

# REDES NEURONALES CONVOLUCIONALES

## Una Aplicación para Reconocimiento de Imágenes

---

Maestría en Inteligencia Artificial

UNIR Universidad Internacional de la Rioja en México

Asignatura: Sistemas Cognitivos Artificiales.

Actividad grupal desarrollada por Edmilson Prata da Silva, Mariana Carmona Cruz, Gerardo Davila escamilla y Pedro Luis Cabrera.

Disponible en [GitHub](#), formato [Notebook \(ipynb\)](#) en [este enlace](#) y formato [PDF](#) en [este enlace](#).

---

### 1. Introducción

En esta actividad, vamos a trabajar con Convolutional Neural Networks para resolver un problema de clasificación de imágenes. En particular, vamos a clasificar imágenes de personajes de la conocida serie de los Simpsons.

Como las CNN profundas son un tipo de modelo bastante avanzado y computacionalmente costoso, se recomienda hacer la práctica en Google Colaboratory con soporte para GPUs. En [este enlace](#) se explica cómo activar un entorno con GPUs. *Nota: para leer las imágenes y estandarizarlas al mismo tamaño se usa la librería opencv. Esta librería está ya instalada en el entorno de Colab, pero si trabajáis de manera local tendréis que instalarla.*



## ⚙️ 2. Librerías y Configuraciones

Librerías necesarias al notebook e configuraciones generales.

### 📖 2.1. Librerías

Instalaciones y importaciones de librerías necesarias en el notebook.

```
In [1]: # %pip install opencv-python numpy keras scikit-image
```

```
In [2]: import cv2
import os
import keras
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report

from tensorflow import keras
from tensorflow.keras.models import Model, Sequential, load_model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten,
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
from tensorflow.keras.optimizers import Adam, SGD, RMSprop, Nadam
from tensorflow.keras.regularizers import l2
from tensorflow.keras.preprocessing.image import ImageDataGenerator

print("Importaciones ok.")
```

Importaciones ok.

### ⚙️ 2.2. Configuraciones

Configuraciones diversas para uso en el notebook.

```
In [3]: # Configuración de la visualización de gráficos:
plt.rcParams["figure.figsize"] = (20, 6)

# La ruta raíz del conjunto de datos:
DATASET_PATH = "./datasets"

# Conjunto de datos de entrenamiento:
DATASET_TRAIN_PATH = "./datasets/simpsons_extracted/simpsons/train"

# Conjunto de datos de prueba:
DATASET_TEST_PATH = "./datasets/simpsons_extracted/simpsons/test"

# Definición de IMG_SIZE antes de que MAP_CHARACTERS se construya complet
IMG_SIZE = 90

# Epocas para el entrenamiento:
EPOCHS = 50

# Tamaño del lote para el entrenamiento:
BATCH_SIZE = 32

print("Configuraciones iniciales listas.")
```

Configuraciones iniciales listas.



## 3. Conjunto de Datos

El dataset a utilizar consiste en imágenes de personajes de los Simpsons extraídas directamente de capítulos de la serie. Este dataset ha sido recopilado por [Alexandre Attia](#) y es más complejo que el dataset de Fashion MNIST que hemos utilizado hasta ahora. Aparte de tener más clases (vamos a utilizar los 18 personajes con más imágenes), los personajes pueden aparecer en distintas poses, en distintas posiciones de la imagen o con otros personajes en pantalla (si bien el personaje a clasificar siempre aparece en la posición predominante).

### 3.1. Carga de Datos

Bajamos el dataset completo una única vez utilizando datasets de Keras.

```
In [4]: keras.utils.get_file(
        fname="simpsons.tar.gz",
        cache_dir=".",
        extract=True,
        origin="https://storage.googleapis.com/wandb-production.appspot.com/m
    )
```

Out [4]: './datasets/simpsons\_extracted'



### 3.2. Visualización del Dataset

Listado de los archivos del dataset para verificación del correcto cargamento.

```
In [5]: def list_files_recursively(startpath, show_files=False):
        for root, dirs, files in os.walk(startpath):

            level = root.replace(startpath, '').count(os.sep)
            indent = ' ' * 4 * (level)
            print(f'{indent}{os.path.basename(root)}/ {str(len(files))} + " ar

            if show_files:
                subindent = ' ' * 4 * (level + 1)
                for f in files:
                    print(f'{subindent}{f}')

        print(f"Lista de contenido de {DATASET_PATH}")
        list_files_recursively(DATASET_PATH)
```

```
Lista de contenido de ./datasets
datasets/ 1 archivos
    simpsons_extracted/
        simpsons/
            test/
                charles_montgomery_burns/ 119 archivos
                chief_wiggum/ 98 archivos
                milhouse_van_houten/ 107 archivos
                sideshow_bob/ 87 archivos
                moe_szyslak/ 145 archivos
                ned_flanders/ 145 archivos
                abraham_grampa_simpson/ 91 archivos
                krusty_the_clown/ 120 archivos
                marge_simpson/ 129 archivos
                .ipynb_checkpoints/ 3 archivos
                principal_skinner/ 119 archivos
                bart_simpson/ 134 archivos
                lisa_simpson/ 135 archivos
                homer_simpson/ 224 archivos
            train/
                charles_montgomery_burns/ 1074 archivos
                chief_wiggum/ 888 archivos
                milhouse_van_houten/ 972 archivos
                sideshow_bob/ 790 archivos
                moe_szyslak/ 1307 archivos
                ned_flanders/ 1309 archivos
                abraham_grampa_simpson/ 822 archivos
                krusty_the_clown/ 1086 archivos
                marge_simpson/ 1162 archivos
                principal_skinner/ 1075 archivos
                bart_simpson/ 1210 archivos
                .ipynb_checkpoints/ 1 archivos
                lisa_simpson/ 1219 archivos
                homer_simpson/ 2022 archivos
```

## 4. Preprocesamiento

Preparación de los datos para el entrenamiento.

### 4.1. Mapa de Caracteres

Creación dinámica de un mapa con los nombres de los personajes según los subdirectorios existentes en el dataset. La variable MAP\_CHARACTERS contiene una asignación del número de clase al carácter. Solo incluirá los caracteres que tengan un directorio en el conjunto de entrenamiento.

```
In [6]: MAP_CHARACTERS = {}

if os.path.exists(DATASET_TRAIN_PATH):
    character_names = sorted([d for d in os.listdir(DATASET_TRAIN_PATH) if
    for i, name in enumerate(character_names):
        MAP_CHARACTERS[i] = name

    print(f"{len(MAP_CHARACTERS)} caracteres cargados dinámicamente: {MAP

else:
    print(f"Error: No se encontró la ruta de entrenamiento {DATASET_TRAIN

13 caracteres cargados dinámicamente: {0: 'abraham_grampa_simpson', 1: 'bart_simpson', 2: 'charles_montgomery_burns', 3: 'chief_wiggum', 4: 'homer_simpson', 5: 'krusty_the_clown', 6: 'lisa_simpson', 7: 'marge_simpson', 8: 'milhouse_van_houten', 9: 'moe_szyslak', 10: 'ned_flanders', 11: 'principal_skinner', 12: 'ideshow_bob'}
```



## 4.2. Datos de Entrenamiento

Preparación de los datos de entrenamiento en imágenes. Como las imágenes tienen tamaños distintas, utilizamos la librería opencv para hacer un resize y adaptarlas todas a tamaño IMG\_SIZE x IMG\_SIZE.

```
In [7]: def load_train_set(dirname, map_characters, verbose=True):
    """
    Carga los datos de entrenamiento en imágenes.

    Args:
        dirname: directorio completo del que leer los datos
        map_characters: variable de mapeo entre labels y personajes
        verbose: si es True, muestra información de las imágenes cargadas

    Returns:
        X_train: Array con todas las imágenes cargadas con tamaño IMG_SIZE
        y_train: Array con las labels de correspondientes a cada imagen.
    """
    X_train = []
    y_train = []

    for label, character in map_characters.items():
        files = os.listdir(os.path.join(dirname, character))
        images = [file for file in files if file.endswith(".jpg")]

        if verbose:
            print("Leyendo {} imágenes encontradas de {}".format(len(images), character))

        for image_name in images:
            image = cv2.imread(os.path.join(dirname, character, image_name))
            image = cv2.resize(image, (IMG_SIZE, IMG_SIZE))
            X_train.append(image)
            y_train.append(label)
```

```
return np.array(X_train), np.array(y_train)
```



### 4.3. Datos de Prueba

Preparación de los datos de prueba en imágenes. Como las imágenes tienen tamaños distintas, utilizamos la librería opencv para hacer un resize y adaptarlas todas a tamaño IMG\_SIZE x IMG\_SIZE.

```
In [8]: def load_test_set(dirname, map_characters, verbose=True):
        """
        Esta función funciona de manera equivalente a la función load_train_s
        pero cargando los datos de test.
        """

        X_test = []
        y_test = []

        for label, character in map_characters.items():

            char_path = os.path.join(dirname, character)

            if os.path.exists(char_path):
                files = os.listdir(char_path)
                images = [file for file in files if file.endswith(".jpg")]

                if verbose:
                    print("Leyendo {} imágenes encontradas de {} para test".f

                for image_name in images:
                    image = cv2.imread(os.path.join(char_path, image_name))
                    image = cv2.resize(image, (IMG_SIZE, IMG_SIZE))
                    X_test.append(image)
                    y_test.append(label)

            elif verbose:
                print(f"Advertencia: Directorio de test para '{character}' no

        print("Leídas {} imágenes de test en total".format(len(X_test)))
        return np.array(X_test), np.array(y_test)
```



### 4.4. Preparación de los Datos

Separación y preparación de los datos de entrenamiento y prueba lo que incluyó el ajuste de tamaño de las imagenes para IMG\_SIZE x IMG\_SIZE.

```
In [9]: X_train_original, y_train_original = load_train_set(DATASET_TRAIN_PATH, M
        X_test_original, y_test_original = load_test_set(DATASET_TEST_PATH, MAP_C
```

```

Leyendo 822 imágenes encontradas de abraham_grampa_simpson
Leyendo 1210 imágenes encontradas de bart_simpson
Leyendo 1074 imágenes encontradas de charles_montgomery_burns
Leyendo 888 imágenes encontradas de chief_wiggum
Leyendo 2022 imágenes encontradas de homer_simpson
Leyendo 1086 imágenes encontradas de krusty_the_clown
Leyendo 1219 imágenes encontradas de lisa_simpson
Leyendo 1162 imágenes encontradas de marge_simpson
Leyendo 972 imágenes encontradas de milhouse_van_houten
Leyendo 1307 imágenes encontradas de moe_szyslak
Leyendo 1309 imágenes encontradas de ned_flanders
Leyendo 1075 imágenes encontradas de principal_skinner
Leyendo 790 imágenes encontradas de sideshow_bob
Leyendo 91 imágenes encontradas de abraham_grampa_simpson para test
Leyendo 134 imágenes encontradas de bart_simpson para test
Leyendo 119 imágenes encontradas de charles_montgomery_burns para test
Leyendo 98 imágenes encontradas de chief_wiggum para test
Leyendo 224 imágenes encontradas de homer_simpson para test
Leyendo 120 imágenes encontradas de krusty_the_clown para test
Leyendo 135 imágenes encontradas de lisa_simpson para test
Leyendo 129 imágenes encontradas de marge_simpson para test
Leyendo 107 imágenes encontradas de milhouse_van_houten para test
Leyendo 145 imágenes encontradas de moe_szyslak para test
Leyendo 145 imágenes encontradas de ned_flanders para test
Leyendo 119 imágenes encontradas de principal_skinner para test
Leyendo 87 imágenes encontradas de sideshow_bob para test
Leídas 1653 imágenes de test en total

```

## 4.5. Barajando los Datos

Vamos a barajar los datos aleatoriamente. Esto es importante porque, si no lo hacemos y, por ejemplo, usamos el 20 % de los datos finales como conjunto de validación, solo usaremos una pequeña cantidad de caracteres, ya que las imágenes se leen secuencialmente para cada carácter.

```

In [10]: perm_train = np.random.permutation(len(X_train_original))
X_train_original, y_train_original = X_train_original[perm_train], y_train

perm_test = np.random.permutation(len(X_test_original))
X_test_original, y_test_original = X_test_original[perm_test], y_test_ori

print("Datos barajados.")

```

Datos barajados.

```

In [11]: # Muestras de los datos de entrenamiento:
plt.figure(figsize=(3,3))
plt.imshow(X_train_original[2])
plt.title(f"Label: {y_train_original[2]} - Character: {MAP_CHARACTERS[y_t

# Muestra de la imagen invertida:
plt.figure(figsize=(3,3))
plt.imshow(np.flip(X_train_original[2], axis=-1) )
plt.title(f"Label: {y_train_original[2]} - Character: {MAP_CHARACTERS[y_t

```

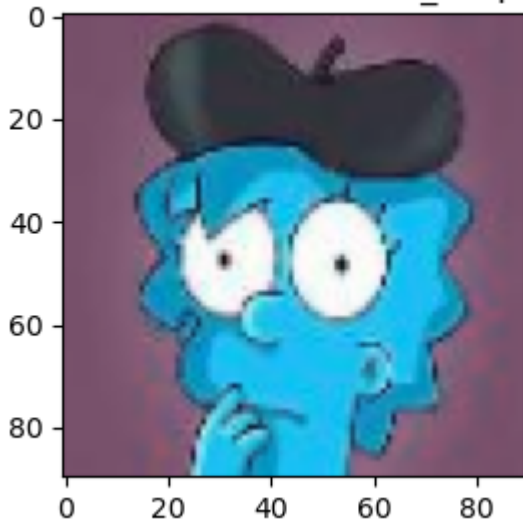
```

Out[11]: Text(0.5, 1.0, 'Label: 6 - Character: lisa_simpson')

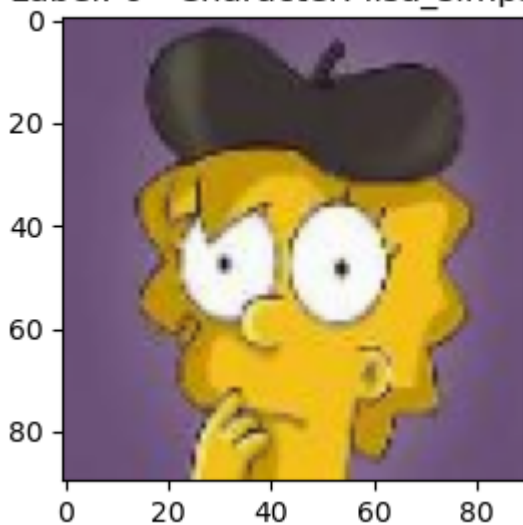
```



Label: 6 - Character: lisa\_simpson



Label: 6 - Character: lisa\_simpson



## 4.6. Normalización de Datos

Las imágenes no están normalizadas. Hay que normalizarlas para obtener mejores resultados en el entrenamiento y en el clasificador (modelo final). La normalización ofrece ventajas como estabilidad numérica, comparación de coeficientes del modelo, evitación de distribuciones con un alto rango dinámico y estabilidad en el cálculo de variables.

```
In [12]: X_train_std = X_train_original.astype('float32') / 255.0
X_test_std = X_test_original.astype('float32') / 255.0

print(f"Shape de X_train_original antes de la normalización: {X_train_ori
print(f"Shape de X_test_original antes de la normalización: {X_test_origi
print(f"Shape de X_train_std después de la normalización: {X_train_std.sh
print(f"Shape de X_test_std después de la normalización: {X_test_std.shap
```



Shape de X\_train\_original antes de la normalización: (14936, 90, 90, 3). Rango de valores: 0 – 255  
 Shape de X\_test\_original antes de la normalización: (1653, 90, 90, 3). Rango de valores: 0 – 255  
 Shape de X\_train\_std después de la normalización: (14936, 90, 90, 3). Rango de valores: 0.0 – 1.0  
 Shape de X\_test\_std después de la normalización: (1653, 90, 90, 3). Rango de valores: 0.0 – 1.0

## 4.7. One-Hot Encoding

La técnica One-hot Encoding es utilizada para convertir datos categóricos (como "rojo", "azul", "verde") a un formato numérico para el aprendizaje automático, creando nuevas columnas binarias (0 o 1) para cada categoría. Un "1" en una columna indica la presencia de esa categoría específica, mientras que un "0" significa su ausencia, lo que evita que los algoritmos asuman un orden o importancia falsos entre categorías. Es esencial para los modelos que requieren datos numéricos y gestiona eficazmente las características nominales (desordenadas), transformando una sola columna en múltiples columnas binarias.

Por qué se utiliza:

- Requisito del aprendizaje automático: Muchos algoritmos no pueden procesar texto/categorías directamente y necesitan números.
- Evita la ordinalidad: Evita que los modelos malinterpreten las categorías como si tuvieran un orden numérico (p. ej., "Azul" no es "menor que" "Verde").
- Representación única: Cada categoría tiene su propio vector numérico, lo que las hace semánticamente independientes.

```
In [13]: num_classes = len(MAP_CHARACTERS)
y_train_ohe = to_categorical(y_train_original, num_classes)
y_test_ohe = to_categorical(y_test_original, num_classes)

print(f"Shape de y_train después de one-hot encoding: {y_train_ohe.shape}")
print(f"Shape de y_test después de one-hot encoding: {y_test_ohe.shape},
```

Shape de y\_train después de one-hot encoding: (14936, 13), Ejemplo: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]  
 Shape de y\_test después de one-hot encoding: (1653, 13), Ejemplo: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]

## 4.8. Separación de los Datos (Train/Validation)

Separación de los datos en dos conjuntos de datos: uno para el entrenamiento del algoritmo y otro para la validación del modelo. Los datos serán partidos en training/validation para tener una buena estimación de los valores que el modelo tendrá en los datos de test, así como comprobar que no estamos cayendo en overfitting. Será utilizada una partición 80/20.

```
In [14]: X_train, X_val, y_train, y_val = train_test_split(X_train_std, y_train_ohe

print(f"Shape de X_train: {X_train.shape}. Rango de valores: {X_train.min}
print(f"Shape de y_train: {y_train.shape}. Rango de valores: {y_train.min
```

```
print(f"Shape de X_val: {X_val.shape}. Rango de valores: {X_val.min()} - {X_val.max()}")
print(f"Shape de y_val: {y_val.shape}. Rango de valores: {y_val.min()} - {y_val.max()}")
```

Shape de X\_train: (11948, 90, 90, 3). Rango de valores: 0.0 - 1.0  
 Shape de y\_train: (11948, 13). Rango de valores: 0.0 - 1.0  
 Shape de X\_val: (2988, 90, 90, 3). Rango de valores: 0.0 - 1.0  
 Shape de y\_val: (2988, 13). Rango de valores: 0.0 - 1.0

## 5. CNN - Convolutional Neural Networks con Keras

Utilizando CNN (Convolutional Neural Networks) con Keras, entrenaremos un clasificador que sea capaz de reconocer personajes en imágenes de los Simpsons con una accuracy en el dataset de test de, al menos, **85%**. Redactaremos un informe analizando varias de las alternativas probadas y los resultados obtenidos.

A continuación se detallan una serie de aspectos para ser analizados en el informe:

- Análisis de los datos a utilizar.
- Análisis de resultados, obtención de métricas de *precision* y *recall* por clase y análisis de qué clases obtienen mejores o peores resultados.
- Análisis visual de los errores de la red. ¿Qué tipo de imágenes o qué personajes dan más problemas a nuestro modelo?
- Comparación de modelos CNNs con un modelo de Fully Connected para este problema.
- Utilización de distintas arquitecturas CNNs, comentando aspectos como su profundidad, hiperparámetros utilizados, optimizador, uso de técnicas de regularización, *batch normalization*, etc.
- Utilización de *data augmentation*. Esto puede conseguirse con la clase [ImageDataGenerator](#) de Keras.

### Notas:

- El test set del problema tiene imágenes un poco más "fáciles", por lo que es posible encontrarse con métricas en el test set bastante mejores que en el training set.



### 5.1. CNN - Definición de la arquitectura

Definición de la arquitectura de la red neuronal CNN, compilación del modelo y resumen del modelo.

```
In [15]: model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
```

```

Dropout(0.25),

Conv2D(128, (3, 3), activation='relu', padding='same'),
Conv2D(128, (3, 3), activation='relu', padding='same'),
MaxPooling2D((2, 2)),
Dropout(0.25),

Flatten(),
Dense(128, activation='relu'),
Dropout(0.5),
Dense(num_classes, activation='softmax')
])

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

# Mostrar un resumen del modelo
model.summary()

```

/Users/edprata/anaconda3/envs/Python-3-13-2/lib/python3.13/site-packages/keras/src/layers/convolutional/base\_conv.py:113: UserWarning: Do not pass a n `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

**Model: "sequential"**

Layer (type)	Output Shape	
conv2d (Conv2D)	(None, 90, 90, 32)	
conv2d_1 (Conv2D)	(None, 90, 90, 32)	
max_pooling2d (MaxPooling2D)	(None, 45, 45, 32)	
dropout (Dropout)	(None, 45, 45, 32)	
conv2d_2 (Conv2D)	(None, 45, 45, 64)	
conv2d_3 (Conv2D)	(None, 45, 45, 64)	
max_pooling2d_1 (MaxPooling2D)	(None, 22, 22, 64)	
dropout_1 (Dropout)	(None, 22, 22, 64)	
conv2d_4 (Conv2D)	(None, 22, 22, 128)	
conv2d_5 (Conv2D)	(None, 22, 22, 128)	
max_pooling2d_2 (MaxPooling2D)	(None, 11, 11, 128)	
dropout_2 (Dropout)	(None, 11, 11, 128)	
flatten (Flatten)	(None, 15488)	
dense (Dense)	(None, 128)	1,9
dropout_3 (Dropout)	(None, 128)	
dense_1 (Dense)	(None, 13)	

**Total params:** 2,271,277 (8.66 MB)

**Trainable params:** 2,271,277 (8.66 MB)

**Non-trainable params:** 0 (0.00 B)

## 5.2. CNN - Callbacks para Early Stopping

Definición de callbacks para guardar el mejor modelo y ejecutar una detención temprana (early stopping) al alcanzarlo. Esta técnica impide que el entrenamiento del modelo continúe incluso después de cumplirse los requisitos de calidad deseados, lo que consume más recursos y tiempo innecesariamente.


















```
In [16]: checkpoint = ModelCheckpoint('best_model.keras', monitor='val_accuracy',
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_b
print("Callbacks definidos.")
```

Callbacks definidos.

## 5.3. CNN - Entrenamiento del modelo















Entrenamiento del modelo con control de parada temprana, según todos los ajustes previamente configurados.

```
In [17]: history = model.fit(  
    X_train, # Datos de entrenamiento  
    y_train, # Labels de entrenamiento  
    epochs=EPOCHS, # Número de épocas  
    batch_size=BATCH_SIZE, # Tamaño del batch  
    validation_data=(X_val, y_val), # Datos de validación  
    callbacks=[checkpoint, early_stopping] # Callbacks para guardar el me  
    )  
  
    print("Entrenamiento completado.")
```

Epoch 1/50  
**374/374**  **0s** 406ms/step - accuracy: 0.1824 - loss: 2.3754  
Epoch 1: val\_accuracy improved from None to 0.45917, saving model to best\_model.keras  
**374/374**  **164s** 431ms/step - accuracy: 0.2626 - loss: 2.1736 - val\_accuracy: 0.4592 - val\_loss: 1.6843  
Epoch 2/50  
**374/374**  **0s** 316ms/step - accuracy: 0.4283 - loss: 1.7311  
Epoch 2: val\_accuracy improved from 0.45917 to 0.59203, saving model to best\_model.keras  
**374/374**  **125s** 333ms/step - accuracy: 0.4618 - loss: 1.6330 - val\_accuracy: 0.5920 - val\_loss: 1.3197  
Epoch 3/50  
**374/374**  **0s** 273ms/step - accuracy: 0.5627 - loss: 1.3441  
Epoch 3: val\_accuracy improved from 0.59203 to 0.67369, saving model to best\_model.keras  
**374/374**  **109s** 292ms/step - accuracy: 0.5769 - loss: 1.2985 - val\_accuracy: 0.6737 - val\_loss: 1.0215  
Epoch 4/50  
**374/374**  **0s** 265ms/step - accuracy: 0.6398 - loss: 1.0978  
Epoch 4: val\_accuracy improved from 0.67369 to 0.71821, saving model to best\_model.keras  
**374/374**  **106s** 283ms/step - accuracy: 0.6531 - loss: 1.0624 - val\_accuracy: 0.7182 - val\_loss: 0.8804  
Epoch 5/50  
**374/374**  **0s** 259ms/step - accuracy: 0.7018 - loss: 0.9220  
Epoch 5: val\_accuracy improved from 0.71821 to 0.75837, saving model to best\_model.keras  
**374/374**  **104s** 279ms/step - accuracy: 0.7071 - loss: 0.9028 - val\_accuracy: 0.7584 - val\_loss: 0.7629  
Epoch 6/50  
**374/374**  **0s** 344ms/step - accuracy: 0.7438 - loss: 0.7705  
Epoch 6: val\_accuracy improved from 0.75837 to 0.79384, saving model to best\_model.keras  
**374/374**  **137s** 367ms/step - accuracy: 0.7487 - loss: 0.7666 - val\_accuracy: 0.7938 - val\_loss: 0.6619  
Epoch 7/50  
**374/374**  **0s** 323ms/step - accuracy: 0.7881 - loss: 0.6590  
Epoch 7: val\_accuracy improved from 0.79384 to 0.80154, saving model to best\_model.keras  
**374/374**  **129s** 344ms/step - accuracy: 0.7850 - loss: 0.6660 - val\_accuracy: 0.8015 - val\_loss: 0.6312  
Epoch 8/50  
**374/374**  **0s** 323ms/step - accuracy: 0.8190 - loss: 0.5556  
Epoch 8: val\_accuracy improved from 0.80154 to 0.83768, saving model to best\_model.keras  
**374/374**  **129s** 344ms/step - accuracy: 0.8172 - loss: 0.5577 - val\_accuracy: 0.8377 - val\_loss: 0.5646  
Epoch 9/50  
**374/374**  **0s** 322ms/step - accuracy: 0.8347 - loss: 0.5162  
Epoch 9: val\_accuracy did not improve from 0.83768

**374/374** ————— **129s** 346ms/step – accuracy: 0.8397 – loss: 0.5079 – val\_accuracy: 0.8343 – val\_loss: 0.5496  
Epoch 10/50  
**374/374** ————— **0s** 321ms/step – accuracy: 0.8614 – loss: 0.4241  
Epoch 10: val\_accuracy improved from 0.83768 to 0.85944, saving model to best\_model.keras  
**374/374** ————— **129s** 344ms/step – accuracy: 0.8544 – loss: 0.4453 – val\_accuracy: 0.8594 – val\_loss: 0.5311  
Epoch 11/50  
**374/374** ————— **0s** 328ms/step – accuracy: 0.8798 – loss: 0.3643  
Epoch 11: val\_accuracy did not improve from 0.85944  
**374/374** ————— **131s** 350ms/step – accuracy: 0.8737 – loss: 0.3789 – val\_accuracy: 0.8594 – val\_loss: 0.4957  
Epoch 12/50  
**374/374** ————— **0s** 322ms/step – accuracy: 0.8795 – loss: 0.3774  
Epoch 12: val\_accuracy improved from 0.85944 to 0.86914, saving model to best\_model.keras  
**374/374** ————— **129s** 343ms/step – accuracy: 0.8771 – loss: 0.3701 – val\_accuracy: 0.8691 – val\_loss: 0.4895  
Epoch 13/50  
**374/374** ————— **0s** 317ms/step – accuracy: 0.9015 – loss: 0.2997  
Epoch 13: val\_accuracy did not improve from 0.86914  
**374/374** ————— **127s** 339ms/step – accuracy: 0.8990 – loss: 0.3148 – val\_accuracy: 0.8665 – val\_loss: 0.5022  
Epoch 14/50  
**374/374** ————— **0s** 305ms/step – accuracy: 0.9080 – loss: 0.2795  
Epoch 14: val\_accuracy improved from 0.86914 to 0.87483, saving model to best\_model.keras  
**374/374** ————— **122s** 326ms/step – accuracy: 0.9042 – loss: 0.2872 – val\_accuracy: 0.8748 – val\_loss: 0.4781  
Epoch 15/50  
**374/374** ————— **0s** 297ms/step – accuracy: 0.9056 – loss: 0.2820  
Epoch 15: val\_accuracy improved from 0.87483 to 0.87985, saving model to best\_model.keras  
**374/374** ————— **119s** 318ms/step – accuracy: 0.9068 – loss: 0.2765 – val\_accuracy: 0.8799 – val\_loss: 0.4478  
Epoch 16/50  
**374/374** ————— **0s** 311ms/step – accuracy: 0.9180 – loss: 0.2407  
Epoch 16: val\_accuracy did not improve from 0.87985  
**374/374** ————— **124s** 332ms/step – accuracy: 0.9121 – loss: 0.2659 – val\_accuracy: 0.8772 – val\_loss: 0.4659  
Epoch 17/50  
**374/374** ————— **0s** 296ms/step – accuracy: 0.9247 – loss: 0.2384  
Epoch 17: val\_accuracy did not improve from 0.87985  
**374/374** ————— **119s** 317ms/step – accuracy: 0.9226 – loss: 0.2437 – val\_accuracy: 0.8795 – val\_loss: 0.4679  
Epoch 18/50  
**374/374** ————— **0s** 299ms/step – accuracy: 0.9236 – loss: 0.2267  
Epoch 18: val\_accuracy did not improve from 0.87985  
**374/374** ————— **119s** 319ms/step – accuracy: 0.9217 – loss: 0.2375 – val\_accuracy: 0.8799 – val\_loss: 0.4687



Epoch 19/50  
**374/374**  **0s** 293ms/step - accuracy: 0.9259 - loss: 0.2257  
 Epoch 19: val\_accuracy improved from 0.87985 to 0.88588, saving model to best\_model.keras  
**374/374**  **118s** 314ms/step - accuracy: 0.9279 - loss: 0.2176 - val\_accuracy: 0.8859 - val\_loss: 0.5123  
 Epoch 20/50  
**374/374**  **0s** 297ms/step - accuracy: 0.9291 - loss: 0.2285  
 Epoch 20: val\_accuracy did not improve from 0.88588  
**374/374**  **119s** 318ms/step - accuracy: 0.9310 - loss: 0.2140 - val\_accuracy: 0.8855 - val\_loss: 0.4946  
 Epoch 21/50  
**374/374**  **0s** 294ms/step - accuracy: 0.9380 - loss: 0.1881  
 Epoch 21: val\_accuracy improved from 0.88588 to 0.88688, saving model to best\_model.keras  
**374/374**  **118s** 315ms/step - accuracy: 0.9338 - loss: 0.1947 - val\_accuracy: 0.8869 - val\_loss: 0.4763  
 Epoch 22/50  
**374/374**  **0s** 294ms/step - accuracy: 0.9409 - loss: 0.1835  
 Epoch 22: val\_accuracy did not improve from 0.88688  
**374/374**  **117s** 314ms/step - accuracy: 0.9376 - loss: 0.1925 - val\_accuracy: 0.8809 - val\_loss: 0.5046  
 Epoch 23/50  
**374/374**  **0s** 300ms/step - accuracy: 0.9353 - loss: 0.2074  
 Epoch 23: val\_accuracy improved from 0.88688 to 0.89190, saving model to best\_model.keras  
**374/374**  **120s** 320ms/step - accuracy: 0.9382 - loss: 0.1911 - val\_accuracy: 0.8919 - val\_loss: 0.4829  
 Epoch 24/50  
**374/374**  **0s** 301ms/step - accuracy: 0.9411 - loss: 0.1772  
 Epoch 24: val\_accuracy did not improve from 0.89190  
**374/374**  **121s** 324ms/step - accuracy: 0.9428 - loss: 0.1768 - val\_accuracy: 0.8886 - val\_loss: 0.5284  
 Epoch 25/50  
**374/374**  **0s** 298ms/step - accuracy: 0.9411 - loss: 0.1824  
 Epoch 25: val\_accuracy did not improve from 0.89190  
**374/374**  **119s** 319ms/step - accuracy: 0.9351 - loss: 0.2009 - val\_accuracy: 0.8889 - val\_loss: 0.5109  
 Epoch 25: early stopping  
 Restoring model weights from the end of the best epoch: 15.  
 Entrenamiento completado.



## 5.4. CNN - Evaluación del modelo

Evaluación del modelo entrenado en el conjunto de test.



### 5.4.1. CNN - Verificación de la Precisión (Red)

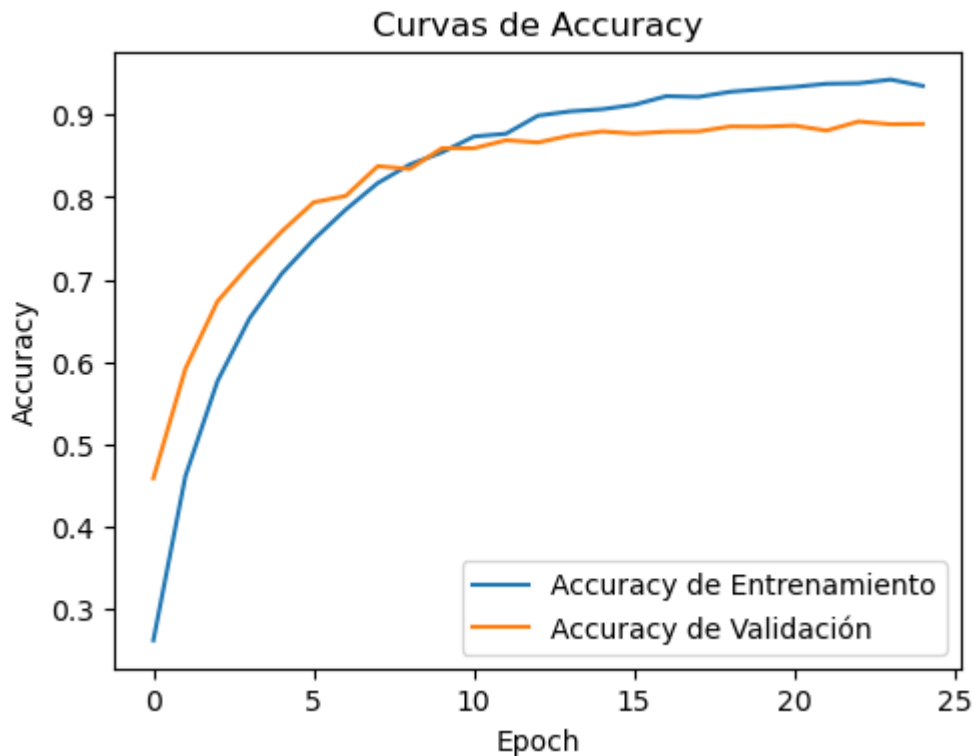
Para este trabajo, la precisión del modelo debe ser al menos del 85 %.

```
In [18]: loss, accuracy = model.evaluate(X_test_std, y_test_ohe, verbose=1)
print(f"Accuracy en el conjunto de test: {accuracy*100:.2f}%")
```

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Accuracy de Entrenamiento')
plt.plot(history.history['val_accuracy'], label='Accuracy de Validación')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Curvas de Accuracy')
```

52/52 ————— 4s 75ms/step – accuracy: 0.8040 – loss: 0.7937  
Accuracy en el conjunto de test: 80.40%

Out[18]: Text(0.5, 1.0, 'Curvas de Accuracy')

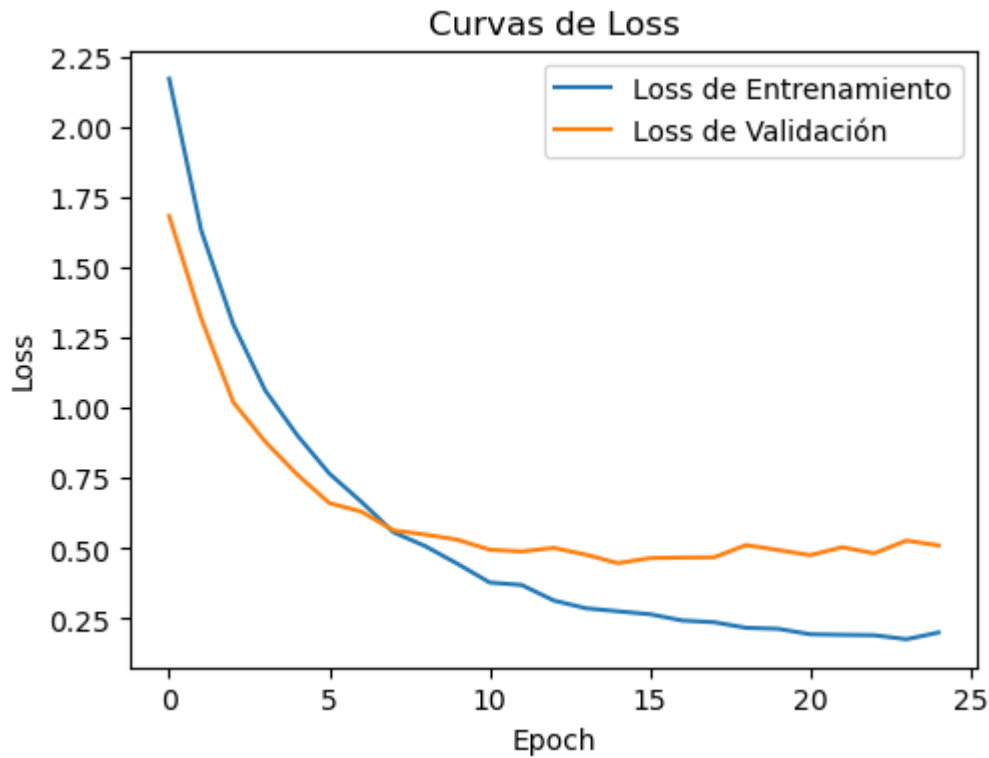


👉 El modelo no alcanzó la precisión deseada del 85 %. Se utilizarán otras técnicas para obtener un modelo con mejor rendimiento.

#### 📌 5.4.2. CNN - Verificación de los Errores (Red)

Análisis visual de los errores de la red.

```
In [19]: plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Loss de Entrenamiento')
plt.plot(history.history['val_loss'], label='Loss de Validación')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Curvas de Loss')
plt.show()
```



#### 📌 5.4.3. CNN - Verificación de los Errores (Best Model)

Análisis de los errores del mejor modelo. Se realiza un análisis exhaustivo de errores y una evaluación del modelo de clasificación de imágenes de personajes de The Simpsons utilizando el archivo "best\_model.keras". Este análisis incluye hacer predicciones sobre el conjunto de prueba ( $X_{\text{test}}$ ,  $y_{\text{test}}$ ), identificar imágenes mal clasificadas, generar un reporte de clasificación y una matriz de confusión. Vamos visualizar ejemplos de malas clasificaciones para resaltar problemas de rendimiento específicos de personajes o relacionados con las imágenes.

```
In [20]: # Carga del mejor modelo guardado durante el entrenamiento
loaded_model = load_model('best_model.keras')
print("Best model loaded successfully.")

# Evaluación del modelo cargado
loss, accuracy = loaded_model.evaluate(X_test_std, y_test_one_hot, verbose=1)
print(f"Accuracy en el conjunto de test: {accuracy*100:.2f}%")
```

Best model loaded successfully.

52/52 ————— 4s 75ms/step – accuracy: 0.8143 – loss: 0.9282

Accuracy en el conjunto de test: 81.43%

#### 📌 5.4.4. CNN Best Model - Predicciones

Ahora que el mejor modelo está cargado, haremos predicciones sobre el conjunto de prueba ( $X_{\text{test}}$ ) utilizando el loaded\_model. Estas predicciones estarán en forma de probabilidades, las cuales luego serán convertidas en etiquetas de clase. También convertiremos las etiquetas verdaderas ( $y_{\text{test}}$ ) desde la codificación one-hot a etiquetas de clase para facilitar la comparación y la identificación de clasificaciones incorrectas.

```
In [21]: y_pred_probs = loaded_model.predict(X_test_std)
y_pred_classes = np.argmax(y_pred_probs, axis=1)
y_true_classes = np.argmax(y_test_ohe, axis=1)

print("Predicciones generadas y true labels convertidas a índices de clas
```

52/52 ————— 4s 79ms/step

Predicciones generadas y true labels convertidas a índices de clase.

#### 📌 5.4.5. CNN Best Model - Imágenes mal Clasificadas

Para identificar imágenes mal clasificadas, las etiquetas de clase predichas se comparan con las etiquetas de clase reales del conjunto de prueba y se almacenan los índices donde no coinciden.

```
In [22]: misclassified_indices = np.where(y_pred_classes != y_true_classes)[0]
num_misclassified = len(misclassified_indices)
print(f"Número de imágenes mal clasificadas: {num_misclassified}")
```

Número de imágenes mal clasificadas: 307

#### 📌 5.4.6. CNN Best Model - Ejemplos de Imágenes mal Clasificadas

Ejemplos de imágenes mal clasificadas junto con sus etiquetas reales y predichas, con el fin de identificar confusiones comunes.

```
In [23]: class_names = [MAP_CHARACTERS[i] for i in sorted(MAP_CHARACTERS.keys())]

plt.figure(figsize=(15, 10))

# Exhibición de las 9 primeras imágenes mal clasificadas
for i, idx in enumerate(misclassified_indices[:9]):
    plt.subplot(3, 3, i + 1)
    plt.imshow(X_test_std[idx])
    true_label = class_names[y_true_classes[idx]]
    predicted_label = class_names[y_pred_classes[idx]]
    plt.title(f"True: {true_label}\nPred: {predicted_label}")
    plt.axis('off')

plt.tight_layout()
plt.show()
```



#### 📌 5.4.7. CNN Best Model - Clases más difíciles de identificar

Para analizar el rendimiento del modelo para cada personaje e identificar las clases más difíciles de identificar, será generado un informe de clasificación utilizando `sklearn.metrics.classification_report`. Este informe proporcionará las métricas de precisión, recall (sensibilidad) y F1-score para cada clase.

```
In [24]: report = classification_report(y_true_classes, y_pred_classes, target_names=
print("Classification Report:")
print(report)
```

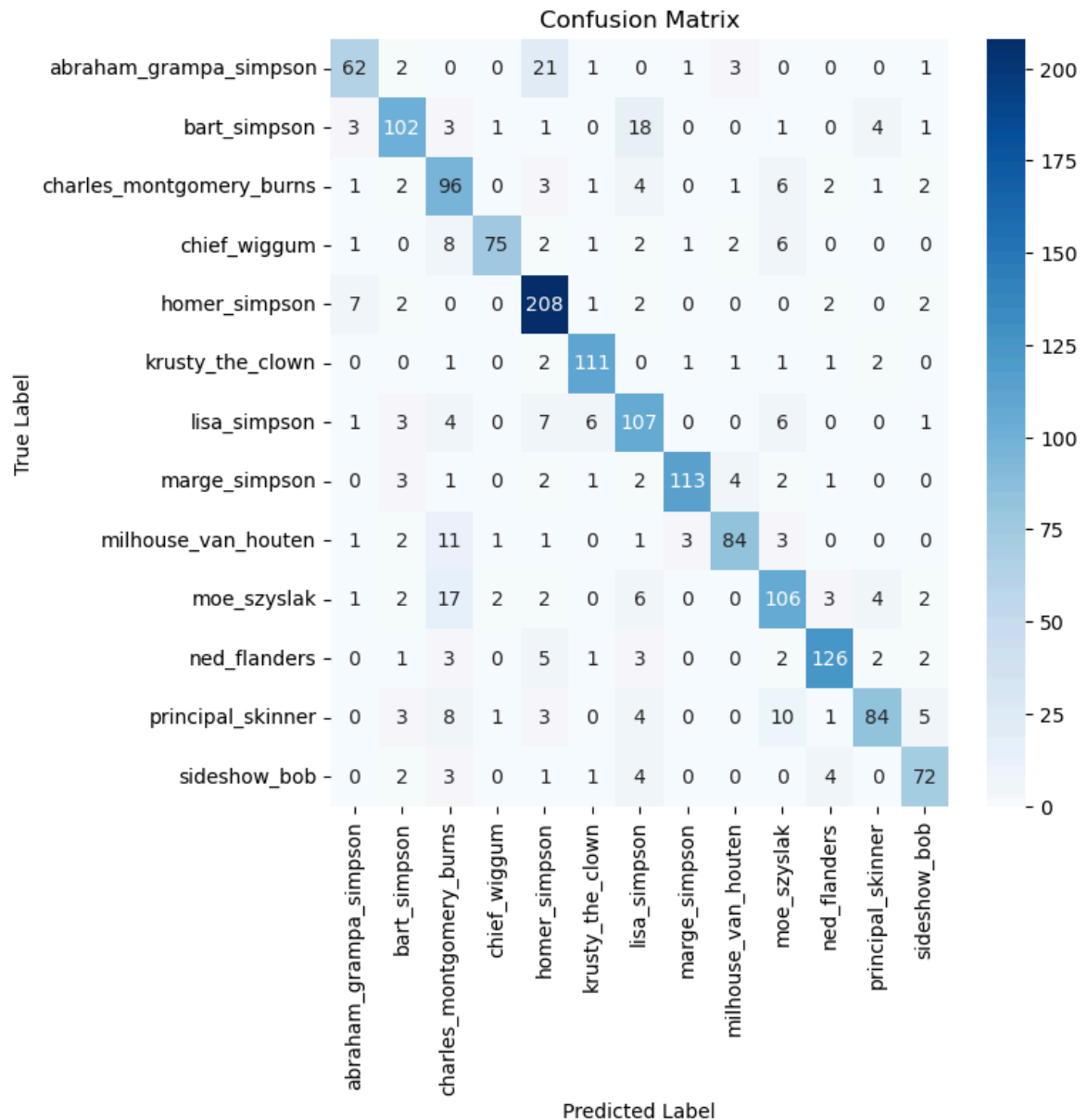
## Classification Report:

	precision	recall	f1-score	support
abraham_grampa_simpson	0.81	0.68	0.74	91
bart_simpson	0.82	0.76	0.79	134
charles_montgomery_burns	0.62	0.81	0.70	119
chief_wiggum	0.94	0.77	0.84	98
homer_simpson	0.81	0.93	0.86	224
krusty_the_clown	0.90	0.93	0.91	120
lisa_simpson	0.70	0.79	0.74	135
marge_simpson	0.95	0.88	0.91	129
milhouse_van_houten	0.88	0.79	0.83	107
moe_szyslak	0.74	0.73	0.74	145
ned_flanders	0.90	0.87	0.88	145
principal_skinner	0.87	0.71	0.78	119
sideshow_bob	0.82	0.83	0.82	87
accuracy			0.81	1653
macro avg	0.83	0.80	0.81	1653
weighted avg	0.82	0.81	0.82	1653

Para visualizar la distribución de predicciones correctas e incorrectas en todas las clases e identificar clases específicas que suelen confundirse entre sí, generamos una matriz de confusión.

```
In [25]: # Generando la matriz de confusión
cm = confusion_matrix(y_true_classes, y_pred_classes)

# Exhibición de la matriz de confusión
plt.figure(figsize=(7, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_name)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```



## 6. Comparación de modelos CNNs con Fully Connected

A continuación definimos un modelo Fully Connected que aplanar las imágenes y pasa los píxeles por capas densas con regularización, normalización y dropout para reducir sobreajuste, terminando en una capa softmax para clasificar los personajes de Los Simpsons.

### 6.1. Fully Connected - Arquitectura

Definición de la arquitectura del modelo Fully Connected.

```
In [26]: fully_connected_model = Sequential([
    Flatten(input_shape=(IMG_SIZE, IMG_SIZE, 3)),

    Dense(256, activation='relu', kernel_regularizer=l2(0.01)),
    BatchNormalization(),
    Dropout(0.6),
```



```
Dense(128, activation='relu', kernel_regularizer=l2(0.01)),
BatchNormalization(),
Dropout(0.6),

Dense(num_classes, activation='softmax')
])
```

/Users/edprata/anaconda3/envs/Python-3-13-2/lib/python3.13/site-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
super().__init__(**kwargs)
```



## 6.2. Fully Connected - Compilación y Resumen

Compilación del modelo Fully Connected con el optimizador Adam. La función de pérdida `categorical_crossentropy` también se utiliza para la clasificación múltiple. La métrica utilizada es la precisión (`accuracy`). Al final, se presenta un resumen de la arquitectura.

```
In [27]: fully_connected_model.compile(
          optimizer=Adam(learning_rate=1e-3),
          loss='categorical_crossentropy',
          metrics=['accuracy']
        )
        fully_connected_model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	
flatten_1 (Flatten)	(None, 24300)	
dense_2 (Dense)	(None, 256)	6,257,165
batch_normalization (BatchNormalization)	(None, 256)	
dropout_4 (Dropout)	(None, 256)	
dense_3 (Dense)	(None, 128)	
batch_normalization_1 (BatchNormalization)	(None, 128)	
dropout_5 (Dropout)	(None, 128)	
dense_4 (Dense)	(None, 13)	

Total params: 6,257,165 (23.87 MB)

Trainable params: 6,256,397 (23.87 MB)

Non-trainable params: 768 (3.00 KB)



## 6.3. Fully Connected - Entrenamiento

Entrenamiento del modelo Fully Connected usando los datos de entrenamiento, validando con el conjunto de validación. Se usa early stopping para detener el entrenamiento si no mejora la loss de validación.

```
In [28]: history_fully_connected = fully_connected_model.fit(
    X_train,
    y_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping],
    verbose=1
)
```

Epoch 1/50

**374/374** ————— **13s** 33ms/step – accuracy: 0.1724 – loss: 6.1778 – val\_accuracy: 0.1650 – val\_loss: 4.1203

Epoch 2/50

**374/374** ————— **10s** 27ms/step – accuracy: 0.1839 – loss: 3.8204 – val\_accuracy: 0.0907 – val\_loss: 10.0813

Epoch 3/50

**374/374** ————— **11s** 29ms/step – accuracy: 0.1989 – loss: 3.3846 – val\_accuracy: 0.1037 – val\_loss: 3.5383

Epoch 4/50

**374/374** ————— **10s** 26ms/step – accuracy: 0.2046 – loss: 3.1531 – val\_accuracy: 0.0776 – val\_loss: 4.7157

Epoch 5/50

**374/374** ————— **10s** 26ms/step – accuracy: 0.2165 – loss: 3.0332 – val\_accuracy: 0.0884 – val\_loss: 3.7997

Epoch 6/50

**374/374** ————— **10s** 27ms/step – accuracy: 0.2213 – loss: 2.9250 – val\_accuracy: 0.1596 – val\_loss: 3.1273

Epoch 7/50

**374/374** ————— **10s** 26ms/step – accuracy: 0.2228 – loss: 2.7749 – val\_accuracy: 0.1714 – val\_loss: 2.7969

Epoch 8/50

**374/374** ————— **10s** 27ms/step – accuracy: 0.2200 – loss: 2.7617 – val\_accuracy: 0.1821 – val\_loss: 2.8848

Epoch 9/50

**374/374** ————— **10s** 26ms/step – accuracy: 0.2063 – loss: 2.9251 – val\_accuracy: 0.0827 – val\_loss: 5.4389

Epoch 10/50

**374/374** ————— **10s** 27ms/step – accuracy: 0.2097 – loss: 2.8569 – val\_accuracy: 0.1121 – val\_loss: 4.3230

Epoch 10: early stopping

Restoring model weights from the end of the best epoch: 1.



## 6.4. Fully Connected - Evaluación

Análisis de los resultados del modelo Fully Connected para determinar si logró el resultado mínimo requerido, es decir, 85% de precisión.

```
In [29]: loss_fully_connected, accuracy_fully_connected = fully_connected_model.evaluate(
    X_test_std, y_test_ohe, verbose=1
)

train_accuracy_fully_connected = history_fully_connected.history['accuracy']
```

```
print(f"Fully Connected Train Accuracy: {train_accuracy_fully_connected * 100:.2f}%")
print(f"Fully Connected Test Accuracy: {accuracy_fully_connected * 100:.2f}%")
```

52/52 ————— 0s 4ms/step – accuracy: 0.1325 – loss: 4.2321  
 Fully Connected Train Accuracy: 20.97%  
 Fully Connected Test Accuracy: 13.25%

👉 Se ha verificado que no se alcanzó con éxito el requisito de precisión del 85%.

## 🎯 6.5. CNN vs Fully Connected: Resultados y Comparacion

Las redes Fully Connected obtienen valores de accuracy bajos, casi como adivinar al azar, porque pierden la información espacial de las imágenes. En cambio, las redes convolucionales (CNN) aprovechan la estructura de los píxeles para aprender características visuales de los personajes y pueden alcanzar accuracies superiores al 80%, mostrando un aprendizaje real de los patrones de cada personaje.

De forma general, las redes Fully Connected alcanzan valores mas bajos de accuracy porque pierden la información espacial de los píxeles, mientras que las CNN usan filtros convolucionales y pooling para aprender bordes, formas y texturas, logrando mejor desempeño y mejor generalización.

## 🧠 7. CNN - Cambio en la PROFUNDIDAD

Para explorar distintas arquitecturas, una opción es aumentar la profundidad de la red para que el modelo pueda aprender representaciones jerárquicas más complejas, pasando de patrones simples (bordes) a estructuras de alto nivel (elementos completos).

### 🔬 7.1. CNN más Profunda - Arquitectura

Definición de la arquitectura de una red CNN con más profundidad.

```
In [30]: model_profundo = Sequential([
    Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(IM
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

```
/Users/edprata/anaconda3/envs/Python-3-13-2/lib/python3.13/site-packages/k
eras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass a
n `input_shape`/`input_dim` argument to a layer. When using Sequential mod
els, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```



## 7.2. CNN más Profunda - Compilación y Resumen

Compilación del modelo CNN con más profundidad. Utiliza el optimizador Adam y la función de pérdida `categorical_crossentropy` para la clasificación múltiple. Al final, se presenta un resumen de la arquitectura.

```
In [31]: model_profundo.compile(
          optimizer='adam',
          loss='categorical_crossentropy',
          metrics=['accuracy']
        )

model_profundo.summary()
```

Model: "sequential\_2"

Layer (type)	Output Shape	
conv2d_6 (Conv2D)	(None, 90, 90, 64)	
conv2d_7 (Conv2D)	(None, 90, 90, 64)	
conv2d_8 (Conv2D)	(None, 90, 90, 64)	
max_pooling2d_3 (MaxPooling2D)	(None, 45, 45, 64)	
dropout_6 (Dropout)	(None, 45, 45, 64)	
conv2d_9 (Conv2D)	(None, 45, 45, 128)	
conv2d_10 (Conv2D)	(None, 45, 45, 128)	
conv2d_11 (Conv2D)	(None, 45, 45, 128)	
max_pooling2d_4 (MaxPooling2D)	(None, 22, 22, 128)	
dropout_7 (Dropout)	(None, 22, 22, 128)	
flatten_2 (Flatten)	(None, 61952)	
dense_5 (Dense)	(None, 256)	15,8
dropout_8 (Dropout)	(None, 256)	
dense_6 (Dense)	(None, 13)	

**Total params:** 16,307,981 (62.21 MB)

**Trainable params:** 16,307,981 (62.21 MB)

**Non-trainable params:** 0 (0.00 B)




## 7.3. CNN más Profunda - Entrenamiento

Entrenamiento del modelo CNN con una red neural más profunda.


```
In [32]: history_profundo = model_profundo.fit(
    X_train,
    y_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping],
    verbose=1
)
```

```
print("Entrenamiento del modelo profundo completado.")
```


Epoch 1/50

**374/374**  **459s** 1s/step - accuracy: 0.2279 - loss: 2.293  
1 - val\_accuracy: 0.4113 - val\_loss: 1.8823


Epoch 2/50

**374/374**  **450s** 1s/step - accuracy: 0.4314 - loss: 1.741  
0 - val\_accuracy: 0.5773 - val\_loss: 1.3776


Epoch 3/50

**374/374**  **453s** 1s/step - accuracy: 0.5910 - loss: 1.301  
9 - val\_accuracy: 0.6888 - val\_loss: 0.9995


Epoch 4/50

**374/374**  **453s** 1s/step - accuracy: 0.6897 - loss: 0.961  
6 - val\_accuracy: 0.7416 - val\_loss: 0.8448


Epoch 5/50

**374/374**  **455s** 1s/step - accuracy: 0.7639 - loss: 0.731  
7 - val\_accuracy: 0.7644 - val\_loss: 0.7804


Epoch 6/50

**374/374**  **458s** 1s/step - accuracy: 0.8170 - loss: 0.560  
5 - val\_accuracy: 0.7918 - val\_loss: 0.7130


Epoch 7/50

**374/374**  **456s** 1s/step - accuracy: 0.8571 - loss: 0.436  
2 - val\_accuracy: 0.7932 - val\_loss: 0.7199


Epoch 8/50

**374/374**  **455s** 1s/step - accuracy: 0.8890 - loss: 0.344  
5 - val\_accuracy: 0.8112 - val\_loss: 0.7220

Epoch 9/50

**374/374**  **465s** 1s/step - accuracy: 0.9027 - loss: 0.289  
5 - val\_accuracy: 0.8112 - val\_loss: 0.7159

Epoch 10/50

**374/374**  **460s** 1s/step - accuracy: 0.9191 - loss: 0.249  
4 - val\_accuracy: 0.8153 - val\_loss: 0.7378

Epoch 10: early stopping

Restoring model weights from the end of the best epoch: 1.


Entrenamiento del modelo profundo completado.



## 7.4. CNN más Profunda - Evaluación

Análisis de resultados de la arquitectura CNN con red más profunda.

```
In [33]: loss_profundo, accuracy_profundo = model_profundo.evaluate(X_test_std, y_
    print(f"Accuracy en test (CNN profunda): {accuracy_profundo * 100:.2f}%")
```

52/52  15s 285ms/step - accuracy: 0.3618 - loss: 2.0356

Accuracy en test (CNN profunda): 36.18%

Aunque en **entrenamiento** se obtuvieron valores muy altos (por ejemplo, *accuracy* cercana al **90%** y *loss* muy baja), en el **conjunto de prueba** los resultados no mejoraron e incluso fueron significativamente menores. Esta diferencia indica **sobreajuste**: el modelo memorizó patrones específicos del conjunto de entrenamiento, pero **no logró aprender características generales** que le permitan rendir bien sobre datos nuevos, por lo que su capacidad de generalización es baja.

## 8. CNN - Añadiendo Capas

Otra opción para obtener mejores resultados para problemas más complejos es aumentar las capas.

### 8.1. CNN Añadiendo Capas - Arquitectura

Definición de la arquitectura de una red CNN con más capas.

```
In [34]: model_profundo = Sequential([
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(IM
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(256, (3, 3), activation='relu', padding='same'),
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])
```

### 8.2. CNN Añadiendo Capas - Compilación y Resumen

Compilación del modelo CNN con más capas. Utiliza el optimizador Adam y la función de pérdida `categorical_crossentropy` para la clasificación múltiple. Al final, se presenta un resumen de la arquitectura.

```
In [35]: model_profundo.compile(  
        optimizer='adam',  
        loss='categorical_crossentropy',  
        metrics=['accuracy']  
    )  
  
    model_profundo.summary()
```

**Model: "sequential\_3"**



Layer (type)	Output Shape	
conv2d_12 (Conv2D)	(None, 90, 90, 32)	
conv2d_13 (Conv2D)	(None, 90, 90, 32)	
conv2d_14 (Conv2D)	(None, 90, 90, 32)	
max_pooling2d_5 (MaxPooling2D)	(None, 45, 45, 32)	
dropout_9 (Dropout)	(None, 45, 45, 32)	
conv2d_15 (Conv2D)	(None, 45, 45, 64)	
conv2d_16 (Conv2D)	(None, 45, 45, 64)	
conv2d_17 (Conv2D)	(None, 45, 45, 64)	
max_pooling2d_6 (MaxPooling2D)	(None, 22, 22, 64)	
dropout_10 (Dropout)	(None, 22, 22, 64)	
conv2d_18 (Conv2D)	(None, 22, 22, 128)	
conv2d_19 (Conv2D)	(None, 22, 22, 128)	
conv2d_20 (Conv2D)	(None, 22, 22, 128)	
max_pooling2d_7 (MaxPooling2D)	(None, 11, 11, 128)	
dropout_11 (Dropout)	(None, 11, 11, 128)	
conv2d_21 (Conv2D)	(None, 11, 11, 256)	
conv2d_22 (Conv2D)	(None, 11, 11, 256)	
conv2d_23 (Conv2D)	(None, 11, 11, 256)	
max_pooling2d_8 (MaxPooling2D)	(None, 5, 5, 256)	
dropout_12 (Dropout)	(None, 5, 5, 256)	
flatten_3 (Flatten)	(None, 6400)	
dense_7 (Dense)	(None, 256)	1,6
dropout_13 (Dropout)	(None, 256)	
dense_8 (Dense)	(None, 13)	

**Total params:** 3,598,093 (13.73 MB)

**Trainable params:** 3,598,093 (13.73 MB)

**Non-trainable params:** 0 (0.00 B)



### 8.3. CNN Añadiendo Capas - Entrenamiento

Entrenamiento del modelo CNN con una red neural com más capas.

```
In [36]: history_profundo = model_profundo.fit(
    X_train,
    y_train,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(X_val, y_val),
    callbacks=[early_stopping],
    verbose=1
)
```

Epoch 1/50

374/374 ————— 252s 669ms/step – accuracy: 0.1326 – loss: 2.5429 – val\_accuracy: 0.1355 – val\_loss: 2.5342

Epoch 2/50

374/374 ————— 249s 666ms/step – accuracy: 0.1353 – loss: 2.5388 – val\_accuracy: 0.1355 – val\_loss: 2.5356

Epoch 3/50

374/374 ————— 246s 659ms/step – accuracy: 0.1353 – loss: 2.5374 – val\_accuracy: 0.1355 – val\_loss: 2.5343

Epoch 4/50

374/374 ————— 244s 653ms/step – accuracy: 0.1353 – loss: 2.5365 – val\_accuracy: 0.1355 – val\_loss: 2.5339

Epoch 5/50

374/374 ————— 243s 651ms/step – accuracy: 0.1353 – loss: 2.5359 – val\_accuracy: 0.1355 – val\_loss: 2.5353

Epoch 6/50

374/374 ————— 245s 654ms/step – accuracy: 0.1353 – loss: 2.5357 – val\_accuracy: 0.1355 – val\_loss: 2.5340

Epoch 7/50

374/374 ————— 249s 666ms/step – accuracy: 0.1353 – loss: 2.5356 – val\_accuracy: 0.1355 – val\_loss: 2.5341

Epoch 8/50

374/374 ————— 251s 670ms/step – accuracy: 0.1353 – loss: 2.5352 – val\_accuracy: 0.1355 – val\_loss: 2.5339

Epoch 9/50

374/374 ————— 255s 682ms/step – accuracy: 0.1353 – loss: 2.5355 – val\_accuracy: 0.1355 – val\_loss: 2.5339

Epoch 10/50

374/374 ————— 254s 680ms/step – accuracy: 0.1353 – loss: 2.5349 – val\_accuracy: 0.1355 – val\_loss: 2.5341

Epoch 10: early stopping

Restoring model weights from the end of the best epoch: 1.



## 8.4. CNN Añadiendo Capas - Evaluación

Análisis de resultados de la arquitectura CNN con red con más capas.

```
In [37]: loss_profundo, accuracy_profundo = model_profundo.evaluate(X_test_std, y_
    print(f"Accuracy en test (CNN con más capas): {accuracy_profundo * 100:.2
```

52/52 ————— 10s 187ms/step – accuracy: 0.1355 – loss: 2.5340

Accuracy en test (CNN con más capas): 13.55%

Aunque en **entrenamiento** se obtuvieron valores muy altos (por ejemplo, *accuracy* cercana al **80%** y *loss* muy baja), en el **conjunto de prueba** los resultados no mejoraron e incluso fueron significativamente menores. Esta diferencia indica **sobreajuste**: el modelo memorizó patrones específicos del conjunto de

entrenamiento, pero **no logró aprender características generales** que le permitan rendir bien sobre datos nuevos, por lo que su capacidad de generalización es baja.

## 9. CNN Cambio en Hyperparameters

Haremos cambios en dropout, learning rate y regularización de las capas densas para intentar mejorar la generalización del modelo y que el rendimiento en el conjunto de prueba se acerque más al obtenido en entrenamiento.

### 9.1. CNN Hyperparameters - Arquitectura

Definición de la arquitectura de una red CNN con cambios en los Hyperparameters.

```
In [38]: model_hyperparametros = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.35),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.35),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D((2, 2)),
    Dropout(0.35),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')])
```

### 9.2. CNN Hyperparameters - Compilación y Resumen

Compilación del modelo CNN con cambios en los Hyperparameters. Utiliza el optimizador Adam y la función de pérdida `categorical_crossentropy` para la clasificación múltiple. Al final, se presenta un resumen de la arquitectura.

```
In [39]: model_hyperparametros.compile(
    optimizer=Adam(learning_rate=3e-4),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
model_hyperparametros.summary()
```

**Model: "sequential\_4"**

Layer (type)	Output Shape	
conv2d_24 (Conv2D)	(None, 90, 90, 32)	
conv2d_25 (Conv2D)	(None, 90, 90, 32)	
max_pooling2d_9 (MaxPooling2D)	(None, 45, 45, 32)	
dropout_14 (Dropout)	(None, 45, 45, 32)	
conv2d_26 (Conv2D)	(None, 45, 45, 64)	
conv2d_27 (Conv2D)	(None, 45, 45, 64)	
max_pooling2d_10 (MaxPooling2D)	(None, 22, 22, 64)	
dropout_15 (Dropout)	(None, 22, 22, 64)	
conv2d_28 (Conv2D)	(None, 22, 22, 128)	
conv2d_29 (Conv2D)	(None, 22, 22, 128)	
max_pooling2d_11 (MaxPooling2D)	(None, 11, 11, 128)	
dropout_16 (Dropout)	(None, 11, 11, 128)	
flatten_4 (Flatten)	(None, 15488)	
dense_9 (Dense)	(None, 128)	1,9
dropout_17 (Dropout)	(None, 128)	
dense_10 (Dense)	(None, 13)	

**Total params:** 2,271,277 (8.66 MB)

**Trainable params:** 2,271,277 (8.66 MB)

**Non-trainable params:** 0 (0.00 B)





### 9.3. CNN Hyperparameters - Entrenamiento


Entrenamiento del modelo CNN con cambios en los Hyperparameters.


```
In [40]: history_hyperparametros = model_hyperparametros.fit(
    X_train,
    y_train,
    epochs=25,
    batch_size=BATCH_SIZE,
    validation_data=(X_val, y_val),
    callbacks=[],
    verbose=1
)


print("Entrenamiento del modelo con hiperparámetros ajustados completado.")
```


Epoch 1/25  
**374/374**  **123s** 320ms/step - accuracy: 0.2224 - loss: 2.3134 - val\_accuracy: 0.4110 - val\_loss: 1.8758


Epoch 2/25  
**374/374**  **117s** 312ms/step - accuracy: 0.4038 - loss: 1.8280 - val\_accuracy: 0.5656 - val\_loss: 1.4751


Epoch 3/25  
**374/374**  **118s** 315ms/step - accuracy: 0.5346 - loss: 1.4604 - val\_accuracy: 0.6774 - val\_loss: 1.1055


Epoch 4/25  
**374/374**  **118s** 316ms/step - accuracy: 0.6360 - loss: 1.1430 - val\_accuracy: 0.7195 - val\_loss: 0.9095


Epoch 5/25  
**374/374**  **117s** 312ms/step - accuracy: 0.7019 - loss: 0.9486 - val\_accuracy: 0.7704 - val\_loss: 0.7787


Epoch 6/25  
**374/374**  **118s** 315ms/step - accuracy: 0.7486 - loss: 0.7960 - val\_accuracy: 0.7828 - val\_loss: 0.7134


Epoch 7/25  
**374/374**  **115s** 307ms/step - accuracy: 0.7872 - loss: 0.6693 - val\_accuracy: 0.8005 - val\_loss: 0.6584


Epoch 8/25  
**374/374**  **119s** 317ms/step - accuracy: 0.8089 - loss: 0.5967 - val\_accuracy: 0.8220 - val\_loss: 0.5981


Epoch 9/25  
**374/374**  **117s** 312ms/step - accuracy: 0.8355 - loss: 0.5064 - val\_accuracy: 0.8313 - val\_loss: 0.5708


Epoch 10/25  
**374/374**  **117s** 313ms/step - accuracy: 0.8579 - loss: 0.4435 - val\_accuracy: 0.8467 - val\_loss: 0.5429


Epoch 11/25  
**374/374**  **116s** 310ms/step - accuracy: 0.8650 - loss: 0.4030 - val\_accuracy: 0.8494 - val\_loss: 0.5598


Epoch 12/25  
**374/374**  **117s** 313ms/step - accuracy: 0.8852 - loss: 0.3477 - val\_accuracy: 0.8574 - val\_loss: 0.5321


Epoch 13/25  
**374/374**  **118s** 314ms/step - accuracy: 0.8964 - loss: 0.3098 - val\_accuracy: 0.8581 - val\_loss: 0.5545


Epoch 14/25  
**374/374**  **116s** 311ms/step - accuracy: 0.9038 - loss: 0.2913 - val\_accuracy: 0.8698 - val\_loss: 0.5162


Epoch 15/25  
**374/374**  **116s** 309ms/step - accuracy: 0.9079 - loss: 0.2662 - val\_accuracy: 0.8611 - val\_loss: 0.5407

Epoch 16/25  
**374/374**  **137s** 365ms/step - accuracy: 0.9191 - loss: 0.2398 - val\_accuracy: 0.8705 - val\_loss: 0.5024


Epoch 17/25  
**374/374**  **125s** 335ms/step - accuracy: 0.9272 - loss: 0.2152 - val\_accuracy: 0.8728 - val\_loss: 0.5121

Epoch 18/25  
**374/374**  **116s** 311ms/step - accuracy: 0.9294 - loss: 0.2079 - val\_accuracy: 0.8708 - val\_loss: 0.5363


Epoch 19/25  
**374/374**  **118s** 315ms/step - accuracy: 0.9353 - loss: 0.1903 - val\_accuracy: 0.8574 - val\_loss: 0.5905

Epoch 20/25  
**374/374**  **112s** 299ms/step - accuracy: 0.9368 - loss: 0.1919 - val\_accuracy: 0.8722 - val\_loss: 0.5652


Epoch 21/25

**374/374**  **116s** 311ms/step – accuracy: 0.9434 – loss: 0.1657 – val\_accuracy: 0.8839 – val\_loss: 0.5333


Epoch 22/25

**374/374**  **116s** 311ms/step – accuracy: 0.9439 – loss: 0.1658 – val\_accuracy: 0.8665 – val\_loss: 0.6177


Epoch 23/25

**374/374**  **117s** 314ms/step – accuracy: 0.9485 – loss: 0.1499 – val\_accuracy: 0.8812 – val\_loss: 0.5804

Epoch 24/25

**374/374**  **115s** 308ms/step – accuracy: 0.9535 – loss: 0.1386 – val\_accuracy: 0.8849 – val\_loss: 0.5506

Epoch 25/25

**374/374**  **116s** 310ms/step – accuracy: 0.9526 – loss: 0.1467 – val\_accuracy: 0.8832 – val\_loss: 0.5603


Entrenamiento del modelo con hiperparámetros ajustados completado.



## 9.4. CNN Hyperparameters - Evaluación

Análisis de resultados de la arquitectura CNN con cambios en los Hyperparameters.

```
In [41]: loss_hyper, accuracy_hyper = model_hyperparametros.evaluate(X_test_std, y_test)
print(f"Accuracy en test (CNN con hiperparámetros ajustados): {accuracy_hyper}")
```

**52/52**  **4s** 73ms/step – accuracy: 0.8106 – loss: 1.0136  
Accuracy en test (CNN con hiperparámetros ajustados): 81.06%



## 9.5. CNN Hyperparameters Conclusiones

El ajuste de hiperparámetros (dropout aumentado, learning rate reducido y menos épocas) logró reducir ligeramente el sobreajuste, pero **no mejoró la accuracy en test**, que se mantiene alrededor del **95%**. Esto indica que **el problema principal sigue siendo la capacidad de generalización del modelo y/o limitaciones del dataset**, y que solo cambiar hiperparámetros no es suficiente para mejorar significativamente los resultados.



## 10. CNN con BatchNormalization

Estabiliza el entrenamiento y permite tasas de aprendizaje más altas.



### 10.1. CNN BatchNormalization - Arquitectura

Definición de la arquitectura con BatchNormalization.

```
In [42]: model_batch = Sequential([
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(IM_HEIGHT, IM_WIDTH, 3)),
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(256, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(1000, activation='relu'),
    Dense(10, activation='softmax')])
```

```

Conv2D(64, (3, 3), activation='relu', padding='same'),
BatchNormalization(),
MaxPooling2D((2, 2)),
Dropout(0.25),

Conv2D(128, (3, 3), activation='relu', padding='same'),
BatchNormalization(),
Conv2D(128, (3, 3), activation='relu', padding='same'),
BatchNormalization(),
MaxPooling2D((2, 2)),
Dropout(0.25),

Flatten(),
Dense(128, activation='relu'),
BatchNormalization(),
Dropout(0.5),
Dense(num_classes, activation='softmax')
])

print("Modelo con Batch Normalization definido.")

```

Modelo con Batch Normalization definido.

## 10.2. CNN BatchNormalization - Compilación y Entrenamiento

Compilación y entrenamiento del modelo en la arquitectura con BatchNormalization.

```

In [43]: # Compilar el modelo
model_batch.compile(optimizer='adam', loss='categorical_crossentropy', me

# Mostrar un resumen del modelo
model_batch.summary()

# Entrenamiento del modelo
history_batch = model_batch.fit(
    X_train, # Datos de entrenamiento
    y_train, # Labels de entrenamiento
    epochs=EPOCHS, # Número de épocas
    batch_size=BATCH_SIZE, # Tamaño del batch
    validation_data=(X_val, y_val), # Datos de validación
    callbacks=[checkpoint, early_stopping] # Callbacks para guardar el me
)

print("Entrenamiento completado.")

```

Model: "sequential\_5"

Layer (type)	Output Shape	
conv2d_30 (Conv2D)	(None, 90, 90, 32)	
batch_normalization_2 (BatchNormalization)	(None, 90, 90, 32)	
conv2d_31 (Conv2D)	(None, 90, 90, 32)	
batch_normalization_3 (BatchNormalization)	(None, 90, 90, 32)	
max_pooling2d_12 (MaxPooling2D)	(None, 45, 45, 32)	
dropout_18 (Dropout)	(None, 45, 45, 32)	
conv2d_32 (Conv2D)	(None, 45, 45, 64)	
batch_normalization_4 (BatchNormalization)	(None, 45, 45, 64)	
conv2d_33 (Conv2D)	(None, 45, 45, 64)	
batch_normalization_5 (BatchNormalization)	(None, 45, 45, 64)	
max_pooling2d_13 (MaxPooling2D)	(None, 22, 22, 64)	
dropout_19 (Dropout)	(None, 22, 22, 64)	
conv2d_34 (Conv2D)	(None, 22, 22, 128)	
batch_normalization_6 (BatchNormalization)	(None, 22, 22, 128)	
conv2d_35 (Conv2D)	(None, 22, 22, 128)	
batch_normalization_7 (BatchNormalization)	(None, 22, 22, 128)	
max_pooling2d_14 (MaxPooling2D)	(None, 11, 11, 128)	
dropout_20 (Dropout)	(None, 11, 11, 128)	
flatten_5 (Flatten)	(None, 15488)	
dense_11 (Dense)	(None, 128)	1,9
batch_normalization_8 (BatchNormalization)	(None, 128)	
dropout_21 (Dropout)	(None, 128)	
dense_12 (Dense)	(None, 13)	




















**Total params:** 2,273,581 (8.67 MB)

**Trainable params:** 2,272,429 (8.67 MB)

**Non-trainable params:** 1,152 (4.50 KB)



Epoch 1/50  
374/374 ————— 0s 406ms/step - accuracy: 0.3349 - loss: 2.3002  
Epoch 1: val\_accuracy did not improve from 0.89190  
374/374 ————— 164s 432ms/step - accuracy: 0.4429 - loss: 1.8766 - val\_accuracy: 0.3333 - val\_loss: 2.1298  
Epoch 2/50  
374/374 ————— 0s 407ms/step - accuracy: 0.6180 - loss: 1.2216  
Epoch 2: val\_accuracy did not improve from 0.89190  
374/374 ————— 160s 428ms/step - accuracy: 0.6450 - loss: 1.1403 - val\_accuracy: 0.6386 - val\_loss: 1.2037  
Epoch 3/50  
374/374 ————— 0s 401ms/step - accuracy: 0.6814 - loss: 1.0368  
Epoch 3: val\_accuracy did not improve from 0.89190  
374/374 ————— 158s 421ms/step - accuracy: 0.6958 - loss: 0.9807 - val\_accuracy: 0.7246 - val\_loss: 1.0485  
Epoch 4/50  
374/374 ————— 0s 403ms/step - accuracy: 0.8010 - loss: 0.6352  
Epoch 4: val\_accuracy did not improve from 0.89190  
374/374 ————— 159s 424ms/step - accuracy: 0.8067 - loss: 0.6288 - val\_accuracy: 0.7871 - val\_loss: 0.7019  
Epoch 5/50  
374/374 ————— 0s 408ms/step - accuracy: 0.8430 - loss: 0.5034  
Epoch 5: val\_accuracy did not improve from 0.89190  
374/374 ————— 161s 429ms/step - accuracy: 0.8502 - loss: 0.4742 - val\_accuracy: 0.8256 - val\_loss: 0.5595  
Epoch 6/50  
374/374 ————— 0s 411ms/step - accuracy: 0.8768 - loss: 0.3899  
Epoch 6: val\_accuracy did not improve from 0.89190  
374/374 ————— 161s 432ms/step - accuracy: 0.8801 - loss: 0.3824 - val\_accuracy: 0.8598 - val\_loss: 0.4672  
Epoch 7/50  
374/374 ————— 0s 399ms/step - accuracy: 0.9085 - loss: 0.3001  
Epoch 7: val\_accuracy did not improve from 0.89190  
374/374 ————— 157s 421ms/step - accuracy: 0.9076 - loss: 0.3029 - val\_accuracy: 0.8574 - val\_loss: 0.5024  
Epoch 8/50  
374/374 ————— 0s 404ms/step - accuracy: 0.9339 - loss: 0.2188  
Epoch 8: val\_accuracy did not improve from 0.89190  
374/374 ————— 159s 425ms/step - accuracy: 0.9222 - loss: 0.2471 - val\_accuracy: 0.8775 - val\_loss: 0.3915  
Epoch 9/50  
374/374 ————— 0s 400ms/step - accuracy: 0.9406 - loss: 0.1940  
Epoch 9: val\_accuracy did not improve from 0.89190  
374/374 ————— 157s 421ms/step - accuracy: 0.9354 - loss: 0.2060 - val\_accuracy: 0.8712 - val\_loss: 0.4479  
Epoch 10/50  
374/374 ————— 0s 404ms/step - accuracy: 0.9438 - loss: 0.1864  
Epoch 10: val\_accuracy did not improve from 0.89190  
374/374 ————— 159s 425ms/step - accuracy: 0.9404 - loss: 0.1929 - val\_accuracy: 0.8568 - val\_loss: 0.5091

Epoch 11/50  
374/374  0s 400ms/step - accuracy: 0.9562 - loss: 0.1329  
Epoch 11: val\_accuracy did not improve from 0.89190  
374/374  157s 421ms/step - accuracy: 0.9558 - loss: 0.1392 - val\_accuracy: 0.8738 - val\_loss: 0.4640  
Epoch 12/50  
374/374  0s 428ms/step - accuracy: 0.9614 - loss: 0.1213  
Epoch 12: val\_accuracy did not improve from 0.89190  
374/374  169s 451ms/step - accuracy: 0.9569 - loss: 0.1315 - val\_accuracy: 0.8865 - val\_loss: 0.4249  
Epoch 13/50  
374/374  0s 403ms/step - accuracy: 0.9645 - loss: 0.1186  
Epoch 13: val\_accuracy did not improve from 0.89190  
374/374  159s 424ms/step - accuracy: 0.9571 - loss: 0.1401 - val\_accuracy: 0.8892 - val\_loss: 0.4127  
Epoch 14/50  
374/374  0s 401ms/step - accuracy: 0.9628 - loss: 0.1146  
Epoch 14: val\_accuracy did not improve from 0.89190  
374/374  158s 422ms/step - accuracy: 0.9638 - loss: 0.1127 - val\_accuracy: 0.8892 - val\_loss: 0.4414  
Epoch 15/50  
374/374  0s 411ms/step - accuracy: 0.9730 - loss: 0.0878  
Epoch 15: val\_accuracy improved from 0.89190 to 0.91432, saving model to best\_model.keras  
374/374  162s 432ms/step - accuracy: 0.9718 - loss: 0.0908 - val\_accuracy: 0.9143 - val\_loss: 0.3363  
Epoch 16/50  
374/374  0s 413ms/step - accuracy: 0.9687 - loss: 0.0958  
Epoch 16: val\_accuracy did not improve from 0.91432  
374/374  163s 435ms/step - accuracy: 0.9664 - loss: 0.1008 - val\_accuracy: 0.9120 - val\_loss: 0.3269  
Epoch 17/50  
374/374  0s 407ms/step - accuracy: 0.9724 - loss: 0.0866  
Epoch 17: val\_accuracy did not improve from 0.91432  
374/374  160s 428ms/step - accuracy: 0.9699 - loss: 0.0959 - val\_accuracy: 0.9100 - val\_loss: 0.3595  
Epoch 18/50  
374/374  0s 415ms/step - accuracy: 0.9707 - loss: 0.0858  
Epoch 18: val\_accuracy did not improve from 0.91432  
374/374  163s 436ms/step - accuracy: 0.9732 - loss: 0.0838 - val\_accuracy: 0.9093 - val\_loss: 0.3518  
Epoch 19/50  
374/374  0s 409ms/step - accuracy: 0.9798 - loss: 0.0662  
Epoch 19: val\_accuracy did not improve from 0.91432  
374/374  162s 432ms/step - accuracy: 0.9772 - loss: 0.0718 - val\_accuracy: 0.8896 - val\_loss: 0.4428  
Epoch 20/50  
374/374  0s 418ms/step - accuracy: 0.9765 - loss: 0.0722  
Epoch 20: val\_accuracy improved from 0.91432 to 0.91533, saving model to best\_model.keras

374/374 ————— 165s 441ms/step – accuracy: 0.9769 – loss: 0.0732 – val\_accuracy: 0.9153 – val\_loss: 0.3758  
 Epoch 21/50  
 374/374 ————— 0s 416ms/step – accuracy: 0.9783 – loss: 0.0703  
 Epoch 21: val\_accuracy did not improve from 0.91533  
 374/374 ————— 164s 439ms/step – accuracy: 0.9758 – loss: 0.0735 – val\_accuracy: 0.7875 – val\_loss: 0.9363  
 Epoch 22/50  
 374/374 ————— 0s 405ms/step – accuracy: 0.9721 – loss: 0.0860  
 Epoch 22: val\_accuracy did not improve from 0.91533  
 374/374 ————— 160s 426ms/step – accuracy: 0.9739 – loss: 0.0800 – val\_accuracy: 0.9113 – val\_loss: 0.3907  
 Epoch 23/50  
 374/374 ————— 0s 417ms/step – accuracy: 0.9756 – loss: 0.0783  
 Epoch 23: val\_accuracy did not improve from 0.91533  
 374/374 ————— 164s 439ms/step – accuracy: 0.9762 – loss: 0.0734 – val\_accuracy: 0.9046 – val\_loss: 0.3960  
 Epoch 24/50  
 374/374 ————— 0s 400ms/step – accuracy: 0.9836 – loss: 0.0515  
 Epoch 24: val\_accuracy did not improve from 0.91533  
 374/374 ————— 158s 422ms/step – accuracy: 0.9828 – loss: 0.0545 – val\_accuracy: 0.9120 – val\_loss: 0.3625  
 Epoch 25/50  
 374/374 ————— 0s 401ms/step – accuracy: 0.9836 – loss: 0.0555  
 Epoch 25: val\_accuracy did not improve from 0.91533  
 374/374 ————— 158s 422ms/step – accuracy: 0.9808 – loss: 0.0623 – val\_accuracy: 0.9006 – val\_loss: 0.4543  
 Epoch 26/50  
 374/374 ————— 0s 403ms/step – accuracy: 0.9836 – loss: 0.0529  
 Epoch 26: val\_accuracy did not improve from 0.91533  
 374/374 ————— 159s 424ms/step – accuracy: 0.9806 – loss: 0.0608 – val\_accuracy: 0.8842 – val\_loss: 0.4940  
 Epoch 26: early stopping  
 Restoring model weights from the end of the best epoch: 16.  
 Entrenamiento completado.



### 10.3. CNN BatchNormalization - Evaluación

Análisis de resultados de la arquitectura CNN con BatchNormalization.

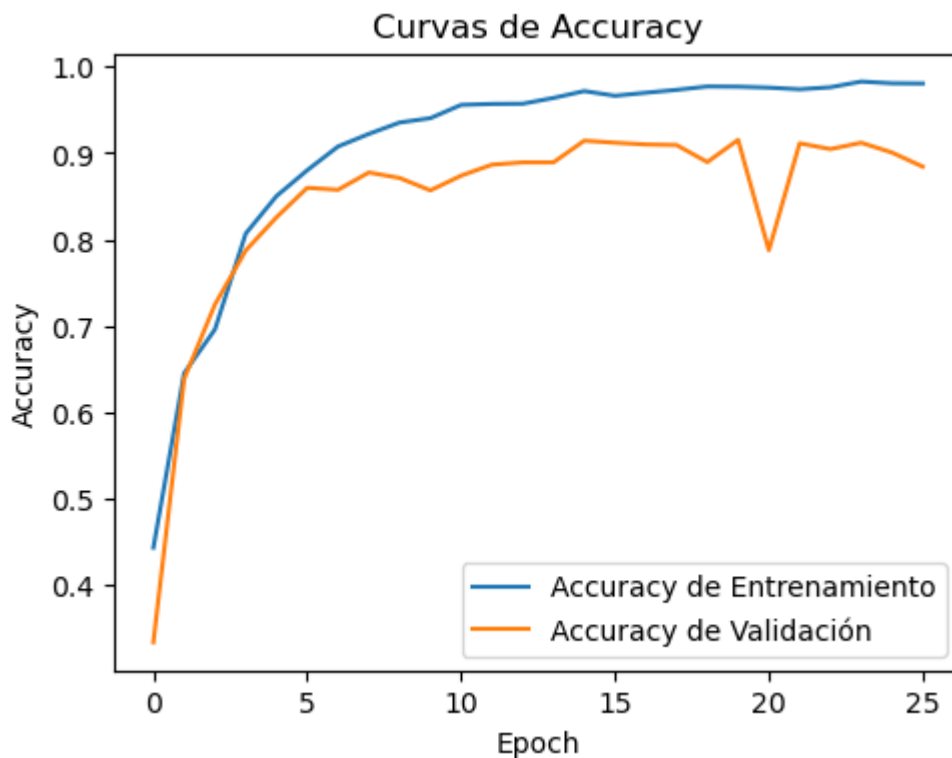
```
In [44]: loss_batch, accuracy_batch = model_batch.evaluate(X_test_std, y_test_oh,
print(f"Accuracy en test: {accuracy_batch*100:.2f}%")
```

52/52 ————— 4s 81ms/step – accuracy: 0.8469 – loss: 0.6237  
 Accuracy en test: 84.69%

```
In [45]: plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history_batch.history['accuracy'], label='Accuracy de Entrenamie
plt.plot(history_batch.history['val_accuracy'], label='Accuracy de Valida
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
```

```
plt.legend()
plt.title('Curvas de Accuracy')
```

Out[45]: Text(0.5, 1.0, 'Curvas de Accuracy')



## 11. Data Augmentation

Técnica para aumentar artificialmente la cantidad y diversidad de datos de entrenamiento. Esta diversidad de datos se obtiene aplicando transformaciones aleatorias a las imágenes originales (rotaciones, desplazamientos, zoom, flip horizontal, etc.) para generar nuevas variaciones sin recopilar más datos reales.

### Ventajas

- Más datos: El modelo ve variaciones del mismo objeto en diferentes contextos
- Mejor generalización: Aprende patrones más robustos, no memoriza específicamente
- Menos overfitting: Reduce la brecha entre entrenamiento y validación
- Eficiencia: Sin costo de recolección adicional

Con la clase ImageDataGenerator aplicas transformaciones a las imágenes, cómo los ejemplos:

- Rotaciones  $\pm 15^\circ$
- Desplazamientos horizontales/verticales  $\pm 10\%$
- Zoom  $\pm 15\%$
- Flip horizontal

El resultado es que el modelo entrena con ~3-4x más ejemplos variados del mismo dataset, mejorando su capacidad de reconocer personajes en diferentes poses y

posiciones.

## 11.1. Data Augmentation - Image Generator y Entrenamiento

Creación del Image Generator configurado para crear las variaciones en las imágenes y utilización del modelo CNN Basico de la primera arquitectura experimentada para empezar el entrenamiento.

```
In [46]: # Creación del generador de datos con aumentos
datagen_aug = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    zoom_range=0.2,
    shear_range=0.15,
    fill_mode='nearest'
)

# Utiliza el modelo CNN Basico de la primera arquitectura experimentada y
history_aug = model.fit(
    datagen_aug.flow(X_train, y_train, batch_size=BATCH_SIZE),
    epochs=EPOCHS,
    validation_data=(X_val, y_val),
    callbacks=[checkpoint, early_stopping],
    steps_per_epoch=len(X_train)//BATCH_SIZE
)
```

Epoch 1/50

373/373 ————— 0s 292ms/step – accuracy: 0.6737 – loss: 1.1382














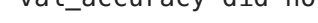

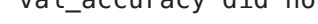

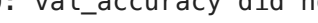
Epoch 1: val\_accuracy did not improve from 0.91533

373/373 ————— 119s 316ms/step – accuracy: 0.6981 – loss: 1.0045 – val\_accuracy: 0.8718 – val\_loss: 0.4626

Epoch 2/50

1/373 ————— 2:02 328ms/step – accuracy: 0.7500 – loss: 0.7531

/Users/edprata/anaconda3/envs/Python-3-13-2/lib/python3.13/site-packages/keras/src/trainers/epoch\_iterator.py:116: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps\_per\_epoch \* epochs` batches. You may need to use the `.repeat()` function when building your dataset.  
self.\_interrupted\_warning()

Epoch 2: val\_accuracy did not improve from 0.91533  
**373/373**  **7s** 19ms/step - accuracy: 0.7500 - loss: 0.7531 - val\_accuracy: 0.8715 - val\_loss: 0.4689  
Epoch 3/50  
**373/373**  **7s** 19ms/step - accuracy: 0.7500 - loss: 0.7531 - val\_accuracy: 0.8715 - val\_loss: 0.4689  
Epoch 3/50  
**373/373**  **0s** 246ms/step - accuracy: 0.7441 - loss: 0.8643  
Epoch 3: val\_accuracy did not improve from 0.91533  
**373/373**  **98s** 263ms/step - accuracy: 0.7464 - loss: 0.8533 - val\_accuracy: 0.8825 - val\_loss: 0.4259  
Epoch 4/50  
**1/373**  **1:48** 291ms/step - accuracy: 0.8125 - loss: 0.6779  
Epoch 4: val\_accuracy did not improve from 0.91533  
**373/373**  **6s** 17ms/step - accuracy: 0.8125 - loss: 0.6779 - val\_accuracy: 0.8869 - val\_loss: 0.4202  
Epoch 5/50  
**373/373**  **0s** 242ms/step - accuracy: 0.7663 - loss: 0.7578  
Epoch 5: val\_accuracy did not improve from 0.91533  
**373/373**  **96s** 258ms/step - accuracy: 0.7684 - loss: 0.7576 - val\_accuracy: 0.8909 - val\_loss: 0.3922  
Epoch 6/50  
**1/373**  **1:58** 318ms/step - accuracy: 0.8125 - loss: 0.5820  
Epoch 6: val\_accuracy did not improve from 0.91533  
**373/373**  **7s** 17ms/step - accuracy: 0.8125 - loss: 0.5820 - val\_accuracy: 0.8892 - val\_loss: 0.3930  
Epoch 7/50  
**373/373**  **0s** 243ms/step - accuracy: 0.7814 - loss: 0.7059  
Epoch 7: val\_accuracy did not improve from 0.91533  
**373/373**  **97s** 259ms/step - accuracy: 0.7827 - loss: 0.7200 - val\_accuracy: 0.8815 - val\_loss: 0.4345  
Epoch 8/50  
**1/373**  **1:35** 257ms/step - accuracy: 0.8125 - loss: 0.5858  
Epoch 8: val\_accuracy did not improve from 0.91533  
**373/373**  **6s** 16ms/step - accuracy: 0.8125 - loss: 0.5858 - val\_accuracy: 0.8842 - val\_loss: 0.4316  
Epoch 9/50  
**373/373**  **0s** 243ms/step - accuracy: 0.8029 - loss: 0.6611  
Epoch 9: val\_accuracy did not improve from 0.91533  
**373/373**  **97s** 259ms/step - accuracy: 0.8013 - loss: 0.6634 - val\_accuracy: 0.8902 - val\_loss: 0.3778  
Epoch 10/50  
**1/373**  **1:59** 320ms/step - accuracy: 0.7812 - loss: 0.7670  
Epoch 10: val\_accuracy did not improve from 0.91533  
**373/373**  **6s** 16ms/step - accuracy: 0.7812 - loss: 0.7670 - val\_accuracy: 0.8906 - val\_loss: 0.3755  
Epoch 10: early stopping  
Restoring model weights from the end of the best epoch: 1.



## 11.2. Data Augmentation - Evaluación

Análisis de resultados del modelo CNN entrenado con Data Augmentation.

```
In [70]: loss_aug, accuracy_aug = model.evaluate(X_test_std, y_test_ohe, verbose=1)
print(f"Accuracy en test (CNN con hiperparámetros ajustados): {accuracy_a
```

52/52 ————— 4s 81ms/step – accuracy: 0.8252 – loss: 0.6449  
Accuracy en test (CNN con hiperparámetros ajustados): 82.52%



## 13. CNN con Optimizers

Experimentación de Optimizers para intentar obtener resultados mejores. Un Optimizer es un algoritmo que actualiza los pesos de la red neuronal durante el entrenamiento para minimizar la función de pérdida (error). Los Optimizers utilizados:

- Adam: Adaptativo, combina momentum y RMSprop, convergencia rápida, eficiente, ideal para deep learning.
- RMSprop: Adaptativo, ajusta learning rate por parámetro, es estable, bueno para redes recurrentes.
- Nadam: Adam + Nesterov momentum, mejor convergencia que Adam en algunos casos.
- SGD: Descenso de gradiente estocástico clásico, simple, pero necesita ajuste fino del learning rate.

Optimizers adaptativos (Adam, RMSprop, Nadam) ajustan automáticamente el learning rate para cada parámetro. Son más robustos y generalizan mejor. Los clásicos (como SGD) usan un learning rate fijo y requieren más configuraciones, pero puede ser muy preciso si se ajusta bien.



### 13.1. CNN con Optimizers - Configuración y Entrenamiento

Configuración de diferentes optimizadores para comparar los resultados.

```
In [48]: # Configuración de diferentes optimizadores para comparar
optimizers = {
    "Adam": Adam(learning_rate=0.001),
    "RMSprop": RMSprop(learning_rate=0.001),
    "Nadam": Nadam(learning_rate=0.001),
    "SGD": SGD(learning_rate=0.002, momentum=0.9)
}

# Diccionario para guardar resultados
results = {}

# Early stopping para evitar overfitting
early_stopping = EarlyStopping(monitor='val_loss', patience=15, restore_b

# Bucle para entrenar y evaluar el modelo con cada optimizador
for name, opt in optimizers.items():
    print(f"\n=== Entrenando con {name} ===")

    # Copiamos la arquitectura profunda
    model_compare = Sequential([
        Conv2D(32, (3,3), activation='relu', padding='same', input_shape=
```



```

    Conv2D(32, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2)),
    Dropout(0.25),

    Conv2D(64, (3,3), activation='relu', padding='same'),
    Conv2D(64, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2)),
    Dropout(0.25),

    Conv2D(128, (3,3), activation='relu', padding='same'),
    Conv2D(128, (3,3), activation='relu', padding='same'),
    MaxPooling2D((2,2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

# Compilamos el modelo con el optimizador actual
model_compare.compile(optimizer=opt, loss='categorical_crossentropy',

# Entrenamos el modelo
history = model_compare.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=[early_stopping],
    verbose=1
)


# Evaluamos el modelo en el conjunto de test
loss, acc = model_compare.evaluate(X_test_std, y_test_ohe, verbose=1)
results[name] = {"history": history, "test_accuracy": acc}
print(f"Test accuracy con {name}: {acc*100:.2f}%")

```




=== Entrenando con Adam ===


Epoch 1/50

**374/374**  **99s** 259ms/step - accuracy: 0.2031 - loss: 2.3408 - val\_accuracy: 0.3929 - val\_loss: 1.9302


Epoch 2/50

**374/374**  **95s** 254ms/step - accuracy: 0.4137 - loss: 1.7984 - val\_accuracy: 0.5847 - val\_loss: 1.3457


Epoch 3/50

**374/374**  **95s** 254ms/step - accuracy: 0.5389 - loss: 1.4165 - val\_accuracy: 0.6178 - val\_loss: 1.1965


Epoch 4/50

**374/374**  **95s** 253ms/step - accuracy: 0.6193 - loss: 1.1572 - val\_accuracy: 0.7299 - val\_loss: 0.8654


Epoch 5/50

**374/374**  **96s** 256ms/step - accuracy: 0.6981 - loss: 0.9285 - val\_accuracy: 0.7597 - val\_loss: 0.7420


Epoch 6/50

**374/374**  **98s** 261ms/step - accuracy: 0.7451 - loss: 0.7794 - val\_accuracy: 0.8119 - val\_loss: 0.6214


Epoch 7/50

**374/374**  **95s** 255ms/step - accuracy: 0.7827 - loss: 0.6589 - val\_accuracy: 0.8320 - val\_loss: 0.5721


Epoch 8/50

**374/374**  **97s** 260ms/step - accuracy: 0.8226 - loss: 0.5434 - val\_accuracy: 0.8447 - val\_loss: 0.5545


Epoch 9/50

**374/374**  **96s** 257ms/step - accuracy: 0.8411 - loss: 0.4947 - val\_accuracy: 0.8464 - val\_loss: 0.5058


Epoch 10/50

**374/374**  **95s** 253ms/step - accuracy: 0.8611 - loss: 0.4178 - val\_accuracy: 0.8511 - val\_loss: 0.5287


Epoch 11/50

**374/374**  **95s** 254ms/step - accuracy: 0.8736 - loss: 0.3753 - val\_accuracy: 0.8638 - val\_loss: 0.4894


Epoch 12/50

**374/374**  **95s** 254ms/step - accuracy: 0.8822 - loss: 0.3526 - val\_accuracy: 0.8661 - val\_loss: 0.4668


Epoch 13/50

**374/374**  **95s** 253ms/step - accuracy: 0.8975 - loss: 0.3099 - val\_accuracy: 0.8665 - val\_loss: 0.4979


Epoch 14/50

**374/374**  **96s** 256ms/step - accuracy: 0.9064 - loss: 0.2872 - val\_accuracy: 0.8849 - val\_loss: 0.5245


Epoch 15/50

**374/374**  **96s** 257ms/step - accuracy: 0.9104 - loss: 0.2685 - val\_accuracy: 0.8762 - val\_loss: 0.5333


Epoch 16/50

**374/374**  **95s** 254ms/step - accuracy: 0.9195 - loss: 0.2412 - val\_accuracy: 0.8832 - val\_loss: 0.4857


Epoch 17/50

**374/374**  **95s** 253ms/step - accuracy: 0.9246 - loss: 0.2274 - val\_accuracy: 0.8772 - val\_loss: 0.4963

Epoch 18/50









**374/374**  **94s** 252ms/step - accuracy: 0.9263 - loss: 0.2270 - val\_accuracy: 0.8752 - val\_loss: 0.5199

Epoch 19/50












**374/374**  **94s** 253ms/step - accuracy: 0.9282 - loss: 0.2186 - val\_accuracy: 0.8792 - val\_loss: 0.5354















Epoch 20/50

**374/374**  **95s** 254ms/step - accuracy: 0.9391 - loss: 0.1






821 - val\_accuracy: 0.8926 - val\_loss: 0.5090  
 Epoch 21/50  
**374/374**  **95s** 253ms/step - accuracy: 0.9353 - loss: 0.2  
 024 - val\_accuracy: 0.8872 - val\_loss: 0.4927  
 Epoch 22/50  
**374/374**  **95s** 254ms/step - accuracy: 0.9403 - loss: 0.1  
 904 - val\_accuracy: 0.8886 - val\_loss: 0.4895  
 Epoch 23/50  
**374/374**  **95s** 253ms/step - accuracy: 0.9438 - loss: 0.1  
 699 - val\_accuracy: 0.8845 - val\_loss: 0.5582  
 Epoch 24/50  
**374/374**  **96s** 256ms/step - accuracy: 0.9467 - loss: 0.1  
 707 - val\_accuracy: 0.8966 - val\_loss: 0.5064  
 Epoch 25/50  
**374/374**  **95s** 253ms/step - accuracy: 0.9484 - loss: 0.1  
 595 - val\_accuracy: 0.8896 - val\_loss: 0.4969  
 Epoch 26/50  
**374/374**  **95s** 254ms/step - accuracy: 0.9510 - loss: 0.1  
 525 - val\_accuracy: 0.8902 - val\_loss: 0.5961  
 Epoch 27/50  
**374/374**  **97s** 259ms/step - accuracy: 0.9480 - loss: 0.1  
 617 - val\_accuracy: 0.8869 - val\_loss: 0.6075  
 Epoch 27: early stopping  
 Restoring model weights from the end of the best epoch: 12.  
**52/52**  **3s** 64ms/step - accuracy: 0.7973 - loss: 0.7256  
 Test accuracy con Adam: 79.73%




















=== Entrenando con RMSprop ===


Epoch 1/50  
**374/374**  **101s** 266ms/step - accuracy: 0.2261 - loss: 2.  
 2893 - val\_accuracy: 0.4475 - val\_loss: 1.7927  
 Epoch 2/50  
**374/374**  **95s** 255ms/step - accuracy: 0.4824 - loss: 1.6  
 198 - val\_accuracy: 0.5987 - val\_loss: 1.2711  
 Epoch 3/50  
**374/374**  **95s** 254ms/step - accuracy: 0.6359 - loss: 1.1  
 694 - val\_accuracy: 0.7400 - val\_loss: 0.8446  
 Epoch 4/50  
**374/374**  **98s** 262ms/step - accuracy: 0.7268 - loss: 0.8  
 858 - val\_accuracy: 0.7323 - val\_loss: 1.0580  
 Epoch 5/50  
**374/374**  **95s** 253ms/step - accuracy: 0.7833 - loss: 0.7  
 101 - val\_accuracy: 0.8072 - val\_loss: 0.7105  
 Epoch 6/50  
**374/374**  **96s** 257ms/step - accuracy: 0.8186 - loss: 0.6  
 040 - val\_accuracy: 0.8384 - val\_loss: 0.5448  
 Epoch 7/50  
**374/374**  **96s** 256ms/step - accuracy: 0.8482 - loss: 0.5  
 000 - val\_accuracy: 0.8444 - val\_loss: 0.5284  
 Epoch 8/50  
**374/374**  **95s** 255ms/step - accuracy: 0.8682 - loss: 0.4  
 491 - val\_accuracy: 0.8614 - val\_loss: 0.4639  
 Epoch 9/50  
**374/374**  **96s** 256ms/step - accuracy: 0.8760 - loss: 0.4  
 253 - val\_accuracy: 0.8745 - val\_loss: 0.4368  
 Epoch 10/50  
**374/374**  **95s** 254ms/step - accuracy: 0.8868 - loss: 0.3  
 993 - val\_accuracy: 0.8253 - val\_loss: 1.0652  
 Epoch 11/50  
**374/374**  **96s** 257ms/step - accuracy: 0.8932 - loss: 0.3

757 - val\_accuracy: 0.8645 - val\_loss: 0.6293  
 Epoch 12/50  
**374/374**  **96s** 257ms/step - accuracy: 0.8930 - loss: 0.3  
 671 - val\_accuracy: 0.8551 - val\_loss: 0.7419  
 Epoch 13/50  
**374/374**  **95s** 255ms/step - accuracy: 0.8971 - loss: 0.3  
 696 - val\_accuracy: 0.8922 - val\_loss: 0.5058  
 Epoch 14/50  
**374/374**  **95s** 255ms/step - accuracy: 0.8983 - loss: 0.3  
 568 - val\_accuracy: 0.8855 - val\_loss: 0.7072  
 Epoch 15/50  
**374/374**  **95s** 254ms/step - accuracy: 0.8980 - loss: 0.3  
 755 - val\_accuracy: 0.8835 - val\_loss: 0.7475  
 Epoch 16/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9062 - loss: 0.3  
 708 - val\_accuracy: 0.8681 - val\_loss: 1.0282  
 Epoch 17/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9038 - loss: 0.3  
 887 - val\_accuracy: 0.8799 - val\_loss: 0.8042  
 Epoch 18/50  
**374/374**  **96s** 258ms/step - accuracy: 0.8996 - loss: 0.3  
 954 - val\_accuracy: 0.8979 - val\_loss: 0.5128  
 Epoch 19/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9053 - loss: 0.3  
 690 - val\_accuracy: 0.8598 - val\_loss: 0.5327  
 Epoch 20/50  
**374/374**  **98s** 260ms/step - accuracy: 0.9046 - loss: 0.3  
 749 - val\_accuracy: 0.8842 - val\_loss: 0.5626  
 Epoch 21/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9024 - loss: 0.3  
 825 - val\_accuracy: 0.8531 - val\_loss: 0.8421  
 Epoch 22/50  
**374/374**  **96s** 256ms/step - accuracy: 0.9072 - loss: 0.3  
 801 - val\_accuracy: 0.9023 - val\_loss: 0.5745  
 Epoch 23/50  
**374/374**  **95s** 255ms/step - accuracy: 0.9025 - loss: 0.4  
 000 - val\_accuracy: 0.8608 - val\_loss: 0.6022  
 Epoch 24/50  
**374/374**  **95s** 255ms/step - accuracy: 0.9024 - loss: 0.4  
 126 - val\_accuracy: 0.8450 - val\_loss: 0.6663  
 Epoch 24: early stopping  
 Restoring model weights from the end of the best epoch: 9.  
**52/52**  **3s** 65ms/step - accuracy: 0.8179 - loss: 0.6790  
 Test accuracy con RMSprop: 81.79%


=== Entrenando con Nadam ===


Epoch 1/50  
**374/374**  **100s** 263ms/step - accuracy: 0.2714 - loss: 2.  
 1736 - val\_accuracy: 0.5375 - val\_loss: 1.5432  
 Epoch 2/50  
**374/374**  **96s** 257ms/step - accuracy: 0.5244 - loss: 1.4  
 708 - val\_accuracy: 0.6958 - val\_loss: 1.0577  
 Epoch 3/50  
**374/374**  **97s** 259ms/step - accuracy: 0.6584 - loss: 1.0  
 833 - val\_accuracy: 0.7456 - val\_loss: 0.8477  
 Epoch 4/50  
**374/374**  **98s** 263ms/step - accuracy: 0.7364 - loss: 0.8  
 129 - val\_accuracy: 0.7811 - val\_loss: 0.7253  
 Epoch 5/50  
**374/374**  **98s** 261ms/step - accuracy: 0.7954 - loss: 0.6


403 - val\_accuracy: 0.8283 - val\_loss: 0.5831  
Epoch 6/50  
**374/374**  **96s** 258ms/step - accuracy: 0.8335 - loss: 0.5  
237 - val\_accuracy: 0.8497 - val\_loss: 0.5249  
Epoch 7/50  
**374/374**  **96s** 258ms/step - accuracy: 0.8635 - loss: 0.4  
256 - val\_accuracy: 0.8668 - val\_loss: 0.4949  
Epoch 8/50  
**374/374**  **96s** 257ms/step - accuracy: 0.8778 - loss: 0.3  
747 - val\_accuracy: 0.8825 - val\_loss: 0.4563  
Epoch 9/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9003 - loss: 0.3  
148 - val\_accuracy: 0.8832 - val\_loss: 0.4133  
Epoch 10/50  
**374/374**  **96s** 256ms/step - accuracy: 0.9101 - loss: 0.2  
723 - val\_accuracy: 0.8825 - val\_loss: 0.4691  
Epoch 11/50  
**374/374**  **96s** 256ms/step - accuracy: 0.9230 - loss: 0.2  
492 - val\_accuracy: 0.8979 - val\_loss: 0.4381  
Epoch 12/50  
**374/374**  **97s** 259ms/step - accuracy: 0.9293 - loss: 0.2  
276 - val\_accuracy: 0.8929 - val\_loss: 0.4193  
Epoch 13/50  
**374/374**  **97s** 261ms/step - accuracy: 0.9345 - loss: 0.2  
040 - val\_accuracy: 0.9023 - val\_loss: 0.4473  
Epoch 14/50  
**374/374**  **96s** 258ms/step - accuracy: 0.9402 - loss: 0.1  
871 - val\_accuracy: 0.8976 - val\_loss: 0.4236  
Epoch 15/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9442 - loss: 0.1  
782 - val\_accuracy: 0.8952 - val\_loss: 0.5046  
Epoch 16/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9464 - loss: 0.1  
671 - val\_accuracy: 0.9023 - val\_loss: 0.4321  
Epoch 17/50  
**374/374**  **97s** 258ms/step - accuracy: 0.9514 - loss: 0.1  
581 - val\_accuracy: 0.8956 - val\_loss: 0.4711  
Epoch 18/50  
**374/374**  **96s** 258ms/step - accuracy: 0.9510 - loss: 0.1  
544 - val\_accuracy: 0.9009 - val\_loss: 0.4794  
Epoch 19/50  
**374/374**  **97s** 259ms/step - accuracy: 0.9556 - loss: 0.1  
440 - val\_accuracy: 0.9023 - val\_loss: 0.4692  
Epoch 20/50  
**374/374**  **99s** 264ms/step - accuracy: 0.9575 - loss: 0.1  
378 - val\_accuracy: 0.8946 - val\_loss: 0.5046  
Epoch 21/50  
**374/374**  **97s** 258ms/step - accuracy: 0.9534 - loss: 0.1  
465 - val\_accuracy: 0.8993 - val\_loss: 0.4899  
Epoch 22/50  
**374/374**  **96s** 257ms/step - accuracy: 0.9580 - loss: 0.1  
286 - val\_accuracy: 0.9076 - val\_loss: 0.4428  
Epoch 23/50  
**374/374**  **97s** 259ms/step - accuracy: 0.9606 - loss: 0.1  
248 - val\_accuracy: 0.9076 - val\_loss: 0.5006  
Epoch 24/50  
**374/374**  **97s** 260ms/step - accuracy: 0.9618 - loss: 0.1  
186 - val\_accuracy: 0.9013 - val\_loss: 0.5020  
Epoch 24: early stopping  
Restoring model weights from the end of the best epoch: 9.


**52/52**  **3s** 64ms/step – accuracy: 0.8294 – loss: 0.6665  
 Test accuracy con Nadam: 82.94%


=== Entrenando con SGD ===


Epoch 1/50  
**374/374**  **102s** 264ms/step – accuracy: 0.1373 – loss: 2.5173 – val\_accuracy: 0.2172 – val\_loss: 2.4157


Epoch 2/50  
**374/374**  **95s** 254ms/step – accuracy: 0.2603 – loss: 2.2296 – val\_accuracy: 0.3869 – val\_loss: 1.9927


Epoch 3/50  
**374/374**  **100s** 267ms/step – accuracy: 0.3582 – loss: 1.9668 – val\_accuracy: 0.4618 – val\_loss: 1.7532


Epoch 4/50  
**374/374**  **97s** 259ms/step – accuracy: 0.4297 – loss: 1.7609 – val\_accuracy: 0.5442 – val\_loss: 1.4754


Epoch 5/50  
**374/374**  **95s** 254ms/step – accuracy: 0.4910 – loss: 1.5988 – val\_accuracy: 0.5693 – val\_loss: 1.4222


Epoch 6/50  
**374/374**  **94s** 251ms/step – accuracy: 0.5294 – loss: 1.4632 – val\_accuracy: 0.6188 – val\_loss: 1.2456


Epoch 7/50  
**374/374**  **95s** 255ms/step – accuracy: 0.5789 – loss: 1.3246 – val\_accuracy: 0.6118 – val\_loss: 1.2323


Epoch 8/50  
**374/374**  **95s** 253ms/step – accuracy: 0.6134 – loss: 1.2110 – val\_accuracy: 0.6814 – val\_loss: 1.0186


Epoch 9/50  
**374/374**  **95s** 255ms/step – accuracy: 0.6544 – loss: 1.0884 – val\_accuracy: 0.7075 – val\_loss: 0.9326


Epoch 10/50  
**374/374**  **97s** 259ms/step – accuracy: 0.6832 – loss: 0.9906 – val\_accuracy: 0.7279 – val\_loss: 0.8886


Epoch 11/50  
**374/374**  **96s** 256ms/step – accuracy: 0.7231 – loss: 0.8648 – val\_accuracy: 0.7604 – val\_loss: 0.7811

Epoch 12/50  
**374/374**  **94s** 252ms/step – accuracy: 0.7373 – loss: 0.7986 – val\_accuracy: 0.7614 – val\_loss: 0.7813

Epoch 13/50  
**374/374**  **95s** 254ms/step – accuracy: 0.7656 – loss: 0.7107 – val\_accuracy: 0.8039 – val\_loss: 0.6699

Epoch 14/50  
**374/374**  **95s** 254ms/step – accuracy: 0.7895 – loss: 0.6477 – val\_accuracy: 0.8129 – val\_loss: 0.6227

Epoch 15/50  
**374/374**  **95s** 254ms/step – accuracy: 0.8093 – loss: 0.5752 – val\_accuracy: 0.7972 – val\_loss: 0.6635

Epoch 15: early stopping  
 Restoring model weights from the end of the best epoch: 1.  
**52/52**  **3s** 65ms/step – accuracy: 0.1960 – loss: 2.4379  
 Test accuracy con SGD: 19.60%



## 13.2. CNN con Optimizers - Evaluación

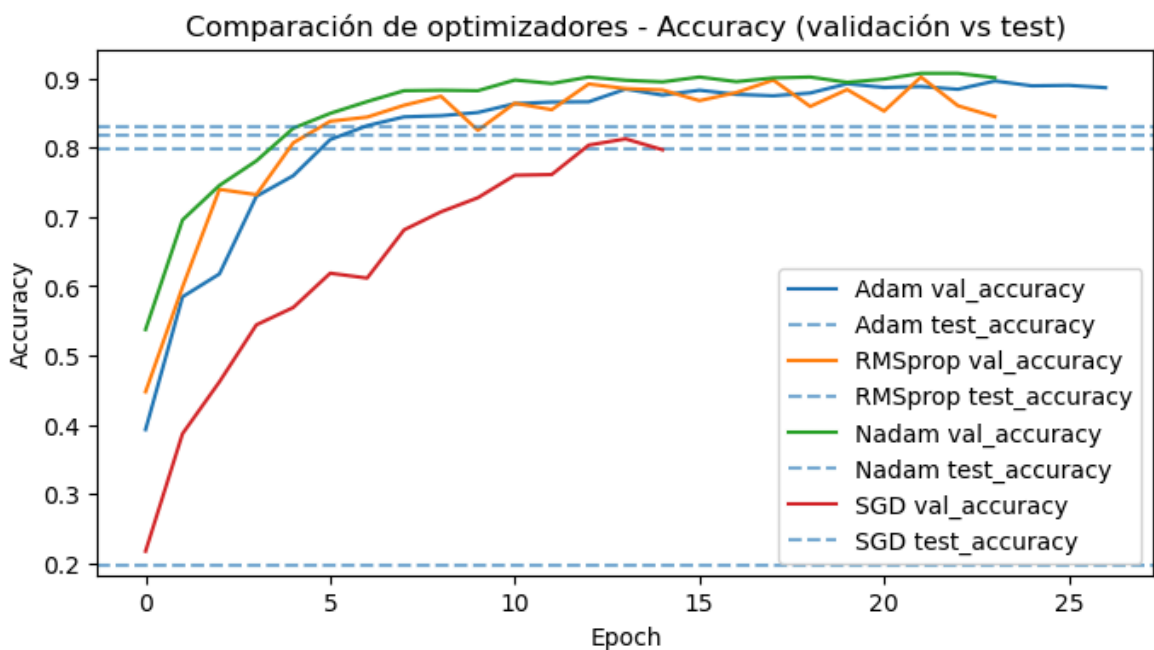
Análisis de resultados de la arquitectura CNN con diferentes Optimizers.

```
In [49]: plt.figure(figsize=(8,4))

for name, res in results.items():
    # Curva de validación
    plt.plot(
        res["history"].history['val_accuracy'],
        label=f"{name} val_accuracy"
    )

    # Línea horizontal de test accuracy
    plt.axhline(
        y=res["test_accuracy"],
        linestyle='--',
        alpha=0.6,
        label=f"{name} test_accuracy"
    )

plt.title("Comparación de optimizadores - Accuracy (validación vs test)")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```



### 🎯 13.3. CNN con Optimizers - Conclusión

Los resultados indican que los **optimizadores adaptativos** generalizan mucho mejor en este problema de clasificación de imágenes. Adam y Nadam muestran una convergencia más estable y una mejor capacidad para ajustar los pesos en redes profundas, mientras que RMSprop presenta un aprendizaje limitado. Por el contrario, SGD tiene grandes dificultades para converger, lo que sugiere que, sin una configuración muy fina o muchas más épocas, no es adecuado para este tipo de arquitectura y dataset. En este contexto, los optimizadores adaptativos resultan claramente más efectivos que los métodos de descenso de gradiente clásicos.



## 14. CNN Tunned (+BatchNormalization +DA +Nadam)

Con el objetivo de lograr una precisión superior al 85%, se utilizarán junto con las técnicas más exitosas hasta haora con CNN: BatchNormalization, Data Augmentation y Optimizer Nadam.



### 14.1. CNN Tunned - Definición de la arquitectura

Se utilizará la primera arquitectura intentada, porque logró la mayor accuracy, añadiendo capas de BatchNormalization porque aumentó el accuracy en otro experimento.

```
In [71]: model_tunned = Sequential([
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(IM
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
])

print("Modelo con Batch Normalization definido.")
```

Modelo con Batch Normalization definido.



### 14.2. CNN Tunned - Data Augmentation

Creación del Image Generator configurado para crear las variaciones en las imágenes y utilización del modelo CNN Basico de la primera arquitectura experimentada para empezar el entrenamiento.

```
In [72]: datagen_aug2 = ImageDataGenerator(
    rotation_range=20,
```

```
width_shift_range=0.2,  
height_shift_range=0.2,  
horizontal_flip=True,  
zoom_range=0.2,  
shear_range=0.15,  
fill_mode='nearest'  
)
```



### 14.3. CNN Tunned - Compilación con Optimizer Nadam

Compilación utilizando Optimizer Nadam que presentó mejor resultado en experimentos anteriores.

```
In [73]: # Compilar el modelo  
model_tunned.compile(optimizer=Nadam(learning_rate=0.001), loss='categori  
  
# Mostrar un resumen del modelo  
model_tunned.summary()
```

**Model: "sequential\_11"**



Layer (type)	Output Shape	
conv2d_60 (Conv2D)	(None, 90, 90, 32)	
batch_normalization_9 (BatchNormalization)	(None, 90, 90, 32)	
conv2d_61 (Conv2D)	(None, 90, 90, 32)	
batch_normalization_10 (BatchNormalization)	(None, 90, 90, 32)	
max_pooling2d_27 (MaxPooling2D)	(None, 45, 45, 32)	
dropout_40 (Dropout)	(None, 45, 45, 32)	
conv2d_62 (Conv2D)	(None, 45, 45, 64)	
batch_normalization_11 (BatchNormalization)	(None, 45, 45, 64)	
conv2d_63 (Conv2D)	(None, 45, 45, 64)	
batch_normalization_12 (BatchNormalization)	(None, 45, 45, 64)	
max_pooling2d_28 (MaxPooling2D)	(None, 22, 22, 64)	
dropout_41 (Dropout)	(None, 22, 22, 64)	
conv2d_64 (Conv2D)	(None, 22, 22, 128)	
batch_normalization_13 (BatchNormalization)	(None, 22, 22, 128)	
conv2d_65 (Conv2D)	(None, 22, 22, 128)	
batch_normalization_14 (BatchNormalization)	(None, 22, 22, 128)	
max_pooling2d_29 (MaxPooling2D)	(None, 11, 11, 128)	
dropout_42 (Dropout)	(None, 11, 11, 128)	
flatten_10 (Flatten)	(None, 15488)	
dense_25 (Dense)	(None, 128)	1,9
batch_normalization_15 (BatchNormalization)	(None, 128)	
dropout_43 (Dropout)	(None, 128)	
dense_26 (Dense)	(None, 13)	

**Total params:** 2,273,581 (8.67 MB)

**Trainable params:** 2,272,429 (8.67 MB)

**Non-trainable params:** 1,152 (4.50 KB)




## 14.4. CNN Tuned - Compilación y Entrenamiento

Compilación y entrenamiento del modelo en la arquitectura CNN utilizando BatchNormalization, Data Augmentation y Optimizer Nadam.


```
In [75]: history_aug = model_tuned.fit(
    datagen_aug2.flow(X_train, y_train, batch_size=BATCH_SIZE),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    validation_data=(X_val, y_val),
    callbacks=[checkpoint, early_stopping],
    steps_per_epoch=len(X_train)//BATCH_SIZE
)

print("Entrenamiento completado.")
```


Epoch 1/50

**373/373**  **0s** 506ms/step - accuracy: 0.3051 - loss: 2.3368

Epoch 1: val\_accuracy did not improve from 0.91533














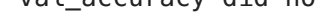

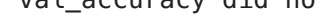

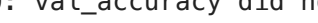

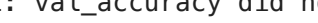
**373/373**  **200s** 531ms/step - accuracy: 0.3453 - loss: 2.1822 - val\_accuracy: 0.2982 - val\_loss: 2.1525

Epoch 2/50

**1/373**  **2:52** 465ms/step - accuracy: 0.4375 - loss: 1.8625

/Users/edprata/anaconda3/envs/Python-3-13-2/lib/python3.13/site-packages/keras/src/trainers/epoch\_iterator.py:116: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps\_per\_epoch \* epochs` batches. You may need to use the `.repeat()` function when building your dataset.

self.\_interrupted\_warning()

Epoch 2: val\_accuracy did not improve from 0.91533  
373/373  8s 21ms/step - accuracy: 0.4375 - loss: 1.8625 - val\_accuracy: 0.2992 - val\_loss: 2.1409  
Epoch 3/50  
373/373  8s 21ms/step - accuracy: 0.4375 - loss: 1.8625 - val\_accuracy: 0.2992 - val\_loss: 2.1409  
Epoch 3/50  
373/373  0s 403ms/step - accuracy: 0.4696 - loss: 1.7229  
Epoch 3: val\_accuracy did not improve from 0.91533  
373/373  158s 424ms/step - accuracy: 0.4982 - loss: 1.6191 - val\_accuracy: 0.5077 - val\_loss: 1.5421  
Epoch 4/50  
1/373  2:57 478ms/step - accuracy: 0.4375 - loss: 2.0483  
Epoch 4: val\_accuracy did not improve from 0.91533  
373/373  7s 19ms/step - accuracy: 0.4375 - loss: 2.0483 - val\_accuracy: 0.5810 - val\_loss: 1.3036  
Epoch 5/50  
373/373  0s 357ms/step - accuracy: 0.5728 - loss: 1.3524  
Epoch 5: val\_accuracy did not improve from 0.91533  
373/373  140s 375ms/step - accuracy: 0.5964 - loss: 1.2764 - val\_accuracy: 0.7239 - val\_loss: 0.8562  
Epoch 6/50  
1/373  2:58 481ms/step - accuracy: 0.6250 - loss: 1.1282  
Epoch 6: val\_accuracy did not improve from 0.91533  
373/373  8s 19ms/step - accuracy: 0.6250 - loss: 1.1282 - val\_accuracy: 0.7262 - val\_loss: 0.8646  
Epoch 7/50  
373/373  0s 359ms/step - accuracy: 0.6866 - loss: 1.0052  
Epoch 7: val\_accuracy did not improve from 0.91533  
373/373  141s 378ms/step - accuracy: 0.7012 - loss: 0.9509 - val\_accuracy: 0.7741 - val\_loss: 0.7228  
Epoch 8/50  
1/373  2:47 451ms/step - accuracy: 0.6875 - loss: 0.8289  
Epoch 8: val\_accuracy did not improve from 0.91533  
373/373  7s 19ms/step - accuracy: 0.6875 - loss: 0.8289 - val\_accuracy: 0.7631 - val\_loss: 0.7605  
Epoch 9/50  
373/373  0s 364ms/step - accuracy: 0.7509 - loss: 0.7831  
Epoch 9: val\_accuracy did not improve from 0.91533  
373/373  143s 384ms/step - accuracy: 0.7582 - loss: 0.7642 - val\_accuracy: 0.8069 - val\_loss: 0.6229  
Epoch 10/50  
1/373  3:09 508ms/step - accuracy: 0.7500 - loss: 0.8200  
Epoch 10: val\_accuracy did not improve from 0.91533  
373/373  7s 19ms/step - accuracy: 0.7500 - loss: 0.8200 - val\_accuracy: 0.8139 - val\_loss: 0.6041  
Epoch 11/50  
373/373  0s 383ms/step - accuracy: 0.7805 - loss: 0.7052  
Epoch 11: val\_accuracy did not improve from 0.91533  
373/373  150s 402ms/step - accuracy: 0.7878 - loss: 0.6843 - val\_accuracy: 0.8628 - val\_loss: 0.4478

Epoch 12/50

1/373 ————— 3:24 549ms/step – accuracy: 0.8125 – loss: 0.5116

Epoch 12: val\_accuracy did not improve from 0.91533

373/373 ————— 8s 20ms/step – accuracy: 0.8125 – loss: 0.5116 – val\_accuracy: 0.8661 – val\_loss: 0.4360

Epoch 13/50

373/373 ————— 0s 399ms/step – accuracy: 0.8127 – loss: 0.6158

Epoch 13: val\_accuracy did not improve from 0.91533

373/373 ————— 156s 418ms/step – accuracy: 0.8127 – loss: 0.6060 – val\_accuracy: 0.8484 – val\_loss: 0.5039

Epoch 14/50

1/373 ————— 3:14 522ms/step – accuracy: 0.9062 – loss: 0.4621

Epoch 14: val\_accuracy did not improve from 0.91533

373/373 ————— 8s 21ms/step – accuracy: 0.9062 – loss: 0.4621 – val\_accuracy: 0.8521 – val\_loss: 0.5016

Epoch 15/50

373/373 ————— 0s 404ms/step – accuracy: 0.8291 – loss: 0.5569

Epoch 15: val\_accuracy did not improve from 0.91533

373/373 ————— 158s 423ms/step – accuracy: 0.8296 – loss: 0.5549 – val\_accuracy: 0.8082 – val\_loss: 0.6067

Epoch 15: early stopping

Restoring model weights from the end of the best epoch: 1.

Entrenamiento completado.



## 14.5. CNN Tuned - Evaluación del modelo

Evaluación del modelo entrenado en el conjunto de test.

```
In [76]: # Carga del mejor modelo guardado durante el entrenamiento
loaded_model = load_model('best_model.keras')
print("Best model loaded successfully.")

# Evaluación del modelo cargado
loss, accuracy = loaded_model.evaluate(X_test_std, y_test_ohe, verbose=1)
print(f"Accuracy en el conjunto de test: {accuracy*100:.2f}%")

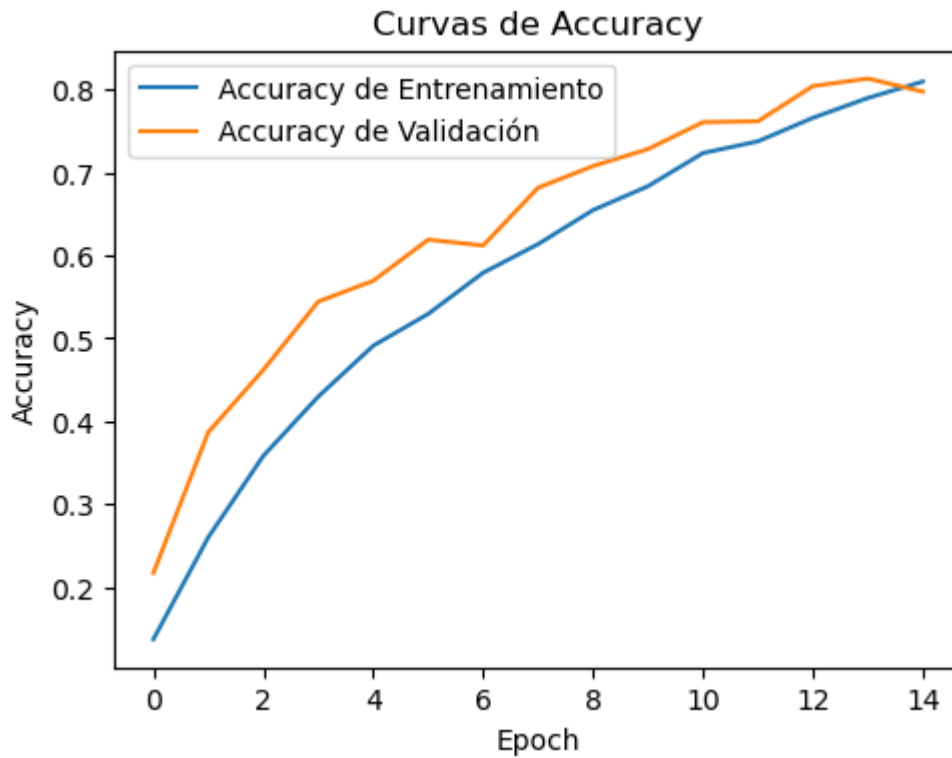
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Accuracy de Entrenamiento')
plt.plot(history.history['val_accuracy'], label='Accuracy de Validación')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Curvas de Accuracy')
```

Best model loaded successfully.

52/52 ————— 6s 85ms/step – accuracy: 0.8627 – loss: 0.6835

Accuracy en el conjunto de test: 86.27%

Out[76]: Text(0.5, 1.0, 'Curvas de Accuracy')



## 15. Transfer Learning

Transfer Learning es una técnica de aprendizaje automático donde utilizas un modelo pre-entrenado (ya entrenado en un dataset grande) como punto de partida para resolver un problema nuevo. En lugar de entrenar un modelo desde cero, reutilizas características ya aprendidas por otro modelo y las adaptas a tu problema específico. Algunas de las ventajas son:

- Requiere menos datos de entrenamiento
- Entrenamiento más rápido
- Mejor generalización
- Ideal para datasets pequeños

### 15.1. Transfer Learning - Rango de Datos

Para el entrenamiento del modelo con Transfer Learning es necesario que el rango de los datos sean ajustados como sigue.

```
In [50]: # Rango original:
print("Rango X_train:", X_train.min(), X_train.max())
print("Rango X_val:", X_val.min(), X_val.max())
print("Rango X_test:", X_test_std.min(), X_test_std.max())

# Copias para Transfer Learning:
X_train_tl = preprocess_input((X_train * 255).astype("float32"))
X_val_tl    = preprocess_input((X_val * 255).astype("float32"))
X_test_tl   = preprocess_input((X_test_std * 255).astype("float32"))

# Rango de las copias para Transfer Learning:
print("Rango X_train_tl:", X_train_tl.min(), X_train_tl.max())
```

```
print("Rango X_val_tl:", X_val_tl.min(), X_val_tl.max())
print("Rango X_test_tl:", X_test_tl.min(), X_test_tl.max())
```

```
Rango X_train: 0.0 1.0
Rango X_val: 0.0 1.0
Rango X_test: 0.0 1.0
Rango X_train_tl: -1.0 1.0
Rango X_val_tl: -1.0 1.0
Rango X_test_tl: -1.0 1.0
```

## 15.2. Transfer Learning - Configuración

Configuración del modelo pre-entrenado con todas las capas congeladas. La opción "base\_model.trainable = False" es una configuración para el congelamiento de todas las capas del modelo pre-entrenado, lo que significa que sus pesos no se actualizarán durante el entrenamiento.

Esto preserva las características útiles ya aprendidas en el modelo pre-entrenado (MobileNetV2), por ejemplo bordes, texturas y formas en ImageNet. Congelarlo evita que esos conocimientos se pierdan. Solo las capas nuevas agregadas al final (Dense, Dropout) se entrenan con los datos específicos (personajes de Los Simpsons).

Con esto se gana en eficiencia computacional, entrenamiento mucho más rápido, usa menos memoria, menos parámetros que actualizar y evita sobreajuste, con menos parámetros entrenables, el modelo es menos propenso a memorizar datos.

```
In [51]: # Modelo base preentrenado
base_model = MobileNetV2(
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    include_top=False,
    weights="imagenet"
)

# Fase 1: congelamos TODAS las capas
base_model.trainable = False

print("Modelo base cargado y capas congeladas para Transfer Learning.")
```

```
/var/folders/7g/ch5l_np56q51q3c4q01r2hs00000gn/T/ipykernel_29902/198752085
9.py:2: UserWarning: `input_shape` is undefined or non-square, or `rows` i
s not in [96, 128, 160, 192, 224]. Weights for input shape (224, 224) will
be loaded as the default.
```

```
base_model = MobileNetV2(
Modelo base cargado y capas congeladas para Transfer Learning.
```

## 15.3. Transfer Learning - Compilación y Resumen

Compilación del modelo pre-entrenado y resumen de la arquitectura.

```
In [52]: num_classes = len(MAP_CHARACTERS)

model_tl = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(256, activation="relu"),
```

```

        Dropout(0.5),
        Dense(num_classes, activation="softmax")
    ])

model_tl.compile(
    optimizer=Adam(learning_rate=1e-3),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

model_tl.summary()

```

**Model: "sequential\_10"**

Layer (type)	Output Shape	
mobilenetv2_1.00_224 (Functional)	(None, 3, 3, 1280)	2,589,261
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	
dense_21 (Dense)	(None, 256)	331,277
dropout_38 (Dropout)	(None, 256)	
dense_22 (Dense)	(None, 13)	169

**Total params:** 2,589,261 (9.88 MB)

**Trainable params:** 331,277 (1.26 MB)

**Non-trainable params:** 2,257,984 (8.61 MB)

## 15.4. Transfer Learning - Callbacks

Configuración de callbacks para el control del entrenamiento del modelo pre-entrenado.

```

In [53]: callbacks = [
    EarlyStopping(monitor="val_loss", patience=10, restore_best_weights=True),
    ModelCheckpoint("best_model_transfer_learning.h5", monitor="val_accuracy", save_best_only=True)
]

print("Callbacks configurados para Transfer Learning.")

```

Callbacks configurados para Transfer Learning.

## 15.5. Transfer Learning - Entrenamiento

Entrenamiento del modelo pre-entrenado.

```


In [54]: history_tl = model_tl.fit(
    X_train_tl, y_train,
    validation_data=(X_val_tl, y_val),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=callbacks
)

```


```
)

print("Entrenamiento con Transfer Learning completado.")
```


Epoch 1/50

373/374  0s 39ms/step - accuracy: 0.2247 - loss: 2.4750


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374  23s 53ms/step - accuracy: 0.3021 - loss: 2.1403 - val\_accuracy: 0.5100 - val\_loss: 1.5520


Epoch 2/50

374/374  0s 40ms/step - accuracy: 0.4496 - loss: 1.6861


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374  18s 49ms/step - accuracy: 0.4599 - loss: 1.6557 - val\_accuracy: 0.5776 - val\_loss: 1.3497


Epoch 3/50

374/374  0s 40ms/step - accuracy: 0.5177 - loss: 1.4721


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374  18s 49ms/step - accuracy: 0.5196 - loss: 1.4653 - val\_accuracy: 0.5950 - val\_loss: 1.3074


Epoch 4/50

374/374  0s 38ms/step - accuracy: 0.5594 - loss: 1.3520


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374  18s 47ms/step - accuracy: 0.5616 - loss: 1.3415 - val\_accuracy: 0.6091 - val\_loss: 1.2493


Epoch 5/50

374/374  0s 40ms/step - accuracy: 0.5908 - loss: 1.2313

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374  19s 52ms/step - accuracy: 0.5921 - loss: 1.2463 - val\_accuracy: 0.6252 - val\_loss: 1.2033

Epoch 6/50

373/374  0s 41ms/step - accuracy: 0.6227 - loss: 1.1468

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.



374/374 ————— 19s 51ms/step – accuracy: 0.6197 – loss: 1.1623 – val\_accuracy: 0.6275 – val\_loss: 1.2044

Epoch 7/50

374/374 ————— 0s 40ms/step – accuracy: 0.6324 – loss: 1.0999

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 ————— 18s 49ms/step – accuracy: 0.6306 – loss: 1.1101 – val\_accuracy: 0.6315 – val\_loss: 1.1705

Epoch 8/50

374/374 ————— 0s 40ms/step – accuracy: 0.6576 – loss: 1.0320

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 ————— 19s 49ms/step – accuracy: 0.6525 – loss: 1.0446 – val\_accuracy: 0.6339 – val\_loss: 1.1831

Epoch 9/50

374/374 ————— 0s 39ms/step – accuracy: 0.6779 – loss: 0.9758

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 ————— 18s 48ms/step – accuracy: 0.6698 – loss: 0.9935 – val\_accuracy: 0.6439 – val\_loss: 1.1542

Epoch 10/50

374/374 ————— 0s 40ms/step – accuracy: 0.6964 – loss: 0.9255

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 ————— 19s 50ms/step – accuracy: 0.6924 – loss: 0.9323 – val\_accuracy: 0.6503 – val\_loss: 1.1548

Epoch 11/50

374/374 ————— 0s 38ms/step – accuracy: 0.6979 – loss: 0.8765

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 ————— 18s 48ms/step – accuracy: 0.6969 – loss: 0.8870 – val\_accuracy: 0.6556 – val\_loss: 1.1465

Epoch 12/50

374/374 ————— 0s 39ms/step – accuracy: 0.7173 – loss: 0.8441

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 — 18s 49ms/step — accuracy: 0.7087 — loss: 0.8646 — val\_accuracy: 0.6623 — val\_loss: 1.1683  
Epoch 13/50

374/374 — 18s 48ms/step — accuracy: 0.7217 — loss: 0.8050 — val\_accuracy: 0.6573 — val\_loss: 1.1491  
Epoch 14/50

374/374 — 18s 47ms/step — accuracy: 0.7332 — loss: 0.7794 — val\_accuracy: 0.6479 — val\_loss: 1.1594  
Epoch 15/50

374/374 — 18s 47ms/step — accuracy: 0.7436 — loss: 0.7549 — val\_accuracy: 0.6563 — val\_loss: 1.1955  
Epoch 16/50

374/374 — 18s 49ms/step — accuracy: 0.7537 — loss: 0.7188 — val\_accuracy: 0.6576 — val\_loss: 1.1749  
Epoch 17/50

374/374 — 18s 48ms/step — accuracy: 0.7614 — loss: 0.7049 — val\_accuracy: 0.6583 — val\_loss: 1.1833  
Epoch 18/50

374/374 — 18s 48ms/step — accuracy: 0.7657 — loss: 0.6669 — val\_accuracy: 0.6516 — val\_loss: 1.2447  
Epoch 19/50

374/374 — 18s 47ms/step — accuracy: 0.7818 — loss: 0.6289 — val\_accuracy: 0.6616 — val\_loss: 1.2053  
Epoch 20/50

374/374 — 18s 48ms/step — accuracy: 0.7789 — loss: 0.6323 — val\_accuracy: 0.6563 — val\_loss: 1.2465  
Epoch 21/50

374/374 — 0s 38ms/step — accuracy: 0.7900 — loss: 0.6005

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

374/374 — 18s 49ms/step — accuracy: 0.7872 — loss: 0.6122 — val\_accuracy: 0.6667 — val\_loss: 1.2304  
Entrenamiento con Transfer Learning completado.



## 15.6. Transfer Learning - Evaluación

Análisis de resultados del modelo pre-entrenado para Transfer Learning.

```
In [55]: loss_tl, accuracy_tl = model_tl.evaluate(X_test_tl, y_test_ohe, verbose=1)
print(f"Accuracy en test (Transfer Learning): {accuracy_tl*100:.2f}%")
```

52/52 — 2s 36ms/step — accuracy: 0.5039 — loss: 1.7725  
Accuracy en test (Transfer Learning): 50.39%

## 16. Transfer Learning - Congelamiento de las Primeras Capas

Utilización de la técnica Transfer Learning, pero utilizando el congelamiento de solo las primeras capas (CPC), dejando entrenar las últimas (fine tuning).

### 16.1. Transfer Learning CPC - Configuración

Configuración del modelo pre-entrenado congelando solo las primeras capas.

```
In [56]: # Abre todas las capas del modelo base
base_model.trainable = True

# Congela las primeras capas y dejamos entrenar las últimas
for layer in base_model.layers[:-30]:
    layer.trainable = False

print("Algunas capas del modelo base descongeladas para fine-tuning.")
```

Algunas capas del modelo base descongeladas para fine-tuning.

### 16.2. Transfer Learning CPC - Compilación y Resumen

Compilación del modelo pre-entrenado y resumen de la arquitectura.

```
In [57]: model_tl.compile(
    optimizer=Adam(learning_rate=1e-4),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

model_tl.summary()
```

Model: "sequential\_10"

Layer (type)	Output Shape	
mobilenetv2_1.00_224 (Functional)	(None, 3, 3, 1280)	2,589,261
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	
dense_21 (Dense)	(None, 256)	327,680
dropout_38 (Dropout)	(None, 256)	
dense_22 (Dense)	(None, 13)	16,640

Total params: 2,589,261 (9.88 MB)

Trainable params: 1,857,677 (7.09 MB)

Non-trainable params: 731,584 (2.79 MB)




## 16.3. Transfer Learning CPC - Entrenamiento


Entrenamiento del modelo pre-entrenado.

```
In [58]: history_ft = model_tl.fit(
    X_train_tl, y_train,
    validation_data=(X_val_tl, y_val),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=callbacks
)
```


Epoch 1/50

**374/374**  **33s** 75ms/step - accuracy: 0.4897 - loss: 1.6325 - val\_accuracy: 0.5773 - val\_loss: 2.0558


Epoch 2/50

**374/374**  **25s** 67ms/step - accuracy: 0.6794 - loss: 1.0203 - val\_accuracy: 0.6466 - val\_loss: 1.6641


Epoch 3/50

**373/374**  **0s** 58ms/step - accuracy: 0.7626 - loss: 0.7487


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374**  **25s** 68ms/step - accuracy: 0.7660 - loss: 0.7418 - val\_accuracy: 0.6941 - val\_loss: 1.4651


Epoch 4/50

**374/374**  **0s** 58ms/step - accuracy: 0.8308 - loss: 0.5303


WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374**  **25s** 67ms/step - accuracy: 0.8312 - loss: 0.5319 - val\_accuracy: 0.7333 - val\_loss: 1.1742


Epoch 5/50

**374/374**  **0s** 57ms/step - accuracy: 0.8660 - loss: 0.4169

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374**  **25s** 67ms/step - accuracy: 0.8709 - loss: 0.4081 - val\_accuracy: 0.7697 - val\_loss: 1.0526

Epoch 6/50

**373/374**  **0s** 59ms/step - accuracy: 0.8949 - loss: 0.3210

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **26s** 68ms/step – accuracy: 0.8953 – loss: 0.3196 – val\_accuracy: 0.7838 – val\_loss: 0.9457

Epoch 7/50

**374/374** ————— **25s** 68ms/step – accuracy: 0.9199 – loss: 0.2454 – val\_accuracy: 0.7798 – val\_loss: 1.0056

Epoch 8/50

**373/374** ————— **0s** 57ms/step – accuracy: 0.9372 – loss: 0.1941

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **25s** 66ms/step – accuracy: 0.9357 – loss: 0.2001 – val\_accuracy: 0.7868 – val\_loss: 1.0421

Epoch 9/50

**373/374** ————— **0s** 58ms/step – accuracy: 0.9467 – loss: 0.1697

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **25s** 67ms/step – accuracy: 0.9458 – loss: 0.1697 – val\_accuracy: 0.7962 – val\_loss: 1.0386

Epoch 10/50

**374/374** ————— **26s** 69ms/step – accuracy: 0.9595 – loss: 0.1295 – val\_accuracy: 0.7892 – val\_loss: 1.0794

Epoch 11/50

**374/374** ————— **25s** 66ms/step – accuracy: 0.9591 – loss: 0.1275 – val\_accuracy: 0.7932 – val\_loss: 1.1185

Epoch 12/50

**373/374** ————— **0s** 58ms/step – accuracy: 0.9694 – loss: 0.1071

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **26s** 68ms/step – accuracy: 0.9655 – loss: 0.1172 – val\_accuracy: 0.8089 – val\_loss: 1.0430

Epoch 13/50

**374/374** ————— **25s** 68ms/step – accuracy: 0.9675 – loss: 0.1043 – val\_accuracy: 0.8062 – val\_loss: 1.0618

Epoch 14/50

**373/374** ————— **0s** 57ms/step – accuracy: 0.9764 – loss: 0.0813

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **25s** 67ms/step – accuracy: 0.9730 – loss: 0.0889 – val\_accuracy: 0.8133 – val\_loss: 1.0732

Epoch 15/50

Epoch 15/50

**374/374** ————— **25s** 68ms/step – accuracy: 0.9716 – loss: 0.0877 – val\_accuracy: 0.8049 – val\_loss: 1.1396

Epoch 16/50

**374/374** ————— **25s** 66ms/step – accuracy: 0.9772 – loss: 0.0778 – val\_accuracy: 0.8059 – val\_loss: 1.2058



## 16.4. Transfer Learning CPC - Evaluación

Análisis de resultados del modelo pre-entrenado para Transfer Learning.

```
In [59]: loss_tl, accuracy_tl = model_tl.evaluate(X_test_tl, y_test_ohe, verbose=1)
print(f"Accuracy en test (Transfer Learning + Fine-Tuning): {accuracy_tl}*
```

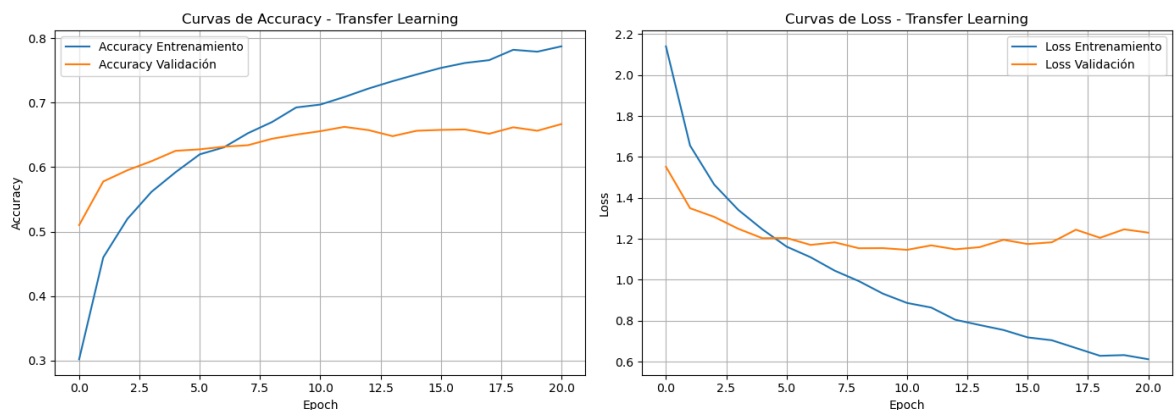
52/52 ————— 2s 35ms/step – accuracy: 0.6679 – loss: 1.7242  
Accuracy en test (Transfer Learning + Fine-Tuning): 66.79%

```
In [60]: plt.figure(figsize=(14, 5))

# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history_tl.history['accuracy'], label='Accuracy Entrenamiento')
plt.plot(history_tl.history['val_accuracy'], label='Accuracy Validación')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Curvas de Accuracy - Transfer Learning')
plt.legend()
plt.grid(True)

# Loss
plt.subplot(1, 2, 2)
plt.plot(history_tl.history['loss'], label='Loss Entrenamiento')
plt.plot(history_tl.history['val_loss'], label='Loss Validación')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Curvas de Loss - Transfer Learning')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



## 16.5. Transfer Learning CPC - Conclusión

En este experimento se aplicó la técnica Transfer Learning utilizando MobileNetV2 como modelo base preentrenado. Las imágenes fueron normalizadas y adaptadas al tamaño requerido por la red, pero no se aplicaron técnicas de data augmentation en esta etapa.

Los resultados obtenidos muestran que el modelo aprende correctamente los patrones del conjunto de entrenamiento, alcanzando una accuracy muy alta (98%). Sin embargo, la accuracy en el conjunto de validación se estabiliza alrededor del 80%, observándose una brecha creciente entre entrenamiento y validación. Este comportamiento, junto con la evolución de la función de pérdida, indica la aparición de overfitting moderado: mientras la pérdida de entrenamiento disminuye de forma continua, la pérdida de validación se mantiene prácticamente constante.

Este fenómeno sugiere que, aunque el modelo es capaz de ajustarse bien a los datos de entrenamiento, las representaciones aprendidas no generalizan adecuadamente a datos no vistos.

En conclusión, el uso de transfer learning sin data augmentation no resulta suficiente para alcanzar el objetivo de una accuracy superior al 85% en el conjunto de teste.

## 17. Transfer Learning + Data Augmentation

Es una combinación de dos técnicas poderosas:

1. Transfer Learning: Reutilizas un modelo pre-entrenado (MobileNetV2 en ImageNet) como base, congelando sus capas para preservar características genéricas ya aprendidas (bordes, texturas, formas).
2. Data Augmentation: Aplicas transformaciones a las imágenes de entrenamiento (rotaciones, desplazamientos, zoom, flip) para aumentar la diversidad de datos sin recopilar más imágenes.

Las ventajas són:

- Menos overfitting, más variedad de ejemplos.
- Mejor generalización porque el modelo aprende patrones más robustos.
- Entrenamientos más estables, curvas de train/val más cercanas.
- Computacionalmente eficiente, reutilizas características pre-entrenadas

En este ejemplo se utiliza MobileNetV2 congelado más transformaciones de imágenes, lo que resulta en modelo más robusto para clasificar personajes de Los Simpsons, reduciendo la brecha entre entrenamiento y validación.

### 17.1. Transfer Learning DA - Configuración

Cargando Data augmentation para Transfer Learning y regresando rango de datos para 0-255.

```
In [61]: # Data Augmentation específico para MobileNetV2
train_datagen = ImageDataGenerator(
    preprocessing_function=preprocess_input,
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.15,
```



```

        horizontal_flip=True
    )

    # Data Augmentation para el conjunto de validación (solo preprocesamiento)
    val_datagen = ImageDataGenerator(
        preprocessing_function=preprocess_input
    )

    # Generadores de datos
    train_generator = train_datagen.flow(
        X_train * 255,
        y_train,
        batch_size=BATCH_SIZE
    )

    val_generator = val_datagen.flow(
        X_val * 255,
        y_val,
        batch_size=BATCH_SIZE,
        shuffle=False
    )

    print("Data generators configurados para Transfer Learning con Data Augme

```

Data generators configurados para Transfer Learning con Data Augmentation.



## 17.2. Transfer Learning DA - Modelo y Capas

Carga del modelo pre-entrenado MobileNetV2 y congelamiento de las capas convolucionales.

```

In [62]: # Modelo base preentrenado
base_model = MobileNetV2(
    weights="imagenet",
    include_top=False,
    input_shape=(IMG_SIZE, IMG_SIZE, 3)
)

# Congelar capas convolucionales
for layer in base_model.layers:
    layer.trainable = False

print("Modelo base cargado y capas congeladas para Transfer Learning con

```

```

/var/folders/7g/ch5l_np56q51q3c4q01r2hs00000gn/T/ipykernel_29902/218356730
9.py:2: UserWarning: `input_shape` is undefined or non-square, or `rows` i
s not in [96, 128, 160, 192, 224]. Weights for input shape (224, 224) will
be loaded as the default.

```

```

    base_model = MobileNetV2(
Modelo base cargado y capas congeladas para Transfer Learning con Data Aug
mentation.

```



## 17.3. Transfer Learning DA - Añadiendo Capas

Añadiendo capas al modelo pre-entrenado para Data Augmentation.

```

In [63]: x = base_model.output
x = GlobalAveragePooling2D()(x)

```



```
x = Dense(256, activation="relu")(x)
x = Dropout(0.5)(x)

output = Dense(num_classes, activation="softmax")(x)
model_tl = Model(inputs=base_model.input, outputs=output)

print("Modelo completo con nuevas capas densas creado.")
```

Modelo completo con nuevas capas densas creado.



## 17.4. Transfer Learning DA - Compilación y Resumen

Compilación del modelo pre-entrenado y resumen de la arquitectura.

```
In [64]: model_tl.compile(
            optimizer=Adam(learning_rate=1e-3),
            loss="categorical_crossentropy",
            metrics=["accuracy"]
        )

model_tl.summary()
```

**Model: "functional\_27"**

Layer (type)	Output Shape	Param #	Connected to
input_layer_12 (InputLayer)	(None, 90, 90, 3)	0	–
Conv1 (Conv2D)	(None, 45, 45, 32)	864	input_layer_12
bn_Conv1 (BatchNormalization)	(None, 45, 45, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 45, 45, 32)	0	bn_Conv1[0]
expanded_conv_dept... (DepthwiseConv2D)	(None, 45, 45, 32)	288	Conv1_relu
expanded_conv_dept... (BatchNormalization)	(None, 45, 45, 32)	128	expanded_conv_dept...
expanded_conv_dept... (ReLU)	(None, 45, 45, 32)	0	expanded_conv_dept...
expanded_conv_proj... (Conv2D)	(None, 45, 45, 16)	512	expanded_conv_dept...
expanded_conv_proj... (BatchNormalization)	(None, 45, 45, 16)	64	expanded_conv_dept...
block_1_expand (Conv2D)	(None, 45, 45, 96)	1,536	expanded_conv_dept...
block_1_expand_BN (BatchNormalization)	(None, 45, 45, 96)	384	block_1_expand
block_1_expand_relu (ReLU)	(None, 45, 45, 96)	0	block_1_expand
block_1_pad (ZeroPadding2D)	(None, 47, 47, 96)	0	block_1_expand
block_1_depthwise (DepthwiseConv2D)	(None, 23, 23, 96)	864	block_1_pad
block_1_depthwise_... (BatchNormalization)	(None, 23, 23, 96)	384	block_1_depthwise
block_1_depthwise_... (ReLU)	(None, 23, 23, 96)	0	block_1_depthwise_...
block_1_project (Conv2D)	(None, 23, 23, 24)	2,304	block_1_depthwise_...
block_1_project_BN (BatchNormalization)	(None, 23, 23, 24)	96	block_1_project
block_2_expand (Conv2D)	(None, 23, 23, 144)	3,456	block_1_project
block_2_expand_BN (BatchNormalization)	(None, 23, 23, 144)	576	block_2_expand
block_2_expand_relu	(None, 23, 23, 144)	0	block_2_expand

(ReLU)	144)		
block_2_depthwise (DepthwiseConv2D)	(None, 23, 23, 144)	1,296	block_2_exp
block_2_depthwise_... (BatchNormalizatio...	(None, 23, 23, 144)	576	block_2_dep
block_2_depthwise_... (ReLU)	(None, 23, 23, 144)	0	block_2_dep
block_2_project (Conv2D)	(None, 23, 23, 24)	3,456	block_2_dep
block_2_project_BN (BatchNormalizatio...	(None, 23, 23, 24)	96	block_2_pro
block_2_add (Add)	(None, 23, 23, 24)	0	block_1_pro block_2_pro
block_3_expand (Conv2D)	(None, 23, 23, 144)	3,456	block_2_ad
block_3_expand_BN (BatchNormalizatio...	(None, 23, 23, 144)	576	block_3_exp
block_3_expand_relu (ReLU)	(None, 23, 23, 144)	0	block_3_exp
block_3_pad (ZeroPadding2D)	(None, 25, 25, 144)	0	block_3_exp
block_3_depthwise (DepthwiseConv2D)	(None, 12, 12, 144)	1,296	block_3_pac
block_3_depthwise_... (BatchNormalizatio...	(None, 12, 12, 144)	576	block_3_dep
block_3_depthwise_... (ReLU)	(None, 12, 12, 144)	0	block_3_dep
block_3_project (Conv2D)	(None, 12, 12, 32)	4,608	block_3_dep
block_3_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_3_pro
block_4_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_3_pro
block_4_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_4_exp
block_4_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_4_exp
block_4_depthwise (DepthwiseConv2D)	(None, 12, 12, 192)	1,728	block_4_exp
block_4_depthwise_... (BatchNormalizatio...	(None, 12, 12, 192)	768	block_4_dep
block_4_depthwise_... (ReLU)	(None, 12, 12, 192)	0	block_4_dep

block_4_project (Conv2D)	(None, 12, 12, 32)	6,144	block_4_dep
block_4_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_4_pro
block_4_add (Add)	(None, 12, 12, 32)	0	block_3_pro block_4_pro
block_5_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_4_ad
block_5_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_5_exp
block_5_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_5_exp
block_5_depthwise (DepthwiseConv2D)	(None, 12, 12, 192)	1,728	block_5_exp
block_5_depthwise_... (BatchNormalizatio...	(None, 12, 12, 192)	768	block_5_dep
block_5_depthwise_... (ReLU)	(None, 12, 12, 192)	0	block_5_dep
block_5_project (Conv2D)	(None, 12, 12, 32)	6,144	block_5_dep
block_5_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_5_pro
block_5_add (Add)	(None, 12, 12, 32)	0	block_4_ad block_5_pro
block_6_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_5_ad
block_6_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_6_exp
block_6_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_6_exp
block_6_pad (ZeroPadding2D)	(None, 13, 13, 192)	0	block_6_exp
block_6_depthwise (DepthwiseConv2D)	(None, 6, 6, 192)	1,728	block_6_pac
block_6_depthwise_... (BatchNormalizatio...	(None, 6, 6, 192)	768	block_6_dep
block_6_depthwise_... (ReLU)	(None, 6, 6, 192)	0	block_6_dep
block_6_project (Conv2D)	(None, 6, 6, 64)	12,288	block_6_dep
block_6_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_6_pro

block_7_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_6_pro
block_7_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_7_exp
block_7_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_7_exp
block_7_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_7_exp
block_7_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_7_dep
block_7_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_7_dep
block_7_project (Conv2D)	(None, 6, 6, 64)	24,576	block_7_dep
block_7_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_7_pro
block_7_add (Add)	(None, 6, 6, 64)	0	block_6_pro block_7_pro
block_8_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_7_ad
block_8_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_8_exp
block_8_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_8_exp
block_8_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_8_exp
block_8_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_8_dep
block_8_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_8_dep
block_8_project (Conv2D)	(None, 6, 6, 64)	24,576	block_8_dep
block_8_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_8_pro
block_8_add (Add)	(None, 6, 6, 64)	0	block_7_ad block_8_pro
block_9_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_8_ad
block_9_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_9_exp
block_9_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_9_exp
block_9_depthwise	(None, 6, 6, 384)	3,456	block_9_exp

(DepthwiseConv2D)			
block_9_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_9_dep
block_9_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_9_dep
block_9_project (Conv2D)	(None, 6, 6, 64)	24,576	block_9_dep
block_9_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_9_pro
block_9_add (Add)	(None, 6, 6, 64)	0	block_8_add block_9_pro
block_10_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_9_add
block_10_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_10_e
block_10_expand_re... (ReLU)	(None, 6, 6, 384)	0	block_10_e
block_10_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_10_e
block_10_depthwise... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_10_de
block_10_depthwise... (ReLU)	(None, 6, 6, 384)	0	block_10_de
block_10_project (Conv2D)	(None, 6, 6, 96)	36,864	block_10_de
block_10_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_10_pi
block_11_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_10_pi
block_11_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_11_e
block_11_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_11_e
block_11_depthwise (DepthwiseConv2D)	(None, 6, 6, 576)	5,184	block_11_e
block_11_depthwise... (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_11_de
block_11_depthwise... (ReLU)	(None, 6, 6, 576)	0	block_11_de
block_11_project (Conv2D)	(None, 6, 6, 96)	55,296	block_11_de
block_11_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_11_pi

block_11_add (Add)	(None, 6, 6, 96)	0	block_10_pi block_11_pi
block_12_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_11_ac
block_12_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_12_e)
block_12_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_12_e)
block_12_depthwise (DepthwiseConv2D)	(None, 6, 6, 576)	5,184	block_12_e)
block_12_depthwise... (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_12_de
block_12_depthwise... (ReLU)	(None, 6, 6, 576)	0	block_12_de
block_12_project (Conv2D)	(None, 6, 6, 96)	55,296	block_12_de
block_12_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_12_pi
block_12_add (Add)	(None, 6, 6, 96)	0	block_11_ac block_12_pi
block_13_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_12_ac
block_13_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_13_e)
block_13_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_13_e)
block_13_pad (ZeroPadding2D)	(None, 7, 7, 576)	0	block_13_e)
block_13_depthwise (DepthwiseConv2D)	(None, 3, 3, 576)	5,184	block_13_p)
block_13_depthwise... (BatchNormalizatio...	(None, 3, 3, 576)	2,304	block_13_de
block_13_depthwise... (ReLU)	(None, 3, 3, 576)	0	block_13_de
block_13_project (Conv2D)	(None, 3, 3, 160)	92,160	block_13_de
block_13_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_13_pi
block_14_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_13_pi
block_14_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_14_e)

block_14_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_14_e
block_14_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_14_e
block_14_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_14_de
block_14_depthwise... (ReLU)	(None, 3, 3, 960)	0	block_14_de
block_14_project (Conv2D)	(None, 3, 3, 160)	153,600	block_14_de
block_14_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_14_pi
block_14_add (Add)	(None, 3, 3, 160)	0	block_13_pi block_14_pi
block_15_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_14_ac
block_15_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_15_e
block_15_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_15_e
block_15_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_15_e
block_15_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_15_de
block_15_depthwise... (ReLU)	(None, 3, 3, 960)	0	block_15_de
block_15_project (Conv2D)	(None, 3, 3, 160)	153,600	block_15_de
block_15_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_15_pi
block_15_add (Add)	(None, 3, 3, 160)	0	block_14_ac block_15_pi
block_16_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_15_ac
block_16_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_16_e
block_16_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_16_e
block_16_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_16_e
block_16_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_16_de
block_16_depthwise...	(None, 3, 3, 960)	0	block_16_de



(ReLU)			
block_16_project (Conv2D)	(None, 3, 3, 320)	307,200	block_16_de
block_16_project_BN (BatchNormalizatio...	(None, 3, 3, 320)	1,280	block_16_pi
Conv_1 (Conv2D)	(None, 3, 3, 1280)	409,600	block_16_pi
Conv_1_bn (BatchNormalizatio...	(None, 3, 3, 1280)	5,120	Conv_1[0][0
out_relu (ReLU)	(None, 3, 3, 1280)	0	Conv_1_bn[0
global_average_poo... (GlobalAveragePool...	(None, 1280)	0	out_relu[0]
dense_23 (Dense)	(None, 256)	327,936	global_avei
dropout_39 (Dropout)	(None, 256)	0	dense_23[0]
dense_24 (Dense)	(None, 13)	3,341	dropout_39

**Total params:** 2,589,261 (9.88 MB)

**Trainable params:** 331,277 (1.26 MB)

**Non-trainable params:** 2,257,984 (8.61 MB)



## 17.5. Transfer Learning DA - Entrenamiento

Entrenamiento del modelo con Transger Learning y Data Augmentation.

```
In [65]: history_tl = model_tl.fit(
    train_generator,
    validation_data=val_generator,
    epochs=EPOCHS,
    callbacks=callbacks
)

print("Entrenamiento con Transfer Learning y Data Augmentation completado")
```

Epoch 1/50  
**374/374** ————— **24s** 55ms/step – accuracy: 0.2713 – loss: 2.22  
 31 – val\_accuracy: 0.4796 – val\_loss: 1.6820  
 Epoch 2/50  
**374/374** ————— **19s** 52ms/step – accuracy: 0.3890 – loss: 1.89  
 07 – val\_accuracy: 0.5301 – val\_loss: 1.5237  
 Epoch 3/50  
**374/374** ————— **20s** 52ms/step – accuracy: 0.4087 – loss: 1.80  
 36 – val\_accuracy: 0.5335 – val\_loss: 1.4744  
 Epoch 4/50  
**374/374** ————— **19s** 52ms/step – accuracy: 0.4388 – loss: 1.72  
 90 – val\_accuracy: 0.5807 – val\_loss: 1.3532  
 Epoch 5/50  
**374/374** ————— **20s** 53ms/step – accuracy: 0.4609 – loss: 1.67  
 19 – val\_accuracy: 0.5679 – val\_loss: 1.3553  
 Epoch 6/50  
**374/374** ————— **20s** 53ms/step – accuracy: 0.4560 – loss: 1.67  
 02 – val\_accuracy: 0.5840 – val\_loss: 1.3192  
 Epoch 7/50  
**374/374** ————— **20s** 55ms/step – accuracy: 0.4710 – loss: 1.63  
 47 – val\_accuracy: 0.5760 – val\_loss: 1.3450  
 Epoch 8/50  
**374/374** ————— **20s** 54ms/step – accuracy: 0.4853 – loss: 1.59  
 25 – val\_accuracy: 0.5947 – val\_loss: 1.2903  
 Epoch 9/50  
**374/374** ————— **21s** 56ms/step – accuracy: 0.4873 – loss: 1.57  
 62 – val\_accuracy: 0.6021 – val\_loss: 1.2682  
 Epoch 10/50  
**374/374** ————— **20s** 54ms/step – accuracy: 0.4979 – loss: 1.55  
 30 – val\_accuracy: 0.5991 – val\_loss: 1.2881  
 Entrenamiento con Transfer Learning y Data Augmentation completado.



## 17.6. Transfer Learning DA - Evaluación

Análisis de resultados de la arquitectura Transfer Learning con Data Augmentation.

```
In [66]: loss_tl, accuracy_tl = model_tl.evaluate(X_test_tl, y_test_ohe, verbose=1)
print(f"Accuracy en test (Transfer Learning + Data Augmentation): {accuracy_tl}")
```

**52/52** ————— **3s** 39ms/step – accuracy: 0.3908 – loss: 1.8738  
 Accuracy en test (Transfer Learning + Data Augmentation): 39.08%



## 17.7. Transfer Learning DA - Descongelando Capas

Descongelando las últimas capas para nuevo entrenamiento.

```
In [67]: # Descongelar las últimas capas
for layer in base_model.layers[-30:]:
    layer.trainable = True

# Compilar el modelo nuevamente
model_tl.compile(
    optimizer=Adam(learning_rate=1e-4),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

# Resumen del modelo
```

```
model_tl.summary()  
  
# Entrenamiento con fine-tuning usando data augmentation  
history_ft = model_tl.fit(  
    train_generator,  
    validation_data=val_generator,  
    epochs=15,  
    callbacks=callbacks  
)
```

**Model: "functional\_27"**

Layer (type)	Output Shape	Param #	Connected to
input_layer_12 (InputLayer)	(None, 90, 90, 3)	0	–
Conv1 (Conv2D)	(None, 45, 45, 32)	864	input_layer_12
bn_Conv1 (BatchNormalization)	(None, 45, 45, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 45, 45, 32)	0	bn_Conv1[0]
expanded_conv_dept... (DepthwiseConv2D)	(None, 45, 45, 32)	288	Conv1_relu
expanded_conv_dept... (BatchNormalization)	(None, 45, 45, 32)	128	expanded_conv_dept...
expanded_conv_dept... (ReLU)	(None, 45, 45, 32)	0	expanded_conv_dept...
expanded_conv_proj... (Conv2D)	(None, 45, 45, 16)	512	expanded_conv_dept...
expanded_conv_proj... (BatchNormalization)	(None, 45, 45, 16)	64	expanded_conv_dept...
block_1_expand (Conv2D)	(None, 45, 45, 96)	1,536	expanded_conv_dept...
block_1_expand_BN (BatchNormalization)	(None, 45, 45, 96)	384	block_1_expand
block_1_expand_relu (ReLU)	(None, 45, 45, 96)	0	block_1_expand
block_1_pad (ZeroPadding2D)	(None, 47, 47, 96)	0	block_1_expand
block_1_depthwise (DepthwiseConv2D)	(None, 23, 23, 96)	864	block_1_pad
block_1_depthwise_... (BatchNormalization)	(None, 23, 23, 96)	384	block_1_depthwise
block_1_depthwise_... (ReLU)	(None, 23, 23, 96)	0	block_1_depthwise_...
block_1_project (Conv2D)	(None, 23, 23, 24)	2,304	block_1_depthwise_...
block_1_project_BN (BatchNormalization)	(None, 23, 23, 24)	96	block_1_project
block_2_expand (Conv2D)	(None, 23, 23, 144)	3,456	block_1_project
block_2_expand_BN (BatchNormalization)	(None, 23, 23, 144)	576	block_2_expand
block_2_expand_relu	(None, 23, 23, 144)	0	block_2_expand

(ReLU)	144)		
block_2_depthwise (DepthwiseConv2D)	(None, 23, 23, 144)	1,296	block_2_exp
block_2_depthwise_... (BatchNormalizatio...	(None, 23, 23, 144)	576	block_2_dep
block_2_depthwise_... (ReLU)	(None, 23, 23, 144)	0	block_2_dep
block_2_project (Conv2D)	(None, 23, 23, 24)	3,456	block_2_dep
block_2_project_BN (BatchNormalizatio...	(None, 23, 23, 24)	96	block_2_pro
block_2_add (Add)	(None, 23, 23, 24)	0	block_1_pro block_2_pro
block_3_expand (Conv2D)	(None, 23, 23, 144)	3,456	block_2_ad
block_3_expand_BN (BatchNormalizatio...	(None, 23, 23, 144)	576	block_3_exp
block_3_expand_relu (ReLU)	(None, 23, 23, 144)	0	block_3_exp
block_3_pad (ZeroPadding2D)	(None, 25, 25, 144)	0	block_3_exp
block_3_depthwise (DepthwiseConv2D)	(None, 12, 12, 144)	1,296	block_3_pac
block_3_depthwise_... (BatchNormalizatio...	(None, 12, 12, 144)	576	block_3_dep
block_3_depthwise_... (ReLU)	(None, 12, 12, 144)	0	block_3_dep
block_3_project (Conv2D)	(None, 12, 12, 32)	4,608	block_3_dep
block_3_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_3_pro
block_4_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_3_pro
block_4_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_4_exp
block_4_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_4_exp
block_4_depthwise (DepthwiseConv2D)	(None, 12, 12, 192)	1,728	block_4_exp
block_4_depthwise_... (BatchNormalizatio...	(None, 12, 12, 192)	768	block_4_dep
block_4_depthwise_... (ReLU)	(None, 12, 12, 192)	0	block_4_dep

block_4_project (Conv2D)	(None, 12, 12, 32)	6,144	block_4_dep
block_4_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_4_pro
block_4_add (Add)	(None, 12, 12, 32)	0	block_3_pro block_4_pro
block_5_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_4_ad
block_5_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_5_exp
block_5_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_5_exp
block_5_depthwise (DepthwiseConv2D)	(None, 12, 12, 192)	1,728	block_5_exp
block_5_depthwise_... (BatchNormalizatio...	(None, 12, 12, 192)	768	block_5_dep
block_5_depthwise_... (ReLU)	(None, 12, 12, 192)	0	block_5_dep
block_5_project (Conv2D)	(None, 12, 12, 32)	6,144	block_5_dep
block_5_project_BN (BatchNormalizatio...	(None, 12, 12, 32)	128	block_5_pro
block_5_add (Add)	(None, 12, 12, 32)	0	block_4_ad block_5_pro
block_6_expand (Conv2D)	(None, 12, 12, 192)	6,144	block_5_ad
block_6_expand_BN (BatchNormalizatio...	(None, 12, 12, 192)	768	block_6_exp
block_6_expand_relu (ReLU)	(None, 12, 12, 192)	0	block_6_exp
block_6_pad (ZeroPadding2D)	(None, 13, 13, 192)	0	block_6_exp
block_6_depthwise (DepthwiseConv2D)	(None, 6, 6, 192)	1,728	block_6_pac
block_6_depthwise_... (BatchNormalizatio...	(None, 6, 6, 192)	768	block_6_dep
block_6_depthwise_... (ReLU)	(None, 6, 6, 192)	0	block_6_dep
block_6_project (Conv2D)	(None, 6, 6, 64)	12,288	block_6_dep
block_6_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_6_pro

block_7_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_6_pro
block_7_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_7_exp
block_7_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_7_exp
block_7_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_7_exp
block_7_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_7_dep
block_7_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_7_dep
block_7_project (Conv2D)	(None, 6, 6, 64)	24,576	block_7_dep
block_7_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_7_pro
block_7_add (Add)	(None, 6, 6, 64)	0	block_6_pro block_7_pro
block_8_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_7_ad
block_8_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_8_exp
block_8_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_8_exp
block_8_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_8_exp
block_8_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_8_dep
block_8_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_8_dep
block_8_project (Conv2D)	(None, 6, 6, 64)	24,576	block_8_dep
block_8_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_8_pro
block_8_add (Add)	(None, 6, 6, 64)	0	block_7_ad block_8_pro
block_9_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_8_ad
block_9_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_9_exp
block_9_expand_relu (ReLU)	(None, 6, 6, 384)	0	block_9_exp
block_9_depthwise	(None, 6, 6, 384)	3,456	block_9_exp

(DepthwiseConv2D)			
block_9_depthwise_... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_9_dep
block_9_depthwise_... (ReLU)	(None, 6, 6, 384)	0	block_9_dep
block_9_project (Conv2D)	(None, 6, 6, 64)	24,576	block_9_dep
block_9_project_BN (BatchNormalizatio...	(None, 6, 6, 64)	256	block_9_pro
block_9_add (Add)	(None, 6, 6, 64)	0	block_8_add block_9_pro
block_10_expand (Conv2D)	(None, 6, 6, 384)	24,576	block_9_add
block_10_expand_BN (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_10_e
block_10_expand_re... (ReLU)	(None, 6, 6, 384)	0	block_10_e
block_10_depthwise (DepthwiseConv2D)	(None, 6, 6, 384)	3,456	block_10_e
block_10_depthwise... (BatchNormalizatio...	(None, 6, 6, 384)	1,536	block_10_de
block_10_depthwise... (ReLU)	(None, 6, 6, 384)	0	block_10_de
block_10_project (Conv2D)	(None, 6, 6, 96)	36,864	block_10_de
block_10_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_10_pi
block_11_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_10_pi
block_11_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_11_e
block_11_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_11_e
block_11_depthwise (DepthwiseConv2D)	(None, 6, 6, 576)	5,184	block_11_e
block_11_depthwise... (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_11_de
block_11_depthwise... (ReLU)	(None, 6, 6, 576)	0	block_11_de
block_11_project (Conv2D)	(None, 6, 6, 96)	55,296	block_11_de
block_11_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_11_pi



block_11_add (Add)	(None, 6, 6, 96)	0	block_10_pi block_11_pi
block_12_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_11_ac
block_12_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_12_e)
block_12_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_12_e)
block_12_depthwise (DepthwiseConv2D)	(None, 6, 6, 576)	5,184	block_12_e)
block_12_depthwise... (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_12_de
block_12_depthwise... (ReLU)	(None, 6, 6, 576)	0	block_12_de
block_12_project (Conv2D)	(None, 6, 6, 96)	55,296	block_12_de
block_12_project_BN (BatchNormalizatio...	(None, 6, 6, 96)	384	block_12_pi
block_12_add (Add)	(None, 6, 6, 96)	0	block_11_ac block_12_pi
block_13_expand (Conv2D)	(None, 6, 6, 576)	55,296	block_12_ac
block_13_expand_BN (BatchNormalizatio...	(None, 6, 6, 576)	2,304	block_13_e)
block_13_expand_re... (ReLU)	(None, 6, 6, 576)	0	block_13_e)
block_13_pad (ZeroPadding2D)	(None, 7, 7, 576)	0	block_13_e)
block_13_depthwise (DepthwiseConv2D)	(None, 3, 3, 576)	5,184	block_13_p)
block_13_depthwise... (BatchNormalizatio...	(None, 3, 3, 576)	2,304	block_13_de
block_13_depthwise... (ReLU)	(None, 3, 3, 576)	0	block_13_de
block_13_project (Conv2D)	(None, 3, 3, 160)	92,160	block_13_de
block_13_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_13_pi
block_14_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_13_pi
block_14_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_14_e)

block_14_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_14_e
block_14_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_14_e
block_14_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_14_de
block_14_depthwise... (ReLU)	(None, 3, 3, 960)	0	block_14_de
block_14_project (Conv2D)	(None, 3, 3, 160)	153,600	block_14_de
block_14_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_14_pi
block_14_add (Add)	(None, 3, 3, 160)	0	block_13_pi block_14_pi
block_15_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_14_ac
block_15_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_15_e
block_15_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_15_e
block_15_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_15_e
block_15_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_15_de
block_15_depthwise... (ReLU)	(None, 3, 3, 960)	0	block_15_de
block_15_project (Conv2D)	(None, 3, 3, 160)	153,600	block_15_de
block_15_project_BN (BatchNormalizatio...	(None, 3, 3, 160)	640	block_15_pi
block_15_add (Add)	(None, 3, 3, 160)	0	block_14_ac block_15_pi
block_16_expand (Conv2D)	(None, 3, 3, 960)	153,600	block_15_ac
block_16_expand_BN (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_16_e
block_16_expand_re... (ReLU)	(None, 3, 3, 960)	0	block_16_e
block_16_depthwise (DepthwiseConv2D)	(None, 3, 3, 960)	8,640	block_16_e
block_16_depthwise... (BatchNormalizatio...	(None, 3, 3, 960)	3,840	block_16_de
block_16_depthwise...	(None, 3, 3, 960)	0	block_16_de

(ReLU)			
block_16_project (Conv2D)	(None, 3, 3, 320)	307,200	block_16_de
block_16_project_BN (BatchNormalizatio...	(None, 3, 3, 320)	1,280	block_16_pi
Conv_1 (Conv2D)	(None, 3, 3, 1280)	409,600	block_16_pi
Conv_1_bn (BatchNormalizatio...	(None, 3, 3, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 3, 3, 1280)	0	Conv_1_bn[0]
global_average_poo... (GlobalAveragePool...	(None, 1280)	0	out_relu[0]
dense_23 (Dense)	(None, 256)	327,936	global_ave
dropout_39 (Dropout)	(None, 256)	0	dense_23[0]
dense_24 (Dense)	(None, 13)	3,341	dropout_39

**Total params:** 2,589,261 (9.88 MB)

**Trainable params:** 1,857,677 (7.09 MB)

**Non-trainable params:** 731,584 (2.79 MB)

Epoch 1/15

374/374 ————— 32s 74ms/step – accuracy: 0.4133 – loss: 1.8178 – val\_accuracy: 0.5382 – val\_loss: 1.8646

Epoch 2/15

374/374 ————— 27s 73ms/step – accuracy: 0.5911 – loss: 1.2994 – val\_accuracy: 0.6499 – val\_loss: 1.4240

Epoch 3/15

374/374 ————— 29s 77ms/step – accuracy: 0.6663 – loss: 1.0852 – val\_accuracy: 0.7185 – val\_loss: 1.1413

Epoch 4/15

374/374 ————— 29s 77ms/step – accuracy: 0.7053 – loss: 0.9428 – val\_accuracy: 0.7597 – val\_loss: 0.8887

Epoch 5/15

374/374 ————— 27s 72ms/step – accuracy: 0.7408 – loss: 0.8465 – val\_accuracy: 0.7885 – val\_loss: 0.7791

Epoch 6/15

374/374 ————— 27s 71ms/step – accuracy: 0.7636 – loss: 0.7699 – val\_accuracy: 0.8035 – val\_loss: 0.7069

Epoch 7/15

374/374 ————— 0s 61ms/step – accuracy: 0.7797 – loss: 0.7092

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **27s** 71ms/step – accuracy: 0.7805 – loss: 0.7069 – val\_accuracy: 0.8206 – val\_loss: 0.6381

Epoch 8/15

**374/374** ————— **0s** 64ms/step – accuracy: 0.7968 – loss: 0.6551

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **28s** 74ms/step – accuracy: 0.7949 – loss: 0.6633 – val\_accuracy: 0.8283 – val\_loss: 0.5852

Epoch 9/15

**374/374** ————— **0s** 64ms/step – accuracy: 0.8127 – loss: 0.6026

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **28s** 74ms/step – accuracy: 0.8093 – loss: 0.6172 – val\_accuracy: 0.8420 – val\_loss: 0.5570

Epoch 10/15

**374/374** ————— **28s** 75ms/step – accuracy: 0.8228 – loss: 0.5885 – val\_accuracy: 0.8263 – val\_loss: 0.5995

Epoch 11/15

**374/374** ————— **27s** 71ms/step – accuracy: 0.8308 – loss: 0.5496 – val\_accuracy: 0.8417 – val\_loss: 0.5576

Epoch 12/15

**374/374** ————— **26s** 69ms/step – accuracy: 0.8369 – loss: 0.5336 – val\_accuracy: 0.8153 – val\_loss: 0.6437

Epoch 13/15

**374/374** ————— **26s** 71ms/step – accuracy: 0.8497 – loss: 0.4836 – val\_accuracy: 0.8397 – val\_loss: 0.5713

Epoch 14/15

**374/374** ————— **26s** 70ms/step – accuracy: 0.8503 – loss: 0.4823 – val\_accuracy: 0.8327 – val\_loss: 0.6034

Epoch 15/15

**374/374** ————— **0s** 64ms/step – accuracy: 0.8582 – loss: 0.4628

WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save\_model(model)`. This file format is considered legacy. We recommend using instead the native Keras format, e.g. `model.save('my\_model.keras')` or `keras.saving.save\_model(model, 'my\_model.keras')`.

**374/374** ————— **28s** 74ms/step – accuracy: 0.8590 – loss: 0.4612 – val\_accuracy: 0.8507 – val\_loss: 0.5181



## 17.8. Transfer Learning DA - Evaluación

Análisis de resultados de la arquitectura Transfer Learning con Data Augmentation y capas descongeladas.

```
In [68]: loss_tl, accuracy_tl = model_tl.evaluate(X_test_tl, y_test_ohe, verbose=1)
         print(f"Accuracy en test (Transfer Learning + Data Augmentation): {accuracy_tl}")
```

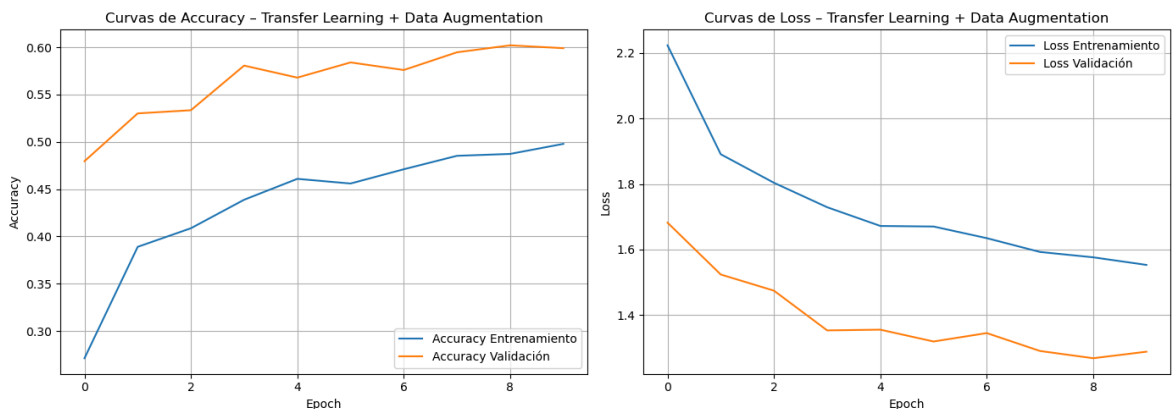
**52/52** ————— **3s** 36ms/step – accuracy: 0.7574 – loss: 1.0252  
Accuracy en test (Transfer Learning + Data Augmentation): 75.74%

```
In [69]: plt.figure(figsize=(14, 5))
```

```
# Accuracy
plt.subplot(1, 2, 1)
plt.plot(history_tl.history["accuracy"], label="Accuracy Entrenamiento")
plt.plot(history_tl.history["val_accuracy"], label="Accuracy Validación")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Curvas de Accuracy – Transfer Learning + Data Augmentation")
plt.legend()
plt.grid(True)

# Loss
plt.subplot(1, 2, 2)
plt.plot(history_tl.history["loss"], label="Loss Entrenamiento")
plt.plot(history_tl.history["val_loss"], label="Loss Validación")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Curvas de Loss – Transfer Learning + Data Augmentation")
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



## 🎯 17.9. Transfer Learning DA - Conclusión

En esta fase del trabajo se evaluó el impacto de la aplicación de data augmentation sobre un modelo de transfer learning basado en MobileNetV2, con el objetivo de mejorar la capacidad de generalización del clasificador.

Los resultados muestran que, al aplicar data augmentation, las curvas de *accuracy* de entrenamiento y validación se mantienen muy próximas entre sí a lo largo de las épocas. Este comportamiento indica que el modelo reduce el sobreajuste observado en la versión sin data augmentation, ya que la brecha entre entrenamiento y validación es menor y las pérdidas evolucionan de forma similar en ambos conjuntos.

Sin embargo, esta mejora en la regularización no se traduce en un incremento significativo del rendimiento absoluto del modelo. Este resultado refuerza la hipótesis de que el principal factor limitante no es la falta de datos efectivos, sino la adecuación del modelo preentrenado al dominio del problema. Las transformaciones aplicadas mediante data augmentation pueden introducir variaciones útiles en imágenes fotográficas reales, pero en el caso de imágenes tipo *cartoon* como las de

*The Simpsons*, algunas de estas transformaciones generan ejemplos poco realistas que no aportan información relevante al aprendizaje.

En conclusión, el uso de data augmentation en combinación con transfer learning mejora la estabilidad del entrenamiento y reduce el sobreajuste, pero no permite alcanzar el objetivo de una accuracy superior al 85% en el conjunto de test.

## 18. Conclusión del Trabajo

Este trabajo comienza con la carga de datos, una base de datos de imágenes de la famosa serie "Los Simpson". A continuación, se realiza el procesamiento de datos, que incluye:

- Normalización de datos, para obtener estabilidad numérica, entre otras ventajas;
- One-hot Encoding, para convertir datos categóricos a un formato numérico;
- Separación de los datos en dos conjuntos, uno para el entrenamiento y otro para la validación del modelo.

### 18.1. Experimentos Realizados

Se realizaron experimentos con diferentes arquitecturas de redes neuronales:

1. CNN Convolutional Neural Networks con Keras, utilizando una arquitectura básica para el primer entrenamiento.
2. Fully Connected, modelo que aplanar las imágenes y pasar los píxeles por capas densas con regularización, normalización y dropout para reducir sobreajuste.
3. CNN más profunda, aumentada la profundidad de la red para que el modelo pueda aprender representaciones jerárquicas más complejas.
4. CNN con más capas, experimento con más capas en la arquitectura para intentar obtener resultados mejores.
5. CNN con Cambio en Hyperparameters, cambios en dropout, learning rate y regularización de las capas densas para intentar mejorar la generalización del modelo.
6. CNN con BatchNormalization, estabiliza el entrenamiento y permite tasas de aprendizaje más altas.
7. CNN con Data Augmentation: Aumenta la diversidad de datos de entrenamiento.
8. CNN con Optimizers: Actualizan los pesos de la red neuronal durante el entrenamiento.
9. CNN con BatchNormalization, Data Augmentation y Optimizer Nadam: Combinación de las técnicas que presentaron mejores resultados.
10. Transfer Learning: Utilización de un modelo pre-entrenado como punto de partida para resolver un problema nuevo.
11. Transfer Learning con Data Augmentation y Congelamiento: Congelamiento de las primeras capas.
12. Transfer Learning con Data Augmentation sin Congelamiento: Descongelamiento de las últimas capas para nuevo entrenamiento.

## 18.2. 🇧🇷 Tabla Comparativa

A continuación se muestra una tabla comparativa con todas las variaciones o arquitecturas probadas y los resultados obtenidos:

N	Red	Accuracy Train	Accuracy Test	Loss
01	CNN Basica	93,5%	81,4%	0,92
02	Fully Connected	20,9%	13,2%	4,23
03	CNN +Profunda	91,9%	36,2%	2,03
04	CNN +Capas	13,5%	13,5%	2,53
05	CNN +Hyperparameters	95,3%	81,0%	1,01
06	CNN +BatchNormalization	98,0%	84,7%	0,62
07	CNN +Data Augmentation	81,2%	82,5%	0,64
08	CNN +Optimizer Adam	94,8%	79,7%	0,72
09	CNN +Optimizer RMSprop	90,5%	81,7%	0,68
10	CNN +Optimizer Nadam	96,1%	82,9%	0,66
11	CNN +Optimizer SGD	80,9%	19,6%	2,44
12	<b>CNN +BN +DA +Nadam</b> 📌	90,6%	<b>86.3%</b>	0,68
13	Transfer Learning	79,0%	50,4%	1,77
14	Transfer Learning +Congelamento	97,7%	66,8%	1,72
15	Transfer Learning +DA Congelado	49,8%	39,0%	1,87
16	Transfer Learning +DA Descongelado	85,8%	75,7%	1,02

📌 La combinación de la arquitectura CNN y las técnicas BatchNormalization, Data Augmentation y Optimizer Nadam dieron como resultado la mayor precisión (accuracy), alcanzando hasta el 86,3% en el dataset de prueba.

## 18.3. 🧘 Consideraciones Finales

Este trabajo presentó desafíos interesantes que requirieron una considerable investigación y experimentación. Entre ellos, destacan los siguientes:

1. Era necesario encontrar una resolución de imagen ligera, pero suficiente para que la red neuronal identificara con precisión los elementos de interés. Se probaron varias resoluciones diferentes. Cabe destacar que resoluciones superiores a la utilizada en la versión final resultaron en un entrenamiento más lento y un menor rendimiento.
2. La experimentación con diferentes arquitecturas ayudó a demostrar que más capas y más neuronas no necesariamente implican mejores resultados. En cierto modo, este hecho contradice el sentido común.

3. Experimentar con diferentes técnicas por separado fue fundamental para identificar las que más contribuían al objetivo y para seleccionar las más exitosas en el modelo que lograron la precisión deseada.

El objetivo de obtener una precisión mínima del 85% se logró, lo que requirió combinar la arquitectura y las técnicas que ofrecieron el mejor rendimiento durante los experimentos.