UNIVERSIDAD ALFONSO X EL SABIO

Business Tech

Máster Universitario en Inteligencia Artificial

ÁREAS DE APLICACIÓN Y CASOS DE USO: ENTORNOS CON DATOS NO ESTRUCTURADOS

Ejercicio de Feedback 2

Sergi Zarzuelo Abelló

03/05/2025

UAX Universidad Alfonso X el Sabio

Índice general

UAX Universidad Alfonso X el Sabio

CAPÍTULO 1

Introducción y Objetivos

Para el desarrollo de este ejercicio de *Feedback*, se ha proporcionado un conjunto de datos con imágenes de personajes de la serie *Los Simpson*. El objetivo es diseñar y entrenar al menos dos modelos de clasificación que logren una precisión mínima del 90 % en el conjunto de *test*.

Se han explorado diversas estrategias, ajustando tanto el preprocesamiento de los datos como la arquitectura de las redes neuronales. Se han evaluado técnicas como la normalización, regularización, aumento de datos (*data augmentation*) y variaciones en la profundidad de las redes convolucionales (CNN).

Este informe resume los enfoques utilizados, los modelos desarrollados, las métricas obtenidas y un análisis comparativo entre ellos, destacando las decisiones clave que permitieron alcanzar el rendimiento deseado.

Aunque herramientas como *Keras* o *TensorFlow* ofrecen una sintaxis más accesible, en este ejercicio se ha optado por utilizar el *framework* de aprendizaje profundo *PyTorch*, precisamente por su mayor complejidad. Esta decisión responde al objetivo formativo de adquirir una comprensión más profunda del flujo de entrenamiento y evaluación de modelos en dicho entorno.

Al tratarse de una herramienta más exigente a nivel técnico, se ha puesto especial atención en explicar de forma clara tanto la arquitectura implementada como las funciones utilizadas para el entrenamiento y la validación.

UAX Universidad Alfonso X el Sabio

CAPÍTULO 2

Preparación de los datos

En primer lugar, se ha descargado el dataset desde la plataforma *Kaggle*[1], obteniéndose una carpeta con múltiples subcarpetas, cada una correspondiente a un personaje de la serie. Dentro de cada subcarpeta se encuentran imágenes individuales asociadas a dicho personaje.

Posteriormente, el conjunto de datos se ha reorganizado para dividirse en tres subconjuntos principales: **Train** (70 %), **Validation** (15 %) y **Test** (15 %).

Consecuentemente, se definen tres rutas principales en el sistema de archivos: `train_dir`, `val_dir` y `test_dir`, que apuntan a los respectivos subconjuntos, desde donde se cargan los datasets con la función `ImageFolder`.

```
train_dataset = datasets.ImageFolder(train_dir, transform=transform)
val_dataset   = datasets.ImageFolder(val_dir, transform=transform)
test_dataset  = datasets.ImageFolder(test_dir, transform=transform)
```

Listing 2.1: Carga de datasets con `ImageFolder`

Adicionalmente, se han establecido ciertas transformaciones en `transform` que ajustan las imagenes para su utilización en el modelo.

[1] https://www.kaggle.com/datasets/alexattia/the-simpsons-characters-dataset?resource=download&select=simpsons_dataset

Concretamente, las transformaciones aplicadas son:

- **Resize (Necesario)**: redimensiona todas las imágenes a `64×64` píxeles. Las redes convolucionales (CNN) requieren entradas de tamaño fijo, por lo que esta transformación garantiza la consistencia en las dimensiones de entrada.
- **ToTensor (Necesario)**: convierte una imagen en formato *PIL* o *NumPy* a un tensor de PyTorch con forma `(C, H, W)`, y escala automáticamente los valores de píxel del rango `[0, 255]` a `[0, 1]`.
- **Normalize (Opcional)**: normaliza los valores de cada canal de la imagen, una práctica común que ayuda a estabilizar y acelerar el entrenamiento. En este caso, al aplicarse después de `ToTensor()`, transforma el rango `[0, 1]` al rango `[-1, 1]`.

```
transform = transforms.Compose([
    transforms.Resize((64, 64)),          # Redimensiona todas las imagenes a 64
        x64 pixeles.
    transforms.ToTensor(),                # Convierte la imagen en un tensor
        PyTorch y normaliza a [0, 1].
    transforms.Normalize([0.5], [0.5])    # Normaliza los valores de cada canal
        a un rango de [-1, 1].])
```

Listing 2.2: Transformaciones aplicadas a las imágenes

Cada uno de estos *dataset* está compuesto por elementos de la forma `(img_tensor, etiqueta)`, donde `img_tensor` es un tensor tridimensional con forma `(C, H, W)`, siendo:

- `C`: número de canales de la imagen (3 para RGB)
- `H`: altura de la imagen (64)
- `W`: anchura de la imagen (64)

Por entender mejor la estructura de los datos, se analiza un ejemplo en detalle del dataset de entrenamiento:

```
img_tensor, etiqueta = train_dataset[0]
```

Listing 2.3: Ejemplo de elemento en dataset

De esta manera, se tiene por un lado una `etiqueta` numérica que representa la clase a la que pertenece la imagen (por ejemplo, `0` para *abraham_grampa_simpson*), y por otro lado un tensor tridimensional `img_tensor` con dimensiones `(3, 64, 64)`.

- `img_tensor[0]`: matriz `(64, 64)` con la información del canal rojo (R)
- `img_tensor[1]`: matriz `(64, 64)` con la información del canal verde (G)
- `img_tensor[2]`: matriz `(64, 64)` con la información del canal azul (B)

```
Tamano tensor: torch.Size([3, 64, 64])
Etiqueta: 0
Visualizacion numerica de ejemplo del canal rojo:
tensor([[-0.2863, -0.2627, -0.2627, ..., 0.1059, 0.0980, 0.0980],
        [-0.3020, -0.2706, -0.2706, ..., 0.0980, 0.0980, 0.1137],
        ...,
        [ 0.4275, 0.4667, 0.3804, ..., 0.6314, 0.5216, 0.1059],
        [ 0.3647, 0.4667, 0.3098, ..., 0.6784, 0.4745, 0.0980],
        [-0.0667, 0.3725, 0.3020, ..., 0.6863, 0.5294, 0.0039]])
```

Listing 2.4: Visualización numérica del canal rojo de una imagen

Figura 2.1: Ejemplo elemento dataset.

Una vez cargados los conjuntos de datos, utilizar las imágenes de forma individual no resulta eficiente, ya que implicaría acceder a ellas una por una. Para optimizar este proceso, se introduce un paso adicional mediante el uso del objeto `DataLoader`, el cual agrupa las imágenes en bloques o *batches* que se procesan conjuntamente durante el entrenamiento del modelo, mejorando significativamente la eficiencia computacional.

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)  # Se
    barajan los datos en cada epoca para evitar sobreajuste
val_loader   = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader  = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Listing 2.5: Creación de los `DataLoader` para cada conjunto

Una vez generados los batches, se visualiza uno como ejemplo:

Figura 2.2: Ejemplo Batch DataLoader.

UAX Universidad Alfonso X el Sabio

CAPÍTULO 3

Modelo 1 (CNN simple)

En esta sección se desarrollará un modelo sencillo basado en redes neuronales convolucionales (CNN).

3.1 › Entrenamiento inicial del modelo

Se desarrollará una red neuronal convolucional (CNN) básica, estructurada en dos partes principales:

1. `self.conv_layer`: compuesto por **6 capas en total**, que se repiten dos veces:
   - **Conv2D**: capas convolucionales encargadas de extraer patrones visuales como bordes, colores o texturas. La primera transforma una imagen con 3 canales RGB en 32 mapas de características, y la segunda incrementa esta representación de 32 a 64 canales.
   - **ReLU**: funciones de activación no lineales que permiten a la red aprender relaciones complejas, conservando los valores positivos y anulando los negativos.

- **MaxPool2D**: capas de reducción espacial que disminuyen las dimensiones de la imagen a la mitad en cada bloque, reduciendo así la carga computacional y conservando las características más relevantes.

En conjunto, estos bloques procesan una imagen de entrada de tamaño `(3, 64, 64)` y la transforman en un tensor de tamaño `(64, 16, 16)` al finalizar esta sección.

2. `self.fc_layer`: compuesto por **4 capas secuenciales**, encargadas de realizar la clasificación final a partir de las características extraídas:

   - **Flatten**: aplana el tensor de salida `(64, 16, 16)` proveniente del bloque convolucional, convirtiéndolo en un vector de `16.384` elementos para su uso en capas densas.
   - **Linear1 (Fully Connected)**: reduce los `16.384` valores a `128` neuronas intermedias, condensando la información extraída.
   - **ReLU**: se aplica entre las capas densas para introducir no linealidad y mejorar la capacidad de aprendizaje del modelo.
   - **Linear2 (Fully Connected)**: proyecta los `128` valores intermedios al número final de salidas, correspondiente al número de clases del problema (`num_classes`).

   Este bloque toma la representación profunda generada por las convoluciones y la traduce en una predicción de clase.

Una vez definida la arquitectura del modelo (ver Anexo 8.1 para el código completo), se procede con la **instanciacion del modelo**, es decir, la definición de las variables necesarias para su ejecución:

- `num_clases`: número total de categorías que el modelo debe predecir. Este valor corresponde al número de carpetas o clases distintas en el conjunto de datos.
- `criterion` (función de pérdida): se utiliza `CrossEntropyLoss`, una función ampliamente empleada en problemas de clasificación multiclase. Esta mide la discrepancia entre la distribución de probabilidad predicha por el modelo y la distribución real (etiqueta verdadera), penalizando más fuertemente las predicciones incorrectas con alta confianza.
- `optimizer` (optimizador): se emplea un optimizador como `Adam`, que combina las ventajas del descenso del gradiente estocástico y el momentum. Ajusta automáticamente la tasa de aprendizaje para cada parámetro, facilitando una convergencia más rápida y estable durante el entrenamiento.

```
num_classes = len(train_dataset.classes)                # Determina cuantas
    categorias debe predecir el modelo.
model_1 = SimpleCNN(num_classes).to(device)             # Se crea y se envia al
    dispositivo (GPU/CPU)

# Esto ultimo esta pensado para que el modelo pueda ser ejecutado en GPU si esta
     disponible, lo que acelerara el entrenamiento y la inferencia.
# Si no hay GPU, el modelo se ejecutara en la CPU, pero se deja programado de
    esta manera para que sea mas eficiente en caso de que el hardware lo permita
    .

criterion_1 = nn.CrossEntropyLoss()                         # Funcion de perdida
     que combina softmax y cross-entropy en una sola funcion. Ideal para
    clasificacion multiclase.
optimizer_1 = optim.Adam(model_1.parameters(), lr=0.001)    # Optimizador Adam,
    que es una mejora del SGD. Se adapta automaticamente a cada parametro y
    tiene buen rendimiento en la mayoria de los casos.
```

Listing 3.1: Instanciacion del modelo, funcion de perdida y optimizador

Por último, se define la función `train_model` encargada de entrenar el modelo previamente instanciado. Esta función contempla tanto el proceso de entrenamiento utilizando las imágenes del conjunto `train`, como la evaluación del rendimiento del modelo empleando las imágenes del conjunto `val` (ver Anexo 8.2 para el código completo).

Con todo ello, se procede a entrenar el modelo con la arquitectura y las funciones definidas.

```
# Entrena el modelo
modelo_entrenado_1, resumen_1 = train_model(model_1, train_loader, val_loader,
    criterion_1, optimizer_1, epochs=10)
```

Listing 3.2: Ejecucion del entrenamiento del modelo

De manera preliminar, se han ejecutado únicamente 10 *epochs* con el objetivo de observar el comportamiento inicial del modelo. Los resultados muestran una discrepancia significativa entre el *accuracy* del conjunto de entrenamiento y el de validación, lo cual sugiere que el modelo no está generalizando correctamente.

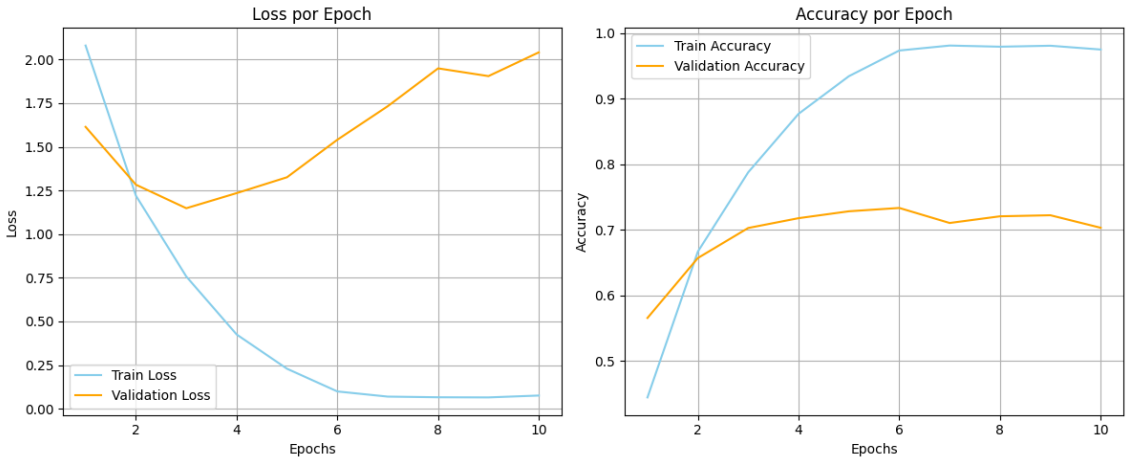

Figura 3.1: Métricas del entrenamiento inicial

Esta diferencia excesiva en el rendimiento puede ser indicativa de un posible *overfitting* durante el entrenamiento, lo cual impide que el modelo generalice adecuadamente al conjunto de validación. Esto puede deberse al desbalanceo que hay entre las distintas clases (personajes) en el dataset, tal y como se profundizará más adelante.

Adicionalmente, se ha evaluado su rendimiento sobre la muestra de test utilizando la función `test_model` (ver Anexo 8.3 para el código completo), obteniéndose un valor de *accuracy* del **70,4 %**.

A continuación, se detallan los resultados obtenidos por clase al evaluar el modelo sobre el conjunto de test, incluyendo métricas como la *precision*, el *recall*, la puntuación *F1* y el *support* (número de imágenes por clase). Los resultados se han ordenado en función del valor de *F1-score*, ya que el objetivo es obtener un modelo lo más equilibrado posible, con un buen compromiso entre precisión y exhaustividad en la clasificación.

✅ Accuracy total en test: 70.40%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
troy_mcclure	1.000000	1.000000	1.000000	2.0
marge_simpson	0.873016	0.850515	0.861619	194.0
krusty_the_clown	0.808511	0.839779	0.823848	181.0
ned_flanders	0.841346	0.799087	0.819672	219.0
milhouse_van_houten	0.780347	0.833333	0.805970	162.0
kent_brockman	0.838235	0.760000	0.797203	75.0
apu_nahasapeemapetilon	0.764706	0.829787	0.795918	94.0
sideshow_bob	0.751773	0.803030	0.776557	132.0
edna_krabappel	0.790323	0.710145	0.748092	69.0
principal_skinner	0.755682	0.738889	0.747191	180.0
abraham_grampa_simpson	0.842593	0.664234	0.742857	137.0
chief_wiggum	0.708075	0.770270	0.737864	148.0
homer_simpson	0.698795	0.688427	0.693572	337.0
mayor_quimby	0.814815	0.594595	0.687500	37.0
moe_szyslak	0.566038	0.825688	0.671642	218.0
comic_book_guy	0.653333	0.690141	0.671233	71.0
carl_carlson	0.750000	0.600000	0.666667	15.0
charles_montgomery_burns	0.744526	0.569832	0.645570	179.0
selma_bouvier	0.888889	0.500000	0.640000	16.0
martin_prince	0.750000	0.545455	0.631579	11.0
groundskeeper_willie	0.520000	0.684211	0.590909	19.0
bart_simpson	0.791667	0.470297	0.590062	202.0
lisa_simpson	0.491803	0.735294	0.589391	204.0
rainier_wolfcastle	0.500000	0.714286	0.588235	7.0
patty_bouvier	0.600000	0.545455	0.571429	11.0
snake_jailbird	0.500000	0.555556	0.526316	9.0
miss_hoover	1.000000	0.333333	0.500000	3.0
nelson_muntz	0.791667	0.351852	0.487179	54.0
waylon_smithers	0.481481	0.464286	0.472727	28.0
professor_john_frink	1.000000	0.300000	0.461538	10.0
gil	0.500000	0.400000	0.444444	5.0
lenny_leonard	0.321839	0.595745	0.417910	47.0
agnes_skinner	0.666667	0.285714	0.400000	7.0
cletus_spuckler	1.000000	0.250000	0.400000	8.0
maggie_simpson	0.714286	0.250000	0.370370	20.0
barney_gumble	0.352941	0.375000	0.363636	16.0
fat_tony	1.000000	0.200000	0.333333	5.0
sideshow_mel	1.000000	0.166667	0.285714	6.0
ralph_wiggum	0.500000	0.142857	0.222222	14.0
disco_stu	0.000000	0.000000	0.000000	2.0
otto_mann	0.000000	0.000000	0.000000	5.0

Figura 3.2: Resultados por clases en test

Como puede observarse, las clases con peores resultados corresponden a aquellas con muy poca representatividad en el conjunto de datos, como por ejemplo *professor_john_frink*, que cuenta con únicamente 10 imágenes en el conjunto de test.

De hecho, se identifican ciertos casos, como *gil*, *disco_stu* o *troy_mcclure*, en los que todas las métricas aparecen con valor cero. Esto se debe a que el modelo no ha seleccionado en ninguna ocasión dichas clases como predicción, lo cual impide calcular medidas como precisión o *recall*.

Adicionalmente, se evalúa el poder predictivo del modelo mediante la siguiente matriz de confusión.

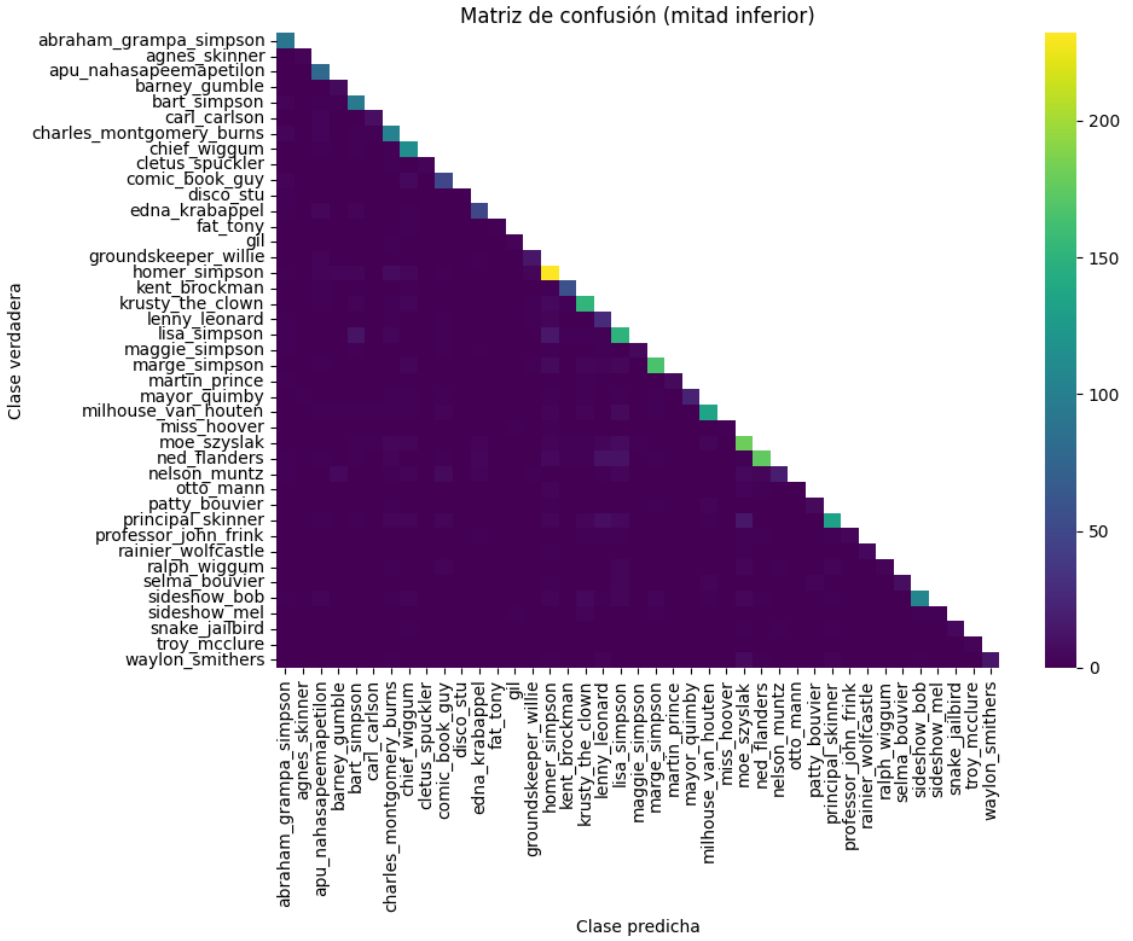

Figura 3.3: Matriz de confusión por clases en test

En ella se observan clases con un rendimiento notable, como es el caso de *homer_simpson* o *ned_flanders*. No obstante, también se evidencian clases en las que el modelo presenta confusiones frecuentes, como ocurre con *lisa_simpson*, que suele ser clasificada erróneamente como *bart_simpson*. Esta confusión podría deberse a que ambos personajes aparecen juntos en múltiples imágenes del conjunto de datos.

Todo ello indica que el modelo no está funcionando correctamente debido al desblanceo de las clases, tal y como se había anticipado anteriormente. En consecuencia, en las próximas secciones se exploran distintos ajustes para solucionar el problema.

3.2 › Ajuste por desbalanceo

En esta sección se exploran distintos ajustes y metodologías orientadas a mitigar el problema de desbalanceo presente en el conjunto de datos, donde se observa que

algunas clases cuentan con un número muy reducido de imágenes, lo que dificulta que el modelo pueda aprender correctamente sus patrones.

Si bien los resultados obtenidos podrían variar al emplear un mayor número de *epochs*, el análisis comparativo se ha llevado a cabo utilizando únicamente 10, por cuestiones de tiempo, memoria y eficiencia computacional.

Una vez seleccionada la estrategia más adecuada para abordar el desbalanceo, esta se integrará en el modelo final, el cual será entrenado durante un mayor número de *epochs*, permitiendo así que el modelo disponga del tiempo necesario para aprender de forma más efectiva.

3.2.1 › Data Augmentation dinámico

Para la primera prueba, además de las transformaciones utilizadas anteriormente, se incorporan técnicas adicionales de *Data Augmentation*, con el objetivo de aumentar la variabilidad de las imágenes de entrenamiento y mejorar la capacidad de generalización del modelo.

Las transformaciones aplicadas son las siguientes:

- **RandomHorizontalFlip**: invierte horizontalmente las imágenes de forma aleatoria. Se aplica únicamente durante el entrenamiento.
- **RandomRotation**: rota aleatoriamente las imágenes hasta un ángulo máximo especificado. Se aplica únicamente durante el entrenamiento.
- **ColorJitter**: modifica aleatoriamente el brillo, contraste y tonalidad de las imágenes, generando variaciones en los colores. Se aplica únicamente durante el entrenamiento.

Es importante destacar que estas transformaciones se aplican exclusivamente sobre la muestra de entrenamiento, ya que el *Data Augmentation* no debe utilizarse en los conjuntos de validación ni test, con el fin de evaluar el rendimiento del modelo sobre datos no alterados.

Cabe señalar que, al aplicar estas transformaciones, no se incrementa el tamaño del conjunto de datos de entrenamiento, de ahí el nombre de *dinámico*. En su lugar, cada vez que el modelo accede a una imagen durante el entrenamiento, se le aplica una transformación aleatoria distinta. Esto permite generar múltiples versiones de una misma imagen a lo largo de las *epochs*, mejorando la robustez del modelo sin necesidad de almacenar imágenes adicionales.

Con este ajuste, y manteniendo tanto la función de pérdida como el optimizador definidos inicialmente, el modelo presenta una mejora significativa. Se observa un

incremento del *accuracy* tanto en el conjunto de entrenamiento como en el de validación, además de una menor diferencia entre ambos, lo que indica una reducción del *overfitting*.

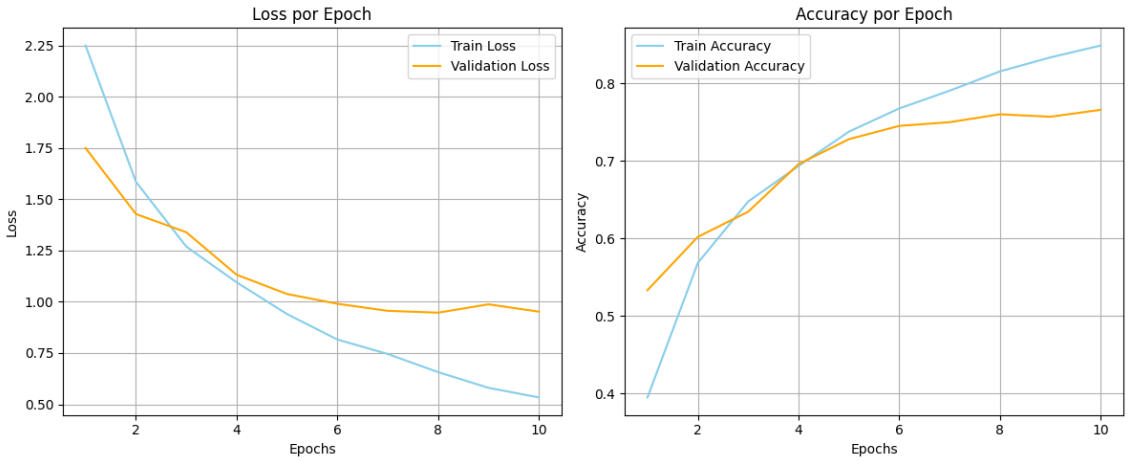

Figura 3.4: Métricas del entrenamiento con Data Augmentation dinámico

Asimismo, el rendimiento sobre el conjunto de test también mejora, alcanzando una exactitud final del **76,6 %**, frente al **70,4 %** obtenido previamente.

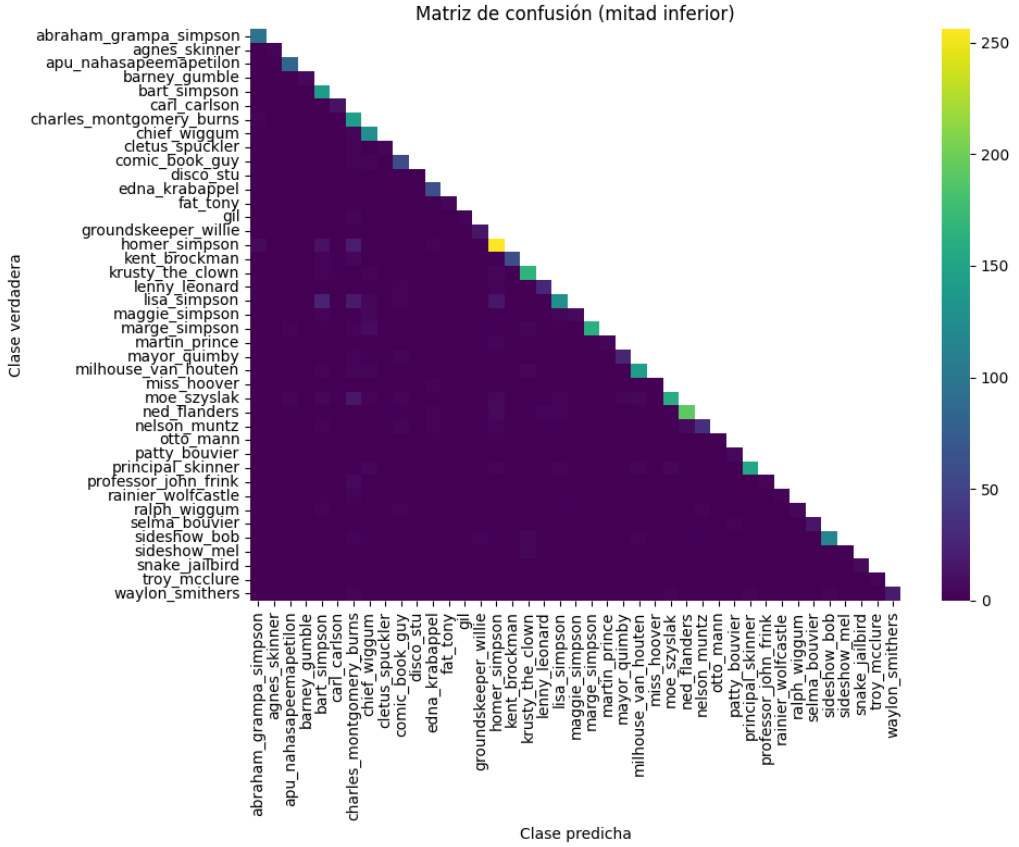

Figura 3.5: Matriz de confusión por clases en test

✅ Accuracy total en test: 76.61%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
troy_mcclure	1.000000	1.000000	1.000000	2.0
marge_simpson	0.941860	0.835052	0.885246	194.0
apu_nahasapeemapetilon	0.881720	0.872340	0.877005	94.0
sideshow_bob	0.877863	0.871212	0.874525	132.0
krusty_the_clown	0.816832	0.911602	0.861619	181.0
principal_skinner	0.869318	0.850000	0.859551	180.0
kent_brockman	0.936508	0.786667	0.855072	75.0
ned_flanders	0.813559	0.876712	0.843956	219.0
edna_krabappel	0.779221	0.869565	0.821918	69.0
milhouse_van_houten	0.755319	0.876543	0.811429	162.0
selma_bouvier	0.764706	0.812500	0.787879	16.0
moe_szyslak	0.854839	0.729358	0.787129	218.0
chief_wiggum	0.728324	0.851351	0.785047	148.0
abraham_grampa_simpson	0.850877	0.708029	0.772908	137.0
homer_simpson	0.748538	0.759644	0.754050	337.0
comic_book_guy	0.640449	0.802817	0.712500	71.0
groundskeeper_willie	0.666667	0.736842	0.700000	19.0
bart_simpson	0.695000	0.688119	0.691542	202.0
carl_carlson	0.647059	0.733333	0.687500	15.0
charles_montgomery_burns	0.592437	0.787709	0.676259	179.0
fat_tony	0.750000	0.600000	0.666667	5.0
mayor_quimby	0.657895	0.675676	0.666667	37.0
lenny_leonard	0.771429	0.574468	0.658537	47.0
lisa_simpson	0.686486	0.622549	0.652956	204.0
snake_jailbird	0.545455	0.666667	0.600000	9.0
waylon_smithers	0.562500	0.642857	0.600000	28.0
nelson_muntz	0.603774	0.592593	0.598131	54.0
patty_bouvier	0.555556	0.454545	0.500000	11.0
miss_hoover	1.000000	0.333333	0.500000	3.0
cletus_spuckler	0.600000	0.375000	0.461538	8.0
barney_gumble	0.833333	0.312500	0.454545	16.0
agnes_skinner	1.000000	0.285714	0.444444	7.0
martin_prince	0.750000	0.272727	0.400000	11.0
gil	0.400000	0.400000	0.400000	5.0
rainier_wolfcastle	0.666667	0.285714	0.400000	7.0
ralph_wiggum	0.500000	0.285714	0.363636	14.0
maggie_simpson	0.428571	0.300000	0.352941	20.0
professor_john_frink	0.500000	0.200000	0.285714	10.0
otto_mann	0.500000	0.200000	0.285714	5.0
sideshow_mel	1.000000	0.166667	0.285714	6.0
disco_stu	0.000000	0.000000	0.000000	2.0

Figura 3.6: Métricas del testing con Data Augmentation dinámico

3.2.2 › Data Augmentation estático

Para esta prueba, en lugar de aplicar el *data augmentation* de manera dinámica, se opta por una estrategia estática. Es decir, en lugar de transformar aleatoriamente una misma imagen cada vez que es procesada por el modelo, se generan copias adicionales de las imágenes pertenecientes a las clases con menor representatividad (hasta llegar a 200 imagenes mínimo por clase, como umbral de prueba), aplicándoles transformaciones previamente definidas.

En otras palabras, en esta ocasión sí se incrementa explícitamente el número de imágenes en el conjunto de entrenamiento, con el objetivo de equilibrar la distribución de clases y mejorar la capacidad del modelo para aprender patrones en categorías con pocos ejemplos.

Manteniendo tanto la función de pérdida como el optimizador definidos inicialmente, los resultados de esta alternativa son peores que los obtenidos con el enfoque dinámico, encontrándose overfitting en el entrenamiento y un valor más bajo de accuracy en el testing final.

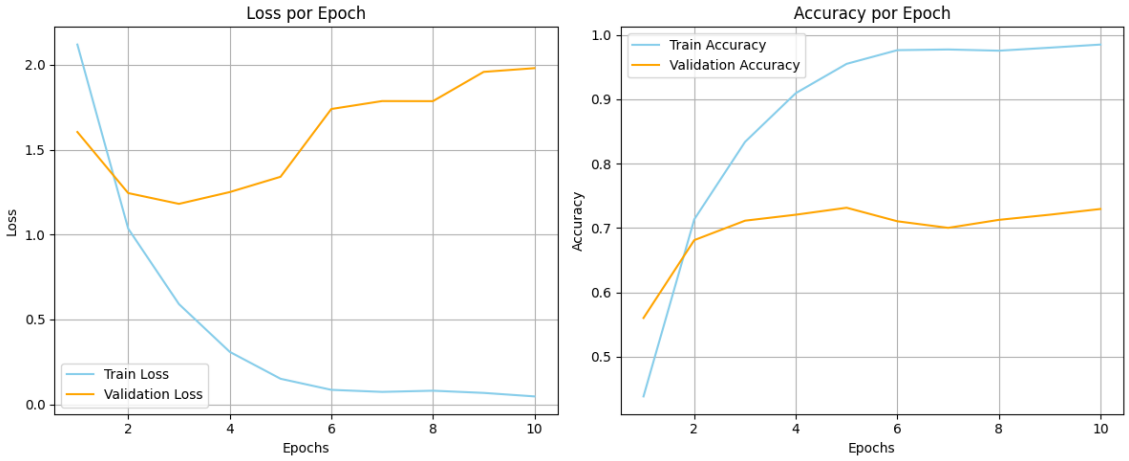

Figura 3.7: Métricas del entrenamiento con Data Augmentation estático

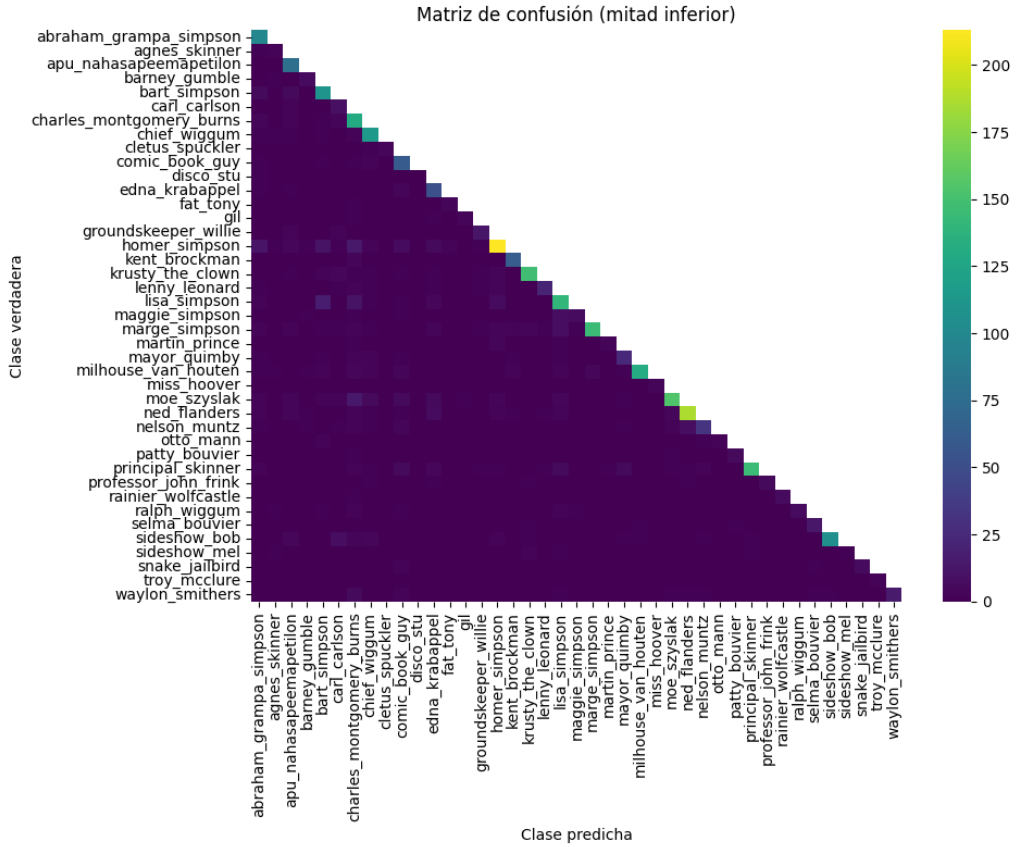

Figura 3.8: Matriz de confusión por clases en test

✅ Accuracy total en test: 71.57%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
troy_mcclure	1.000000	1.000000	1.000000	2.0
miss_hoover	1.000000	1.000000	1.000000	3.0
rainier_wolfcastle	0.857143	0.857143	0.857143	7.0
krusty_the_clown	0.886228	0.817680	0.850575	181.0
milhouse_van_houten	0.879195	0.808642	0.842444	162.0
sideshow_bob	0.876033	0.803030	0.837945	132.0
marge_simpson	0.935484	0.747423	0.830946	194.0
kent_brockman	0.775000	0.826667	0.800000	75.0
ned_flanders	0.697761	0.853881	0.767967	219.0
chief_wiggum	0.765101	0.770270	0.767677	148.0
principal_skinner	0.732323	0.805556	0.767196	180.0
apu_nahasapeemapetilon	0.706422	0.819149	0.758621	94.0
snake_jailbird	0.857143	0.666667	0.750000	9.0
abraham_grampa_simpson	0.722628	0.722628	0.722628	137.0
moe_szyslak	0.729858	0.706422	0.717949	218.0
homer_simpson	0.825581	0.632047	0.715966	337.0
comic_book_guy	0.560748	0.845070	0.674157	71.0
gil	0.750000	0.600000	0.666667	5.0
cletus_spuckler	1.000000	0.500000	0.666667	8.0
edna_krabappel	0.577778	0.753623	0.654088	69.0
charles_montgomery_burns	0.581081	0.720670	0.643392	179.0
lisa_simpson	0.588235	0.686275	0.633484	204.0
bart_simpson	0.723684	0.544554	0.621469	202.0
selma_bouvier	0.481481	0.812500	0.604651	16.0
fat_tony	0.600000	0.600000	0.600000	5.0
groundskeeper_willie	0.571429	0.631579	0.600000	19.0
mayor_quimby	0.545455	0.648649	0.592593	37.0
patty_bouvier	0.833333	0.454545	0.588235	11.0
otto_mann	1.000000	0.400000	0.571429	5.0
nelson_muntz	0.533333	0.592593	0.561404	54.0
lenny_leonard	0.724138	0.446809	0.552632	47.0
ralph_wiggum	0.750000	0.428571	0.545455	14.0
waylon_smithers	0.517241	0.535714	0.526316	28.0
professor_john_frink	0.500000	0.500000	0.500000	10.0
barney_gumble	0.500000	0.375000	0.428571	16.0
carl_carlson	0.264706	0.600000	0.367347	15.0
martin_prince	0.500000	0.272727	0.352941	11.0
maggie_simpson	0.400000	0.300000	0.342857	20.0
agnes_skinner	0.250000	0.285714	0.266667	7.0
sideshow_mel	0.200000	0.166667	0.181818	6.0
disco_stu	0.000000	0.000000	0.000000	2.0

Figura 3.9: Métricas del testing con Data Augmentation estático

3.2.3 › Data Augmentation dinámico + CrossEntropy balanceado

Se concluye que el enfoque de *Data Augmentation* dinámico proporciona mejores resultados en comparación con la estrategia estática.

Como último análisis, se evalúa si esta opción puede mejorarse aún más mediante el ajuste del criterio de pérdida utilizado. En concreto, se mantiene la función de pérdida `CrossEntropyLoss`, pero se incorpora un vector de pesos para penalizar más los errores en clases minoritarias (`class_weight='balanced'`), con el objetivo de compensar el desbalanceo presente en los datos durante el entrenamiento.

No obstante, los resultados obtenidos son peores que aplicando *Data Augmentation* dinámico de manera individual.

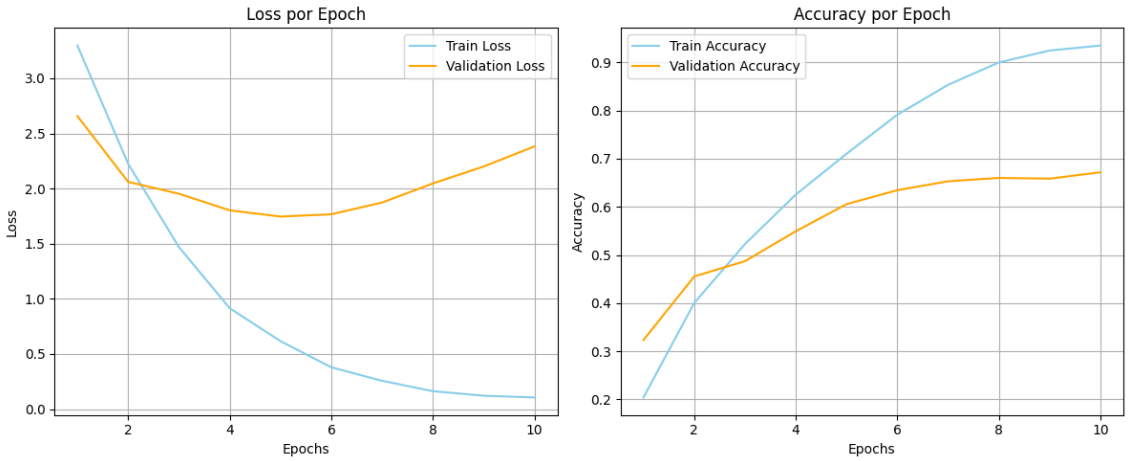

Figura 3.10: Métricas del entrenamiento con Data Augmentation dinámico y CrossEntropy balanceado

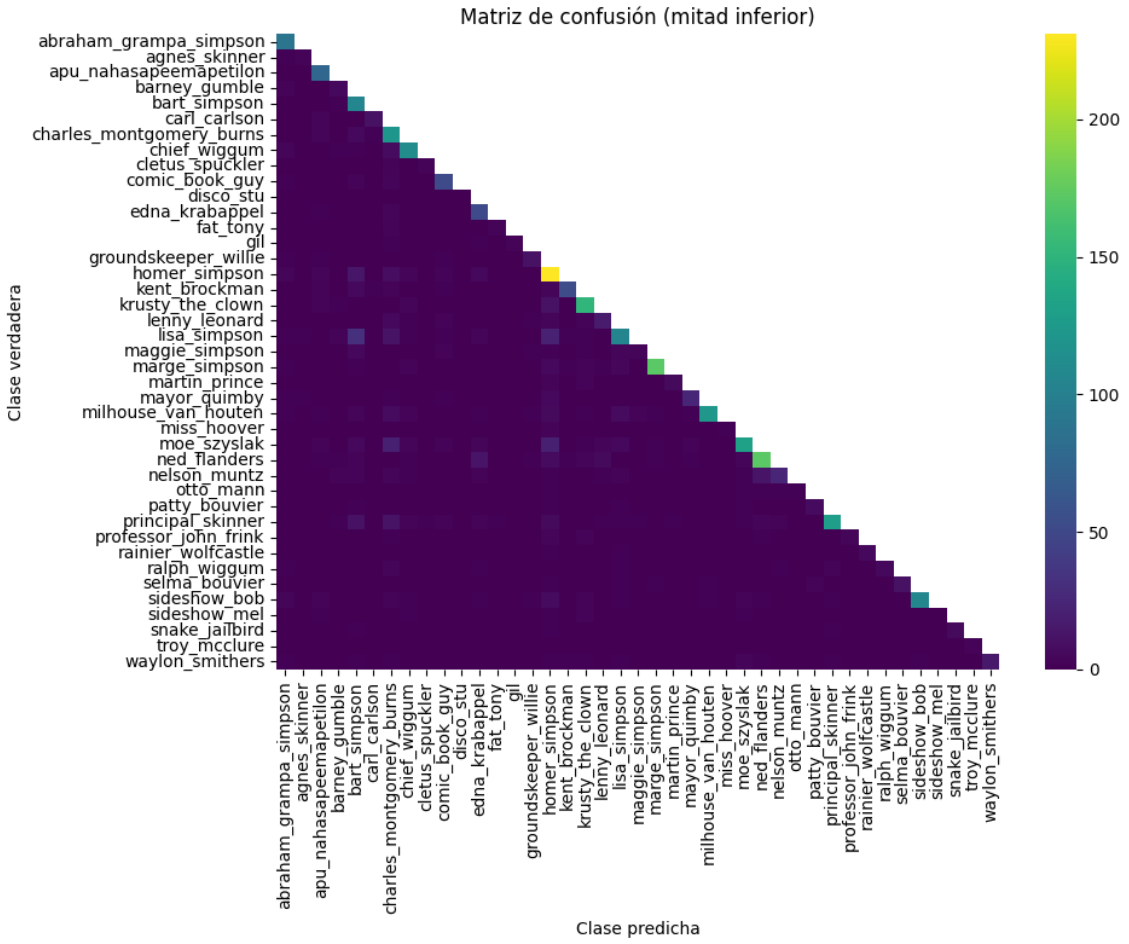

Figura 3.11: Matriz de confusión por clases en test

✅ Accuracy total en test: 67.87%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
troy_mcclure	1.000000	1.000000	1.000000	2.0
marge_simpson	0.873096	0.886598	0.879795	194.0
sideshow_bob	0.856000	0.810606	0.832685	132.0
krusty_the_clown	0.751244	0.834254	0.790576	181.0
apu_nahasapeemapetilon	0.765306	0.797872	0.781250	94.0
milhouse_van_houten	0.792208	0.753086	0.772152	162.0
rainier_wolfcastle	0.833333	0.714286	0.769231	7.0
chief_wiggum	0.770833	0.750000	0.760274	148.0
kent_brockman	0.815385	0.706667	0.757143	75.0
ned_flanders	0.704918	0.785388	0.742981	219.0
comic_book_guy	0.753623	0.732394	0.742857	71.0
abraham_grampa_simpson	0.818182	0.656934	0.728745	137.0
selma_bouvier	0.833333	0.625000	0.714286	16.0
snake_jailbird	0.750000	0.666667	0.705882	9.0
principal_skinner	0.689840	0.716667	0.702997	180.0
carl_carlson	0.714286	0.666667	0.689655	15.0
moe_szyslak	0.785714	0.605505	0.683938	218.0
edna_krabappel	0.565217	0.753623	0.645963	69.0
patty_bouvier	0.636364	0.636364	0.636364	11.0
mayor_quimby	0.595238	0.675676	0.632911	37.0
martin_prince	0.750000	0.545455	0.631579	11.0
homer_simpson	0.556627	0.685460	0.614362	337.0
waylon_smithers	0.714286	0.535714	0.612245	28.0
charles_montgomery_burns	0.542601	0.675978	0.601990	179.0
gil	1.000000	0.400000	0.571429	5.0
lisa_simpson	0.567568	0.514706	0.539846	204.0
bart_simpson	0.535354	0.524752	0.530000	202.0
groundskeeper_willie	0.526316	0.526316	0.526316	19.0
fat_tony	0.666667	0.400000	0.500000	5.0
nelson_muntz	0.510638	0.444444	0.475248	54.0
ralph_wiggum	0.454545	0.357143	0.400000	14.0
miss_hoover	0.500000	0.333333	0.400000	3.0
professor_john_frink	0.600000	0.300000	0.400000	10.0
lenny_leonard	0.425000	0.361702	0.390805	47.0
agnes_skinner	0.500000	0.285714	0.363636	7.0
cletus_spuckler	0.666667	0.250000	0.363636	8.0
barney_gumble	0.357143	0.312500	0.333333	16.0
otto_mann	1.000000	0.200000	0.333333	5.0
maggie_simpson	0.333333	0.150000	0.206897	20.0
disco_stu	0.000000	0.000000	0.000000	2.0
sideshow_mel	0.000000	0.000000	0.000000	6.0

Figura 3.12: Métricas del testing con Data Augmentation dinámico y CrossEntropy balanceado

En base a los resultados obtenidos, se concluye que el primer ajuste, basado en *Data Augmentation* dinámico, representa la opción más eficaz para abordar el desbalanceo de clases.

Además de su efectividad, destaca por su simplicidad de implementación e integración en el flujo de entrenamiento del modelo, por lo que se adopta como solución definitiva para las siguientes evaluaciones.

3.3 › Entrenamiento final del modelo

Una vez seleccionado el ajuste más adecuado, se procede a ejecutar nuevamente el modelo, esta vez incrementando el número de *epochs* a 50, con el objetivo de evaluar el rendimiento máximo que puede alcanzar.

El propósito de esta ejecución final es comprobar si el modelo es capaz de alcanzar una *accuracy* del **90 %**, umbral fijado como objetivo principal en este trabajo.

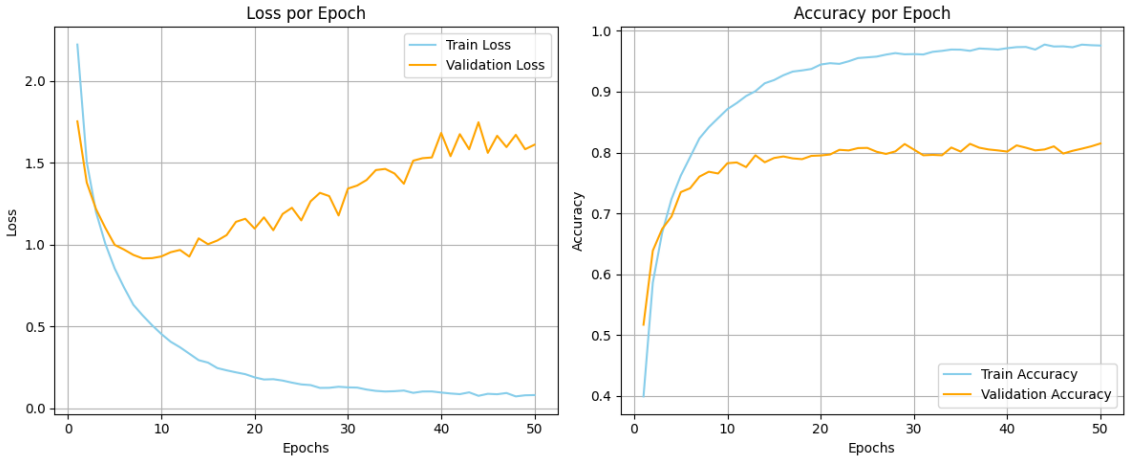

Figura 3.13: Métricas del entrenamiento con Data Augmentation dinámico y 50 *epochs*

Con el ajuste implementado y tras haber incrementado el número de *epochs*, se observa una mejora en la *accuracy* sobre el conjunto de validación. No obstante, persiste una diferencia notable respecto al conjunto de entrenamiento, lo que sugiere la presencia de cierto *overfitting*.

Este comportamiento indica que, a pesar de haber mitigado el desbalanceo de clases, el modelo continúa sobreajustándose durante el entrenamiento. Como posible línea de mejora, se propone modificar la arquitectura de la red convolucional, incorporando capas de `Dropout` para reducir el sobreajuste, o bien aumentar la profundidad de la red mediante capas convolucionales adicionales, lo que permitiría extraer patrones más complejos y mejorar la capacidad del modelo para diferenciar adecuadamente entre todas las clases.

✅ Accuracy total en test: 81.92%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
marge_simpson	0.872642	0.953608	0.911330	194.0
sideshow_bob	0.884058	0.924242	0.903704	132.0
krusty_the_clown	0.909605	0.889503	0.899441	181.0
apu_nahasapeemapetilon	0.903226	0.893617	0.898396	94.0
milhouse_van_houten	0.910256	0.876543	0.893082	162.0
ned_flanders	0.858369	0.913242	0.884956	219.0
kent_brockman	0.822785	0.866667	0.844156	75.0
edna_krabappel	0.929825	0.768116	0.841270	69.0
principal_skinner	0.780952	0.911111	0.841026	180.0
homer_simpson	0.807163	0.869436	0.837143	337.0
comic_book_guy	0.840580	0.816901	0.828571	71.0
chief_wiggum	0.850000	0.804054	0.826389	148.0
lenny_leonard	0.897436	0.744681	0.813953	47.0
abraham_grampa_simpson	0.825758	0.795620	0.810409	137.0
moe_szyslak	0.822115	0.784404	0.802817	218.0
groundskeeper_willie	0.761905	0.842105	0.800000	19.0
troy_mcclure	0.666667	1.000000	0.800000	2.0
lisa_simpson	0.804233	0.745098	0.773537	204.0
selma_bouvier	0.846154	0.687500	0.758621	16.0
fat_tony	1.000000	0.600000	0.750000	5.0
nelson_muntz	0.822222	0.685185	0.747475	54.0
charles_montgomery_burns	0.688679	0.815642	0.746803	179.0
bart_simpson	0.755102	0.732673	0.743719	202.0
mayor_quimby	0.717949	0.756757	0.736842	37.0
carl_carlson	0.900000	0.600000	0.720000	15.0
rainier_wolfcastle	0.714286	0.714286	0.714286	7.0
patty_bouvier	0.777778	0.636364	0.700000	11.0
barney_gumble	0.687500	0.687500	0.687500	16.0
waylon_smithers	0.739130	0.607143	0.666667	28.0
snake_jailbird	0.545455	0.666667	0.600000	9.0
miss_hoover	1.000000	0.333333	0.500000	3.0
otto_mann	0.666667	0.400000	0.500000	5.0
maggie_simpson	0.615385	0.400000	0.484848	20.0
martin_prince	0.666667	0.363636	0.470588	11.0
sideshow_mel	0.428571	0.500000	0.461538	6.0
gil	0.500000	0.400000	0.444444	5.0
cletus_spuckler	0.500000	0.375000	0.428571	8.0
agnes_skinner	0.500000	0.285714	0.363636	7.0
professor_john_frink	0.285714	0.200000	0.235294	10.0
ralph_wiggum	0.400000	0.142857	0.210526	14.0
disco_stu	0.000000	0.000000	0.000000	2.0

Figura 3.14: Métricas del testing con Data Augmentation dinámico y 50 *epochs*

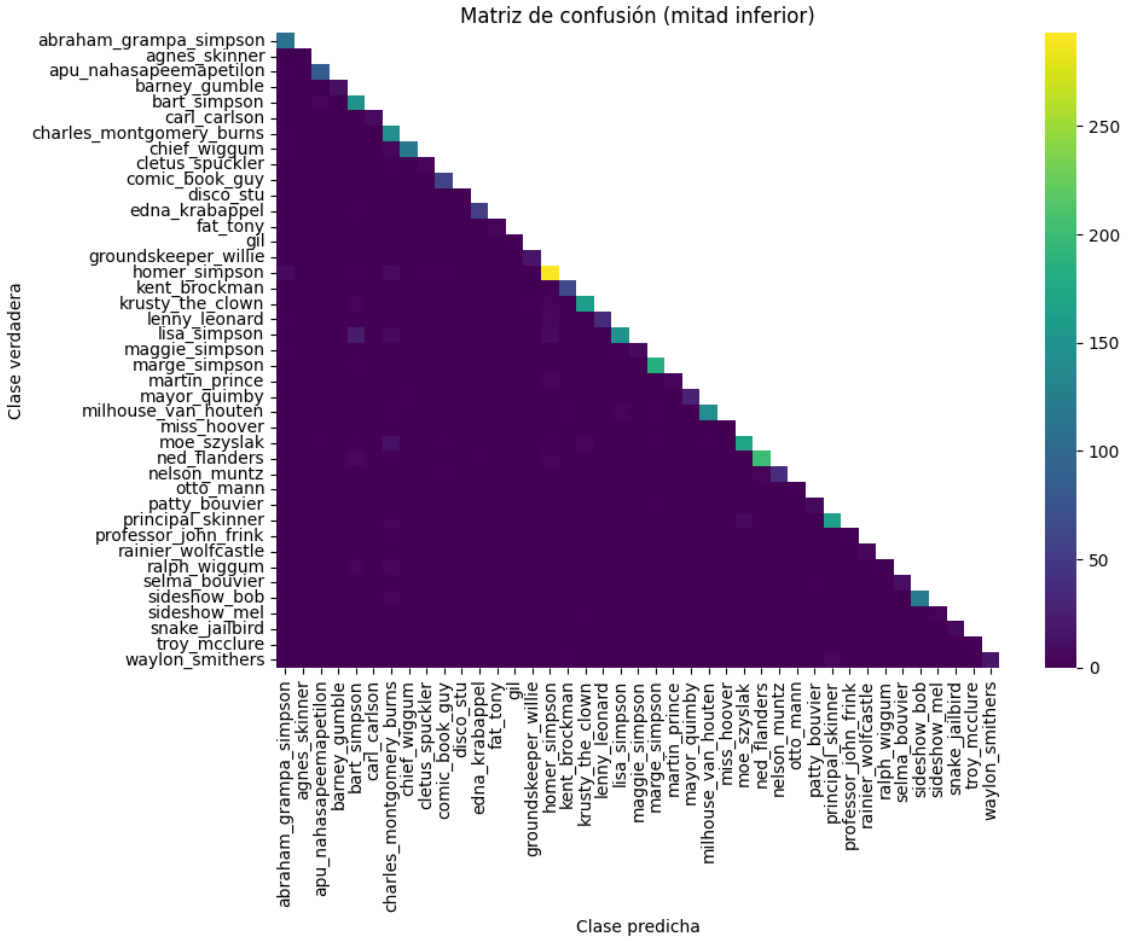

Figura 3.15: Matriz de confusión por clases en test

Por último, se observa que la *accuracy* en el conjunto de test ha aumentado desde un valor inicial del **70,4 %** hasta alcanzar aproximadamente un **81,9 %** en la versión final del modelo.

Si bien esta mejora es significativa, aún se encuentra por debajo del umbral objetivo del **90 %**, lo que sugiere que existen márgenes de mejora adicionales, ya sea a través de ajustes en la arquitectura del modelo, técnicas de regularización o estrategias avanzadas de preprocesamiento, tal y como se explorará en las próximas secciones.

UAX Universidad Alfonso X el Sabio

CAPÍTULO 4

Modelo 2 (CNN medio)

Partiendo de la estructura del modelo previamente definido, en esta etapa se explora una arquitectura más avanzada mediante el incremento del número de capas, con el objetivo de evaluar si dicho ajuste permite mejorar el rendimiento del modelo.

Se mantiene tanto la función de pérdida como el optimizador utilizados anteriormente, de modo que cualquier mejora observada pueda atribuirse directamente a los cambios en la arquitectura de la red.

4.1 › Entrenamiento del modelo

Se desarrolla una red neuronal convolucional (CNN) más profunda y robusta que las exploradas anteriormente, con una arquitectura dividida en dos bloques funcionales principales:

1. `self.features`: compuesto por **9 capas en total**, organizadas en **3 bloques convolucionales** secuenciales:
    - **Conv2D**: capas convolucionales encargadas de extraer patrones visuales relevantes. La primera transforma 3 canales RGB en 32 mapas de

características, la segunda de 32 a 64, y la tercera de 64 a 128. Todas emplean filtros de tamaño 3×3 con *padding* para mantener las dimensiones espaciales.

- **BatchNorm2D**: se aplica tras las dos primeras convoluciones para estabilizar y acelerar el entrenamiento, reduciendo la variación interna de las activaciones.
- **ReLU**: funciones de activación no lineales que anulan los valores negativos y mantienen los positivos, introduciendo no linealidad al modelo.
- **MaxPool2D**: capas de *pooling* que reducen las dimensiones espaciales a la mitad después de los dos primeros bloques, preservando las características más significativas.
- **AdaptiveAvgPool2D**: ajusta dinámicamente el tamaño de salida a un tensor fijo de tamaño (4, 4), permitiendo que la arquitectura sea adaptable a distintas resoluciones de entrada.

Al finalizar esta etapa, una imagen de entrada de tamaño (3, 64, 64) se transforma en un tensor de tamaño fijo (128, 4, 4).

2. `self.classifier`: compuesto por **5 capas secuenciales** que realizan la clasificación final:

   - **Flatten**: aplana el tensor (128, 4, 4) en un vector de 2048 elementos para permitir su procesamiento por capas densas.
   - **Linear1 (fully connected)**: reduce los 2048 valores a 256 neuronas intermedias, condensando la información extraída.
   - **ReLU**: introduce no linealidad en las capas densas para mejorar la capacidad de aprendizaje del modelo.
   - **Dropout**: desactiva aleatoriamente el 50 % de las neuronas durante el entrenamiento para reducir el riesgo de sobreajuste.
   - **Linear2 (fully connected)**: transforma las 256 neuronas intermedias en una salida de dimensión igual al número de clases del problema (`num_classes`).

Este modelo resulta considerablemente más potente que la `SimpleCNN`, gracias a su mayor profundidad, la inclusión de técnicas de normalización y regularización, y una mejor adaptación al tamaño de entrada. Estas características lo hacen especialmente adecuado para trabajar con conjuntos de datos complejos o desbalanceados (ver Anexo 8.4 para el código completo).

Adicionalmente, se incrementa el número de *epochs* a 100, dado que en ejecuciones previas con 50 épocas se observó que el modelo aún continuaba mejorando ligeramente su rendimiento.

Como resultado de estos ajustes, se obtienen métricas notablemente superiores tanto en el conjunto de entrenamiento como en el de validación. En particular, se evidencia una reducción significativa del *overfitting* presente en la versión anterior del modelo, atribuible a la incorporación de capas `Dropout`.

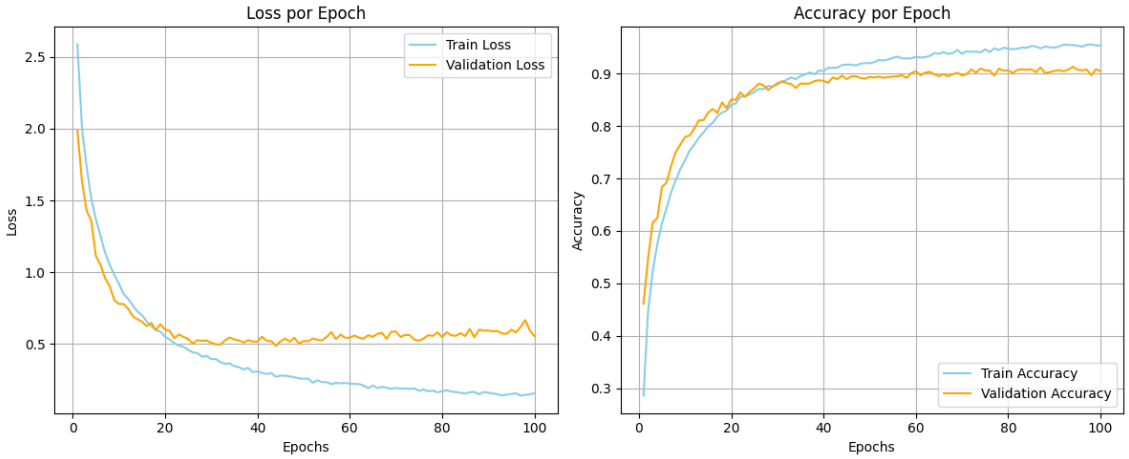

Figura 4.1: Métricas del entrenamiento para la nueva arquitectura y 100 *epochs*

Sin embargo, lo verdaderamente relevante es que la mejora observada durante el entrenamiento también se refleja en el rendimiento sobre el conjunto de test, donde el modelo alcanza una *accuracy* del **90,3 %**.

Este resultado confirma la capacidad del modelo para generalizar correctamente a datos no vistos, cumpliendo así con el objetivo planteado al inicio del proyecto.

✅ Accuracy total en test: 90.25%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
troy_mcclure	1.000000	1.000000	1.000000	2.0
marge_simpson	0.945545	0.984536	0.964646	194.0
sideshow_bob	0.954545	0.954545	0.954545	132.0
principal_skinner	0.945355	0.961111	0.953168	180.0
krusty_the_clown	0.935829	0.966851	0.951087	181.0
milhouse_van_houten	0.962025	0.938272	0.950000	162.0
groundskeeper_willie	0.947368	0.947368	0.947368	19.0
apu_nahasapeemapetilon	0.927083	0.946809	0.936842	94.0
ned_flanders	0.920354	0.949772	0.934831	219.0
kent_brockman	0.932432	0.920000	0.926174	75.0
chief_wiggum	0.902597	0.939189	0.920530	148.0
lisa_simpson	0.947917	0.892157	0.919192	204.0
waylon_smithers	0.960000	0.857143	0.905660	28.0
abraham_grampa_simpson	0.916667	0.883212	0.899628	137.0
homer_simpson	0.884058	0.905045	0.894428	337.0
lenny_leonard	0.893617	0.893617	0.893617	47.0
charles_montgomery_burns	0.875676	0.905028	0.890110	179.0
bart_simpson	0.874396	0.896040	0.885086	202.0
moe_szyslak	0.849785	0.908257	0.878049	218.0
edna_krabappel	0.820513	0.927536	0.870748	69.0
mayor_quimby	0.906250	0.783784	0.840580	37.0
comic_book_guy	0.797468	0.887324	0.840000	71.0
nelson_muntz	0.783333	0.870370	0.824561	54.0
selma_bouvier	0.846154	0.687500	0.758621	16.0
fat_tony	1.000000	0.600000	0.750000	5.0
gil	1.000000	0.600000	0.750000	5.0
professor_john_frink	0.777778	0.700000	0.736842	10.0
agnes_skinner	1.000000	0.571429	0.727273	7.0
martin_prince	1.000000	0.545455	0.705882	11.0
snake_jailbird	0.750000	0.666667	0.705882	9.0
carl_carlson	0.818182	0.600000	0.692308	15.0
rainier_wolfcastle	0.625000	0.714286	0.666667	7.0
sideshow_mel	1.000000	0.500000	0.666667	6.0
patty_bouvier	0.700000	0.636364	0.666667	11.0
ralph_wiggum	1.000000	0.500000	0.666667	14.0
cletus_spuckler	0.800000	0.500000	0.615385	8.0
barney_gumble	0.727273	0.500000	0.592593	16.0
maggie_simpson	0.777778	0.350000	0.482759	20.0
otto_mann	1.000000	0.200000	0.333333	5.0
disco_stu	0.000000	0.000000	0.000000	2.0
miss_hoover	0.000000	0.000000	0.000000	3.0

Figura 4.2: Métricas del testing para la nueva arquitectura y 100 *epochs*

No obstante, se siguen identificando ciertas clases, como *disco_ stu* o *troy_ mcclure*, para las cuales el modelo no realiza ninguna predicción, a pesar de haber aplicado estrategias para corregir el desbalanceo y de haber empleado una arquitectura más avanzada.

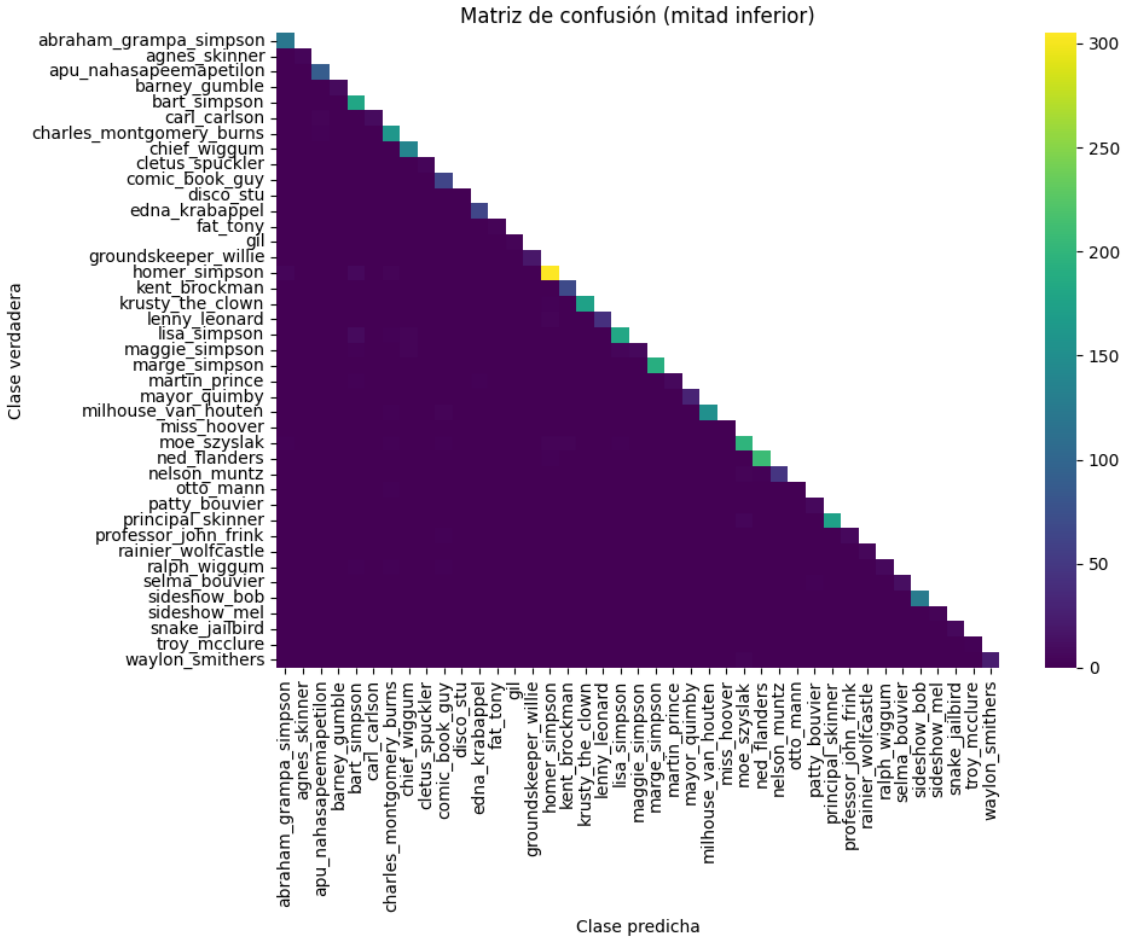

Figura 4.3: Matriz de confusión por clases en test

Adicionalmente, se puede observar como el modelo sigue confundiendo clases como *homer_simpson* o *bart_simpson* con *lisa_simpson*, posiblemente porque estos personajes se encuentren juntos en varias imágenes. No obstante, se puede observar que la matriz otorga mejores resultados respecto a la primera analizada en 3.3, con una diagonal más poblada.

Con todo ello, se explora un último modelo más avanzado, con un número aún mayor de capas, con el objetivo de evaluar si de esta manera se puede abordar correctamente las clases más problemáticas que siguen sin ser reconocidas por el modelo.

CAPÍTULO 5

Modelo 3 (CNN avanzado)

5.1 › Entrenamiento del modelo

Por último, se desarrolla una red neuronal convolucional (CNN) aún más profunda y potente que las anteriores, inspirada en arquitecturas como **VGG**. El modelo se divide en dos bloques principales:

1. `self.features`: compuesto por **14 capas convolucionales y de activación**, organizadas en **4 bloques convolucionales**.

   Cada bloque incluye dos capas `Conv2D`, seguidas de `BatchNorm2D`, una función de activación `ReLU` y una capa de `pooling`. El último bloque finaliza con una capa `AdaptiveAvgPool2D`, que permite obtener una salida de tamaño fijo independientemente del tamaño de entrada.

   - **Bloque 1**:
     - `Conv2D`: de 3 canales RGB a 64 filtros, seguido de otra `Conv2D` de 64 a 64.
     - `BatchNorm2D` y `ReLU` tras cada convolución.
     - `MaxPool2D`: reduce de 64×64 a 32×32.
   - **Bloque 2**:

- `Conv2D`: de 64 a 128, seguido de otra `Conv2D` de 128 a 128.
- `BatchNorm2D` y `ReLU` tras ambas.
- `MaxPool2D`: reduce de 32×32 a 16×16.

- **Bloque 3**:
  - `Conv2D`: de 128 a 256, seguido de otra `Conv2D` de 256 a 256.
  - `BatchNorm2D` y `ReLU`.
  - `MaxPool2D`: reduce de 16×16 a 8×8.
- **Bloque 4**:
  - `Conv2D`: de 256 a 512, seguido de otra `Conv2D` de 512 a 512.
  - `BatchNorm2D` y `ReLU`.
  - `AdaptiveAvgPool2D`: reduce dinamicamente a un tamaño fijo de (4, 4) por canal.

Esta parte toma una imagen de entrada de tamaño (3, 64, 64) y la convierte en un tensor de tamaño (512, 4, 4), equivalente a 8192 características profundas.

2. `self.classifier`: compuesto por **7 capas secuenciales** que procesan el vector extraído y realizan la clasificación:
   - `Flatten`: convierte el tensor (512, 4, 4) en un vector de tamaño 8192.
   - `Linear1`: mapea de 8192 a 512 neuronas.
   - `ReLU`.
   - `Dropout` (p=0.5).
   - `Linear2`: reduce de 512 a 256.
   - `ReLU + Dropout` (p=0.5).
   - `Linear final`: genera una salida de tamaño `num_classes`.

Este modelo mejora significativamente respecto a `DeepCNN` al incorporar:

- Mayor **profundidad y capacidad de representación**.
- **Batch normalization** tras cada convolución para mejorar la estabilidad del entrenamiento.
- **Dropout doble** en la parte densa para mitigar el sobreajuste.
- **Pooling adaptable**, que permite una mayor flexibilidad ante distintas resoluciones de entrada.

En conjunto, esta arquitectura está diseñada para alcanzar un alto rendimiento en tareas de clasificación de imágenes complejas (ver Anexo 8.5 para el código completo).

Al igual que con el modelo previo, se incrementa el número de *epochs* a 100, dado que en ejecuciones previas con 50 épocas se observó que el modelo aún continuaba mejorando ligeramente su rendimiento.

Como resultado de estos ajustes, se obtienen métricas aún superiores a las alcanzadas con el modelo anterior, logrando una *accuracy* superior al **90 %** en los conjuntos de entrenamiento, validación y test.

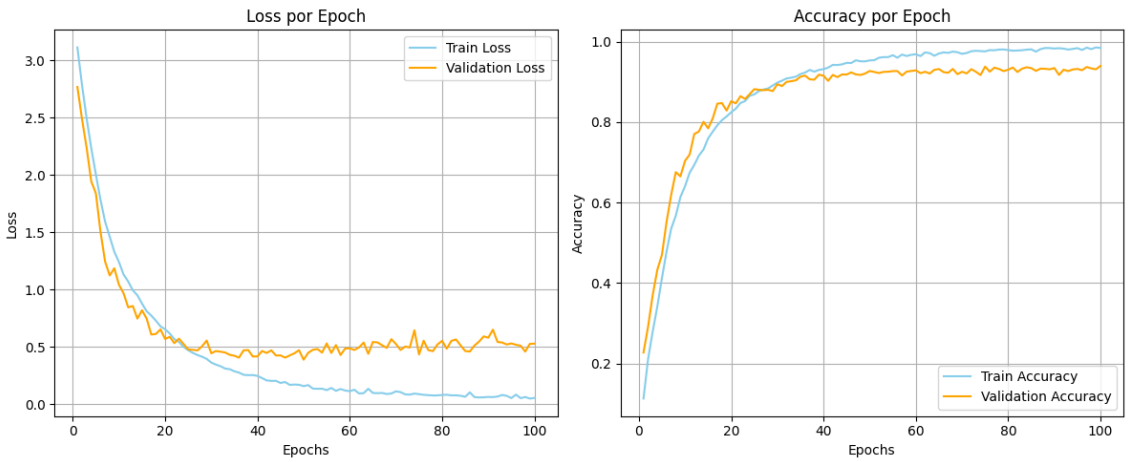

Figura 5.1: Métricas del entrenamiento para la nueva arquitectura y 100 *epochs*

Esta mejora en el rendimiento puede atribuirse, principalmente, a la mayor profundidad de la arquitectura, que permite capturar patrones visuales más complejos y específicos, así como a la inclusión sistemática de técnicas de regularización como `Batch Normalization` y `Dropout`, que contribuyen a estabilizar el entrenamiento y reducir el sobreajuste.

En el conjunto de test, el modelo alcanza una *accuracy* del **93,5 %**, lo que representa una mejora considerable en comparación con las métricas obtenidas por el modelo inicial, y confirma la eficacia de la nueva arquitectura propuesta.

✅ Accuracy total en test: 93.51%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
groundskeeper_willie	1.000000	1.000000	1.000000	19.0
sideshow_mel	1.000000	1.000000	1.000000	6.0
ned_flanders	0.972973	0.986301	0.979592	219.0
marge_simpson	0.989474	0.969072	0.979167	194.0
krusty_the_clown	0.967033	0.972376	0.969697	181.0
principal_skinner	0.961749	0.977778	0.969697	180.0
sideshow_bob	0.976923	0.962121	0.969466	132.0
milhouse_van_houten	0.969136	0.969136	0.969136	162.0
apu_nahasapeemapetilon	0.988636	0.925532	0.956044	94.0
homer_simpson	0.933140	0.952522	0.942731	337.0
chief_wiggum	0.958042	0.925676	0.941581	148.0
edna_krabappel	0.969231	0.913043	0.940299	69.0
bart_simpson	0.963158	0.905941	0.933673	202.0
kent_brockman	0.933333	0.933333	0.933333	75.0
rainier_wolfcastle	0.875000	1.000000	0.933333	7.0
moe_szyslak	0.923423	0.940367	0.931818	218.0
lenny_leonard	0.870370	1.000000	0.930693	47.0
waylon_smithers	0.961538	0.892857	0.925926	28.0
abraham_grampa_simpson	0.953488	0.897810	0.924812	137.0
comic_book_guy	0.941176	0.901408	0.920863	71.0
lisa_simpson	0.888889	0.941176	0.914286	204.0
charles_montgomery_burns	0.869792	0.932961	0.900270	179.0
nelson_muntz	0.847458	0.925926	0.884956	54.0
mayor_quimby	0.888889	0.864865	0.876712	37.0
selma_bouvier	0.875000	0.875000	0.875000	16.0
ralph_wiggum	0.857143	0.857143	0.857143	14.0
professor_john_frink	0.888889	0.800000	0.842105	10.0
maggie_simpson	0.800000	0.800000	0.800000	20.0
carl_carlson	0.700000	0.933333	0.800000	15.0
patty_bouvier	0.647059	1.000000	0.785714	11.0
otto_mann	1.000000	0.600000	0.750000	5.0
barney_gumble	0.785714	0.687500	0.733333	16.0
martin_prince	0.666667	0.727273	0.695652	11.0
gil	0.750000	0.600000	0.666667	5.0
troy_mcclure	1.000000	0.500000	0.666667	2.0
agnes_skinner	1.000000	0.428571	0.600000	7.0
snake_jailbird	0.625000	0.555556	0.588235	9.0
cletus_spuckler	0.666667	0.500000	0.571429	8.0
fat_tony	1.000000	0.400000	0.571429	5.0
miss_hoover	1.000000	0.333333	0.500000	3.0
disco_stu	0.000000	0.000000	0.000000	2.0

Figura 5.2: Métricas del testing para la nueva arquitectura y 100 *epochs*

Finalmente, se siguen identificando ciertas clases, como *disco_ stu* o *troy_ mcclure*, para las cuales ninguno de los modelos explorados ha sido capaz de realizar ninguna predicción, a pesar de haber aplicado estrategias para corregir el desbalanceo y de haber empleado arquitecturas más avanzadas.

Este comportamiento sugiere que, en casos de clases extremadamente minoritarias, puede ser necesario recurrir a enfoques adicionales, como técnicas de sobremuestreo específicas o el uso de *loss functions* más adaptadas a escenarios altamente desbalanceados.

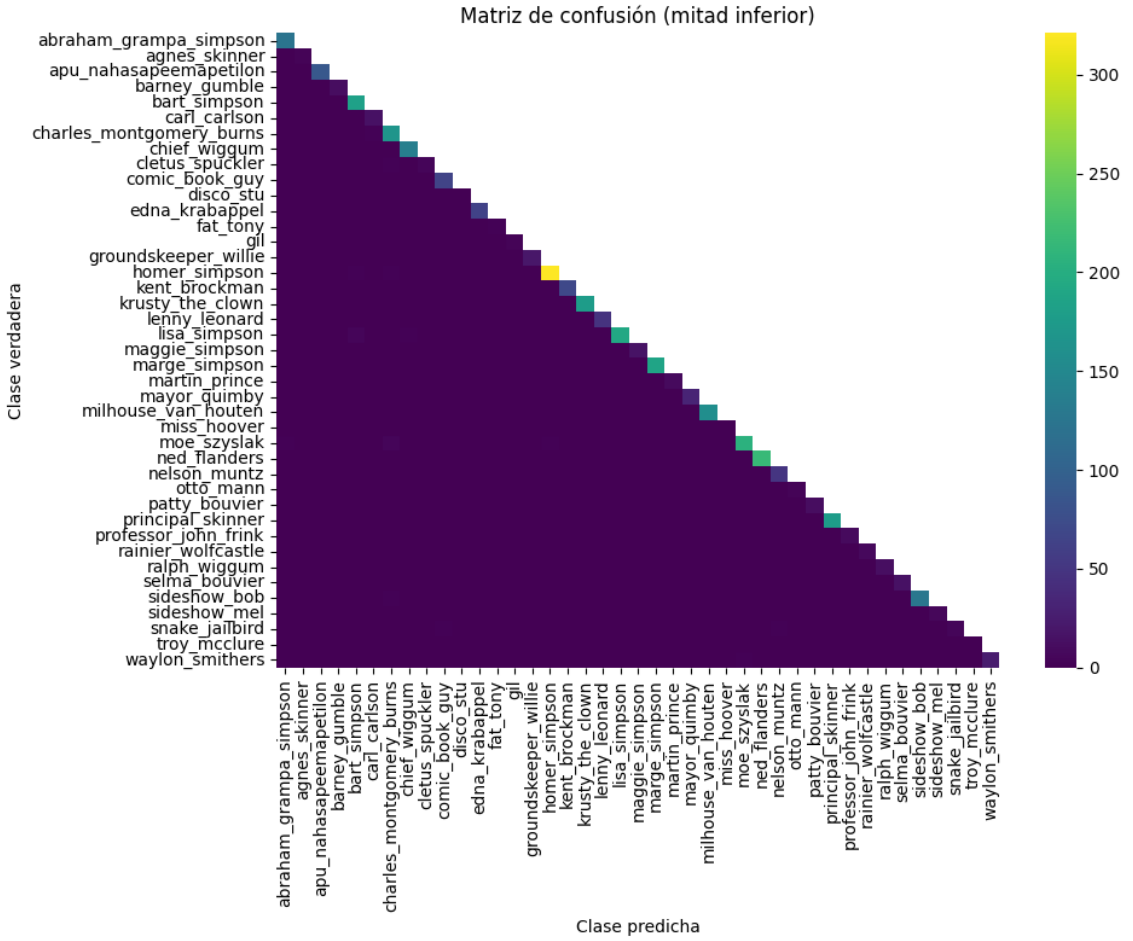

Figura 5.3: Matriz de confusión por clases en test

En cuanto a la matriz de confusión, se observan los mejores resultados obtenidos hasta el momento, con un modelo que apenas incurre en errores de clasificación y cuyas predicciones se concentran prácticamente a lo largo de la diagonal principal. Esto indica un alto nivel de precisión en la identificación de las clases.

Con todo ello, se concluye que este tercer modelo constituye la alternativa más óptima desarrollada, al lograr una alta precisión en el conjunto de *test* y minimizar los errores de clasificación entre clases similares. La combinación adecuada de preprocesamiento, arquitectura convolucional y técnicas de regularización ha permitido obtener un modelo robusto y generalizable, cumpliendo con los objetivos planteados al inicio del trabajo.

CAPÍTULO 6

Futuras líneas de mejora

A pesar de los buenos resultados obtenidos con el modelo final, existen varias estrategias adicionales que podrían explorarse en trabajos futuros con el fin de seguir mejorando el rendimiento, la estabilidad y la capacidad de generalización del modelo:

- **Exploración de distintos optimizadores**: actualmente se ha utilizado el optimizador `Adam`, pero podrían probarse alternativas como `SGD` con momentum, `RMSprop` o incluso optimizadores adaptativos más recientes, con el fin de evaluar su impacto en la convergencia del entrenamiento.
- **Evaluación de distintos criterios de pérdida**: más allá de la función `CrossEntropyLoss`, podrían implementarse funciones ajustadas al desbalanceo como `Focal Loss`, que penaliza más los errores en clases minoritarias, o variantes ponderadas de la entropía cruzada.
- **Diseño de arquitecturas más complejas**: se podrían explorar modelos aún más profundos, o con conexiones residuales (como en *ResNet*), para mejorar la capacidad representacional sin incurrir en problemas de degradación por profundidad.
- **Uso de samplers balanceados en el `DataLoader`**: en lugar de modificar directamente la distribución del conjunto de datos, se podrían utilizar técnicas como `WeightedRandomSampler` para asegurar una representación equitativa de todas las clases durante el entrenamiento.

- **Transfer learning**: utilizar modelos preentrenados sobre grandes datasets como *ImageNet* puede mejorar significativamente la precisión, especialmente en tareas con conjuntos de datos limitados o desbalanceados, reduciendo además los tiempos de entrenamiento.
- **Filtrado de clases minoritarias**: eliminar aquellas clases que cuenten con un número muy reducido de imágenes (por debajo de un umbral establecido) podría contribuir a reducir el ruido en el entrenamiento y a obtener un modelo más estable y equilibrado.
- **Limpieza de fotos**: eliminar aquellas imágenes en las que aparezcan varios personajes a la vez o que no quede claro quién es el personaje principal.
- **Uso de distintos esquemas de *padding***: se podría experimentar con esquemas de *padding* distintos a `same` o `valid`, o incluso eliminar el *padding* para evaluar su impacto sobre la extracción de características en las primeras capas.
- **Alternativas a `MaxPooling`**: se podría sustituir o complementar el `MaxPooling` por otras técnicas de reducción espacial como `AveragePooling`.

CAPÍTULO 7

Extra: Modelo sin convolución (`FullyConnectedNN`)

Como ejercicio adicional, se ha implementado un modelo basado exclusivamente en capas totalmente conectadas, prescindiendo por completo de capas convolucionales. Este modelo, denominado `FullyConnectedNN`, recibe como entrada la imagen ya aplanada en un vector unidimensional, y procesa la información únicamente mediante capas densas.

El objetivo de este experimento es ilustrar empíricamente la importancia de las convoluciones en tareas de reconocimiento de imágenes, y proporcionar una referencia comparativa respecto a los modelos convolucionales desarrollados en capítulos anteriores.

7.1 › Entrenamiento del modelo

Se desarrolla una red neuronal completamente conectada (`FullyConnectedNN`) que prescinde totalmente de capas convolucionales. El objetivo de esta arquitectura es servir como línea base para comparar el rendimiento de una red sin convoluciones frente a las CNN utilizadas previamente. El modelo se divide en dos bloques principales:

1. `self.model`: compuesto por **7 capas secuenciales**, organizadas en **3 bloques funcionales**.

   La imagen de entrada se aplana desde su forma original `(3, 64, 64)` a un vector de `12288` elementos mediante una capa `Flatten`. Posteriormente, se procesan estos datos mediante capas densas con activaciones y regularización.

   - **Entrada y aplanado**:
     - `Flatten`: convierte la imagen de entrada `(3, 64, 64)` en un vector unidimensional de tamaño `12288`.
   - **Bloque 1**:
     - `Linear`: 12288 → 1024 neuronas.
     - `ReLU`: función de activación no lineal.
     - `Dropout(0.5)`: regularización para reducir el sobreajuste.
   - **Bloque 2**:
     - `Linear`: 1024 → 512 neuronas.
     - `ReLU`: activa las salidas positivas.
     - `Dropout(0.5)`: segunda capa de regularización.
   - **Capa de salida**:
     - `Linear`: 512 → `num_classes`.
     - Genera la predicción final del modelo, una por clase.

   Esta arquitectura toma como entrada un vector plano y lo transforma a través de múltiples capas densas hasta obtener un vector de logits con una salida por clase.

Este modelo sirve como punto de comparación con las CNN y presenta las siguientes limitaciones:

- No considera la **estructura espacial** de las imágenes, perdiendo relaciones locales clave como bordes, formas o texturas.
- Tiene una **capacidad de representación limitada** para datos visuales complejos.
- Aunque incorpora `Dropout` como regularización, su rendimiento es **notablemente inferior** al de los modelos convolucionales.

En resumen, este enfoque valida empíricamente la importancia del uso de convoluciones para tareas de visión por computador, al demostrar el pobre rendimiento de un modelo puramente denso en comparación con las CNN desarrolladas anteriormente (ver Anexo 8.6 para el código completo).

Dado que el objetivo de este apartado es únicamente realizar una exploración a alto nivel, se ha limitado el entrenamiento del modelo a **10 épocas**. Este número de iteraciones es suficiente para observar de forma clara las limitaciones del enfoque sin convoluciones y constatar su rendimiento inferior en comparación con los modelos basados en redes convolucionales.

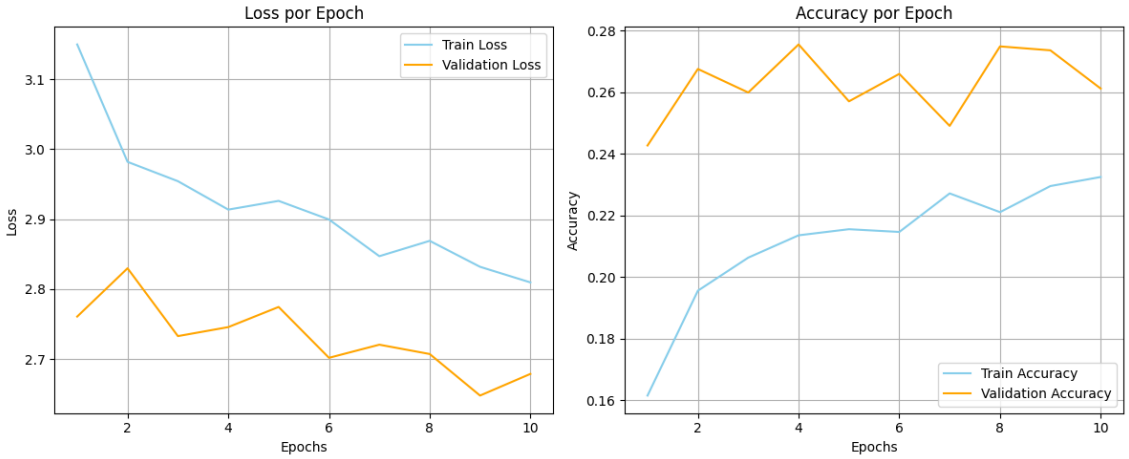

Figura 7.1: Métricas del entrenamiento del modelo sin convolución

A pesar de contar con una arquitectura relativamente profunda (tres capas densas con `ReLU` y `Dropout`), los resultados obtenidos son considerablemente inferiores. El modelo no es capaz de capturar la estructura espacial de las imágenes, ya que el aplanamiento previo destruye la información de correlación local entre píxeles, que es fundamental para detectar patrones visuales como bordes, texturas o formas.

✅ Accuracy total en test: 25.36%
📊 Reporte de clasificación en test (ordenador por f1):

	precision	recall	f1-score	support
marge_simpson	0.762500	0.628866	0.689266	194.0
kent_brockman	1.000000	0.360000	0.529412	75.0
apu_nahasapeemapetilon	0.707317	0.308511	0.429630	94.0
chief_wiggum	0.528736	0.310811	0.391489	148.0
ned_flanders	0.416107	0.283105	0.336957	219.0
sideshow_bob	0.870968	0.204545	0.331288	132.0
principal_skinner	0.348485	0.255556	0.294872	180.0
krusty_the_clown	0.756098	0.171271	0.279279	181.0
homer_simpson	0.142648	0.869436	0.245086	337.0
lisa_simpson	0.473684	0.132353	0.206897	204.0
moe_szyslak	0.179487	0.160550	0.169492	218.0
bart_simpson	0.265957	0.123762	0.168919	202.0
milhouse_van_houten	0.342105	0.080247	0.130000	162.0
edna_krabappel	0.384615	0.072464	0.121951	69.0
charles_montgomery_burns	0.352941	0.067039	0.112676	179.0
comic_book_guy	0.500000	0.014085	0.027397	71.0
abraham_grampa_simpson	0.000000	0.000000	0.000000	137.0
cletus_spuckler	0.000000	0.000000	0.000000	8.0
carl_carlson	0.000000	0.000000	0.000000	15.0
agnes_skinner	0.000000	0.000000	0.000000	7.0
barney_gumble	0.000000	0.000000	0.000000	16.0
gil	0.000000	0.000000	0.000000	5.0
groundskeeper_willie	0.000000	0.000000	0.000000	19.0
disco_stu	0.000000	0.000000	0.000000	2.0
fat_tony	0.000000	0.000000	0.000000	5.0
mayor_quimby	0.000000	0.000000	0.000000	37.0
martin_prince	0.000000	0.000000	0.000000	11.0
lenny_leonard	0.000000	0.000000	0.000000	47.0
maggie_simpson	0.000000	0.000000	0.000000	20.0
nelson_muntz	0.000000	0.000000	0.000000	54.0
otto_mann	0.000000	0.000000	0.000000	5.0
patty_bouvier	0.000000	0.000000	0.000000	11.0
miss_hoover	0.000000	0.000000	0.000000	3.0
professor_john_frink	0.000000	0.000000	0.000000	10.0
rainier_wolfcastle	0.000000	0.000000	0.000000	7.0
ralph_wiggum	0.000000	0.000000	0.000000	14.0
selma_bouvier	0.000000	0.000000	0.000000	16.0
sideshow_mel	0.000000	0.000000	0.000000	6.0
snake_jailbird	0.000000	0.000000	0.000000	9.0
troy_mcclure	0.000000	0.000000	0.000000	2.0
waylon_smithers	0.000000	0.000000	0.000000	28.0

Figura 7.2: Métricas de evaluación en el conjunto de test del modelo sin convolución

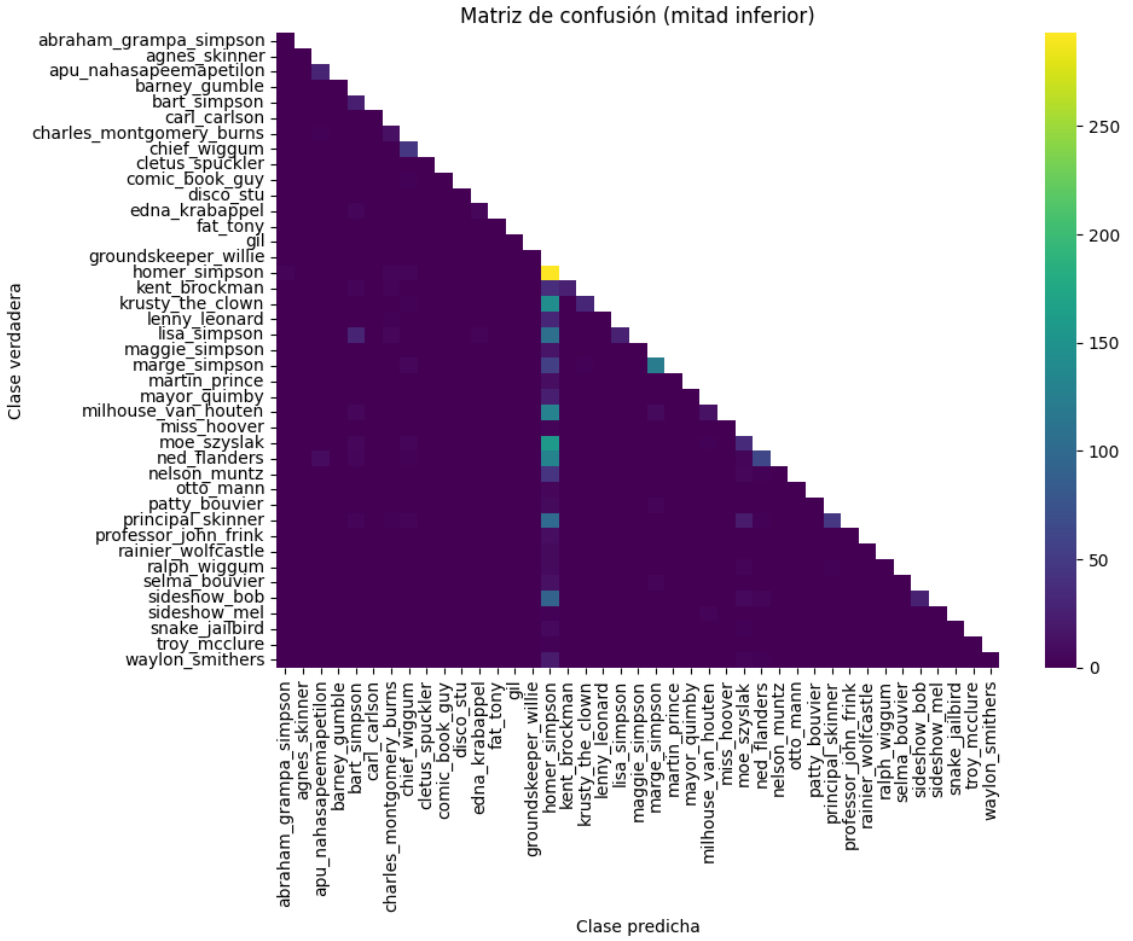

Figura 7.3: Matriz de confusión por clases en test

En comparación con las redes convolucionales, este enfoque obtiene una *accuracy* significativamente más baja tanto en el conjunto de validación como en el de test, evidenciando así que las CNN son más adecuadas para tareas de clasificación de imágenes, debido a su capacidad de extraer jerárquicamente características espaciales a diferentes niveles de abstracción.

Este experimento sirve, por tanto, como una validación del diseño arquitectónico adoptado en los modelos principales, y refuerza la elección de las convoluciones como componente clave para abordar eficazmente problemas de visión por computador.

CAPÍTULO 8

Anexos

```
#  Modelo CNN simple
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        """
        Inicializa la red convolucional simple.

        :param num_classes: Numero de clases de salida (10 en este caso).
        """
        super(SimpleCNN, self).__init__()
        self.conv_layer = nn.Sequential(

            nn.Conv2d(3, 32, kernel_size=3, padding=1),     # Toma 3 canales de
                entrada (RGB), aplica 32 filtros de convolucion de tamano 3x3, y
                 padding de 1 para mantener el tamano de la imagen.
            nn.ReLU(),                                      # Modifica el valor
                de cada celda: Si x < 0, entonces f(x) = 0. Si x >= 0, entonces
                f(x) = x.
            nn.MaxPool2d(2),                                # Reduce las
                dimensiones de la imagen a la mitad (de 64x64     32x32)
                mediante max pooling con kernel 2x2

            nn.Conv2d(32, 64, kernel_size=3, padding=1),    # Toma 32 canales de
                 entrada (del bloque anterior), aplica 64 filtros de convolucion
                 de tamano 3x3, y padding de 1 para mantener el tamano de la
                imagen.
            nn.ReLU(),                                      # Modifica el valor
                de cada celda: Si x < 0, entonces f(x) = 0. Si x >= 0, entonces
                f(x) = x.
            nn.MaxPool2d(2),                                # Reduce las
                dimensiones de la imagen a la mitad (de 32x32     16x16)
                mediante max pooling con kernel 2x2
        )
```

```
        # Output: (64, 16, 16) (64 canales de salida, cada uno de tamano 16x16)

        self.fc_layer = nn.Sequential(

            nn.Flatten(),                                   # Aplana el tensor
                de salida de la parte convolucional (64, 16, 16) en un vector de
                 tamano 64*16*16 para poder pasarlo a capas lineales.
            nn.Linear(64 * 16 * 16, 128),                   # Capa totalmente
                conectada que reduce el vector de entrada (16.384 elementos) a
                128 neuronas intermedias.
            nn.ReLU(),                                      # Activacion no
                lineal que permite a la red aprender relaciones complejas.
            nn.Linear(128, num_classes)                     # Capa de salida que
                 mapea las 128 neuronas a tantas salidas como clases tenga el
                problema (una por clase).

        )

        # Output: Un vector 1D con 10 valores (uno por clase) que representa la
            probabilidad de que la imagen pertenezca a cada clase.

    def forward(self, x):
        """
        Metodo de propagacion hacia adelante. Define como se transforma la
            entrada a traves de la red.

        :param x: Tensor de entrada (batch de imagenes).
        :return: Tensor de salida (predicciones de clase).
        """
        x = self.conv_layer(x)                              # Pasa la entrada a
            traves de las capas convolucionales y de pooling.
        x = self.fc_layer(x)                                # Pasa la salida a
            traves de las capas totalmente conectadas.
        return x                                            # Devuelve la salida
             final de la red.
```

Listing 8.1: Definicion del modelo `SimpleCNN` en PyTorch

```
def train_model(model, train_loader, val_loader, criterion, optimizer, epochs
    =10):

    history = []  # Lista para acumular dicts con metricas por epoca (resumen
        para posterior grafica)

    # Bucle principal de entrenamiento por cada epoca
    for epoch in range(epochs):                             # En cada epoch se
        recorre todo el dataset de entrenamiento
        model.train()                                       # Activa modo
            entrenamiento (habilita dropout, batchnorm, etc.)
        running_loss = 0.0                                  # Inicializacion
            variable para acumular la perdida total (loss) de cada batch a lo
            largo de toda la epoca actual
        correct = 0                                         # Inicializacion
            variable para acumular predicciones correctas en cada batch a lo
            largo de toda la epoca actual
        total = 0                                           # Inicializacion
            variable para contar el numero total de ejemplos procesados durante
            toda la epoca

        # Bucle sobre todos los batches del conjunto de entrenamiento (que esta
            en formato DataLoader)
        for images, labels in tqdm(train_loader, desc=f"Epoca {epoch+1}/{epochs}
            "):
```

```
            images, labels = images.to(device), labels.to(device)  # Como antes
                hemos movido el modelo a device (CPU o GPU), se mueven los datos
                 (imagenes y etiquetas) tambien

            optimizer.zero_grad()                          # Se quiere un gradiente
                 por cada batch. Se reinicia gradiente para no acumular el del
                batch anterior
            outputs = model(images)                        # Se obtiene la
                prediccion del modelo para el batch actual de imagenes
            loss = criterion(outputs, labels)              # Calcula la perdida
                entre prediccion y etiqueta real (-log(probabilidad de la clase
                correcta))
            loss.backward()                                # Backpropagation: se
                calculan los gradientes de la funcion de perdida respecto a los
                parametros del modelo (regla de la cadena)
            optimizer.step()                               # Gradient descent: toma
                 los gradientes de loss.backward() y actualiza los pesos del
                modelo para minimizar la perdida

            # Esta ultima parte se entiende como:
            #   - Forward pass (outputs)     el modelo intenta una prediccion.
            #   - Calculo de perdida     mide que tan mal lo hizo.
            #   - Backward pass (loss.backward())     calcula como corregir los
                pesos para mejorar (Backpropragation).
            #   - Optimizer step     realiza el cambio real en los pesos.

            running_loss += loss.item()                    # Acumula la perdida por
                 batch para tener un loss final por epoca
            _, preds = torch.max(outputs, 1)               # Obtiene la clase con
                mayor probabilidad (prediccion)
            correct += (preds == labels).sum().item()    # Acumula el numero de
                cuantas predicciones fueron correctas por batch para tener un
                total de aciertos por epoca
            total += labels.size(0)                        # Suma la cantidad total
                 de ejemplos

        # Calcula rendimiento en entrenamiento para la epoca actual
        train_loss = running_loss / len(train_loader)
        train_acc = correct / total

        print(f" Entrenamiento - Loss: {train_loss:.4f} | Accuracy: {train_acc
            *100:.2f}%")

        # --- Validacion ---  (similar a entrenamiento)

        model.eval()
        val_running_loss = 0.0
        val_correct = 0
        val_total = 0

        # No se calculan gradientes durante validacion (ahorra memoria)
        with torch.no_grad():

            # Se repiten los pasos de la validacion para cada batch del conjunto
                 de validacion (que esta en formato DataLoader)
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)

                val_running_loss += loss.item()
                _, preds = torch.max(outputs, 1)
                val_correct += (preds == labels).sum().item()
                val_total += labels.size(0)

        # Calcula rendimiento en validacion para la epoca actual
        val_loss = val_running_loss / len(val_loader)
```

```
        val_acc = val_correct / val_total

        print(f" Validacion - Loss: {val_loss:.4f} | Accuracy: {val_acc*100:.2f
            }%\n")

        # Guardar metricas en dict
        history.append({
            'epoch': epoch + 1,
            'train_loss': train_loss,
            'val_loss': val_loss,
            'train_acc': train_acc,
            'val_acc': val_acc
        })

    history_df = pd.DataFrame(history)

    return model, history_df  # Devuelve el modelo ya entrenado y resumen de
        metricas a lo largo de las epocas
```

Listing 8.2: Funcion de entrenamiento del modelo con seguimiento de metricas

```
def test_model(model, test_loader, class_names):    # Procedimiento basicamente
    identico al de validacion
    model.eval()
    model.to(device)

    all_preds = []
    all_labels = []

    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Matriz de confusi n
    cm = confusion_matrix(all_labels, all_preds)

    # Crear m scara para la parte superior
    mask = np.triu(np.ones_like(cm, dtype=bool), k=1)

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, mask=mask, annot=False, fmt='d',
                xticklabels=class_names, yticklabels=class_names, cmap='viridis'
                    )
    plt.xlabel("Clase predicha")
    plt.ylabel("Clase verdadera")
    plt.title("Matriz de confusi n (mitad inferior)")
    plt.xticks(rotation=90)
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.show()

    acc = accuracy_score(all_labels, all_preds)
    print(f" Accuracy total en test: {acc * 100:.2f}%")

    return classification_report(all_labels, all_preds, target_names=class_names
        , output_dict=True) # En validacion no se imprime por eficiencia, pero
        en test si (ya hay un modelo final)
```

Listing 8.3: Evaluacion del modelo final sobre el conjunto de test

```
#  Modelo CNN mas profundo con batch normalization, dropout y adaptive pooling
class DeepCNN(nn.Module):
    def __init__(self, num_classes):
        """
        Inicializa una red convolucional mas profunda que la SimpleCNN.

        :param num_classes: Numero de clases de salida.
        """
        super().__init__()

        self.features = nn.Sequential(

            nn.Conv2d(3, 32, 3, padding=1),              # Primer bloque: 3
                canales RGB     32 mapas de caracteristicas (filtros 3x3,
                padding 1 mantiene tamano: 64x64).
            nn.BatchNorm2d(32),                          # Normaliza las
                activaciones para estabilizar y acelerar el entrenamiento.
            nn.ReLU(),                                   # Funcion de activacion
                 ReLU para introducir no linealidad.
            nn.MaxPool2d(2),                             # Reduce las
                dimensiones espaciales a la mitad (64x64     32x32).

            nn.Conv2d(32, 64, 3, padding=1),             # Segundo bloque: 32
                canales de entrada     64 filtros 3x3 (tamano permanece: 32x32).
            nn.BatchNorm2d(64),                          # Batch normalization
                sobre los 64 mapas de caracteristicas.
            nn.ReLU(),                                   # Activacion no lineal.
            nn.MaxPool2d(2),                             # Reduce tamano a la
                mitad nuevamente (32x32     16x16).

            nn.Conv2d(64, 128, 3, padding=1),            # Tercer bloque: 64
                 128 filtros, mantiene dimensiones: 16x16.
            nn.ReLU(),                                   # Activacion ReLU (sin
                batch norm esta vez).
            nn.AdaptiveAvgPool2d((4, 4))                 # Reduce dinamicamente
                cada mapa a tamano fijo 4x4 (sin importar el tamano de entrada
                exacto).
        )

        # Output de features: (128, 4, 4)     128 canales de salida, cada uno de
             4x4

        self.classifier = nn.Sequential(

            nn.Flatten(),                                # Aplana los tensores
                128x4x4 en un vector de tamano 2048 (128 * 4 * 4).
            nn.Linear(128 * 4 * 4, 256),                 # Capa totalmente
                conectada de 2048     256 neuronas.
            nn.ReLU(),                                   # Activacion no lineal.
            nn.Dropout(0.5),                             # Desactiva
                aleatoriamente el 50% de las neuronas para reducir el
                sobreajuste.
            nn.Linear(256, num_classes)                  # Capa de salida que
                proyecta a `num_classes` (una salida por clase).

        )

        # Output final: vector 1D con `num_classes` elementos que representa las
             predicciones para cada clase

    def forward(self, x):
        """
        Metodo de propagacion hacia adelante. Define como se transforma la
            entrada a traves de la red.

        :param x: Tensor de entrada (batch de imagenes).
        :return: Tensor de salida (predicciones de clase).
```

```
        """
        x = self.features(x)                                    # Pasa la entrada por
            las capas convolucionales + pooling + normalization.
        x = self.classifier(x)                                  # Pasa el vector a
            traves de las capas densas para obtener prediccion final.
        return x                                                # Devuelve el resultado
             (logits por clase).
```

Listing 8.4: Definicion del modelo `DeepCNN` con normalizacion, dropout y pooling adaptativo

```
# Modelo CNN aun mas profundo con multiples bloques convolucionales, batch
    normalization y dropout
class VeryDeepCNN(nn.Module):
    def __init__(self, num_classes):
        """
        Inicializa una red convolucional profunda con varios bloques, similar a
            VGG.

        :param num_classes: Numero de clases de salida.
        """
        super().__init__()

        self.features = nn.Sequential(

            # Primer bloque convolucional: entrada RGB     64 filtros
            nn.Conv2d(3, 64, 3, padding=1),                 # Filtros 3x3, padding
                1 mantiene tamano: 64x64
            nn.BatchNorm2d(64),                             # Normalizacion de
                activaciones
            nn.ReLU(),                                      # Activacion ReLU
            nn.Conv2d(64, 64, 3, padding=1),                # Segundo conv del
                bloque (64x64)
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2),                                # Reduccion espacial:
                64x64     32x32

            # Segundo bloque convolucional: 64     128 filtros
            nn.Conv2d(64, 128, 3, padding=1),               # Tamano se mantiene:
                32x32
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(2),                                # 32x32     16x16

            # Tercer bloque convolucional: 128     256 filtros
            nn.Conv2d(128, 256, 3, padding=1),              # Tamano se mantiene:
                16x16
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 256, 3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(2),                                # 16x16     8x8

            # Cuarto bloque convolucional: 256     512 filtros
            nn.Conv2d(256, 512, 3, padding=1),              # Mantiene tamano: 8x8
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.Conv2d(512, 512, 3, padding=1),
            nn.BatchNorm2d(512),
```

```
            nn.ReLU(),
            nn.AdaptiveAvgPool2d((4, 4))                    # Reduccion adaptable a
                tamano fijo: 4x4 por canal (512 x 4 x 4)
        )

        # Salida de la seccion convolucional: (512, 4, 4) = 8192 elementos

        self.classifier = nn.Sequential(

            nn.Flatten(),                                   # Convierte tensor 512
                x4x4 en vector de 8192
            nn.Linear(512 * 4 * 4, 512),                    # Primera capa densa:
                8192     512
            nn.ReLU(),                                      # Activacion ReLU
            nn.Dropout(0.5),                                # Dropout 50% para
                combatir el sobreajuste

            nn.Linear(512, 256),                            # Segunda capa densa:
                512     256
            nn.ReLU(),                                      # Activacion ReLU
            nn.Dropout(0.5),                                # Otro dropout

            nn.Linear(256, num_classes)                     # Capa final: 256
                numero de clases
        )

    def forward(self, x):
        """
        Metodo de propagacion hacia adelante. Define como se transforma la
            entrada a traves de la red.

        :param x: Tensor de entrada (batch de imagenes).
        :return: Tensor de salida (predicciones de clase).
        """
        x = self.features(x)                                # Pasa la entrada por
            capas convolucionales
        x = self.classifier(x)                              # Clasifica usando
            capas densas
        return x                                             # Devuelve logits (una
             prediccion por clase)
```

Listing 8.5: Modelo `VeryDeepCNN` inspirado en VGG con mayor profundidad y regularizacion

```
# Definicion del modelo completamente conectado (sin convoluciones)
class FullyConnectedNN(nn.Module):
    def __init__(self, num_classes):
        """
        Red neuronal totalmente conectada sin capas convolucionales.
        La entrada se aplana desde (3, 64, 64) a un vector de 12288 elementos.
        """
        super().__init__()
        self.model = nn.Sequential(
            nn.Flatten(),                           # Aplana la imagen (3,64,64)
                12288
            nn.Linear(3 * 64 * 64, 1024),           # Capa densa: 12288     1024
            nn.ReLU(),                              # Activacion ReLU
            nn.Dropout(0.5),                        # Regularizacion

            nn.Linear(1024, 512),                   # Capa intermedia
            nn.ReLU(),
            nn.Dropout(0.5),

            nn.Linear(512, num_classes)             # Capa de salida
```

```
        )

    def forward(self, x):
        return self.model(x)
```

Listing 8.6: Definición del modelo `FullyConnectedNN` sin capas convolucionales