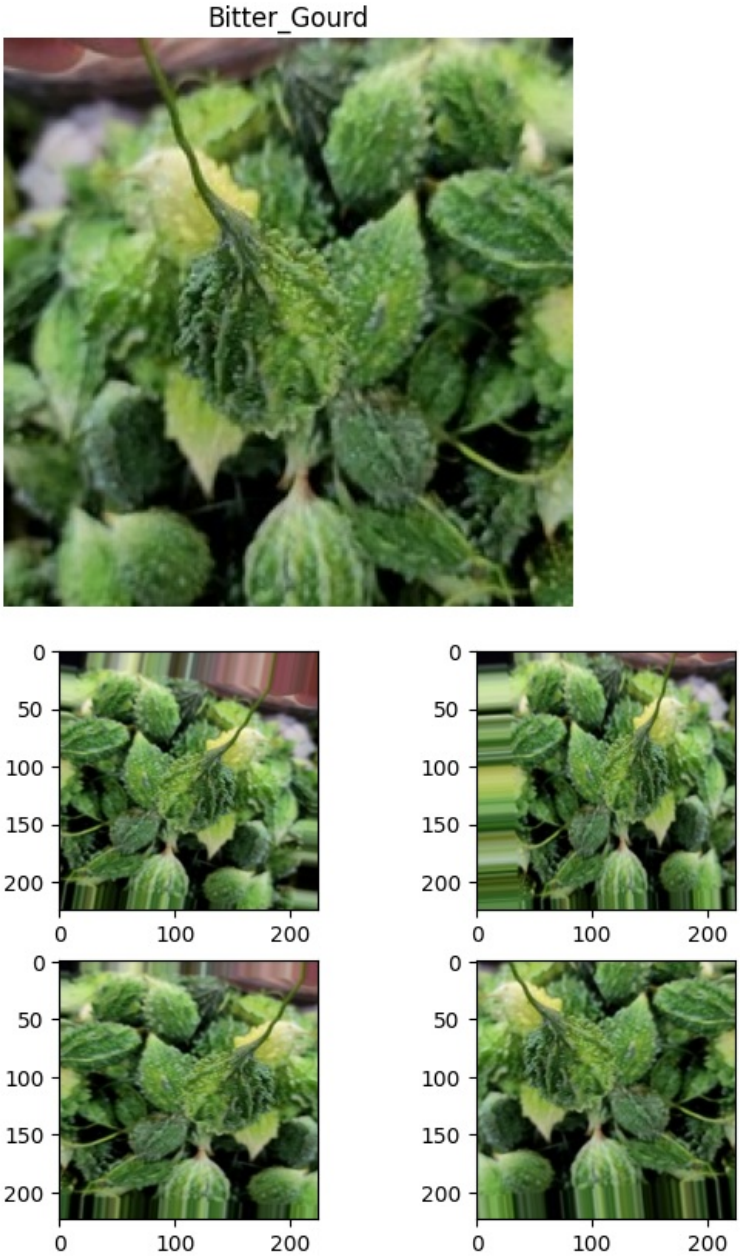
In [ ]:
```
# Carga del conjunto de test
# Los datos de train y validation se cargan durante el entrenamiento del modelo
import numpy as np

test_aug = load_dataset('/test', None)
x_test_aug = []
y_test_aug = []
# norm_test = test_aug.map(lambda x,y: (rescaling_layer(x), y))
for image, label in test_aug.take(len(test_aug)):
  x_test_aug.append(image)
  y_test_aug.append(label)

x_test_aug = np.array(x_test_aug)
y_test_aug = np.array(y_test_aug)

# Guardamos las clases en una variable
class_names_aug = test_aug.class_names
```

```
Found 3000 files belonging to 15 classes.
```

> Inspección de muestras generadas sintéticamente

In [ ]:
```
from tensorflow.keras.preprocessing import image
from tensorflow.keras.backend import expand_dims
import matplotlib.pyplot as plt
%matplotlib inline

# Tomamos uno de los ejemplos de imágenes de train
for images, labels in train_ds.take(1):
  sample = np.random.randint(len(images))
  sample_train_x = images[sample]
  sample_class = class_names_aug[labels[sample]]
plt.imshow(sample_train_x.numpy().astype('uint8'))
plt.title(sample_class)
plt.axis('off')
plt.show()

# Visualizamos las transformaciones hechas por la técnica de Data Augmentation
fig, axes = plt.subplots(2,2)
i = 0
for batch in datagen.flow(expand_dims(sample_train_x, axis=0),batch_size=1):
    #plt.figure(i)
    axes[i//2,i%2].imshow(image.array_to_img(batch[0]))
    i += 1
    if i == 4:
```

```
        break
plt.show()
```



Definición del modelo

In [ ]:
```
# Unimos las entradas y el modelo mediante la función Model con parámetros inputs y ouputs (Consultar la docume
model_aug = Model(inputs=inputs, outputs=predictions) #(X)

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
model_aug.compile(loss="categorical_crossentropy",
#                 loss='sparse_categorical_crossentropy',
                  optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08),
                  metrics=["accuracy"]) #(X)

callbacks_aug = [
    tf.keras.callbacks.ModelCheckpoint(filepath='/content/data_aug',
                                       monitor='val_loss',
                                       save_best_only=True,
                                       mode='min'),
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

num_epoch=20

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
# IMPORTANTE (Consultar la documentación)
H_aug = model_aug.fit(datagen.flow_from_directory(path + '/train',
                                                  target_size=(224, 224),
                                                  color_mode='rgb',
                                                  classes=class_names_aug,
```

```
                                        class_mode='categorical',
                                        batch_size=32,
                                        shuffle=True,
                                        seed=0),
                    # steps_per_epoch = len(norm_train)/ 128,
                    validation_data=datagen.flow_from_directory(path + '/validation',
                                        target_size=(224, 224),
                                        color_mode='rgb',
                                        classes=class_names_aug,
                                        class_mode='categorical',
                                        batch_size=32,
                                        shuffle=True,
                                        seed=0),
                    epochs=num_epoch, verbose=1, callbacks=callbacks_aug)
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Found 15000 images belonging to 15 classes.
Found 3000 images belonging to 15 classes.
Epoch 1/20
469/469 [==============================] - 309s 620ms/step - loss: 1.3553 - accuracy: 0.6007 - val_loss: 1.2031
- val_accuracy: 0.6950
Epoch 2/20
469/469 [==============================] - 305s 650ms/step - loss: 0.5570 - accuracy: 0.8233 - val_loss: 0.8004
- val_accuracy: 0.7783
Epoch 3/20
469/469 [==============================] - 297s 633ms/step - loss: 0.3912 - accuracy: 0.8771 - val_loss: 0.7667
- val_accuracy: 0.7770
Epoch 4/20
469/469 [==============================] - 283s 603ms/step - loss: 0.3023 - accuracy: 0.9016 - val_loss: 0.9474
- val_accuracy: 0.7470
Epoch 5/20
469/469 [==============================] - 312s 665ms/step - loss: 0.2483 - accuracy: 0.9198 - val_loss: 0.2422
- val_accuracy: 0.9133
Epoch 6/20
469/469 [==============================] - 277s 590ms/step - loss: 0.2107 - accuracy: 0.9323 - val_loss: 0.3644
- val_accuracy: 0.8860
Epoch 7/20
469/469 [==============================] - 278s 592ms/step - loss: 0.1864 - accuracy: 0.9395 - val_loss: 0.2470
- val_accuracy: 0.9210
Epoch 8/20
469/469 [==============================] - 280s 596ms/step - loss: 0.1779 - accuracy: 0.9432 - val_loss: 1.3944
- val_accuracy: 0.7090
Epoch 9/20
469/469 [==============================] - 319s 679ms/step - loss: 0.1581 - accuracy: 0.9481 - val_loss: 0.2124
- val_accuracy: 0.9323
Epoch 10/20
469/469 [==============================] - 300s 639ms/step - loss: 0.1555 - accuracy: 0.9521 - val_loss: 0.1739
- val_accuracy: 0.9490
Epoch 11/20
469/469 [==============================] - 278s 592ms/step - loss: 0.1363 - accuracy: 0.9547 - val_loss: 0.2057
- val_accuracy: 0.9470
Epoch 12/20
469/469 [==============================] - 304s 647ms/step - loss: 0.1343 - accuracy: 0.9576 - val_loss: 0.1294
- val_accuracy: 0.9557
Epoch 13/20
469/469 [==============================] - 315s 672ms/step - loss: 0.1218 - accuracy: 0.9603 - val_loss: 0.1183
- val_accuracy: 0.9617
Epoch 14/20
469/469 [==============================] - 276s 589ms/step - loss: 0.1144 - accuracy: 0.9631 - val_loss: 0.2989
- val_accuracy: 0.9117
Epoch 15/20
469/469 [==============================] - 281s 599ms/step - loss: 0.1232 - accuracy: 0.9618 - val_loss: 0.3839
- val_accuracy: 0.9083
Epoch 16/20
469/469 [==============================] - 310s 660ms/step - loss: 0.0950 - accuracy: 0.9687 - val_loss: 0.0940
- val_accuracy: 0.9707
Epoch 17/20
469/469 [==============================] - 277s 590ms/step - loss: 0.0912 - accuracy: 0.9725 - val_loss: 0.6575
- val_accuracy: 0.8500
Epoch 18/20
469/469 [==============================] - 309s 658ms/step - loss: 0.0996 - accuracy: 0.9681 - val_loss: 0.0756
- val_accuracy: 0.9793
Epoch 19/20
469/469 [==============================] - 307s 655ms/step - loss: 0.0767 - accuracy: 0.9751 - val_loss: 0.0712
- val_accuracy: 0.9820
Epoch 20/20
469/469 [==============================] - 288s 613ms/step - loss: 0.0905 - accuracy: 0.9733 - val_loss: 0.0917
- val_accuracy: 0.9727
```

Evaluación del modelo

En este modelo se han obtenido valores de f1-score perfectos para la clasificación de patata y calabaza amarga y casi perfecta para el

pimiento, la zanahoria y el pepino. Sin embargo, la clasificación de otros vegetales como la calabaza o el coliflor se ha visto impactada negativamente. Esto puede deberse a que las imágenes generadas durante el proceso de Data Augmentation han tenido poca representación de estos vegetales.

Destacamos neuvamente el pico de validation loss en la época ocho. El hecho de que el pico se repita en la misma época puede deberse a que el bache de validación (que no cambia entre modelos) contiene imágenes especialmente complicadas de detectar para la red.

In [ ]:
```
# Gráfica de accuracy y losses
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, num_epoch), H_aug.history["loss"], label="train_loss")
plt.plot(np.arange(0, num_epoch), H_aug.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, num_epoch), H_aug.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, num_epoch), H_aug.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

In [ ]:
```
# Evaluación del modelo
print("[INFO]: Evaluando el modelo de red neuronal...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = model_aug.predict(x_test_aug, batch_size=128) #(X)
# Sacamos el report para test
print(classification_report(y_test_aug, predictions.argmax(axis=1), target_names=class_names_aug))
```

```
[INFO]: Evaluando el modelo de red neuronal...
24/24 [==============================] - 12s 287ms/step
              precision    recall  f1-score   support

        Bean       0.96      0.99      0.98       200
Bitter_Gourd       0.99      0.91      0.95       200
Bottle_Gourd       1.00      1.00      1.00       200
     Brinjal       0.99      0.97      0.98       200
    Broccoli       0.67      1.00      0.80       200
     Cabbage       0.91      0.88      0.90       200
    Capsicum       0.98      1.00      0.99       200
      Carrot       0.99      1.00      0.99       200
 Cauliflower       1.00      0.86      0.93       200
    Cucumber       0.99      0.98      0.99       200
      Papaya       1.00      0.94      0.97       200
      Potato       1.00      0.99      1.00       200
     Pumpkin       0.98      0.86      0.92       200
      Radish       0.99      0.97      0.98       200
      Tomato       1.00      0.92      0.96       200

    accuracy                           0.95      3000
   macro avg       0.96      0.95      0.96      3000
weighted avg       0.96      0.95      0.96      3000
```

En caso de que se requiera utilizar el modelo recién entrenado (con el cual se aplicó Data augmentation), se ha guardado en la siguiente

ruta.

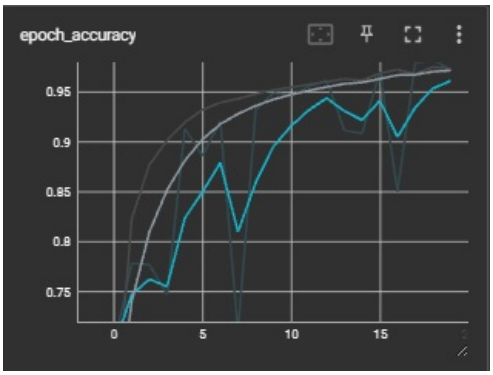
In [ ]:
```
# Guardamos el modelo si así lo deseamos
model_aug.save('/content/model_aug.h5')
```

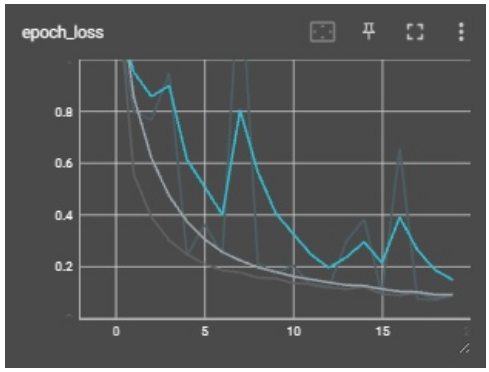
Luego de subirlo al entorno de colab, se puede volver a cargar el modelo y seguir entrenándolo si es necesario.

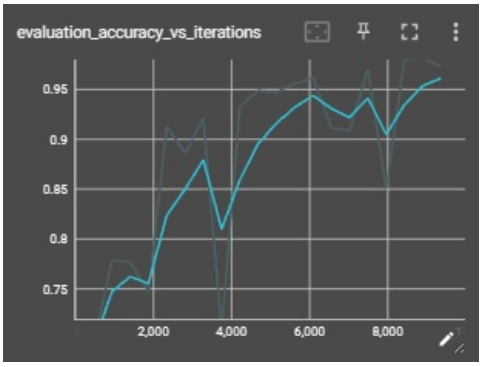
Enlace del archivo contenedor: Modelo entrenado aplicando Data Augmentation

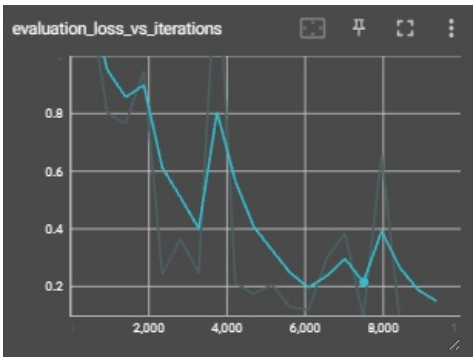
In [ ]:
```
from tensorflow.keras.models import load_model
new_model_aug = load_model('/content/model_aug.h5') # debe verificar la ruta

# Verifique que el estado esté preservado
# new_predictions = new_model_aug.predict(x_test_aug)
# np.testing.assert_allclose(predictions, new_predictions, rtol=1e-6, atol=1e-6)
```

> > Gráficas de Tensorboard mostrando métricas de desempeño del modelo en función de las épocas e iteraciones









4.2 Desarrollo y entrenamiento de un modelo a partir de redes preentrenada

A continuación se hace uso de dos redes preentrenados de ImageNet para aplicar el concepto de Transfer Learning y, posteriormente, el de Fine Tuning para mejorar el overfitting. Las redes a utilizar serán la VGG16 y la ResNet50.

4.2.1 Modelo a partir de la red VGG16

Preprocesamiento de imágenes de acuerdo a la red VGG16

En primer lugar, se normaliza el set de datos de la misma forma en la que los creadores de la red lo hicieron para su entrenamiento.

In [ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.applications import VGG16
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import classification_report
from google.colab import drive
from tensorflow.keras.layers import CategoryEncoding
from tensorflow.keras.applications import VGG16, imagenet_utils

# Conversión a one-hot encoding
OHE = CategoryEncoding(num_tokens=15, output_mode="one_hot")
```

```
# Aplicamos one-hot encoding a las etiquetas
norm_train = train_ds.map(lambda x, y: (imagenet_utils.preprocess_input(x), OHE(y)))
norm_val = val_ds.map(lambda x, y: (imagenet_utils.preprocess_input(x), OHE(y)))

# Separamos el conjunto de test entre imágenes y sus etiquetas
# Se utilizarán para evaluar la precisión del modelo
import numpy as np
x_test = []
y_test = []
norm_test = test_ds.map(lambda x,y: (imagenet_utils.preprocess_input(x), y)) # dejamos intactas las etiquetas (
for image, label in norm_test.take(len(norm_test)):
  x_test.append(image)
  y_test.append(label)

x_test = np.array(x_test)
y_test = np.array(y_test)
```

Carga del base model

Al cargar el modelo se asigna el parámetro include_top=False, ya que solo nos interesa hacer uso del base model de la red.

In [ ]:
```
# Seleccionar modelo preentrenado (VGG16 en este caso)
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_or
dering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 14714688 (56.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Definición del top model para Transfer Learning

Añadimos de manera secuencial un MLP de 256 neuronas en la capa oculta.

In [ ]:
```
# conectarlo a nueva parte densa
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

base_model.trainable = False # Evitar que los pesos se modifiquen en la parte convolucional -> TRANSFER LEARNIN
pre_trained_model = Sequential()
pre_trained_model.add(base_model)
pre_trained_model.add(layers.Flatten())
pre_trained_model.add(layers.Dense(256, activation='relu'))
pre_trained_model.add(layers.Dense(15, activation='softmax'))

pre_trained_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 7, 7, 512)         14714688

 flatten (Flatten)           (None, 25088)             0

 dense (Dense)               (None, 256)               6422784

 dense_1 (Dense)             (None, 15)                3855

=================================================================
Total params: 21141327 (80.65 MB)
Trainable params: 6426639 (24.52 MB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

Compilación del modelo

In [ ]:
```
# Import the necessary packages
import numpy as np
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense, Dropout, BatchNormalization, MaxP
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from google.colab import drive

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
pre_trained_model.compile(loss="categorical_crossentropy", optimizer=Adam(learning_rate=0.0005, weight_decay=0,

# Entrenamiento de la red
print("[INFO]: Entrenando la red...")
H_pre = pre_trained_model.fit(norm_train, batch_size=128, epochs=20, validation_data=norm_val)
```

```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
469/469 [==============================] - 116s 223ms/step - loss: 0.6804 - accuracy: 0.9620 - val_loss: 0.2111
- val_accuracy: 0.9837
Epoch 2/20
469/469 [==============================] - 89s 188ms/step - loss: 0.1295 - accuracy: 0.9921 - val_loss: 0.0841 -
val_accuracy: 0.9957
Epoch 3/20
469/469 [==============================] - 87s 185ms/step - loss: 0.1191 - accuracy: 0.9939 - val_loss: 0.3199 -
val_accuracy: 0.9887
Epoch 4/20
469/469 [==============================] - 79s 168ms/step - loss: 0.1348 - accuracy: 0.9945 - val_loss: 0.1734 -
val_accuracy: 0.9943
Epoch 5/20
469/469 [==============================] - 82s 173ms/step - loss: 0.1020 - accuracy: 0.9959 - val_loss: 0.2836 -
val_accuracy: 0.9900
Epoch 6/20
469/469 [==============================] - 82s 174ms/step - loss: 0.0882 - accuracy: 0.9969 - val_loss: 0.1114 -
val_accuracy: 0.9967
Epoch 7/20
469/469 [==============================] - 83s 177ms/step - loss: 0.0585 - accuracy: 0.9977 - val_loss: 0.3404 -
val_accuracy: 0.9943
Epoch 8/20
469/469 [==============================] - 84s 178ms/step - loss: 0.0631 - accuracy: 0.9975 - val_loss: 0.3944 -
val_accuracy: 0.9907
Epoch 9/20
469/469 [==============================] - 90s 192ms/step - loss: 0.0695 - accuracy: 0.9973 - val_loss: 0.1774 -
val_accuracy: 0.9957
Epoch 10/20
469/469 [==============================] - 83s 177ms/step - loss: 0.1278 - accuracy: 0.9973 - val_loss: 0.2383 -
val_accuracy: 0.9963
Epoch 11/20
469/469 [==============================] - 83s 175ms/step - loss: 0.0173 - accuracy: 0.9990 - val_loss: 0.1284 -
val_accuracy: 0.9970
Epoch 12/20
469/469 [==============================] - 90s 192ms/step - loss: 0.0363 - accuracy: 0.9990 - val_loss: 0.1041 -
val_accuracy: 0.9980
Epoch 13/20
469/469 [==============================] - 83s 176ms/step - loss: 0.0345 - accuracy: 0.9991 - val_loss: 0.1537 -
val_accuracy: 0.9980
Epoch 14/20
469/469 [==============================] - 90s 192ms/step - loss: 0.0932 - accuracy: 0.9982 - val_loss: 0.3380 -
val_accuracy: 0.9947
Epoch 15/20
469/469 [==============================] - 90s 190ms/step - loss: 0.0959 - accuracy: 0.9973 - val_loss: 0.2997 -
val_accuracy: 0.9943
Epoch 16/20
469/469 [==============================] - 83s 177ms/step - loss: 0.1398 - accuracy: 0.9975 - val_loss: 0.2988 -
val_accuracy: 0.9973
Epoch 17/20
469/469 [==============================] - 84s 179ms/step - loss: 0.0026 - accuracy: 0.9999 - val_loss: 0.2150 -
val_accuracy: 0.9970
Epoch 18/20
469/469 [==============================] - 90s 191ms/step - loss: 0.0124 - accuracy: 0.9997 - val_loss: 0.2563 -
val_accuracy: 0.9960
Epoch 19/20
469/469 [==============================] - 83s 177ms/step - loss: 0.0855 - accuracy: 0.9983 - val_loss: 0.5600 -
val_accuracy: 0.9937
Epoch 20/20
469/469 [==============================] - 90s 191ms/step - loss: 0.0676 - accuracy: 0.9988 - val_loss: 0.2971 -
val_accuracy: 0.9950
```

Evaluación del modelo de Transfer Learning

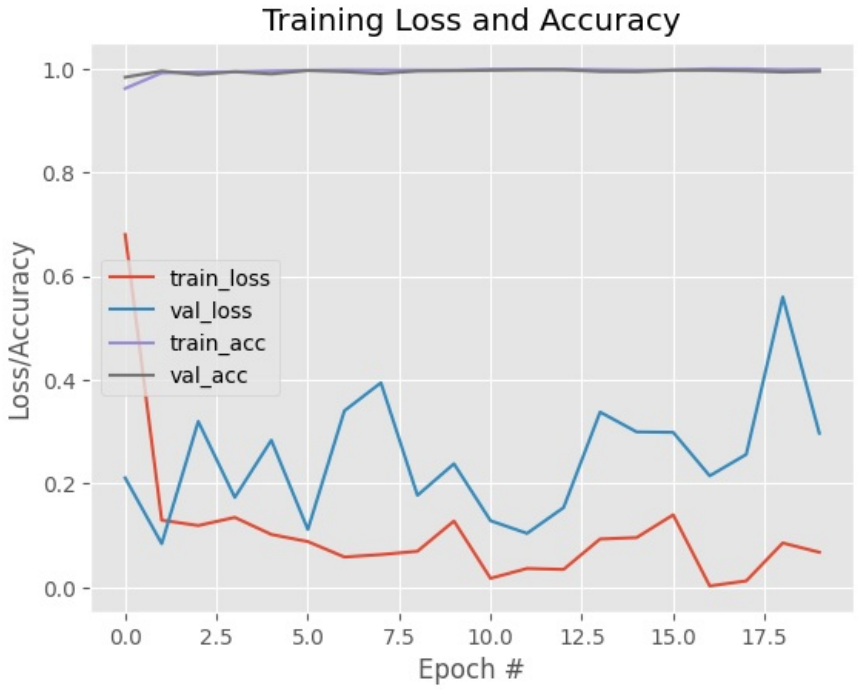
El modelo da resultados excelentes en el test. Esto es sorprendente teniendo en cuenta la simplicidad del top model que se ha añadido y es muestra de la buena calidad del base model, cuyas capas convolucionales contienen pesos entrenados con una cantidad inmensa de datos y son excelentes detectando características base como bordes y texturas. Tenemos unos valores f1-score perfectos para la detección de repollo, pimiento, zanahoria, calabaza y rábano. Esto ya supera con creces los resultados obtenidos en la red from scratch, teniendo todos los f1-score mayores o iguales a 0.99.

El modelo entra en un mínimo local al no observar mejora de la pérdida tras completar la mitad del proceso del entrenamiento. Se buscará salir de este mínimo local aplicando Fine Tuning.

In [ ]:
```
# clases
class_names = test_ds.class_names
class_names

# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
```

```
predictions = pre_trained_model.predict(x_test, batch_size=128)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))

# Gráficas
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H_pre.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H_pre.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H_pre.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H_pre.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

```
[INFO]: Evaluando el modelo...
24/24 [==============================] - 29s 765ms/step
              precision    recall  f1-score   support

        Bean       1.00      0.97      0.99       200
Bitter_Gourd       0.99      0.99      0.99       200
Bottle_Gourd       0.97      1.00      0.99       200
     Brinjal       0.99      0.99      0.99       200
    Broccoli       0.98      0.99      0.99       200
     Cabbage       0.99      1.00      1.00       200
    Capsicum       1.00      1.00      1.00       200
      Carrot       0.99      1.00      1.00       200
 Cauliflower       1.00      0.98      0.99       200
    Cucumber       1.00      0.97      0.98       200
      Papaya       0.99      1.00      0.99       200
      Potato       1.00      0.99      0.99       200
     Pumpkin       0.99      1.00      1.00       200
      Radish       0.99      1.00      1.00       200
      Tomato       1.00      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
```



Aplicación de Fine Tuning y EarlyStopping para reducir overfitting

El base model se congela hasta una capa específica ('block4_pool') para evitar ajustes innecesarios. Adicionalmente se aplica un Early Stopping con patience=3.

In [ ]:
```
# Imports que vamos a necesitar

from tensorflow.keras.applications import VGG16, imagenet_utils
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dropout, Flatten, Dense
from tensorflow.keras import Model
import matplotlib.pyplot as plt
```

```
from sklearn.metrics import classification_report
import numpy as np

####### BASE MODEL ######################
# Importamos VGG16 con pesos de imagenet y sin top_model especificando tamaño de entrada de datos
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Mostramos la arquitectura
base_model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 [==============================] - 0s 0us/step
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 224, 224, 3)]     0

 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792

 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928

 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0

 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856

 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584

 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0

 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168

 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080

 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080

 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0

 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160

 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0

 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0

=================================================================
Total params: 14714688 (56.13 MB)
Trainable params: 14714688 (56.13 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [ ]:

```
%%capture
# congelar capas
for layer in base_model.layers:
  if layer.name == 'block4_pool':
    break
  layer.trainable = False
  print('Capa ' + layer.name + ' congelada...')

# Cogemos la última capa del model y le añadimos nuestro clasificador (top_model)
last = base_model.layers[-1].output # ultima capa del base model
x = Flatten()(last)
x = Dense(1024, activation='relu', name='fc1')(x)
x = Dropout(0.3)(x)
x = Dense(256, activation='relu', name='fc2')(x)
x = Dense(15, activation='softmax', name='predictions')(x)
model = Model(base_model.input, x)

# Compilamos el modelo
model.compile(optimizer=Adam(learning_rate=0.0005, weight_decay=0, beta_1=0.9, beta_2=0.999, epsilon=1e-08), los

model.summary()

my_callbacks = [
```

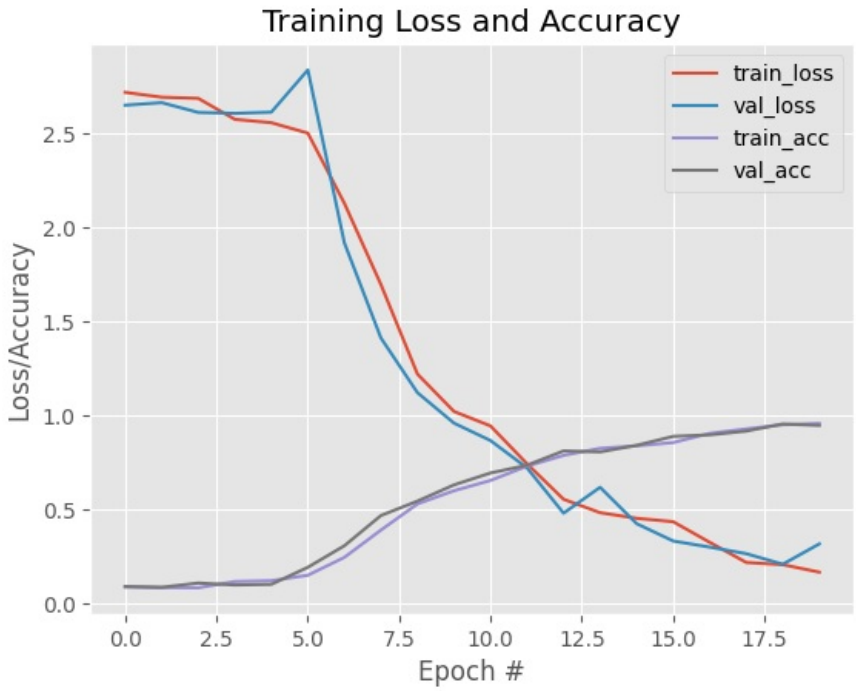
```
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     patience=3, # Valor de patience con mejor resultado entre 2-5
                                     mode='min'),
    tf.keras.callbacks.ModelCheckpoint(filepath='/content/',
                                       monitor='val_accuracy',
                                       save_weights_only=True,
                                       mode='max'),
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
]

n_epochs=20

# Entrenamos el modelo
H = model.fit(norm_train, validation_data = norm_val, batch_size=128,
              epochs=n_epochs, verbose=1, callbacks=my_callbacks)
```

Evaluación del modelo tras el Fine Tuning

Aunque el overfitting haya mejorado, se observa que el accuracy en la etapa de test pasa de 0.99 a 0.94 al aplicar Fine Tuning. Esto puede deberse a que el hecho de reentrenar el último bloque convolucional redujo su capacidad de generalización, quizás por la relativa pequeña cantidad de imágenes usadas para entrenar o porque las imágenes usadas para entrenar al VGG16 hayan sido muy distintas a las usadas en estos entrenamientos.

In [ ]:
```
# clases
class_names = test_ds.class_names
class_names

# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
predictions = model.predict(x_test, batch_size=128)
# Obtener el report de clasificación
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))


# Gráficas
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

```
[INFO]: Evaluando el modelo...
24/24 [==============================] - 28s 724ms/step
              precision    recall  f1-score   support

        Bean       0.87      0.95      0.91       200
Bitter_Gourd       0.99      0.87      0.93       200
Bottle_Gourd       0.99      0.98      0.99       200
     Brinjal       0.89      0.99      0.94       200
    Broccoli       0.98      0.94      0.96       200
     Cabbage       0.82      0.72      0.77       200
    Capsicum       1.00      1.00      1.00       200
      Carrot       1.00      0.99      0.99       200
 Cauliflower       0.79      0.84      0.81       200
    Cucumber       0.93      0.94      0.94       200
      Papaya       0.98      0.99      0.99       200
      Potato       0.99      0.99      0.99       200
     Pumpkin       0.96      1.00      0.98       200
      Radish       1.00      1.00      1.00       200
      Tomato       0.99      0.94      0.97       200

    accuracy                           0.94      3000
   macro avg       0.95      0.94      0.94      3000
weighted avg       0.95      0.94      0.94      3000
```



4.2. Modelo a partir de la red ResNet50

Se empleará adicionalmente, la arquitectura pre-entrenada `ResNet50`, ya que al igual que VGG16, de acuerdo con la documentación, el input shape tiene que ser `(224,224,3)`, siendo justamente la dimensión que tiene las imágenes de nuestro dataset.

Preprocesamiento de imágenes de acuerdo a la red ResNet50

In [ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import classification_report
from google.colab import drive
from tensorflow.keras.layers import CategoryEncoding
from tensorflow.keras.applications import ResNet50, imagenet_utils

# Conversión a one-hot encoding
OHE = CategoryEncoding(num_tokens=15, output_mode="one_hot")

# Aplicamos one-hot encoding a las etiquetas
norm_train = train_ds.map(lambda x, y: (imagenet_utils.preprocess_input(x), OHE(y)))
norm_val = val_ds.map(lambda x, y: (imagenet_utils.preprocess_input(x), OHE(y)))

# Separamos el conjunto de test entre imágenes y sus etiquetas
# Se utilizarán para evaluar la precisión del modelo
import numpy as np
x_test = []
y_test = []
norm_test = test_ds.map(lambda x,y: (imagenet_utils.preprocess_input(x), y)) # dejamos intactas las etiquetas (
for image, label in norm_test.take(len(norm_test)):
  x_test.append(image)
  y_test.append(label)

x_test = np.array(x_test)
y_test = np.array(y_test)
```

Carga del base model

Carga del base model

In [ ]:
```
%%capture
# Seleccionar modelo preentrenado (ResNet50 en este caso)
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

base_model.summary()
```

Definición del top model para Transfer Learning

In [ ]:
```
# conectarlo a nueva parte densa
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers

base_model.trainable = False # Evitar que los pesos se modifiquen en la parte convolucional -> TRANSFER LEARNIN
pre_trained_model = Sequential()
pre_trained_model.add(base_model)
pre_trained_model.add(layers.Flatten())
pre_trained_model.add(layers.Dense(256, activation='relu'))
pre_trained_model.add(layers.Dense(15, activation='softmax'))

pre_trained_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 flatten (Flatten)           (None, 100352)            0

 dense (Dense)               (None, 256)               25690368

 dense_1 (Dense)             (None, 15)                3855

=================================================================
Total params: 49281935 (188.00 MB)
Trainable params: 25694223 (98.02 MB)
Non-trainable params: 23587712 (89.98 MB)
_________________________________________________________________
```

Compilación del modelo aplicando Early Stopping

In [ ]:
```
# Import the necessary packages
import numpy as np
from tensorflow.keras import backend as K
from tensorflow.keras.layers import Input, Conv2D, Activation, Flatten, Dense, Dropout, BatchNormalization, Maxl
from tensorflow.keras.models import Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import SGD, Adam
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from google.colab import drive

# Compilar el modelo
print("[INFO]: Compilando el modelo...")
pre_trained_model.compile(loss="categorical_crossentropy", optimizer=Adam(learning_rate=0.0005, weight_decay=0,

# Entrenamiento de la red
my_callbacks = [
    tf.keras.callbacks.EarlyStopping(monitor='val_loss', #Después de algunas épocas el modelo no mejora
                                     patience=3,
                                     mode='min'),
    tf.keras.callbacks.TensorBoard(log_dir='./logs')
]
print("[INFO]: Entrenando la red...")
H_pre = pre_trained_model.fit(norm_train, batch_size=128, epochs=20, validation_data=norm_val, callbacks=my_cal
```

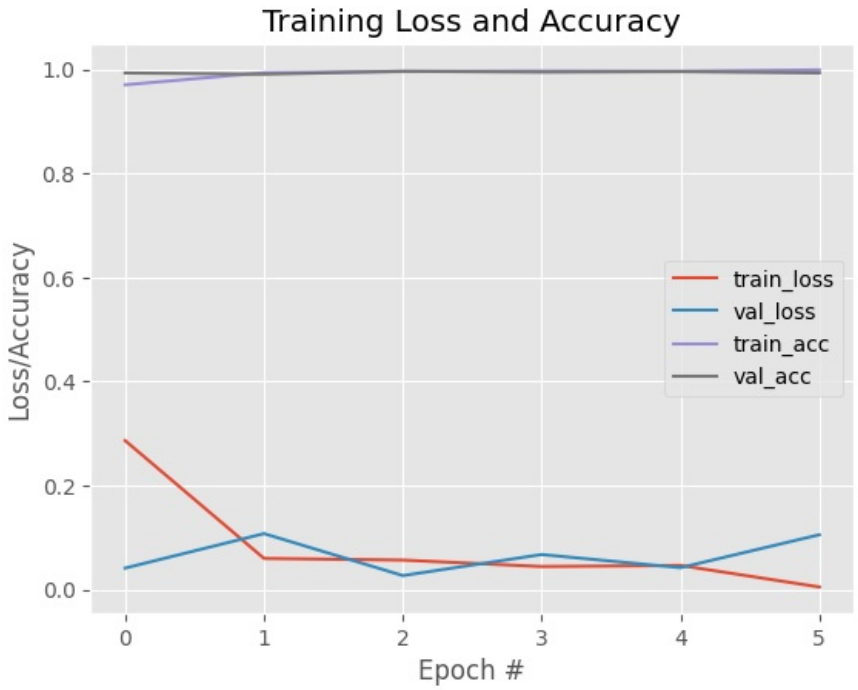
```
[INFO]: Compilando el modelo...
[INFO]: Entrenando la red...
Epoch 1/20
469/469 [==============================] - 89s 162ms/step - loss: 0.2866 - accuracy: 0.9702 - val_loss: 0.0417 -
val_accuracy: 0.9930
Epoch 2/20
469/469 [==============================] - 71s 150ms/step - loss: 0.0603 - accuracy: 0.9935 - val_loss: 0.1079 -
val_accuracy: 0.9907
Epoch 3/20
469/469 [==============================] - 80s 170ms/step - loss: 0.0570 - accuracy: 0.9958 - val_loss: 0.0273 -
val_accuracy: 0.9963
Epoch 4/20
469/469 [==============================] - 70s 148ms/step - loss: 0.0447 - accuracy: 0.9966 - val_loss: 0.0677 -
val_accuracy: 0.9947
Epoch 5/20
469/469 [==============================] - 80s 169ms/step - loss: 0.0467 - accuracy: 0.9962 - val_loss: 0.0425 -
val_accuracy: 0.9957
Epoch 6/20
469/469 [==============================] - 72s 152ms/step - loss: 0.0054 - accuracy: 0.9990 - val_loss: 0.1057 -
val_accuracy: 0.9930
```

Evaluación del modelo de Transfer Learning

Los resultados de este modelo son muy similares a los obtenidos con el Transfer Learning del VGG16 y con menor overfitting. Tiene f1-score perfecto para seis vegetales (brócoli, pimiento, zanahora,patata, rábano y tomate).

In [ ]:
```python
# clases
class_names = test_ds.class_names
class_names

# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
# Efectuamos la predicción (empleamos el mismo valor de batch_size que en training)
predictions = pre_trained_model.predict(x_test, batch_size=128)
# Sacamos el report para test
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))

# Gráficas
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 6), H_pre.history["loss"], label="train_loss")
plt.plot(np.arange(0, 6), H_pre.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 6), H_pre.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 6), H_pre.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

```
[INFO]: Evaluando el modelo...
24/24 [==============================] - 17s 500ms/step
              precision    recall  f1-score   support

        Bean       1.00      0.99      0.99       200
Bitter_Gourd       0.98      0.99      0.99       200
Bottle_Gourd       1.00      1.00      1.00       200
     Brinjal       0.96      1.00      0.98       200
    Broccoli       1.00      1.00      1.00       200
     Cabbage       0.99      1.00      0.99       200
    Capsicum       1.00      1.00      1.00       200
      Carrot       1.00      1.00      1.00       200
 Cauliflower       1.00      0.96      0.98       200
    Cucumber       0.99      0.99      0.99       200
      Papaya       1.00      0.97      0.99       200
      Potato       1.00      1.00      1.00       200
     Pumpkin       1.00      0.98      0.99       200
      Radish       1.00      1.00      1.00       200
      Tomato       1.00      0.99      1.00       200

    accuracy                           0.99      3000
   macro avg       0.99      0.99      0.99      3000
weighted avg       0.99      0.99      0.99      3000
```



Aplicación de Fine Tuning y dropout para reducir overfitting

En esta etapa de Fine Tuning se congela el modelo hasta la capa conv4_block6_out y se redefine el top model para añadir una capa oculta adicional de 1024 neuronas con un dropout del 30%.

In [ ]:
```
%%capture
# Imports que vamos a necesitar

from tensorflow.keras.applications import ResNet50, imagenet_utils
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dropout, Flatten, Dense
from tensorflow.keras import Model
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
import numpy as np

####### BASE MODEL ######################
# Cargamos ResNet50 con pesos de imagenet y sin top_model especificando tamaño de entrada de datos
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(224, 224, 3))

# Mostramos la arquitectura
base_model.summary()
```

In [ ]:
```
for layer in base_model.layers:
  if layer.name == 'conv4_block6_out':
    break
  layer.trainable = False
  print('Capa ' + layer.name + ' congelada...')

# Cogemos la última capa del model y le añadimos nuestro clasificador (top_model)
last = base_model.layers[-1].output # ultima capa del base model
x = Flatten()(last)
x = Dense(1024, activation='relu', name='fc1')(x)
x = Dropout(0.3)(x)
x = Dense(256, activation='relu', name='fc2')(x)
x = Dense(15, activation='softmax', name='predictions')(x)
model = Model(base_model.input, x)

# Compilamos el modelo
model.compile(optimizer=Adam(learning_rate=0.0005, weight_decay=0, beta_1=0.9, beta_2=0.999, epsilon=1e-08), los

model.summary()

# Entrenamos el modelo
```

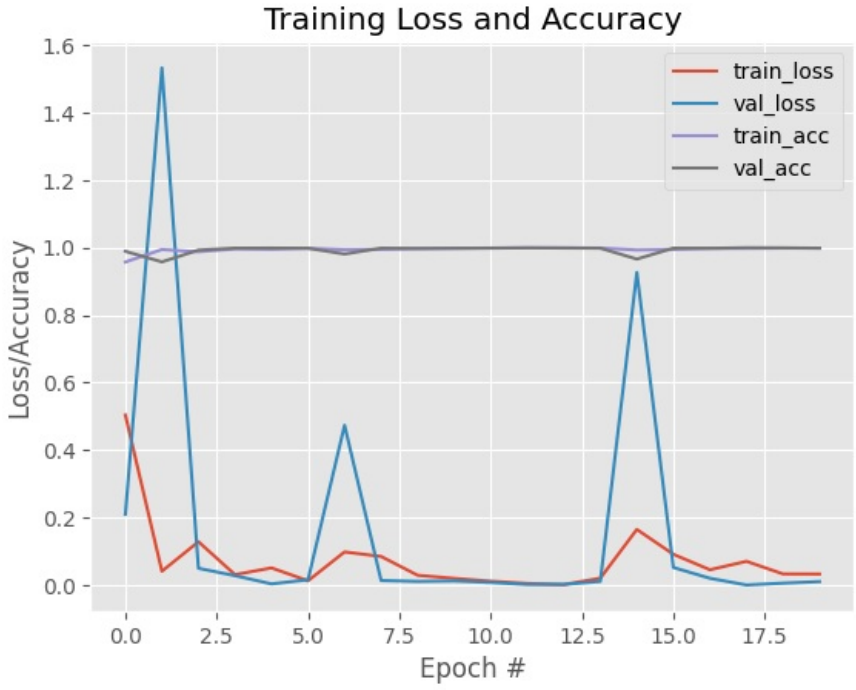
```
H = model.fit(norm_train, validation_data = norm_val, batch_size=128, epochs=20, verbose=1)
```

Evaluación del modelo tras el Fine Tuning

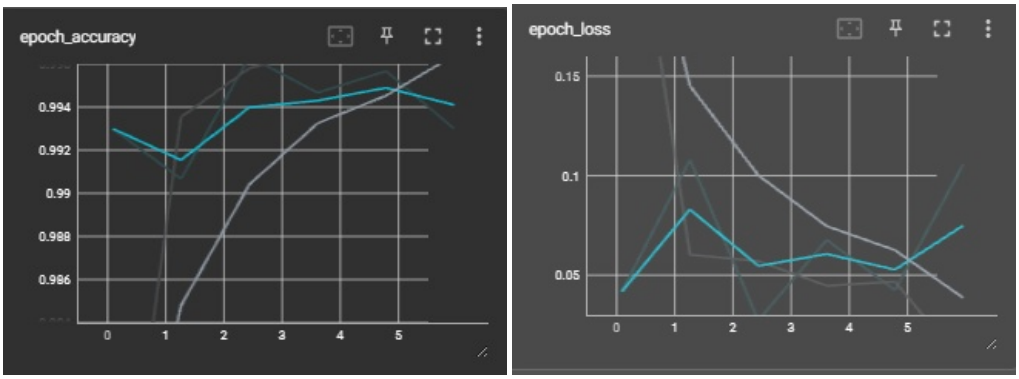
Este es, con diferencia el mejor modelo generado en este proyecto, con valores f1-score perfectos para todos los vegetales, aunque presenta picos de validation loss.

In [ ]:
```
# clases
class_names = test_ds.class_names
class_names

# Evaluación del modelo
print("[INFO]: Evaluando el modelo...")
predictions = model.predict(x_test, batch_size=128)
# Obtener el report de clasificación
print(classification_report(y_test, predictions.argmax(axis=1), target_names=class_names))


# Gráficas
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, 20), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, 20), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
plt.show()
```

```
[INFO]: Evaluando el modelo...
24/24 [==============================] - 10s 398ms/step
              precision    recall  f1-score   support

        Bean       1.00      1.00      1.00       200
Bitter_Gourd       1.00      0.99      1.00       200
Bottle_Gourd       1.00      1.00      1.00       200
     Brinjal       1.00      0.99      1.00       200
    Broccoli       1.00      0.99      1.00       200
     Cabbage       1.00      1.00      1.00       200
    Capsicum       1.00      1.00      1.00       200
      Carrot       1.00      1.00      1.00       200
 Cauliflower       1.00      1.00      1.00       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      0.99      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
```



> Gráficas de Tensorboard mostrando métricas de desempeño del modelo en función de las épocas



5. Conclusiones

En este proyecto se ha generado una serie de modelos de redes neuronales convolucionales utilizando distintas técnicas de regularización para la clasificación de imágenes de quince variedades de vegetales.

Los modelos construidos desde cero mostraron tendencia al overfitting con mejoras notables al introducir las técnicas de batch normalization y dropout. Las técnicas de Early Stopping no mejoró el rendimiento tanto como se esperaba pero esto puede deberse a los movimientos erráticos en el validation loss que se han observado en la mayoría de los modelos, lo que puede suponer que introducir una interrupción temprana al proceso de entrenamiento no permita a la red recomponerse correctamente de estos picos de pérdida para lograr converger satisfactoriamente. En cuanto a la técnica de Data Augmentation, aunque mejoró la clasificación para algunos vegetales, empeoró el accuracy general en la fase test. El riesgo de utilizar Data Augmentation es que puede resultar en overfitting para los datos generados de manera sintéticamente, lo que podría haber afectado a los vegetales poco representados durante la generación.

Los modelos entrenados mediante Transfer Learning han dado resultados superiores a los del modelo construido desde cero. El modelo basado en ResNet50 se destacó como el más exitoso, obteniendo una precisión perfecta para todas las variedades de vegetales. Esto tiene sentido al ser este un modelo mucho más complejo y profundoo en cuanto a arquitectura respecto al VGG16. ResNet50 está compuesto de 50 capas convolucionales y el VGG16 tiene 16. Por esta razón se presume, que el ResNet50 ha respondido mucho mejor al Fine Tuning, al tener las capas que detectan características básicas más alejadas de las capas especializadas se pueden reentrenar las últimas capas de manera que estas se adapten mucho mejor a los nuevos datos de entrenamiento. En el caso de VGG16, entrenar el último bloque convolucional supondría sobreescribir los pesos que, al contar con poca profundidad en la red, aún son muy importantes para extraer características básicas de las imágenes.

Finalmente, se ha dejado en evidencia lo importante que es experimentar con distintas arquitecturas de redes convolucionales para obtener resultados satisfactorios aún contando con limitaciones importantes como una cantidad relativamente pequeña de datos y poco poder de procesamiento.