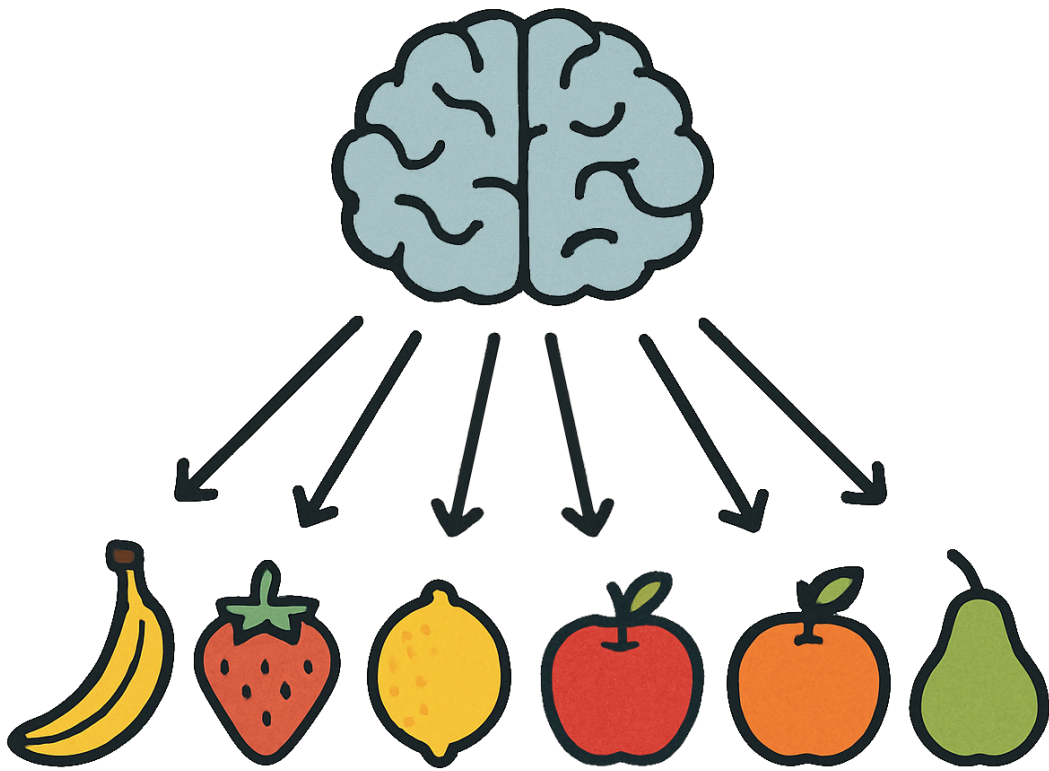


Programación de IA

Proyecto con
Redes Neuronales



Índice:

Análisis del dataset	3
Primer modelo	5
Cargar imágenes con ImageDataGenerator	5
Creación del modelo	6
Entrenamiento	7
Pruebas	8
Segundo modelo mejorado	9
Cargar imágenes con ImageDataGenerator	9
Creación del modelo	10
Entrenamiento	11
Pruebas	12
Pruebas con servidor Flask	13
Pruebas con Gradio	15
Posible método de mejorar el modelo	18

Análisis del dataset

El [dataset](#) se compone de 6 clases: banana, fresa, limón, mandarina, manzana y pera.

Todas las clases tienen 63 fotos de entrenamiento, 10 de validación y 2 de test, dando un total de 450 fotos.

A continuación las dos primeras fotos del conjunto de entrenamiento de cada clase:

Banana:



Fresa:



Limón:



Mandarina:



Manzana:



Pera:



En cada clase se han usado 3 o más frutas distintas.

En la clase manzana hay manzanas rojas y amarillas. He añadido amarillas a propósito para ver si las diferencia bien en caso de usar una CNN que distinga colores.

Primer modelo

Cargar imágenes con ImageDataGenerator

El primer modelo fue entrenado con estos **ImageDataGenerators**:

```
train_datagen_bn = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=15,  
    zoom_range=0.2,  
    width_shift_range=0.1,  
    height_shift_range=0.1,  
    horizontal_flip=True,  
    fill_mode='nearest'  
)  
  
val_datagen_bn = ImageDataGenerator(rescale=1./255)  
  
train_generator_bc = train_datagen_bn.flow_from_directory(  
    train_data_dir,  
    target_size=(128, 128),  
    batch_size=32,  
    class_mode='categorical',  
    color_mode='grayscale',  
    shuffle=True  
)  
  
validation_generator_bc = val_datagen_bn.flow_from_directory(  
    val_data_dir,  
    target_size=(128, 128),  
    batch_size=32,  
    class_mode='categorical',  
    color_mode='grayscale',  
    shuffle=True  
)
```

Las imágenes se reducen a 128 x 128, una resolución más pequeña para trabajar más fácilmente con ellas pero lo suficientemente grandes para no perder muchos detalles.

El **ImageDataGenerator** estandariza los valores de píxeles de [0, 255] a [0, 1], las pone en blanco y negro, rota aleatoriamente las imágenes hasta $\pm 15^\circ$, aplica un zoom aleatorio entre 80% y 120% del tamaño original, desplaza horizontalmente y verticalmente la imagen hasta un 10% del ancho y alto, voltea horizontalmente algunas imágenes y rellena las fotos que han sufrido transformaciones copiando píxeles cercanos.

Creación del modelo

El **CNN** usado tiene la siguiente composición:

```
input_shape = (128, 128, 1)

model_bc = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Conv2D(16, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(32, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dropout(0.4),
    layers.Dense(64, activation='relu'),
    layers.Dense(6, activation='softmax')
])

model_bc.compile(optimizer=keras.optimizers.Adam(),
loss='categorical_crossentropy', metrics=['accuracy'])
```

Capas utilizadas:

1. **Input Layer**: entrada de imágenes en escala de grises de 128x128 píxeles, con 1 canal.
2. **Conv2D** (16 filtros, 3x3): aplica 16 filtros de tamaño 3x3 y extrae bordes y texturas simples. Salida de 126x126x16.
3. **MaxPooling2D** (2x2): Reduce el tamaño espacial a la mitad. Salida de 63x63x16.
4. **Conv2D** (32 filtros, 3x3) + ReLU: detecta patrones complejos. Salida de 61x61x32.
5. **MaxPooling2D** (2x2): 30x30x32.
6. **Conv2D** (64 filtros, 3x3) + ReLU: Aumenta profundidad de la representación. Salida de 28x28x64.
7. **MaxPooling2D** (2x2): Reduce a 14x14x64.
8. **Flatten**: Convierte 14x14x64 = 12,544 elementos a un solo vector 1D.
9. **Dropout** (0.4): Apaga aleatoriamente el 40% de las neuronas en entrenamiento para evitar overfitting.
10. **Dense** (64 neuronas) + ReLU: Capa que aprende combinaciones de las anteriores características extraídas.
11. **Dense** (6 neuronas) + Softmax: Proporciona una salida para las 6 clases. Ideal para clasificación multiclase con **categorical_crossentropy**.

Compilación:

1. **Adam**: optimizador adaptativo eficiente.
2. **Loss**: adecuada para etiquetas codificadas one-hot.
3. **Métrica**: precisión (accuracy) durante entrenamiento y evaluación.

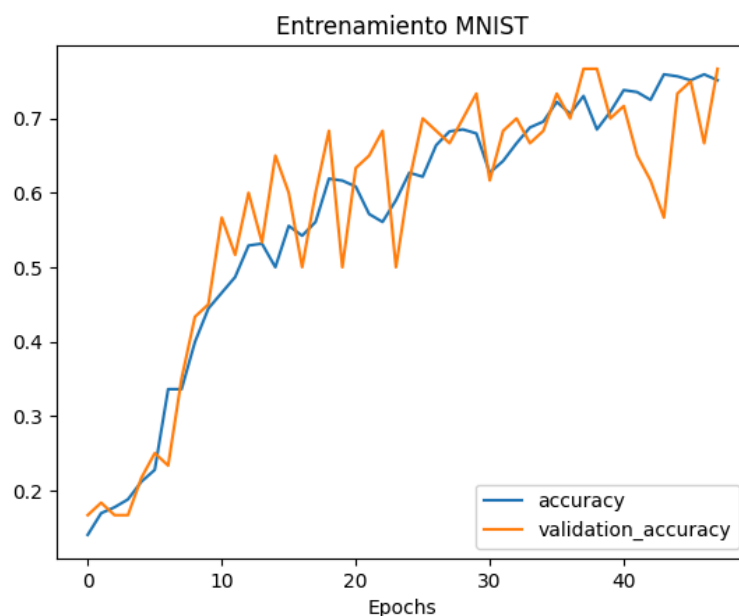
Entrenamiento

El modelo es entrenado con un **EarlyStopping** que cuando en 10 épocas el **val_accuracy** no sube, se queda con la mejor época.

```
es = EarlyStopping(monitor='val_accuracy', mode='max', verbose=1,
patience=10, restore_best_weights=True)

history_bc = model_bc.fit(train_generator_bc,
epochs=200,
validation_data=validation_generator_bc,
callbacks=[es])
```

```
12/12 ————— 79s 5s/step - accuracy: 0.6805 - loss: 0.8412 - val_accuracy: 0.7000 - val_loss: 1.0682
Epoch 38/200
12/12 ————— 57s 5s/step - accuracy: 0.6946 - loss: 0.7739 - val_accuracy: 0.7667 - val_loss: 0.9057
Epoch 39/200
12/12 ————— 56s 5s/step - accuracy: 0.6563 - loss: 0.8826 - val_accuracy: 0.7667 - val_loss: 0.9297
Epoch 40/200
12/12 ————— 55s 5s/step - accuracy: 0.7098 - loss: 0.7888 - val_accuracy: 0.7000 - val_loss: 1.0396
Epoch 41/200
12/12 ————— 82s 5s/step - accuracy: 0.7233 - loss: 0.6874 - val_accuracy: 0.7167 - val_loss: 0.9995
Epoch 42/200
12/12 ————— 56s 5s/step - accuracy: 0.7433 - loss: 0.6518 - val_accuracy: 0.6500 - val_loss: 1.3076
Epoch 43/200
12/12 ————— 55s 5s/step - accuracy: 0.7279 - loss: 0.7866 - val_accuracy: 0.6167 - val_loss: 1.3420
Epoch 44/200
12/12 ————— 59s 5s/step - accuracy: 0.7754 - loss: 0.6207 - val_accuracy: 0.5667 - val_loss: 1.3893
Epoch 45/200
12/12 ————— 81s 5s/step - accuracy: 0.7568 - loss: 0.6886 - val_accuracy: 0.7333 - val_loss: 1.0130
Epoch 46/200
12/12 ————— 56s 5s/step - accuracy: 0.7640 - loss: 0.5937 - val_accuracy: 0.7500 - val_loss: 1.0782
Epoch 47/200
12/12 ————— 55s 5s/step - accuracy: 0.7499 - loss: 0.7012 - val_accuracy: 0.6667 - val_loss: 1.2145
Epoch 48/200
12/12 ————— 82s 5s/step - accuracy: 0.7555 - loss: 0.6215 - val_accuracy: 0.7667 - val_loss: 1.0314
Epoch 48: early stopping
Restoring model weights from the end of the best epoch: 38.
```



En este entrenamiento ha llegado a un **accuracy** de 0.6956 y **val_accuracy** de 0.7667, en otros entrenamientos consiguió alrededor del 0.65 de **val_accuracy**.

Pruebas

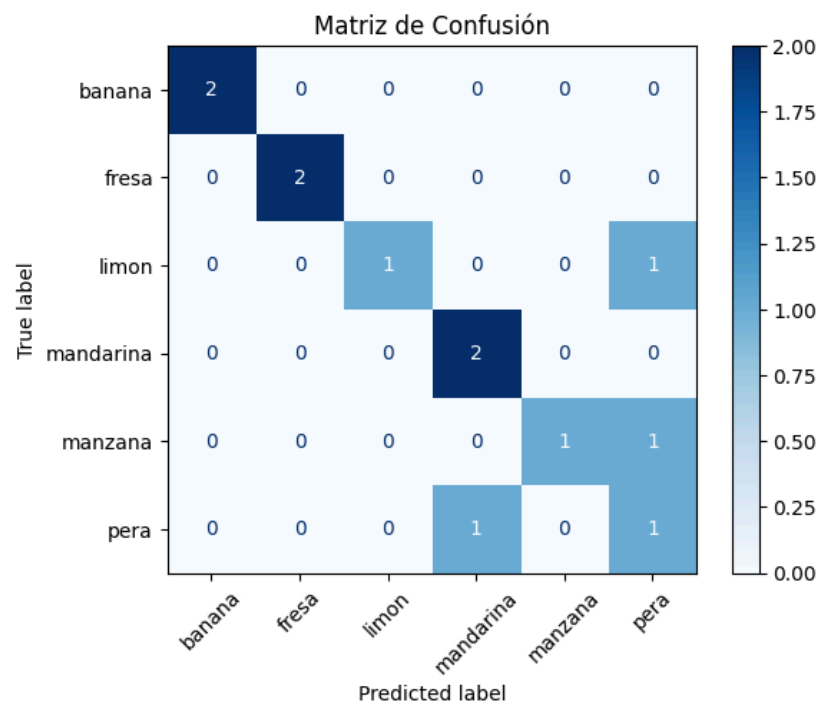
```
Found 12 images belonging to 6 classes.
12/12 2s 132ms/step

Clases: {'banana': 0, 'fresa': 1, 'limon': 2, 'mandarina': 3, 'manzana': 4, 'pera': 5}

Predicciones:

[0.93 0. 0.01 0.01 0.03 0.03] -> [1. 0. 0. 0. 0. 0.]
[0.94 0. 0.02 0.01 0.02 0.01] -> [1. 0. 0. 0. 0. 0.]
[0. 0.85 0. 0. 0.15 0. ] -> [0. 1. 0. 0. 0. 0.]
[0. 0.89 0. 0. 0.11 0. ] -> [0. 1. 0. 0. 0. 0.]
[0.06 0. 0.78 0.02 0.02 0.11] -> [0. 0. 1. 0. 0. 0.]
[0.03 0. 0.29 0.15 0.22 0.3 ] -> [0. 0. 1. 0. 0. 0.] X
[0.02 0. 0.02 0.56 0.08 0.32] -> [0. 0. 0. 1. 0. 0.]
[0. 0.02 0.01 0.8 0.05 0.12] -> [0. 0. 0. 1. 0. 0.]
[0. 0.06 0. 0. 0.94 0. ] -> [0. 0. 0. 0. 1. 0.]
[0.02 0. 0.29 0.25 0.14 0.3 ] -> [0. 0. 0. 0. 1. 0.] X
[0.05 0. 0.2 0.09 0.05 0.61] -> [0. 0. 0. 0. 0. 1.]
[0.01 0. 0.12 0.44 0.22 0.21] -> [0. 0. 0. 0. 0. 1.] X
12/12 2s 177ms/step - accuracy: 0.8902 - loss: 0.2983

Test Loss: 0.5643
Test Accuracy: 0.7500
```



Se han probado dos imágenes de cada clase, y solo ha fallado 3 de 12, lo cual indica que tiene un rendimiento bueno.

Segundo modelo mejorado

Para mejorar el modelo probé añadiendo y quitando capas y modificando el número de neuronas, lo cual hizo que tenga un rendimiento bastante similar.

También cambié el input shape, tamaño del kernel y mezclé estos cambios sin conseguir mejoras.

Finalmente, probé el mismo modelo pero con imágenes a color, dando unos resultados bastante mejores.

Cargar imágenes con ImageDataGenerator

El segundo modelo fue entrenado con estos **ImageDataGenerators**:

```
# Cargar las imágenes

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=15,
    zoom_range=0.2,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True,
    fill_mode='nearest'
)

val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(128, 128),
    batch_size=16,
    class_mode='categorical'
)

validation_generator = val_datagen.flow_from_directory(
    val_data_dir,
    target_size=(128, 128),
    batch_size=16,
    class_mode='categorical'
)
```

El **ImageDataGenerator** estandariza los valores de píxeles de [0, 255] a [0, 1], rota aleatoriamente las imágenes hasta $\pm 15^\circ$, aplica un zoom aleatorio entre 80% y 120% del tamaño original, desplaza horizontalmente y verticalmente la imagen hasta un 10% del ancho y alto, voltea horizontalmente algunas imágenes y rellena las fotos que han sufrido transformaciones copiando píxeles cercanos.

Creación del modelo

El **CNN** usado tiene la siguiente composición:

```
input_shape = (128, 128, 3)

model = models.Sequential([
    layers.Input(shape=(128, 128, 3)),
    layers.Conv2D(16, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(32, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dropout(0.4),
    layers.Dense(64, activation='relu'),
    layers.Dense(6, activation='softmax')
])

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

Capas utilizadas:

12. **Input Layer**: entrada de imágenes de 128x128 píxeles, con 3 canales.
13. **Conv2D** (16 filtros, 3x3): aplica 16 filtros de tamaño 3x3 y extrae bordes y texturas simples. Salida de 126x126x16.
14. **MaxPooling2D** (2x2): Reduce el tamaño espacial a la mitad. Salida de 63x63x16.
15. **Conv2D** (32 filtros, 3x3) + ReLU: detecta patrones complejos. Salida de 61x61x32.
16. **MaxPooling2D** (2x2): 30x30x32.
17. **Conv2D** (64 filtros, 3x3) + ReLU: Aumenta profundidad de la representación. Salida de 28x28x64.
18. **MaxPooling2D** (2x2): Reduce a 14x14x64.
19. **Flatten**: Convierte 14x14x64 = 12,544 elementos a un solo vector 1D.
20. **Dropout** (0.4): Apaga aleatoriamente el 40% de las neuronas en entrenamiento para evitar overfitting.
21. **Dense** (64 neuronas) + ReLU: Capa que aprende combinaciones de las anteriores características extraídas.
22. **Dense** (6 neuronas) + Softmax: Proporciona una salida para las 6 clases. Ideal para clasificación multiclase con **categorical_crossentropy**.

Compilación:

4. **Adam**: optimizador adaptativo eficiente.
5. **Loss**: adecuada para etiquetas codificadas one-hot.
6. **Métrica**: precisión (accuracy) durante entrenamiento y evaluación.

Es básicamente como el anterior, pero con imágenes a color.

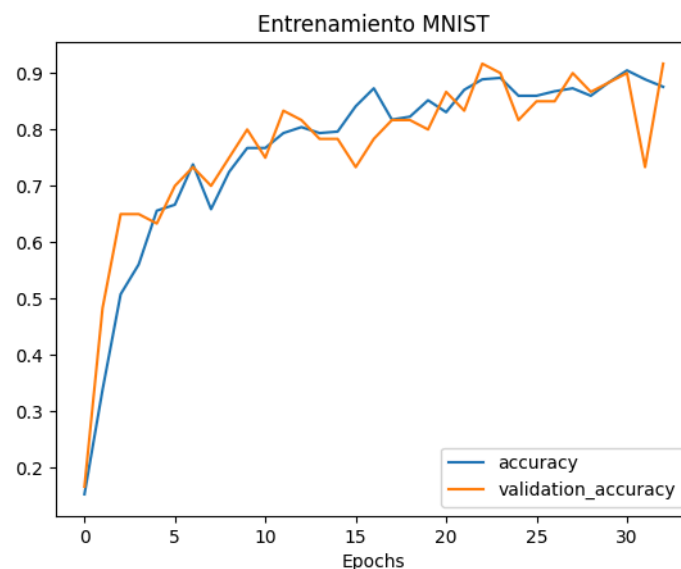
Entrenamiento

El modelo es entrenado con un **EarlyStopping** que cuando en 10 épocas el **val_accuracy** no sube, se queda con la mejor época.

```
es = EarlyStopping(monitor='val_accuracy', mode='max', verbose=1,
patience=10, restore_best_weights=True)

history_bc = model_bc.fit(train_generator_bc,
                           epochs=200,
                           validation_data=validation_generator_bc,
                           callbacks=[es])
```

```
Epoch 23/200
24/24 ————— 52s 2s/step - accuracy: 0.8983 - loss: 0.2407 - val_accuracy: 0.9167 - val_loss: 0.4046
Epoch 24/200
24/24 ————— 53s 2s/step - accuracy: 0.9192 - loss: 0.2562 - val_accuracy: 0.9000 - val_loss: 0.3143
Epoch 25/200
24/24 ————— 53s 2s/step - accuracy: 0.8824 - loss: 0.2873 - val_accuracy: 0.8167 - val_loss: 0.4277
Epoch 26/200
24/24 ————— 56s 2s/step - accuracy: 0.8575 - loss: 0.3700 - val_accuracy: 0.8500 - val_loss: 0.4524
Epoch 27/200
24/24 ————— 53s 2s/step - accuracy: 0.8825 - loss: 0.2996 - val_accuracy: 0.8500 - val_loss: 0.5107
Epoch 28/200
24/24 ————— 52s 2s/step - accuracy: 0.8492 - loss: 0.3529 - val_accuracy: 0.9000 - val_loss: 0.3414
Epoch 29/200
24/24 ————— 51s 2s/step - accuracy: 0.8735 - loss: 0.3021 - val_accuracy: 0.8667 - val_loss: 0.4518
Epoch 30/200
24/24 ————— 48s 2s/step - accuracy: 0.8583 - loss: 0.3607 - val_accuracy: 0.8833 - val_loss: 0.3783
Epoch 31/200
24/24 ————— 52s 2s/step - accuracy: 0.8818 - loss: 0.2469 - val_accuracy: 0.9000 - val_loss: 0.3444
Epoch 32/200
24/24 ————— 55s 2s/step - accuracy: 0.8765 - loss: 0.2826 - val_accuracy: 0.7333 - val_loss: 0.6140
Epoch 33/200
24/24 ————— 50s 2s/step - accuracy: 0.8660 - loss: 0.2911 - val_accuracy: 0.9167 - val_loss: 0.3567
Epoch 33: early stopping
Restoring model weights from the end of the best epoch: 23.
```



En este entrenamiento ha llegado a un **accuracy** de 0.8983 y **val_accuracy** de 0.9167, superando el 0.6956 de **accuracy** y el 0.7667 el **val_accuracy** del modelo anterior.

Pruebas

```

Found 12 images belonging to 6 classes.
12/12 ----- 2s 112ms/step

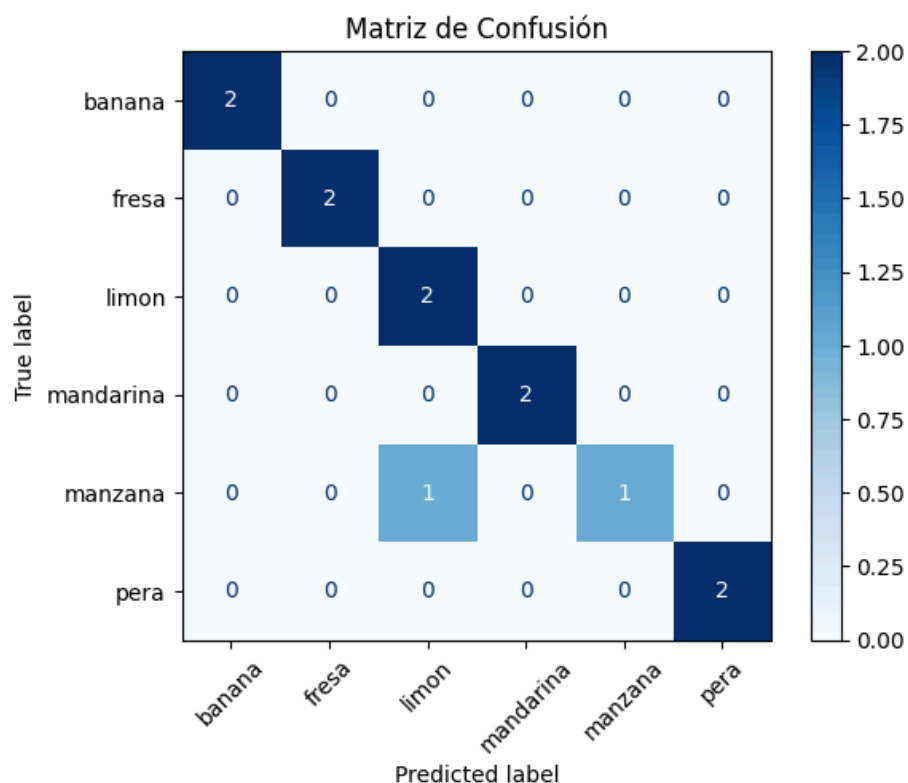
Clases: {'banana': 0, 'fresa': 1, 'limon': 2, 'mandarina': 3, 'manzana': 4, 'pera': 5}

Pruebas:

[0.92 0. 0. 0. 0. 0.07] -> [1. 0. 0. 0. 0. 0.]
[0.99 0. 0. 0. 0. 0.01] -> [1. 0. 0. 0. 0. 0.]
[0. 0.97 0. 0. 0.03 0. ] -> [0. 1. 0. 0. 0. 0.]
[0. 0.52 0. 0. 0.48 0. ] -> [0. 1. 0. 0. 0. 0.]
[0. 0. 1. 0. 0. 0. ] -> [0. 0. 1. 0. 0. 0.]
[0. 0. 0.81 0. 0.19 0. ] -> [0. 0. 1. 0. 0. 0.]
[0. 0.04 0. 0.86 0.1 0. ] -> [0. 0. 0. 1. 0. 0.]
[0. 0. 0. 1. 0. 0. ] -> [0. 0. 0. 1. 0. 0.]
[0. 0.29 0. 0. 0.71 0. ] -> [0. 0. 0. 0. 1. 0.]
[0. 0. 0.87 0. 0.13 0. ] -> [0. 0. 0. 0. 1. 0.] X
[0.03 0. 0. 0. 0. 0.97] -> [0. 0. 0. 0. 0. 1.]
[0. 0. 0. 0. 0. 1. ] -> [0. 0. 0. 0. 0. 1.]
12/12 ----- 2s 123ms/step - accuracy: 0.9725 - loss: 0.1868

Test Loss: 0.2978
Test Accuracy: 0.9167

```



Los resultados de este modelo son excelentes, solo ha fallado una predicción. en concreto ha fallado una imagen de una manzana amarilla, que la habrá confundido con un limón debido al color seguramente, aunque en el anterior modelo en blanco y negro también la confundió.

Pruebas con servidor Flask

Para usar el servidor hay que configurar el token en el código.

El servidor recibe un parámetro "ruta" que es una ruta a una imagen de Google Drive.

Las pruebas están realizadas en el segundo modelo.

Código para llamar a la API:

```
import requests

url = "http://localhost:5000/predict"
base = "/content/drive/MyDrive/Colab
Notebooks/trabajo_final/fruit_recognition_test/"

# Banana
data = {"ruta": f"{base}/banana/20250429_185513.jpg"}
print("\nBanana:\n", requests.post(url, json=data).json())

# Fresa
data = {"ruta": f"{base}/fresa/20250429_190128.jpg"}
print("\nFresa:\n", requests.post(url, json=data).json())

# Limón
data = {"ruta": f"{base}/limon/20250429_184258.jpg"}
print("\nLimón:\n", requests.post(url, json=data).json())

# Mandarina
data = {"ruta": f"{base}/mandarina/20250429_190547.jpg"}
print("\nMandarina:\n", requests.post(url, json=data).json())

# Manzana
data = {"ruta": f"{base}/manzana/20250429_183959.jpg"}
print("\nManzana:\n", requests.post(url, json=data).json())

# Pera
data = {"ruta": f"{base}/pera/20250429_185254.jpg"}
print("\nPera:\n", requests.post(url, json=data).json())
```

Resultado de la ejecución:

```
1/1 _____ 0s 121ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:38] "POST /predict HTTP/1.1" 200 -

Banana:
{'Clase': 'banana', 'Confidence': 0.9249}
1/1 _____ 0s 92ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:39] "POST /predict HTTP/1.1" 200 -

Fresa:
{'Clase': 'fresa', 'Confidence': 0.9705}
1/1 _____ 0s 109ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:40] "POST /predict HTTP/1.1" 200 -

Limón:
{'Clase': 'limon', 'Confidence': 0.9981}
1/1 _____ 0s 103ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:40] "POST /predict HTTP/1.1" 200 -

Mandarina:
{'Clase': 'mandarina', 'Confidence': 0.858}
1/1 _____ 0s 65ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:41] "POST /predict HTTP/1.1" 200 -

Manzana:
{'Clase': 'manzana', 'Confidence': 0.7069}
1/1 _____ 0s 42ms/step
INFO:werkzeug:127.0.0.1 - - [19/May/2025 00:01:41] "POST /predict HTTP/1.1" 200 -


Pera:
{'Clase': 'pera', 'Confidence': 0.9701}
```

Pruebas con Gradio

Las pruebas se han realizado usando imágenes de internet y están usando el segundo modelo.

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

manzana

Confidence

0,6599781


Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

banana

Confidence

0,89909387

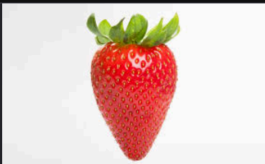
Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

fresa

Confidence

0,9262637


Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

limon

Confidence

0,80985904


Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

mandarina

Confidence

1,0


Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

manzana

Confidence

0,99997663


Marcar

Limpiar

Enviar

Fruit Recognition

Upload an image of a fruit to classify it.



Predicted Class

pera

Confidence

0,99030966

Marcar

Limpiar

Enviar

Por alguna razón, en distintas pruebas el modelo a veces confunde la clase banana con la clase manzana, el resto parece ir bien.

La única razón que le encuentro a este fallo es que todas las fotos de la clase Banana usan el mismo fondo, mientras que el resto de clases usan dos fondos distintos

Finalmente he querido probar manzanas amarillas para ver si es capaz de reconocerlas también y sí, será que la foto elegida para el test le encuentra formas similares a otra fruta.


Esta es la manzana amarilla de test:





A continuación, un par de pruebas con manzanas amarillas:

Fruit Recognition

Upload an image of a fruit to classify it.

 img





Predicted Class

manzana

Confidence

0,96793884


Marcar


Limpiar


Enviar

Fruit Recognition

Upload an image of a fruit to classify it.

 img





Predicted Class

manzana

Confidence

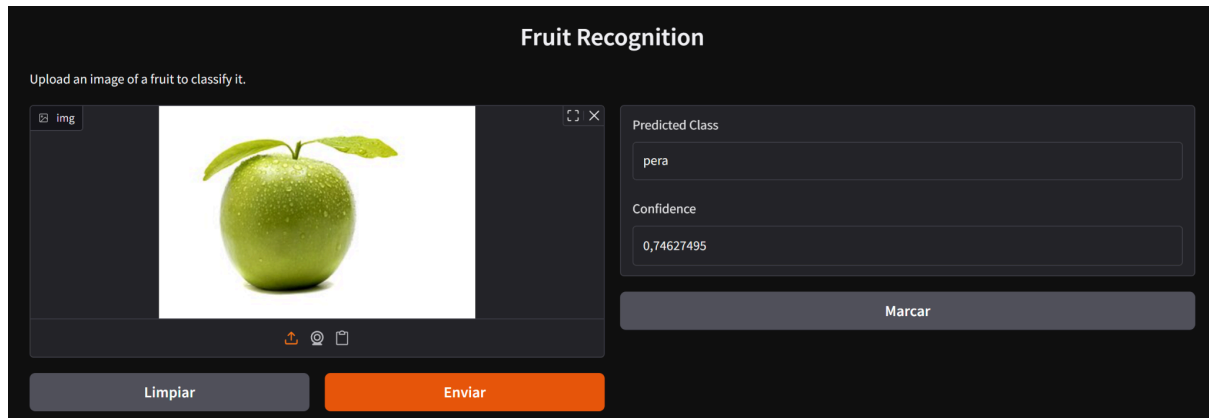
0,6482745

Marcar

Limpiar

Enviar

La verde ya la confunde con la pera, esto indica que los colores tienen mucha importancia, y podría explicar el por qué a veces confunde bananas con manzanas, por las manzanas amarillas del entrenamiento.



He probado también el modelo en blanco y negro pero no acierta ninguna foto de internet, al menos el modelo a color si predice bien.

Posible método de mejorar el modelo

Para mejorar el rendimiento de este modelo se podrían añadir muchas más fotos al dataset, usando múltiples fondos para que detecte mejor las formas de las frutas.

Se podrían seguir dos caminos:

- Continuar con el modelo a color añadiendo más clases de frutas, como arándano rojo y azul, manzana roja, verde y amarilla, aceituna verde y negra y más ejemplos así para sacar más partido al uso de colores.
- Mejorar el modelo en blanco y negro para que distinga las frutas en general sin importar el color y por tanto sin diferenciar sub-especies.