UNIVERSIDAD DE ANTIOQUIA

Facultad de Ingeniería

Informe de la entrega final sobre

Aplicación de una arquitectura CNN para la clasificación de imágenes de alimentos

Fuente de la foto: www.freepik.com

Fundamentos de Deep Learning

Semestre 2024.2

Un curso dirigido por el profesor **Raúl Ramos Pollán -** *raul.ramos@udea.edu.co*

Hugo Maharavo – 20AI87814
hugo.maharavo@udea.edu.co

1. Descripción de la estructura de los notebooks

1.1. Exploracion

```
!pip install kaggle
!kaggle datasets download -d imbikramsaha/food11
import zipfile
with zipfile.ZipFile('food11.zip', 'r') as zip_ref:
    zip_ref.extractall('food11')
```

Esta parte del código descarga en Kaggle el conjunto de datos público puesto en línea por *imbikramsaha* y denominado *food11*. Aquí es el enlace directo para acceder a los datos: https://www.kaggle.com/datasets/imbikramsaha/food11
De esta forma los datos se integran directamente en el colab donde ejecutamos nuestro preproceso y entrenamiento del modelo.

```
# Definir rutas
train_dir = 'food11/food11/train'
test_dir = 'food11/food11/test'
```

Son rutas de directorio que especifican la ubicación de los conjuntos de datos de entrenamiento y prueba. La organización de los datos en carpetas permite al ImageDataGenerator etiquetar automáticamente las imágenes basándose en la estructura del directorio.

1.2. Preprocesado

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16
from tensorflow.keras.optimizers import Adam
```

- **`os`:** Proporciona funciones para interactuar con el sistema de archivos.
- **`numpy`:** Utilizado para operaciones numéricas, especialmente arrays.
- **`tensorflow`:** La biblioteca central para construir y entrenar modelos de machine learning.
- **`tensorflow.keras`:** API de alto nivel para la construcción de redes neuronales, que permite la creación rápida de modelos.
- **`ImageDataGenerator`:** Una clase Keras utilizada para aumentar los datos de imagen para el dataset de entrenamiento y preprocesar imágenes.

Sobre otras importaciones (VGG16 y Adam) hablaremos más adelante en este informe.

```python
# Aumento de datos mejorado para el training set
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=50,
    width_shift_range=0.3,
    height_shift_range=0.3,
    shear_range=0.3,
    zoom_range=0.4,
    horizontal_flip=True,
    brightness_range=[0.7, 1.3]
)
```

- **`rescale=1.0 / 255`**
  Normaliza los valores de los píxeles en el intervalo [0, 1] dividiéndolos por 255

- **`rotation_range=50`**
  Rota aleatoriamente las imágenes hasta 50 grados, lo que ayuda al modelo a aprender características invariantes cuando las imágenes aparecen rotadas.

- **`width_shift_range=0.3 and height_shift_range=0.3`**
  Desplaza las imágenes hasta un 30% de la anchura y la altura, lo que ayuda al modelo a aprender características que se trasladan espacialmente.

- **`shear_range=0.3`**
  Aplica transformaciones de cizallamiento a las imágenes, simulando un efecto de distorsión, útil para el aprendizaje de características que son resistentes a la inclinación.

- **`zoom_range=0.4`**
  Amplía y reduce aleatoriamente las imágenes hasta un 40%, lo que permite al modelo aprender de imágenes que varían en distancia aparente o escala.

- **`horizontal_flip=True`**
  Voltea las imágenes horizontalmente, lo que duplica el tamaño del conjunto de datos y ayuda al modelo a generalizar las orientaciones izquierda-derecha.

- **`brightness_range=[0.7, 1.3]`:** Ajusta aleatoriamente el brillo de las imágenes, lo que proporciona solidez frente a las variaciones de iluminación.

Todo esto permite mejorar la capacidad de generalización del modelo al simular variaciones en los datos de entrenamiento.

```python
# Preprocesamiento para el test set
test_datagen = ImageDataGenerator(rescale=1.0 / 255)
```

El conjunto de test sólo se reescala para normalizar los valores de los píxeles dentro de [0, 1] por coherencia con el conjunto de train que hicimos justo antes. No se aplica ningún otro aumento, ya que el rendimiento del modelo debe evaluarse con datos de prueba inalterados.

```
# Carga training and test data
IMG_SIZE = (128, 128)
BATCH_SIZE = 32
```

IMG_SIZE
Especifica el tamaño objetivo para todas las imágenes de entrada. Aquí, elegí las imágenes para que se redimensionan a 128x128 píxeles para tener una buena eficiencia computacional y detalles suficientes.

BATCH_SIZE
Establece el número de imágenes procesadas a la vez durante el entrenamiento. Elegí lotes de 32 imágenes.

```
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

test_data = test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)
```

flow_from_directory
Esta función carga las imágenes directamente desde la estructura de directorios y aplica las transformaciones definidas en `ImageDataGenerator`.

- **train_dir** y **test_dir:** (Mencionado en la página 2)
- **target_size=IMG_SIZE:** (Mencionado en la página 4)
- **batch_size=BATCH_SIZE:** (Mencionado en la página 4)
- **class_mode='categorical':** Las etiquetas se codificarán en « One-hot », apropiadas para las 11 categorías de comidas de su conjunto de datos.

Conclusión de la parte preprocesado: Estos pasos de preprocesamiento garantizan que el modelo reciba imágenes en un formato coherente, con variaciones en el conjunto de entrenamiento para potenciar la generalización y condiciones estables para el conjunto de pruebas para medir con precisión el rendimiento.

2. Descripción de la solución

Primero, me gustaría indicar que para mejorar la velocidad de entrenamiento de mi modelo, he comprado por un mes la versión de Collab Pro.
Me permitió establecer el modo de ejecución en **A100 GPU**.
Esto facilitó mucho las iteraciones que pude hacer durante este proyecto, ya que me permitió entrenar el modelo final en 30 minutos (en lugar de más de 3 horas que tardaba inicialmente).

2.1. Carga, ajuste y fine-tuning del modelo VGG16

El código utiliza **VGG16**, una red convolucional previamente entrenada en el conjunto de datos de **ImageNet**, excluyendo sus capas superiores.

```
# Cargar el modelo VGG16 con pesos preentrenados
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(128, 128, 3))
```

Por qué VGG16?
Este modelo es especialmente eficaz para tareas de clasificación de imágenes. Al excluirlo de las capas superiores, lo utilizamos como extractor de características para imágenes de alimentos.

Parámetros elegidos :

- **`weights='imagenet'`**: Los pesos se inicializan a partir de los obtenidos tras el entrenamiento en el gran conjunto de datos ImageNet.
- **`include_top=False`**: Esta opción elimina las capas densas de clasificación en la salida, ya que necesitamos adaptar la salida a las 11 categorías de comida del conjunto de datos «food11».
- **`input_shape=(128, 128, 3)`**: Las imágenes se redimensionan a 128x128 píxeles, con 3 canales de color (RGB).

2.2. Construcción del modelo final

El modelo se amplía agregando capas densas encima de la red VGG16 para adaptar la salida a las 11 clases.

```
# Construir el modelo con dense layers
model = Sequential([
    base_model,
    Flatten(),
    Dense(512, activation='relu'),
    Dropout(0.5),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(11, activation='softmax')
])
```

Estructura :

- **`Flatten()`**:
  → Transforma características extraídas por VGG16 (una matriz) en un vector plano para ingresar en capas densas.

- Dos capas **`dense`**:
  → La primera tiene 512 neuronas y utiliza una función de activación **`relu`**. Esta capa aprende representaciones complejas de datos.
  → Una segunda capa con 256 neuronas refina las representaciones aprendidas.

- Dos capas de **`Dropout (0,5):`**
  → Reduce el sobreajuste desactivando aleatoriamente el 50% de las neuronas durante el entrenamiento.

- Una capa final **`Dense`** con 11 neuronas y activación **`softmax`**
  → Produce probabilidades para cada clase

2.3. Compilación del modelo

El modelo se compila con una función de costes y un optimizador adaptado a la clasificación multiclase.

```
# Compilar el modelo con una "learning rate" reducida
model.compile(optimizer=Adam(learning_rate=0.00001), loss='categorical_crossentropy', metrics=['accuracy'])
```

Optimizador: **`Adam`**
→ Esta elección garantiza una convergencia rápida y estable. La tasa de aprendizaje reducida (0,00001) permite actualizaciones fluidas de los pesos para evitar oscilaciones.

Función de pérdida: **`loss = 'categorical_crossentropy'`**
→ Esencial para tareas de clasificación de varias clases donde las etiquetas se codifican en forma categórica.

`metrics=['accuracy']`
Le permite realizar un seguimiento del rendimiento del modelo en términos de predicciones correctas.

Cuando queremos una vista previa del modelo, obtenemos esto:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
vgg16 (Functional)	(None, 4, 4, 512)	14,714,688
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 512)	4,194,816
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 11)	2,827

Total params: 19,043,659 (72.65 MB)
Trainable params: 19,043,659 (72.65 MB)
Non-trainable params: 0 (0.00 B)

2.4. Training del modelo

El modelo se entrena durante 30 **epochs** con los conjuntos de datos de test y de validación.

```
# Entrenar el modelo
history = model.fit(
    train_data,
    epochs=30,
    validation_data=test_data,
)
```

- Número de épocas (**epochs**): Un valor alto (30) permite una convergencia óptima.
- Datos_de_validación: el conjunto de test se utiliza como validación para controlar el rendimiento.

2.5. Evaluación de resultados

Por último, el modelo se evalúa en el conjunto de test

```
# Evaluar el modelo en el conjunto test
test_loss, test_accuracy = model.evaluate(test_data, verbose=1)
print(f"Precisión final del modelo en el conjunto de test: {test_accuracy * 100:.2f}%")
```

La función de evaluación calcula las pérdidas y la precisión del conjunto de pruebas. Estas medidas indican la capacidad de generalización del modelo con nuevos datos.

Conclusión de la parte "Descripción de la solución": El modelo utiliza la potencia de VGG16, combinada con capas personalizadas, para resolver un complejo problema de clasificación de imágenes. El enfoque de ajuste fino explota las ventajas de los pesos preentrenados a la vez que adapta el modelo a datos específicos. La optimización garantiza un rendimiento estable y un entrenamiento eficaz.

3. Descripción de un intento previo con una arquitectura CNN Básica

El primer modelo que hice consistió en una arquitectura CNN desarrollada desde cero y entrenada directamente sobre los datos disponibles.

3.1. Arquitectura

- Tres bloques convolucionales, cada uno compuesto por una capa **Conv2D** seguida de una capa de **MaxPooling2D** para reducir las dimensiones de las características.
- Una capa **Flatten** para convertir los mapas de características en un vector unidimensional.
- Una capa densa con 128 neuronas y activación **ReLU**, seguida de una capa de salida con 11 neuronas y activación **softmax** para la clasificación de múltiples clases.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

# Definir la arquitectura de la CNN
model = Sequential([
    # Primer bloque convolucional
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D(pool_size=(2, 2)),

    # Segundo bloque convolucional
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Tercer bloque convolucional
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),

    # Aplanar los mapas de características a un vector 1D
    Flatten(),

    # Capa completamente conectada
    Dense(128, activation='relu'),
    Dropout(0.5),  # Dropout para la regularización

    # Capa de salida
    Dense(11, activation='softmax')  # 11 clases para las categorías de alimentos
])

# Compilar el modelo
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Mostrar el resumen del modelo
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: U
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 128)	3,211,392
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 11)	1,419

```
Total params: 3,306,059 (12.61 MB)
Trainable params: 3,306,059 (12.61 MB)
Non-trainable params: 0 (0.00 B)
```

3.2. Preprocesamiento

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Definir el tamaño de las imágenes y el tamaño del batch
IMG_SIZE = (128, 128)
BATCH_SIZE = 32

# Aumento de datos y reescalado para el conjunto de entrenamiento
train_datagen = ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

# Reescalado para el conjunto de test
test_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Cargar y preprocesar los datos de entrenamiento
train_data = train_datagen.flow_from_directory(
    train_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# Cargar y preprocesar los datos de test
test_data = test_datagen.flow_from_directory(
    test_dir,
    target_size=IMG_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='categorical'
)

# Entrenar el modelo
history = model.fit(
    train_data,
    epochs=30,
    validation_data=test_data
)
```

Se utilizó **`ImageDataGenerator`** para aplicar aumentos moderados en los datos, como rotaciones, desplazamientos y zoom, además de una normalización básica con **`rescale=1.0 / 255`**.

3.3. Hiperparámetros

Son los mismos que en el modelo con VGG16: `optimizer='Adam', Categorical_crossentropy, IMG_SIZE = (128x128), BATCH_SIZE = 32, epochs=30`

3.4. Resultados y Análisis

- **Precisión en el conjunto de prueba:** 52,36 %.

3.5. Problemas identificados:

- **Entrenamiento desde cero:** Creo que dado el tamaño reducido del conjunto de datos (9900 imágenes para entrenamiento), el modelo no logró aprender características suficientemente representativas.
- **Complejidad:** El modelo tiene más de 3,3 millones de parámetros, lo que puede aumentar el riesgo de sobreajuste.
- **Aumentos de datos limitados:** Los aumentos aplicados no lograron mejorar significativamente la generalización del modelo.

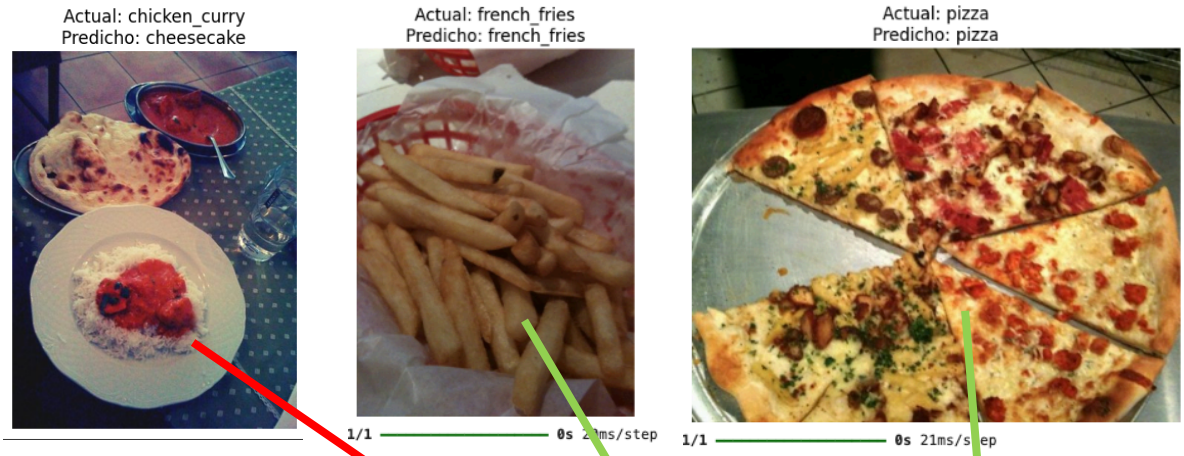
3.6. Comparación de los Modelos (con `epochs = 30`)

	`03 - CNN Modelo básico.ipynb`	**`04 - Modelo final con VGG16.ipynb`**
Base de la arquitectura	CNN desarrollada desde cero	VGG16 preentrenado
Parámetros totales	3,3 millones	19 millones
Aumentos de datos	Rotación, zoom	Rotación, zoom, brillo, etc.
Tasa de aprendizaje	Estándar (0.001)	Reducida (0.00001)
Precisión en prueba	52,36 %	79,55 %

4. Descripción de los resultados.

En mi documento *Entrega_1.pdf*, indiqué que mi objetivo era lograr al menos un 75% de evaluación positiva para el modelo. Cuando ejecuto el modelo (el VGG16 preentrenado), con los datos del kaggle,, obtengo un porcentaje de accuracy de más o menos 80%. Así que estoy muy satisfecho con mi modelo, ya que cumple mis compromisos.

También he escrito codigos que permite visualizar los resultados de nuestro modelo. El código exacto se puede encontrar en mi repositorio de GitHub, en `04 - Modelo final con VGG16.ipynb`, pero aquí quiero mostrar algunos resultados que nos dan, cuando configuro `epochs = 30`:



Resultados con "epochs" diferentes

Para observar la influencia de `epochs`, entrené el modelo con dos valores diferentes.
Primero entrenándolo con **30 épocas**. La precisión obtenida fue del **79,55%.**
Luego otras 50 en el mismo entorno, y eso me da resultados similares: una precisión final del modelo en el conjunto de prueba de **81%.**
Este último Notebook se puede encontrar en el github con el nombre `04 - Modelo final con VGG16.ipynb`