

### Tarea III

#### AdvNNs

7 de noviembre de 2025

#### Resumen

El objetivo de esta tarea es iniciar el proyecto final. Por favor, suba sus soluciones en un archivo comprimido a Classroom antes del 18 de noviembre.

#### Problema 1

Describa en los siguientes puntos clave de su proyecto:

a) Un breve resumen del proyecto. No olvide declarar claramente el objetivo de su proyecto.

En un proyecto para una aplicación que le enseñe a los niños identificar diferentes deportes a través de imágenes, se busca desarrollar modelo que dada la enorme cantidad de imágenes que se requieren para, en primer lugar, la aplicación sea entretenida y, en segundo lugar, pueda identificar si un usuario suba una imagen de un deporte específico, el modelo debe ser capaz de clasificar imágenes en diferentes categorías deportivas (fútbol, baloncesto, tenis, natación, etc.) con alta precisión y eficiencia.

b) Descripción del conjunto de datos. Describa la forma, el tamaño y el preprocesamiento existente de sus datos, incluyendo el proceso de recolección de datos, etc. Añada el enlace URL de los datos.

El conjunto de datos **Sports Image Dataset** contiene imágenes clasificadas en 22 categorías deportivas, cada una representada por una carpeta con el nombre del deporte correspondiente. Cada carpeta incluye entre 400 y 900 imágenes, con un total aproximado de 14,300 imágenes. Estas imágenes fueron recolectadas utilizando un scraper de Google Images, lo que asegura una amplia variedad de ejemplos para cada categoría.

El conjunto de datos está organizado de la siguiente manera:

- **Número de categorías:** 22 (por ejemplo, fútbol, baloncesto, natación, etc.)
- **Número de imágenes por categoría:** Entre 400 y 900
- **Tamaño total:** 494.03 MB
- **Formato de las imágenes:** No especificado, pero típicamente en formatos comunes como JPG o PNG
- **Licencia:** CC BY-NC-SA 4.0

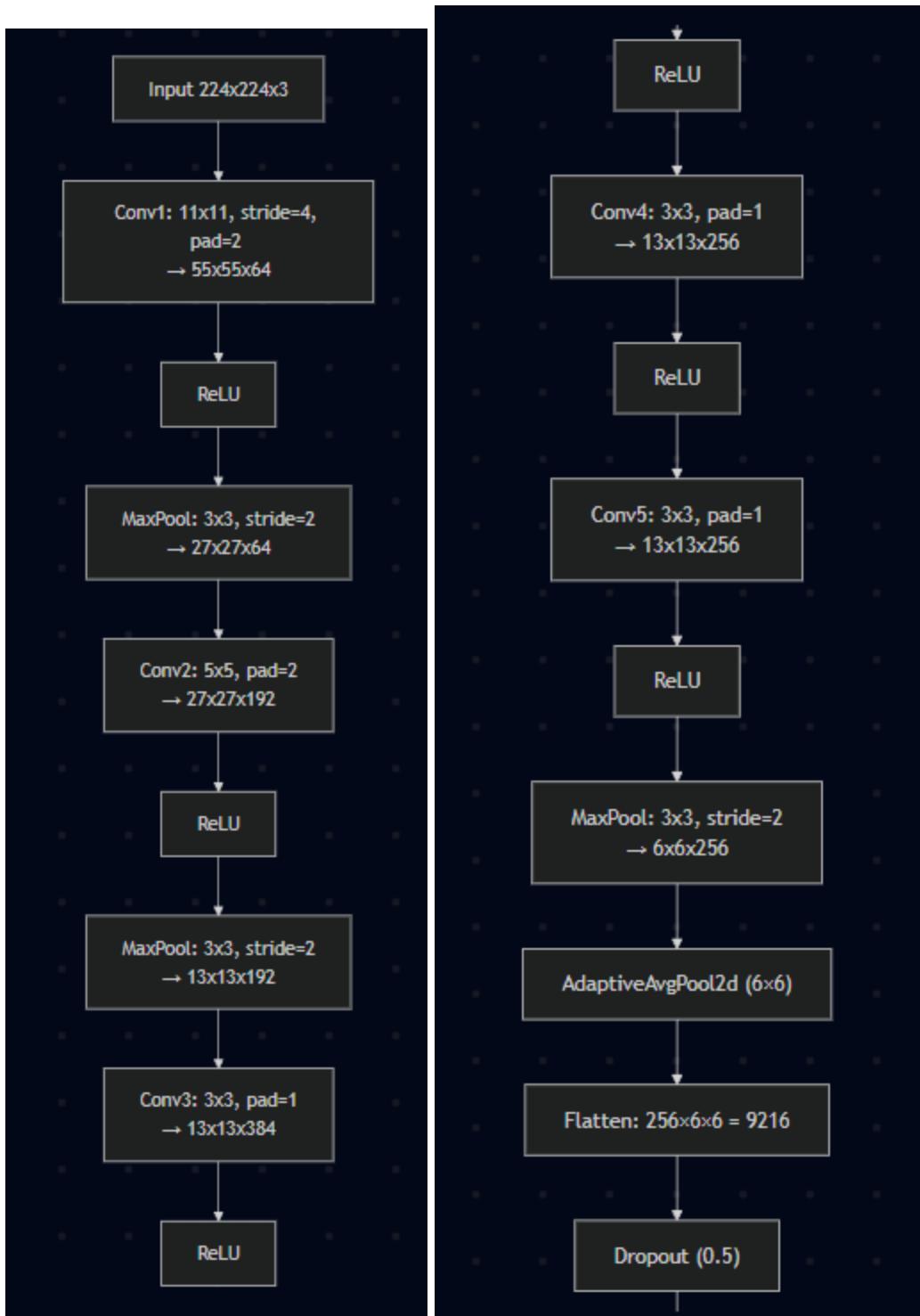
El preprocesamiento inicial incluye la clasificación de las imágenes en carpetas según su categoría. Sin embargo, no se menciona ningún otro tipo de preprocesamiento, como el cambio de tamaño, normalización o eliminación de imágenes duplicadas.

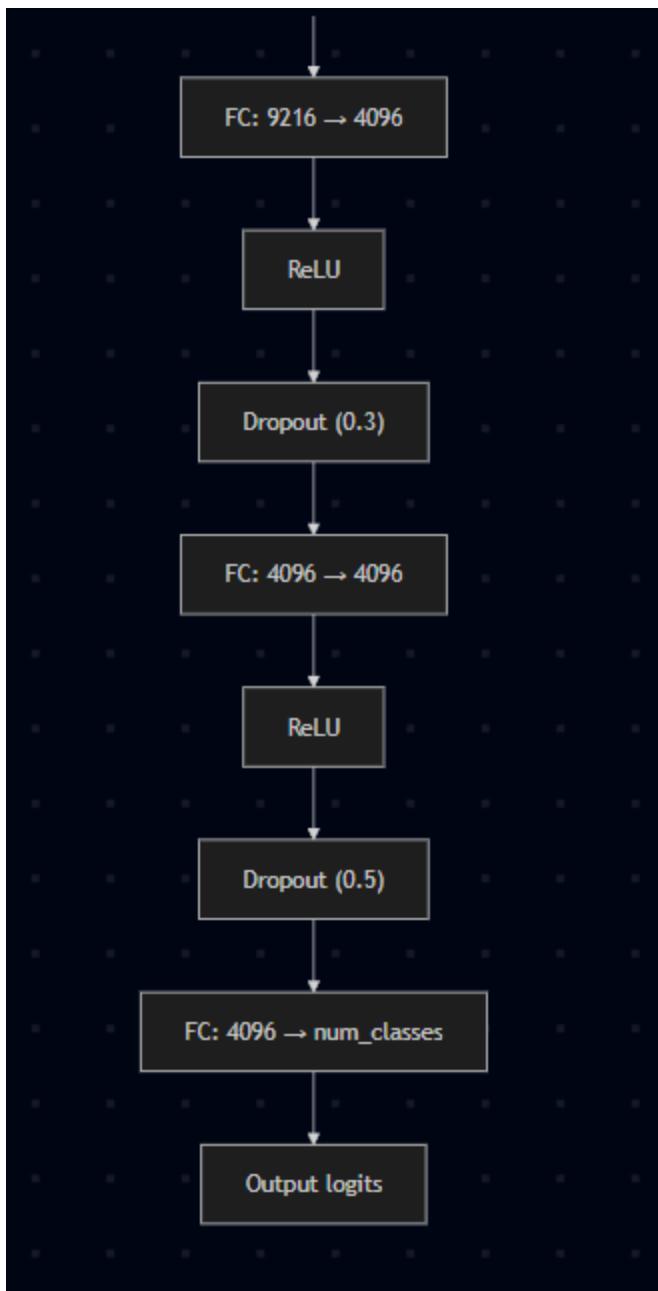
**Enlace al conjunto de datos:** Sports Image Dataset en Kaggle

c) Arquitectura de su modelo. Describa todos los hiperparámetros pertenecientes a su modelo, como el número de capas, neuronas, tipo de red neuronal, etc. Aunque la arquitectura final pueda variar, explique su elección inicial y calcule el número de parámetros entrenables.

Elegire como arquitectura inicial una red neuronal convolucional (CNN) debido a su eficacia comprobada en tareas de clasificación de imágenes. La arquitectura propuesta es la siguiente:

Sera una arquitectura AlexNet, lo considero adecuado debido a su capacidad para aprender características jerárquicas y su eficacia en la clasificación de imágenes.





d) Describa otras consideraciones, incluyendo optimizador, función de coste, etc. Describa por qué los eligió entre otras posibilidades.

Se elegio un optimizador Adam por su capacidad para adaptarse a diferentes tasas de aprendizaje, y una funcion de coste de entropía cruzada categórica, adecuada para problemas de clasificación multiclas, además se programa un scheduler que reduce el learning rate de manera progresiva para mejorar la convergencia y un early stopping para evitar el sobreajuste.

In [ ]:

## Problema 2

Realice un análisis exploratorio de datos. Anote todas sus conclusiones.

In [46]:

```
import os
import kagglehub
import glob

path = kagglehub.dataset_download("rishikeshkonapure/sports-image-dataset")

# Cambia 'images' por 'data' para buscar en todas las carpetas de deportes
imagenes_dir = os.path.join(path, "data")

# Buscar todas las imágenes jpg en todos los subdirectorios
imagenes = glob.glob(os.path.join(imagenes_dir, "**", "*.*"), recursive=True)

print(f"Total de imágenes encontradas: {len(imagenes)}")
```

Total de imágenes encontradas: 12991

In [2]:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import os
import random

imagenes_por_carpeta = {}
for img_path in imagenes:
    carpeta = os.path.basename(os.path.dirname(img_path))
    imagenes_por_carpeta.setdefault(carpeta, []).append(img_path)

imagenes_seleccionadas = []
carpetas = list(imagenes_por_carpeta.keys())
for carpeta in carpetas:
    imgs = imagenes_por_carpeta[carpeta]
    seleccionadas = random.sample(imgs, min(2, len(imgs)))
    for img in seleccionadas:
        imagenes_seleccionadas.append((img, carpeta))

# Muestra las imágenes
plt.figure(figsize=(10, 2 * len(imagenes_seleccionadas) // 2))
for i, (img_path, carpeta) in enumerate(imagenes_seleccionadas):
    img = mpimg.imread(img_path)
    plt.subplot(len(imagenes_seleccionadas) // 2, 2, i + 1)
    plt.imshow(img)
    plt.axis('off')
    plt.title(carpeta, fontsize=10)
plt.tight_layout()
plt.show()
```

badminton



badminton



baseball



baseball



basketball



basketball



boxing



boxing



chess



chess



cricket



cricket



fencing



fencing



football



football



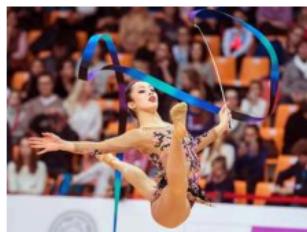
formula1



formula1



gymnastics



gymnastics



hockey



hockey



ice\_hockey



ice\_hockey



kabaddi



kabaddi



motogp



motogp



shooting



shooting



swimming



swimming



table\_tennis



table\_tennis



tennis



tennis



volleyball



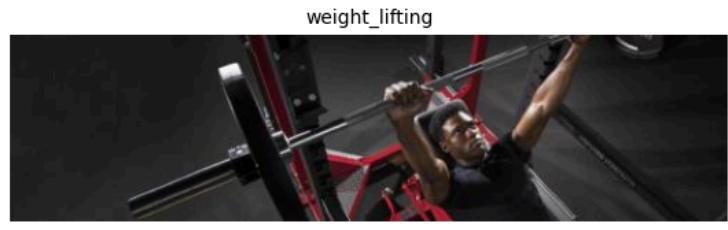
volleyball



weight\_lifting



weight\_lifting



wrestling



wrestling





La anterior salida de código muestra un par de imágenes por categoría deportiva, permitiendo observar la variedad y características visuales de las imágenes en cada categoría.

In [ ]:

```
In [3]: def brillo_medio_color(img_path: str) -> float:
    """
    Calcula el brillo promedio de una imagen RGB.
    El brillo se define como el promedio de los valores de los canales R, G y B.

    Parámetros:
    img_path (str): Ruta de la imagen.

    Retorna:
    float: Brillo promedio.
    """

    try:
        img = mpimg.imread(img_path)
        if img.ndim == 3 and img.shape[2] >= 3:
            # Si la imagen está en formato uint8, normaliza a [0,255]
            if img.dtype == np.uint8:
                brillo = np.mean(img[...,:3])
            else:
                brillo = np.mean(img[...,:3] * 255)
        return brillo
    else:
        # Imagen en escala de grises
        return np.mean(img)
    except Exception as e:
        print(f"Error al procesar {img_path}: {e}")
        return np.nan
```

```
In [4]: # Visualiza la distribución de brillo por carpeta y en general
import matplotlib.pyplot as plt
```

```
# Calcula brillos por carpeta
brillos_por_carpeta = []
for carpeta, rutas in imagenes_por_carpeta.items():
    brillos = [brillo_medio_color(p) for p in rutas if os.path.exists(p)]
    brillos = [b for b in brillos if not np.isnan(b)]
    if brillos:
        brillos_por_carpeta[carpeta] = brillos

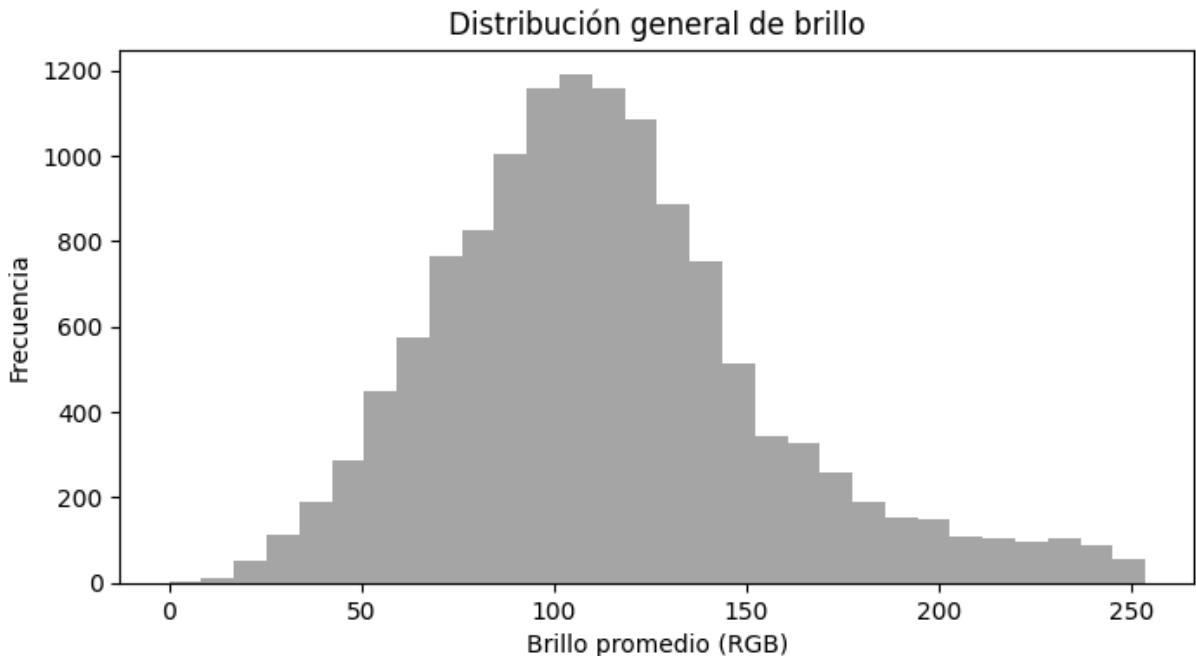
# Histograma general
```

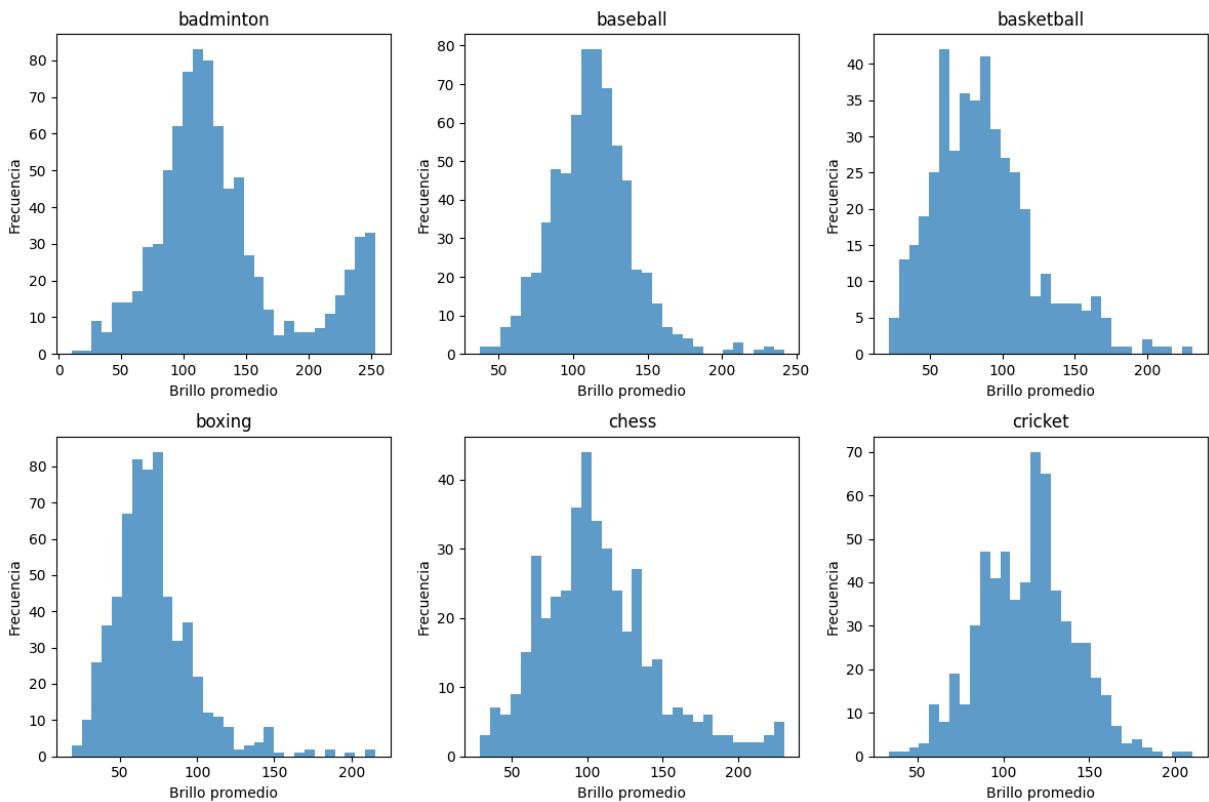
```

todos_los_brillos = [b for brillos in brillos_por_carpeta.values() for b in brillos]
plt.figure(figsize=(8,4))
plt.hist(todos_los_brillos, bins=30, color='gray', alpha=0.7)
plt.title('Distribución general de brillo')
plt.xlabel('Brillo promedio (RGB)')
plt.ylabel('Frecuencia')
plt.show()

# Histogramas por carpeta (máx 6 carpetas para visualización)
plt.figure(figsize=(12,8))
for i, (carpeta, brillos) in enumerate(list(brillos_por_carpeta.items())[:6]):
    plt.subplot(2,3,i+1)
    plt.hist(brillos, bins=30, alpha=0.7)
    plt.title(f'{carpeta}')
    plt.xlabel('Brillo promedio')
    plt.ylabel('Frecuencia')
plt.tight_layout()
plt.show()

```





In [5]:

```

import random
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import os

# Selecciona hasta 10 imágenes por cada carpeta (todas las carpetas incluidas)
pixeles_por_carpeta = {}
for carpeta, rutas in imagenes_por_carpeta.items():
    seleccionadas = random.sample(rutas, min(10, len(rutas)))
    pixeles = []
    for img_path in seleccionadas:
        if os.path.exists(img_path):
            try:
                img = mpimg.imread(img_path)
                if img.ndim == 3 and img.shape[2] >= 3:
                    arr = img[..., :3].reshape(-1)
                    if img.dtype != np.uint8:
                        arr = (arr * 255)
                        arr = arr.astype(np.uint8)
                else:
                    arr = img.reshape(-1)
                    arr = arr.astype(np.uint8)
                pixeles.extend(arr.tolist())
            except Exception as e:
                pass
    if pixeles:
        pixeles_por_carpeta[carpeta] = pixeles

# Histograma global
todos_pixeles = [p for pixs in pixeles_por_carpeta.values() for p in pixs]

```

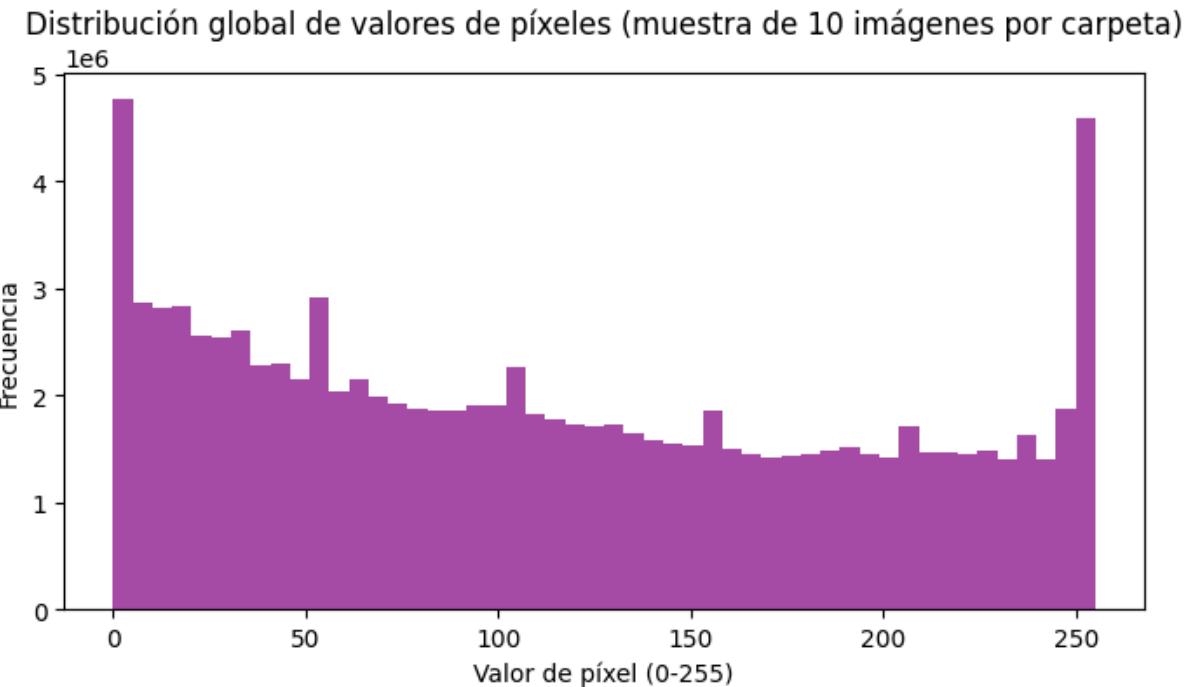
```

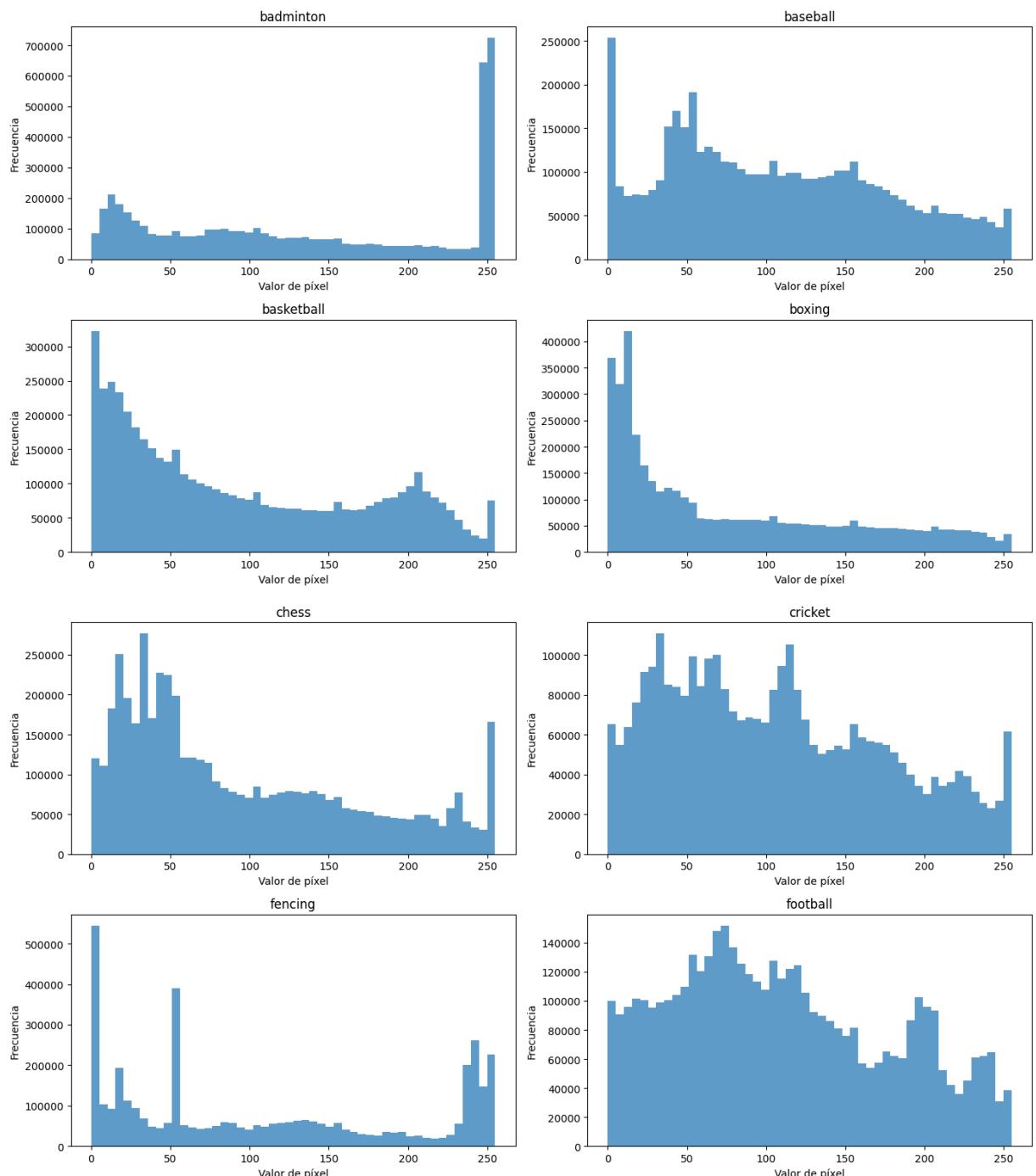
plt.figure(figsize=(8,4))
plt.hist(todos_pixeles, bins=50, color='purple', alpha=0.7)
plt.title('Distribución global de valores de píxeles (muestra de 10 imágenes por carpeta)')
plt.xlabel('Valor de píxel (0-255)')
plt.ylabel('Frecuencia')
plt.show()

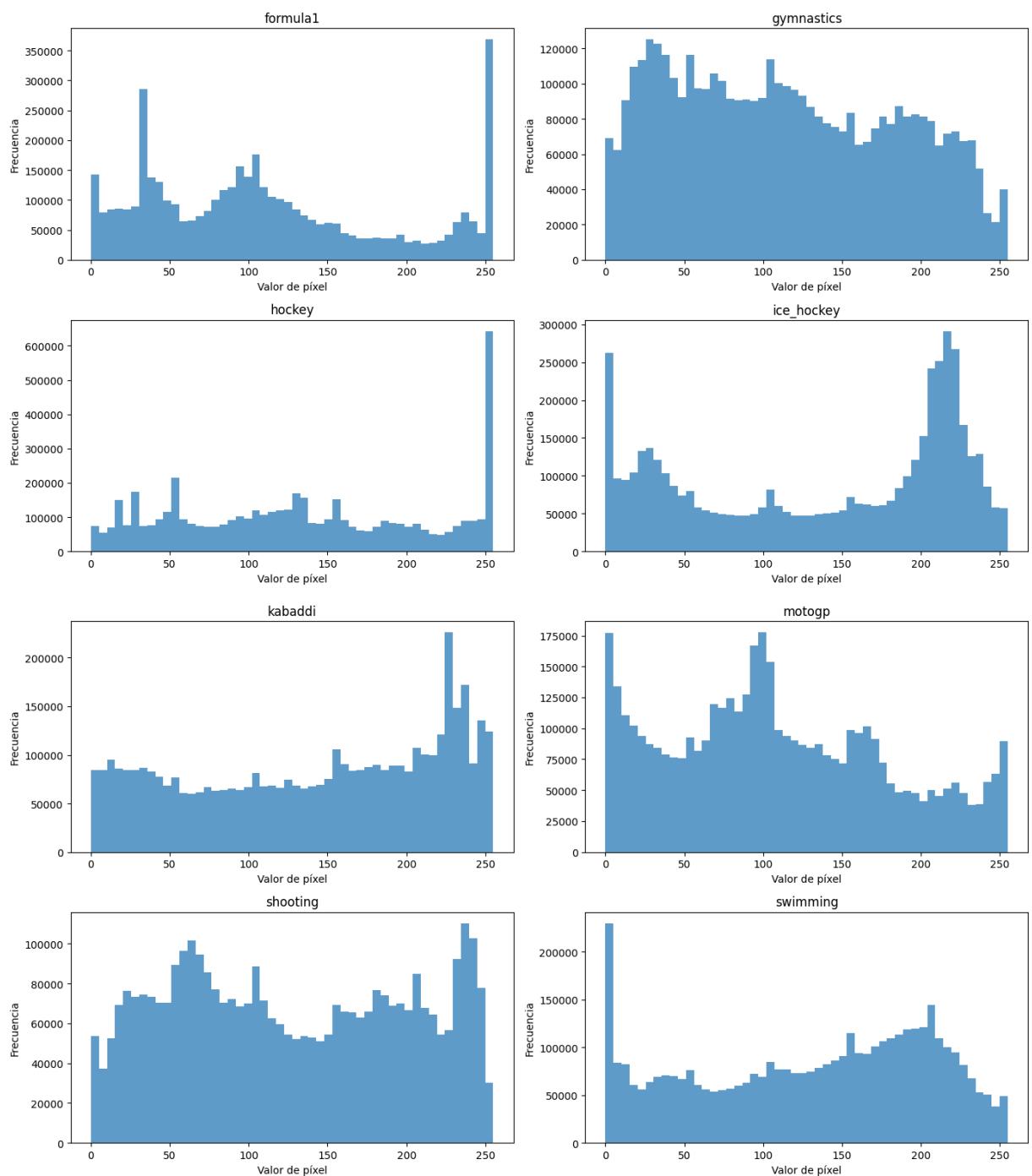
# Visualización: 4 histogramas por figura para mejor claridad
carpetas_a_mostrar = list(pixeles_por_carpeta.items())
num_por_figura = 4
num_figuras = (len(carpetas_a_mostrar) + num_por_figura - 1) // num_por_figura

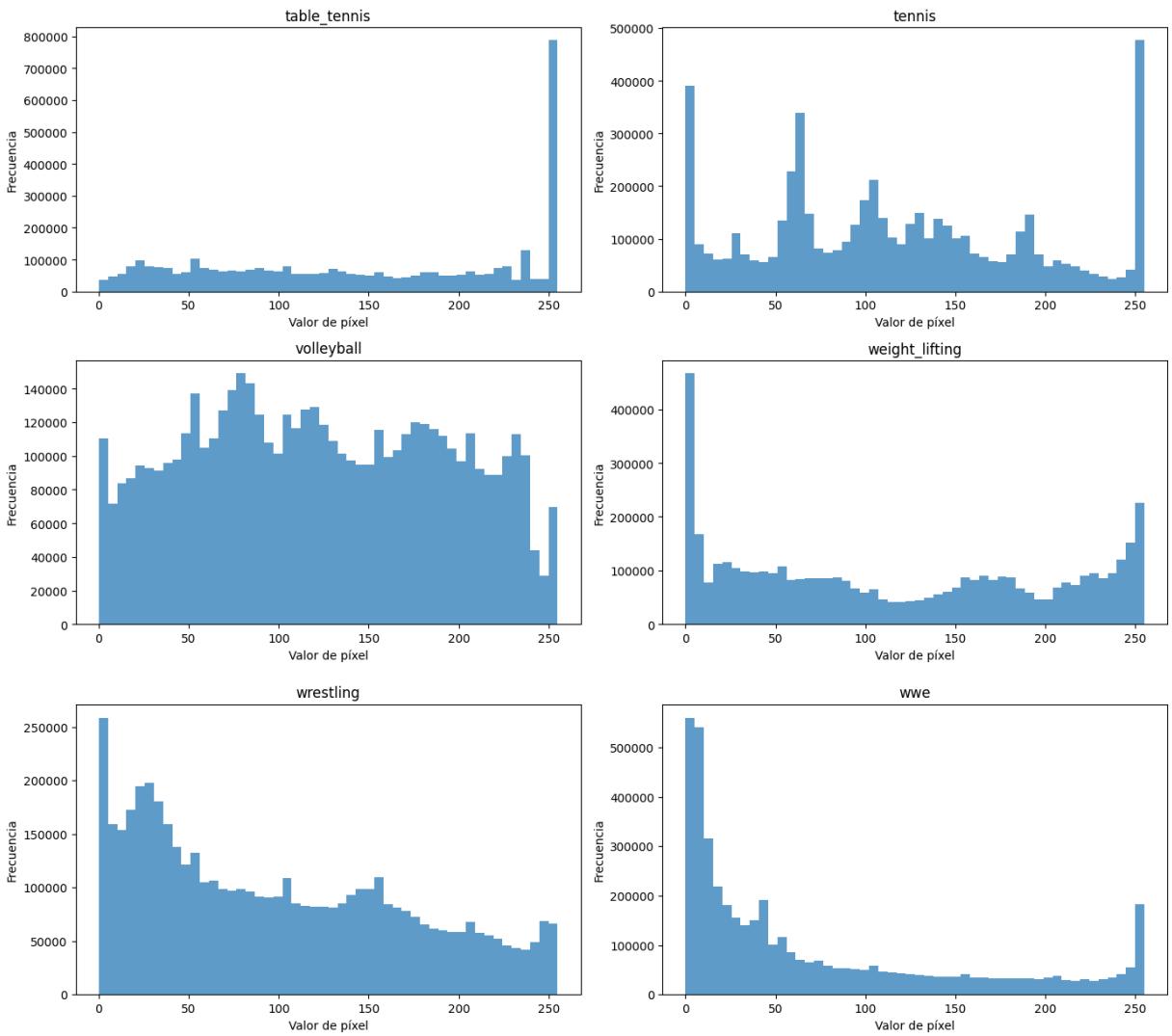
for fig_idx in range(num_figuras):
    plt.figure(figsize=(14, 8))
    for i in range(num_por_figura):
        idx = fig_idx * num_por_figura + i
        if idx >= len(carpetas_a_mostrar):
            break
        carpeta, pixs = carpetas_a_mostrar[idx]
        plt.subplot(2, 2, i + 1)
        plt.hist(pixs, bins=50, alpha=0.7)
        plt.title(f'{carpeta}')
        plt.xlabel('Valor de píxel')
        plt.ylabel('Frecuencia')
    plt.tight_layout()
    plt.show()

```









```
In [6]: import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os
from collections import Counter

anchos = []
altos = []

for img_path in imagenes:
    if os.path.exists(img_path):
        try:
            img = mpimg.imread(img_path)
            altos.append(img.shape[0])
            anchos.append(img.shape[1])
        except Exception as e:
            pass

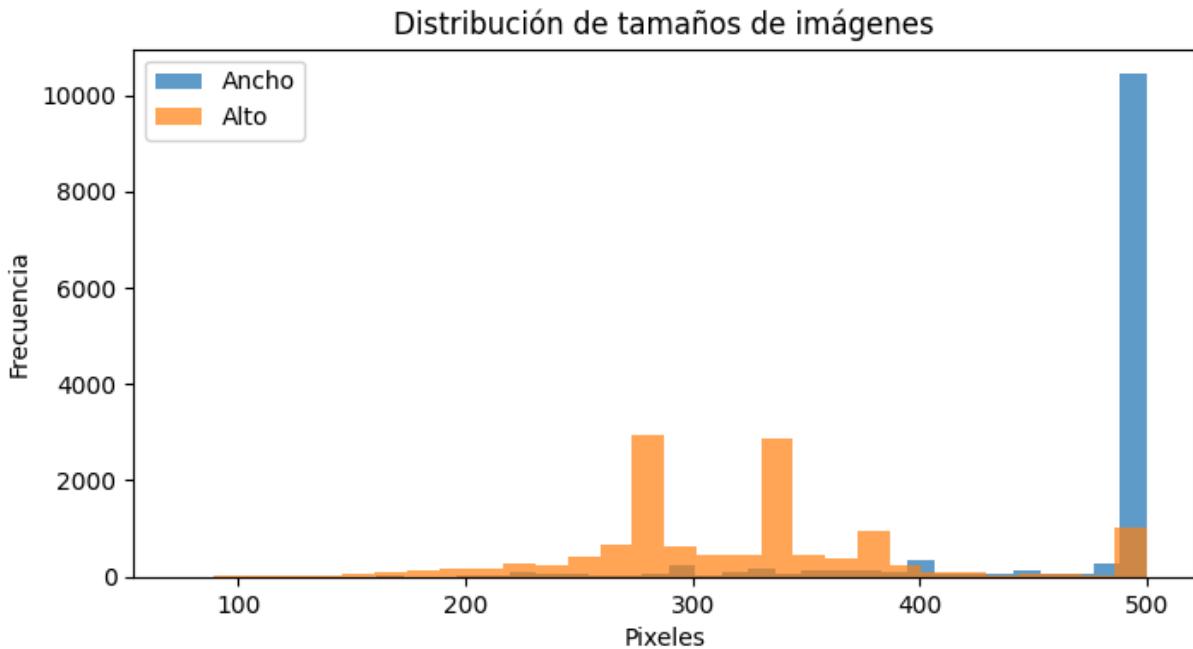
plt.figure(figsize=(8,4))
plt.hist(anchos, bins=30, alpha=0.7, label='Ancho')
plt.hist(altos, bins=30, alpha=0.7, label='Alto')
plt.title('Distribución de tamaños de imágenes')
plt.xlabel('Pixel')
plt.ylabel('Frecuencia')
```

```

plt.legend()
plt.show()

# Corrige el cálculo del tamaño más común
tamanos = list(zip(anchos, altos))
contador = Counter(tamanos)
mas_comun = contador.most_common(1)[0][0] if contador else (None, None)
print(f'Tamaño más común: {mas_comun}')
print(f'Tamaños únicos: {len(contador)}')

```



Tamaño más común: (500, 281)

Tamaños únicos: 1648

La mayoría de imágenes son de (500 x 281) pixeles, hacer un resize de todas las imágenes a (224 x 224) pixeles sera adecuado para el modelo.

In [7]:

```

...
from PIL import Image
import glob
import os

# Carpeta de entrada y salida
input_dir = imagenes_dir
output_dir = "data_224"

os.makedirs(output_dir, exist_ok=True)

# Busca todas las imágenes jpg y png
imagenes = glob.glob(os.path.join(input_dir, "**", "*.jpg"), recursive=True) + \
    glob.glob(os.path.join(input_dir, "**", "*.png"), recursive=True)

for img_path in imagenes:
    with Image.open(img_path) as img:
        img_resized = img.resize((224, 224), Image.LANCZOS)
        # Crea la misma estructura de carpetas en la salida
        rel_path = os.path.relpath(img_path, input_dir)

```

```

        out_path = os.path.join(output_dir, rel_path)
        os.makedirs(os.path.dirname(out_path), exist_ok=True)
        img_resized.save(out_path)
...

```

Out[7]:

```

\lnfrom PIL import Image\lnimport glob\lnimport os\n# Carpeta de entrada y salida
\lninput_dir = imagenes_dir\lnoutput_dir = "data_224"\lnos.makedirs(output_dir, exist_ok=True)\ln# Busca todas las imágenes jpg y png\lnimagenes = glob.glob(os.path.join(input_dir, "**", "*.*"))
glob.glob(os.path.join(input_dir, "**", "*.*"))
for img_path in imagenes:
    with Image.open(img_path) as img:
        img_resized = img.resize((224, 224), Image.LANCZOS)
    # Crea la misma estructura de carpetas en la salida
    rel_path = os.path.relpath(img_path, input_dir)
    out_path = os.path.join(output_dir, rel_path)
    os.makedirs(os.path.dirname(out_path), exist_ok=True)
    img_resized.save(out_path)

```

### Problema 3

Realice una implementación inicial "ingenua" de su proyecto usando PyTorch. Calcule al menos lo siguiente:

- Gráfica de error a lo largo de las iteraciones.
- Precisiones finales de entrenamiento/prueba.
- Tiempo de su simulación (entrenamiento).
- Muestre un dato donde su modelo entrenado exhiba un desempeño excelente.
- Muestre un dato donde su modelo entrenado exhiba un desempeño pobre.
- Identifique el mayor problema/desafío con su modelo actual.
- Proponga un plan para abordar lo anterior.

In [2]:

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Subset
from torchvision import datasets, transforms, models
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import time
import os
from collections import Counter

# Configuración de dispositivo
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Usando dispositivo: {device}')

# Hiperparámetros
BATCH_SIZE = 32
NUM_EPOCHS = 20

```

```

LEARNING_RATE = 1e-4
TRAIN_SPLIT = 0.8

# Transformaciones para las imágenes (con resize incluido)
transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize mediante PyTorch
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Carga del dataset desde el directorio de Kaggle (imagenes_dir ya definido antes)
print(f'Cargando dataset desde: {imagenes_dir}')

dataset = datasets.ImageFolder(root=imagenes_dir, transform=transform)
print(f'Total de imágenes: {len(dataset)}')
print(f'Clases: {dataset.classes}')
print(f'Número de clases: {len(dataset.classes)}')

# División estratificada en entrenamiento y prueba (balance perfecto por clase)
targets = [label for _, label in dataset.samples]
indices = list(range(len(dataset)))

train_indices, test_indices = train_test_split(
    indices,
    test_size=1-TRAIN_SPLIT,
    stratify=targets,
    random_state=42
)

# Crear subsets balanceados
train_dataset = Subset(dataset, train_indices)
test_dataset = Subset(dataset, test_indices)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

print(f'\nTamaño del conjunto de entrenamiento: {len(train_indices)}')
print(f'Tamaño del conjunto de prueba: {len(test_indices)}')

```

Usando dispositivo: cuda  
Cargando dataset desde: C:\Users\sergi\.cache\kagglehub\datasets\rishikeshkonapure\sports-image-dataset\versions\1\data  
Total de imágenes: 14149  
Clases: ['badminton', 'baseball', 'basketball', 'boxing', 'chess', 'cricket', 'fencing', 'football', 'formula1', 'gymnastics', 'hockey', 'ice\_hockey', 'kabaddi', 'motogp', 'shooting', 'swimming', 'table\_tennis', 'tennis', 'volleyball', 'weight\_lifting', 'wrestling', 'wwe']  
Número de clases: 22

Tamaño del conjunto de entrenamiento: 11319  
Tamaño del conjunto de prueba: 2830

In [3]: # --- Data Augmentation: aplicar solo al conjunto de entrenamiento ---  
import random  
torch.manual\_seed(42)  
np.random.seed(42)

```

random.seed(42)

train_transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize mediante PyTorch
    transforms.RandomResizedCrop(224, scale=(0.5, 1.0)), # zoom_range=0.5 equivale
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5), # vertical_flip añadido
    transforms.RandomRotation(degrees=40), # rotation_range=40
    transforms.RandomAffine(degrees=0, translate=(0.3, 0.2)), # width/height shift
    transforms.ColorJitter(brightness=(0.2, 1.0), contrast=0.2, saturation=0.2, hue
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Para validación / test: determinista (solo resize y normalización)
test_transform = transforms.Compose([
    transforms.Resize((224, 224)), # Resize mediante PyTorch
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# Reconstruir datasets con las nuevas transformaciones desde el directorio de Kaggle
train_folder = datasets.ImageFolder(root=imagenes_dir, transform=train_transform)
test_folder = datasets.ImageFolder(root=imagenes_dir, transform=test_transform)

# Usar los mismos índices estratificados que calculamos anteriormente
train_dataset = Subset(train_folder, train_indices)
test_dataset = Subset(test_folder, test_indices)

# Actualizar DataLoaders
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_w
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_w

print('\n>> Data augmentation aplicado: train_transform definido con Resize, Random
print(f"Train samples: {len(train_dataset)}, Test samples: {len(test_dataset)}")
print(f"Dataset cargado desde: {imagenes_dir}")

```

```

>> Data augmentation aplicado: train_transform definido con Resize, RandomResizedCrop, RandomHorizontalFlip y ColorJitter
Train samples: 11319, Test samples: 2830
Dataset cargado desde: C:\Users\sergi\.cache\kagglehub\datasets\rishikeshkonapure\sp
orts-image-dataset\versions\1\data

```

```

In [4]: # === Estadísticas desde DataLoaders (incluye augmentations en train) ===
print('\n== Estadísticas usando DataLoaders (muestras con transforms aplicadas) ==

def compute_loader_stats(loader, n_batches=50):
    """Calcula media y std por canal a partir de batches del loader.
    Se des-normaliza cada batch antes de calcular estadísticas para obtener valores
    """
    means_r, means_g, means_b = [], [], []
    stds_r, stds_g, stds_b = [], [], []
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    batches = 0

```

```

    for images, _ in loader:
        imgs = images.cpu().numpy() # (B, C, H, W)
        imgs = np.transpose(imgs, (0, 2, 3, 1)) # (B, H, W, C)
        # Des-normalizar: x = x * std + mean
        imgs = imgs * std + mean
        imgs = np.clip(imgs, 0.0, 1.0)

        # Calcular por imagen
        for im in imgs:
            means_r.append(im[:, :, 0].mean())
            means_g.append(im[:, :, 1].mean())
            means_b.append(im[:, :, 2].mean())
            stds_r.append(im[:, :, 0].std())
            stds_g.append(im[:, :, 1].std())
            stds_b.append(im[:, :, 2].std())

        batches += 1
        if batches >= n_batches:
            break

    return {
        'mean_r': np.mean(means_r), 'mean_g': np.mean(means_g), 'mean_b': np.mean(means_b),
        'std_r': np.mean(stds_r), 'std_g': np.mean(stds_g), 'std_b': np.mean(stds_b)
    }

# Calcular estadísticas a partir de los loaders (train con augmentations, test sin
train_loader_stats = compute_loader_stats(train_loader, n_batches=50)
test_loader_stats = compute_loader_stats(test_loader, n_batches=50)

# Mostrar tabla comparativa
print(f'{Estadística}':<20} {'Train':<12} {'Test':<12} {'Drift Absoluto':<15} {'Drift Pct':<10.4f}'}
print('-' * 80)
for stat in ['mean_r', 'mean_g', 'mean_b', 'std_r', 'std_g', 'std_b']:
    t = train_loader_stats[stat]
    s = test_loader_stats[stat]
    drift_abs = abs(t - s)
    drift_pct = 100 * drift_abs / (t + 1e-10)
    print(f'{stat}:<20} {t:<12.6f} {s:<12.6f} {drift_abs:<15.6f} {drift_pct:<10.4f}'"

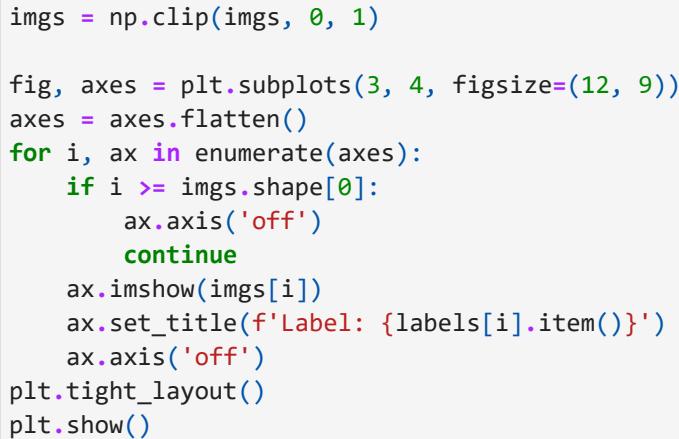
all_drifts = [abs(train_loader_stats[s] - test_loader_stats[s]) / (train_loader_stats[s] + test_loader_stats[s])
              for s in train_loader_stats.keys()]
print(f'\nDrift promedio (loader): {np.mean(all_drifts):.4f}%')
print(f'Drift máximo (loader): {np.max(all_drifts):.4f}%')

# Mostrar algunas imágenes aumentadas (des-normalizadas) para inspección
print('\nMostrando 12 imágenes aumentadas de ejemplo (train)')
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])

batch = next(iter(train_loader))
imgs, labels = batch
imgs = imgs[:12] # tomar hasta 12
imgs = imgs.cpu().numpy()
imgs = np.transpose(imgs, (0, 2, 3, 1))
imgs = imgs * std + mean

```

```



```

== Estadísticas usando DataLoaders (muestras con transforms aplicadas) ==

Estadística	Train	Test	Drift Absoluto	Drift %
mean_r	0.201319	0.450829	0.249510	123.9376
mean_g	0.194017	0.439190	0.245173	126.3665
mean_b	0.187057	0.421824	0.234767	125.5059
std_r	0.177292	0.244122	0.066830	37.6947
std_g	0.167327	0.226756	0.059428	35.5162
std_b	0.163185	0.226976	0.063791	39.0911

Drift promedio (loader): 81.3520%

Drift máximo (loader): 126.3665%

Mostrando 12 imágenes aumentadas de ejemplo (train)



```
In [ ]: train_labels = [targets[i] for i in train_indices]
test_labels = [targets[i] for i in test_indices]

# Contar clases
train_counts = Counter(train_labels)
test_counts = Counter(test_labels)

print('\n==== Distribución de Clases ===')
print(f'{ "Clase":<20} {"Train":<10} {"Test":<10} {"Train %":<10} {"Test %":<10} {"D')
print('-' * 80)

drifts = []
for i, class_name in enumerate(dataset.classes):
    train_count = train_counts[i]
    test_count = test_counts[i]
    train_pct = 100 * train_count / len(train_labels)
    test_pct = 100 * test_count / len(test_labels)
    drift = abs(train_pct - test_pct)
    drifts.append(drift)
    print(f'{class_name:<20} {train_count:<10} {test_count:<10} {train_pct:<10.2f} {test_pct:<10.2f} {abs(drift):<10.2f} {abs(drift)/2:1.2f}%')

print(f'\nDrift promedio: {np.mean(drifts):.4f}%')
print(f'Drift máximo: {np.max(drifts):.4f}%')

# Visualización de la distribución
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# Gráfica de barras comparativa
x = np.arange(len(dataset.classes))
width = 0.35

ax1.bar(x - width/2, [train_counts[i] for i in range(len(dataset.classes))],
         width, label='Train', alpha=0.8, color='blue')
ax1.bar(x + width/2, [test_counts[i] for i in range(len(dataset.classes))],
         width, label='Test', alpha=0.8, color='red')
ax1.set_xlabel('Clase', fontsize=11)
ax1.set_ylabel('Número de imágenes', fontsize=11)
ax1.set_title('Distribución de Clases: Train vs Test', fontsize=13, fontweight='bold')
ax1.set_xticks(x)
ax1.set_xticklabels(dataset.classes, rotation=45, ha='right', fontsize=9)
ax1.legend()
ax1.grid(True, alpha=0.3)

# Gráfica de drift
ax2.bar(x, drifts, color='orange', alpha=0.7)
ax2.set_xlabel('Clase', fontsize=11)
ax2.set_ylabel('Drift (%)', fontsize=11)
ax2.set_title('Drift de Distribución entre Train y Test', fontsize=13, fontweight='bold')
ax2.set_xticks(x)
ax2.set_xticklabels(dataset.classes, rotation=45, ha='right', fontsize=9)
ax2.axhline(y=np.mean(drifts), color='r', linestyle='--', label=f'Promedio: {np.mean(drifts):.2f}%')
ax2.legend()
ax2.grid(True, alpha=0.3)
```

```

plt.tight_layout()
plt.show()

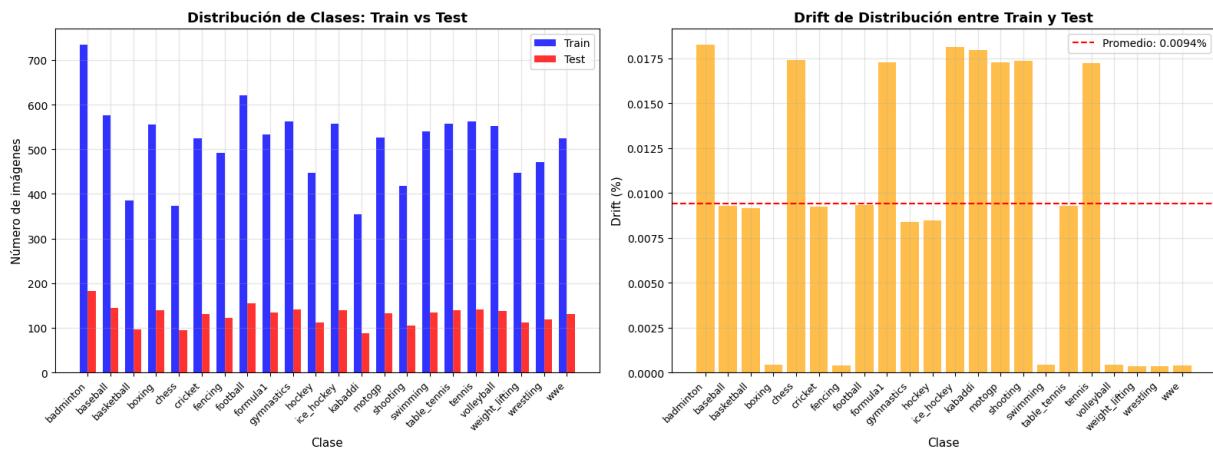
```

== Distribución de Clases ==

Clase	Train	Test	Train %	Test %	Drift %
<hr/>					
badminton	734	183	6.48	6.47	0.0182
baseball	577	144	5.10	5.09	0.0093
basketball	385	96	3.40	3.39	0.0091
boxing	556	139	4.91	4.91	0.0004
chess	374	94	3.30	3.32	0.0174
cricket	525	131	4.64	4.63	0.0092
fencing	492	123	4.35	4.35	0.0004
football	621	155	5.49	5.48	0.0093
formula1	534	134	4.72	4.73	0.0173
gymnastics	563	141	4.97	4.98	0.0084
hockey	447	112	3.95	3.96	0.0085
ice_hockey	558	139	4.93	4.91	0.0181
kabaddi	354	88	3.13	3.11	0.0179
motogp	526	132	4.65	4.66	0.0173
shooting	418	105	3.69	3.71	0.0173
swimming	540	135	4.77	4.77	0.0004
table_tennis	557	139	4.92	4.91	0.0093
tennis	562	141	4.97	4.98	0.0172
volleyball	552	138	4.88	4.88	0.0004
weight_lifting	448	112	3.96	3.96	0.0003
wrestling	472	118	4.17	4.17	0.0004
wwe	524	131	4.63	4.63	0.0004

Drift promedio: 0.0094%

Drift máximo: 0.0182%



Por lo visto las distribuciones entre el test y el conjunto de train, en la mayoría de clases no tienen un drift significativo

```

In [ ]: print('== Análisis de Drift de Píxeles ==')
print('Calculando estadísticas de píxeles (muestra de 50 imágenes por conjunto)...')

# Función para calcular estadísticas de píxeles de manera eficiente
def calcular_estadisticas_píxeles(indices, dataset, n_muestras=50):
    """Calcula media y std de píxeles RGB de una muestra del dataset"""
    muestra_indices = np.random.choice(indices, min(n_muestras, len(indices)), repl

```

```

    medias_r, medias_g, medias_b = [], [], []
    stds_r, stds_g, stds_b = [], [], []

    for idx in muestra_indices:
        img_path = dataset.samples[idx][0]
        try:
            # Leer imagen sin transformaciones para estadísticas originales
            from PIL import Image
            img = Image.open(img_path).convert('RGB')
            img_array = np.array(img).astype(np.float32) / 255.0

            medias_r.append(img_array[:, :, 0].mean())
            medias_g.append(img_array[:, :, 1].mean())
            medias_b.append(img_array[:, :, 2].mean())
            stds_r.append(img_array[:, :, 0].std())
            stds_g.append(img_array[:, :, 1].std())
            stds_b.append(img_array[:, :, 2].std())
        except Exception as e:
            continue

    return {
        'mean_r': np.mean(medias_r), 'mean_g': np.mean(medias_g), 'mean_b': np.mean(medias_b),
        'std_r': np.mean(stds_r), 'std_g': np.mean(stds_g), 'std_b': np.mean(stds_b)
    }

# Calcular estadísticas para train y test
train_stats = calcular_estadisticas_pixeles(train_indices, dataset)
test_stats = calcular_estadisticas_pixeles(test_indices, dataset)

# Calcular drift
print(f'{ "Estadística":<20} {"Train":<12} {"Test":<12} {"Drift Absoluto":<15} {"Dri')
print('-' * 75)

for stat in ['mean_r', 'mean_g', 'mean_b', 'std_r', 'std_g', 'std_b']:
    train_val = train_stats[stat]
    test_val = test_stats[stat]
    drift_abs = abs(train_val - test_val)
    drift_pct = 100 * drift_abs / (train_val + 1e-10)
    print(f'{stat:<20} {train_val:<12.6f} {test_val:<12.6f} {drift_abs:<15.6f} {dri')

# Calcular drift global
all_drifts = [abs(train_stats[s] - test_stats[s]) / (train_stats[s] + 1e-10) * 100
              for s in train_stats.keys()]
print(f'\nDrift promedio de píxeles: {np.mean(all_drifts):.4f}%')
print(f'Drift máximo de píxeles: {np.max(all_drifts):.4f}%')

# Visualización del drift de píxeles
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

# Comparación de medias RGB
canales = ['R', 'G', 'B']
train_means = [train_stats['mean_r'], train_stats['mean_g'], train_stats['mean_b']]
test_means = [test_stats['mean_r'], test_stats['mean_g'], test_stats['mean_b']]

x = np.arange(len(canales))

```

```

width = 0.35

ax1.bar(x - width/2, train_means, width, label='Train', alpha=0.8, color=['red', 'g
ax1.bar(x + width/2, test_means, width, label='Test', alpha=0.6, color=['darkred',
ax1.set_xlabel('Canal', fontsize=12)
ax1.set_ylabel('Media de Píxeles [0-1]', fontsize=12)
ax1.set_title('Comparación de Medias de Píxeles RGB', fontsize=14, fontweight='bold'
ax1.set_xticks(x)
ax1.set_xticklabels(canales)
ax1.legend()
ax1.grid(True, alpha=0.3)

# Comparación de desviaciones estándar
train_stds = [train_stats['std_r'], train_stats['std_g'], train_stats['std_b']]
test_stds = [test_stats['std_r'], test_stats['std_g'], test_stats['std_b']]

ax2.bar(x - width/2, train_stds, width, label='Train', alpha=0.8, color=['red', 'g
ax2.bar(x + width/2, test_stds, width, label='Test', alpha=0.6, color=['darkred',
ax2.set_xlabel('Canal', fontsize=12)
ax2.set_ylabel('Desviación Estándar [0-1]', fontsize=12)
ax2.set_title('Comparación de Desviaciones Estándar RGB', fontsize=14, fontweight='bold'
ax2.set_xticks(x)
ax2.set_xticklabels(canales)
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

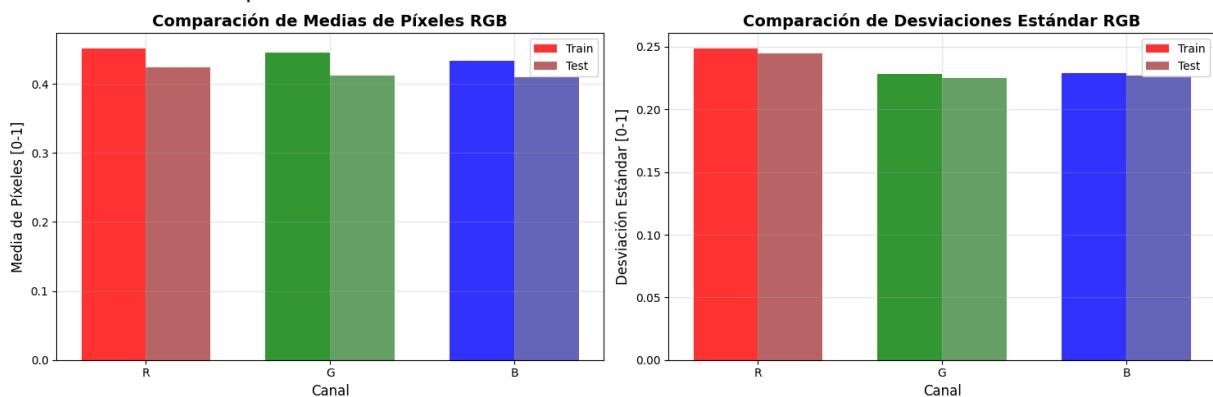
==== Análisis de Drift de Píxeles ===

Calculando estadísticas de píxeles (muestra de 50 imágenes por conjunto)...

Estadística	Train	Test	Drift Absoluto	Drift %
<hr/>				
mean_r	0.451029	0.424263	0.026766	5.9344
mean_g	0.445480	0.412353	0.033127	7.4362
mean_b	0.433185	0.409924	0.023261	5.3698
std_r	0.248482	0.244532	0.003950	1.5897
std_g	0.228080	0.225105	0.002975	1.3045
std_b	0.229111	0.226819	0.002292	1.0004

Drift promedio de píxeles: 3.7725%

Drift máximo de píxeles: 7.4362%



Vemos que existe un porcentaje de drift considerable en el verde, sin embargo vemos en la media estandar de los 3 colores que la del rojo es la que tiene el drift más alto sin embargo considero aceptable estos porcentajes.

Existe un drift moderado en el canal verde y azul en contraste con el rojo

## Modelos Disponibles: VGG16 y AlexNet con Transfer Learning

Implementaremos dos arquitecturas con las siguientes características:

### VGG16:

- Bloques convolucionales 1-3: **Congelados** si se quiere - extracción de características básicas
- Bloques convolucionales 4-5: **Entrenables** - ajuste fino para deportes
- Capas densas personalizadas con regularización L2 y Dropout
- Total: ~134M parámetros | Entrenables: ~132M

### AlexNet:

- Capas convolucionales 1-10: **Congeladas** si se quiere - extracción de características básicas
- Capas 11+: **Entrenables** - ajuste fino para deportes
- Capas densas personalizadas con Dropout
- Total: ~57M parámetros | Entrenables: ~57M

```
In [ ]: MODEL_TYPE = 'alexnet' # Opciones: 'vgg16' o 'alexnet'

import torch
import torch.nn as nn
import torchvision.models as models # (no se usa para pesos; se mantiene por compa

# =====
# Utilidades de congelación
# =====
def freeze_first_n_blocks_sequential(features: nn.Sequential, n_blocks: int):
    """
    Congela parámetros de las primeras n_blocks regiones separadas por MaxPool2d.
    Se recorre 'features' y se incrementa el contador al encontrar MaxPool2d.
    """
    if n_blocks <= 0:
        return
    blocks_frozen = 0
    for layer in features:
        # Mientras no hayamos completado los n bloques, se congela este layer
        if blocks_frozen < n_blocks:
            for p in layer.parameters():
                p.requires_grad = False
        # Al encontrar un MaxPool, consideramos que terminó un bloque
        if isinstance(layer, nn.MaxPool2d):
```

```

        blocks_frozen += 1
    if blocks_frozen >= n_blocks:
        # El resto de capas quedan entrenables
        continue

# =====
# VGG16 desde cero (custom)
# =====
class VGG16Custom(nn.Module):
    """
    Implementación de VGG16 (configuración D) desde cero.
    Entrada esperada: 224x224 RGB.
    """

    def __init__(self, num_classes: int = 22):
        super().__init__()
        self.features = nn.Sequential(
            # Bloque 1
            nn.Conv2d(3, 64, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2), # /2

            # Bloque 2
            nn.Conv2d(64, 128, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(128, 128, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2), # /2

            # Bloque 3
            nn.Conv2d(128, 256, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2), # /2

            # Bloque 4
            nn.Conv2d(256, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2), # /2

            # Bloque 5
            nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2), # /2
        )

        # 224 -> 112 -> 56 -> 28 -> 14 -> 7
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(0.3),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(0.5),
            nn.Linear(4096, num_classes),

```

```

        )

    self._init_weights()

    def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

    def create_vgg16_model(num_classes=22, freeze_blocks=3):
        """
        Crea modelo VGG16 desde cero con opción de congelar los primeros 'freeze_blocks'.
        Mantiene el nombre de la función original.
        """
        vgg16 = VGG16Custom(num_classes=num_classes)
        # Congelar bloques 1..freeze_blocks (cada bloque termina en MaxPool)
        if freeze_blocks and freeze_blocks > 0:
            freeze_first_n_blocks_sequential(vgg16.features, freeze_blocks)
        return vgg16

# =====
# AlexNet desde cero (custom)
# =====
class AlexNetCustom(nn.Module):
    """
    Implementación completa de AlexNet desde cero (Krizhevsky et al., 2012).
    Entrada esperada: 224x224 RGB.
    """
    def __init__(self, num_classes=22):
        super().__init__()
        self.features = nn.Sequential(
            # Conv1: 224 -> 55
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),

            # Conv2: 55 -> 27
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),

            # Conv3: 27 -> 27
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

```

```

# Conv4: 27 -> 27
nn.Conv2d(384, 256, kernel_size=3, padding=1),
nn.ReLU(inplace=True),

# Conv5: 27 -> 13
nn.Conv2d(256, 256, kernel_size=3, padding=1),
nn.ReLU(inplace=True),
nn.MaxPool2d(kernel_size=3, stride=2),
)
self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
self.classifier = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(256 * 6 * 6, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(0.3),
    nn.Linear(4096, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(0.5),
    nn.Linear(4096, num_classes),
)
self._init_weights()

def _init_weights(self):
    for m in self.modules():
        if isinstance(m, nn.Conv2d):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0, 0.01)
            nn.init.constant_(m.bias, 0)

def forward(self, x):
    x = self.features(x)
    x = self.avgpool(x)
    x = torch.flatten(x, 1) # 256*6*6 = 9216
    x = self.classifier(x)
    return x

def create_alexnet_model(num_classes=22, freeze_blocks=3):
    """
    Crea modelo AlexNet desde cero con opción de congelar los primeros 'freeze_blocks'.
    Mantiene el nombre de la función original.
    """
    alexnet = AlexNetCustom(num_classes=num_classes)
    if freeze_blocks and freeze_blocks > 0:
        freeze_first_n_blocks_sequential(alexnet.features, freeze_blocks)
    return alexnet

# Crear el modelo según la configuración
print(f'\n==== Creando modelo {MODEL_TYPE.upper()} con Transfer Learning ===')

if MODEL_TYPE.lower() == 'vgg16':
    model = create_vgg16_model(num_classes=len(dataset.classes), freeze_blocks=3)
    model_name = 'VGG16'

```

```

    elif MODEL_TYPE.lower() == 'alexnet':
        model = create_alexnet_model(num_classes=len(dataset.classes), freeze_blocks=0)
        model_name = 'AlexNet'
    else:
        raise ValueError(f"MODEL_TYPE debe ser 'vgg16' o 'alexnet', no '{MODEL_TYPE}'")

model = model.to(device)

# Contar parámetros entrenables y no entrenables
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
frozen_params = total_params - trainable_params

print(f'\n==== Arquitectura del Modelo ====')
print(f'Modelo: {model_name}')
print(f'\nNúmero total de parámetros: {total_params:,}')
print(f'Parámetros entrenables: {trainable_params:,}')
print(f'Parámetros congelados: {frozen_params:,}')
print(f'\nEstructura:')
if MODEL_TYPE.lower() == 'vgg16':
    print(f' - Bloques 1-3: CONGELADOS (extracción de características)')
    print(f' - Bloques 4-5: ENTRENABLES (ajuste fino)')
    print(f' - Clasificador: 25088 → 4096 → Dropout(0.3) → 4096 → Dropout(0.5) → {')
else:
    print(f' - Capas 1-10: CONGELADAS (extracción de características)')
    print(f' - Capas 11+: ENTRENABLES (ajuste fino)')
    print(f' - Clasificador: 9216 → Dropout(0.5) → 4096 → Dropout(0.3) → 4096 → Dr

```

==== Creando modelo ALEXNET con Transfer Learning ===

==== Arquitectura del Modelo ===

Modelo: AlexNet

Número total de parámetros: 57,093,974

Parámetros entrenables: 57,093,974

Parámetros congelados: 0

Estructura:

- Capas 1-10: CONGELADAS (extracción de características)
- Capas 11+: ENTRENABLES (ajuste fino)
- Clasificador: 9216 → Dropout(0.5) → 4096 → Dropout(0.3) → 4096 → Dropout(0.5) →

22

==== Arquitectura del Modelo ===

Modelo: AlexNet

Número total de parámetros: 57,093,974

Parámetros entrenables: 57,093,974

Parámetros congelados: 0

Estructura:

- Capas 1-10: CONGELADAS (extracción de características)
- Capas 11+: ENTRENABLES (ajuste fino)
- Clasificador: 9216 → Dropout(0.5) → 4096 → Dropout(0.3) → 4096 → Dropout(0.5) →

22

```
In [ ]: # Configuración de entrenamiento

# Hiperparámetros
BATCH_SIZE = 32
NUM_EPOCHS = 150
LEARNING_RATE = 1e-4

# Función de pérdida y optimizador con regularización L2
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)

# Calcular class weights para compensar desbalance
from collections import Counter
train_labels = [targets[i] for i in train_indices]
class_counts = Counter(train_labels)
max_count = float(max(class_counts.values()))
class_weights_dict = {class_id: max_count / num_images for class_id, num_images in class_counts.items()}
class_weights_tensor = torch.tensor([class_weights_dict[i] for i in range(len(datasets))])

# Actualizar criterion con class weights
criterion = nn.CrossEntropyLoss(weight=class_weights_tensor)

# Learning Rate Scheduler: reduce LR cuando test loss no mejora
from torch.optim.lr_scheduler import ReduceLROnPlateau
import copy

scheduler = ReduceLROnPlateau(
    optimizer,
    mode='min',           # minimizar test_loss
    factor=0.5,            # reducir LR a la mitad
    patience=8,             # esperar 8 épocas sin mejora
    verbose=True,           # imprimir cuando se reduzca LR
    min_lr=1e-7              # LR mínimo
)

# Checkpointing: guardar mejor modelo (por test_accuracy)
best_test_acc = 0.0
best_model_wts = copy.deepcopy(model.state_dict())
best_epoch = 0
save_best_path = f'{MODEL_TYPE}_best_checkpoint.pth'

early_stopping_patience = 15 # detener después de 15 épocas sin mejora
no_improve_epochs = 0

print(f'\n==== Configuración de Entrenamiento ===')
print(f'Modelo: {model_name}')
print(f'Batch size: {BATCH_SIZE}')
print(f'Épocas: {NUM_EPOCHS}')
print(f'Learning rate: {LEARNING_RATE}')
print(f'Optimizador: Adam con weight_decay=0')
print(f'Función de pérdida: CrossEntropyLoss con class weights')
print(f'\n==== Mejoras Aplicadas ===')
print(f'✓ ReduceLROnPlateau: factor=0.5, patience=8')
```

```
print(f'✓ Checkpointing: guardando mejor modelo en {save_best_path}')
print(f'✓ Early Stopping: patience={early_stopping_patience} épocas')
```

```
\n==== Configuración de Entrenamiento ===
Modelo: AlexNet
Batch size: 32
Épocas: 150
Learning rate: 0.0001
Optimizador: Adam con weight_decay=0
Función de pérdida: CrossEntropyLoss con class weights
\n==== Mejoras Aplicadas ====
✓ ReduceLROnPlateau: factor=0.5, patience=8
✓ Checkpointing: guardando mejor modelo en alexnet_best_checkpoint.pth
✓ Early Stopping: patience=15 épocas
```

```
In [50]: device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Usando dispositivo: {device}')
```

```
Usando dispositivo: cuda
```

```
In [ ]: train_losses = []
train_accuracies = []
test_losses = []
test_accuracies = []

print(f'\n==== Iniciando Entrenamiento de {model_name} ====')
print(f'Total de épocas: {NUM_EPOCHS}')
print(f'Entrenamiento en: {device}\n')

start_time = time.time()

for epoch in range(NUM_EPOCHS):
    epoch_start = time.time()

    # ===== FASE DE ENTRENAMIENTO =====
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass y optimización
        loss.backward()
        optimizer.step()

        # Estadísticas
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

```

    if (batch_idx + 1) % 50 == 0:
        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}], Step [{batch_idx+1}/{len(train_loader)}]
              f'Loss: {loss.item():.4f}, Acc: {100*correct/total:.2f}%)')

    train_loss = running_loss / len(train_loader)
    train_accuracy = 100 * correct / total
    train_losses.append(train_loss)
    train_accuracies.append(train_accuracy)

model.eval()
test_loss = 0.0
correct = 0
total = 0

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)

        test_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    test_loss = test_loss / len(test_loader)
    test_accuracy = 100 * correct / total
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

epoch_time = time.time() - epoch_start

# ===== SCHEDULER: reducir LR si test_loss no mejora =====
scheduler.step(test_loss)

# ===== CHECKPOINTING: guardar mejor modelo =====
if test_accuracy > best_test_acc:
    best_test_acc = test_accuracy
    best_epoch = epoch + 1
    best_model_wts = copy.deepcopy(model.state_dict())
    torch.save({
        'epoch': best_epoch,
        'model_state_dict': best_model_wts,
        'optimizer_state_dict': optimizer.state_dict(),
        'test_accuracy': best_test_acc,
        'test_loss': test_loss,
        'train_accuracy': train_accuracy,
        'train_loss': train_loss,
        'class_names': dataset.classes,
        'model_type': MODEL_TYPE
    }, save_best_path)
    no_improve_epochs = 0
    print(f'✓ Nuevo mejor modelo guardado: Test Acc = {best_test_acc:.2f}% (época {best_epoch})')
else:

```

```

no_improve_epochs += 1

# Mostrar progreso cada época
print(f'\n{"*80}')
print(f'Época [{epoch+1}/{NUM_EPOCHS}] Completada')
print(f'{"*80}')
print(f'Train Loss: {train_loss:.4f} | Train Acc: {train_accuracy:.2f}%')
print(f'Test Loss: {test_loss:.4f} | Test Acc: {test_accuracy:.2f}%')
print(f'Gap (Overfitting): {train_accuracy - test_accuracy:.2f}%')
print(f'Mejor Test Acc hasta ahora: {best_test_acc:.2f}% (época {best_epoch})')
print(f'Épocas sin mejora: {no_improve_epochs}/{early_stopping_patience}')
print(f'Learning Rate actual: {optimizer.param_groups[0]["lr"]:.2e}')
print(f'Tiempo de época: {epoch_time:.2f}s')
print(f'{"*80}\n')

# ====== EARLY STOPPING ======
if no_improve_epochs >= early_stopping_patience:
    print(f'\n!{"*80}')
    print(f'{EARLY_STOPPING_ACTIVADO}^80')
    print(f'!{"*80}')
    print(f'No se observó mejora en Test Accuracy durante {early_stopping_patience}')
    print(f'Deteniendo entrenamiento en época {epoch+1}.')
    print(f'Mejor modelo: época {best_epoch} con Test Acc = {best_test_acc:.2f}')
    print(f'!{"*80}\n')
    break

total_time = time.time() - start_time

# Cargar mejor modelo para evaluación final
print(f'\n{"*80}')
print(f'Cargando mejor modelo (época {best_epoch}) para evaluación final...')
model.load_state_dict(best_model_wts)
print(f'{"*80}\n')

print(f'\n#"*80')
print(f'{ENTRENAMIENTO_COMPLETADO}^80')
print(f'#"*80')
print(f'Tiempo total: {total_time/60:.2f} minutos ({total_time:.2f} segundos)')
print(f'Épocas completadas: {len(train_losses)}')
print(f'Mejor Test Accuracy: {best_test_acc:.2f}% (época {best_epoch})')
print(f'Precisión final de entrenamiento: {train_accuracies[-1]:.2f}%')
print(f'Precisión final de prueba: {test_accuracies[-1]:.2f}%')
print(f'Gap final: {train_accuracies[-1] - test_accuracies[-1]:.2f}%')
print(f'#"*80\n')

```

==== Iniciando Entrenamiento de AlexNet ===

Total de épocas: 150

Entrenamiento en: cuda

```
Epoch [1/150], Step [50/354], Loss: 3.0912, Acc: 5.12%
Epoch [1/150], Step [50/354], Loss: 3.0912, Acc: 5.12%
Epoch [1/150], Step [100/354], Loss: 3.1110, Acc: 5.06%
Epoch [1/150], Step [100/354], Loss: 3.1110, Acc: 5.06%
Epoch [1/150], Step [150/354], Loss: 3.0636, Acc: 5.23%
Epoch [1/150], Step [150/354], Loss: 3.0636, Acc: 5.23%
Epoch [1/150], Step [200/354], Loss: 2.9660, Acc: 6.00%
Epoch [1/150], Step [200/354], Loss: 2.9660, Acc: 6.00%
Epoch [1/150], Step [250/354], Loss: 3.0736, Acc: 6.60%
Epoch [1/150], Step [250/354], Loss: 3.0736, Acc: 6.60%
Epoch [1/150], Step [300/354], Loss: 2.8892, Acc: 7.12%
Epoch [1/150], Step [300/354], Loss: 2.8892, Acc: 7.12%
Epoch [1/150], Step [350/354], Loss: 2.8521, Acc: 7.72%
Epoch [1/150], Step [350/354], Loss: 2.8521, Acc: 7.72%
✓ Nuevo mejor modelo guardado: Test Acc = 12.47% (época 1)
```

=====

Época [1/150] Completada

=====

```
Train Loss: 3.0027 | Train Acc: 7.75%
Test Loss: 2.8159 | Test Acc: 12.47%
Gap (Overfitting): -4.73%
Mejor Test Acc hasta ahora: 12.47% (época 1)
Épocas sin mejora: 0/15
Learning Rate actual: 1.00e-04
Tiempo de época: 169.90s
```

=====

✓ Nuevo mejor modelo guardado: Test Acc = 12.47% (época 1)

=====

Época [1/150] Completada

=====

```
Train Loss: 3.0027 | Train Acc: 7.75%
Test Loss: 2.8159 | Test Acc: 12.47%
Gap (Overfitting): -4.73%
Mejor Test Acc hasta ahora: 12.47% (época 1)
Épocas sin mejora: 0/15
Learning Rate actual: 1.00e-04
Tiempo de época: 169.90s
```

=====

```
Epoch [2/150], Step [50/354], Loss: 2.7886, Acc: 10.12%
Epoch [2/150], Step [50/354], Loss: 2.7886, Acc: 10.12%
Epoch [2/150], Step [100/354], Loss: 2.7413, Acc: 9.66%
Epoch [2/150], Step [100/354], Loss: 2.7413, Acc: 9.66%
Epoch [2/150], Step [150/354], Loss: 2.7252, Acc: 10.44%
Epoch [2/150], Step [150/354], Loss: 2.7252, Acc: 10.44%
Epoch [2/150], Step [200/354], Loss: 2.5579, Acc: 10.91%
Epoch [2/150], Step [200/354], Loss: 2.5579, Acc: 10.91%
Epoch [2/150], Step [250/354], Loss: 2.7336, Acc: 11.25%
Epoch [2/150], Step [250/354], Loss: 2.7336, Acc: 11.25%
```

```
Epoch [137/150], Step [200/354], Loss: 0.3732, Acc: 73.34%
Epoch [137/150], Step [200/354], Loss: 0.3732, Acc: 73.34%
Epoch [137/150], Step [250/354], Loss: 0.9580, Acc: 73.47%
Epoch [137/150], Step [250/354], Loss: 0.9580, Acc: 73.47%
Epoch [137/150], Step [300/354], Loss: 1.1061, Acc: 73.46%
Epoch [137/150], Step [300/354], Loss: 1.1061, Acc: 73.46%
Epoch [137/150], Step [350/354], Loss: 0.4923, Acc: 73.55%
Epoch [137/150], Step [350/354], Loss: 0.4923, Acc: 73.55%
```

```
=====
Época [137/150] Completada
=====
```

```
Train Loss: 0.8362 | Train Acc: 73.50%
Test Loss: 1.4432 | Test Acc: 66.43%
Gap (Overfitting): 7.07%
Mejor Test Acc hasta ahora: 66.82% (época 122)
Épocas sin mejora: 15/15
Learning Rate actual: 7.81e-07
Tiempo de época: 215.00s
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
EARLY STOPPING ACTIVADO
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
No se observó mejora en Test Accuracy durante 15 épocas consecutivas.
Deteniendo entrenamiento en época 137.
Mejor modelo: época 122 con Test Acc = 66.82%
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```
*****
Cargando mejor modelo (época 122) para evaluación final...
*****
```

```
#####
ENTRENAMIENTO COMPLETADO
#####
Tiempo total: 578.96 minutos (34737.67 segundos)
Épocas completadas: 137
Mejor Test Accuracy: 66.82% (época 122)
Precisión final de entrenamiento: 73.50%
Precisión final de prueba: 66.43%
Gap final: 7.07%
#####
```

```
=====
Época [137/150] Completada
=====
```

```
Train Loss: 0.8362 | Train Acc: 73.50%
Test Loss: 1.4432 | Test Acc: 66.43%
Gap (Overfitting): 7.07%
Mejor Test Acc hasta ahora: 66.82% (época 122)
Épocas sin mejora: 15/15
```

```
Learning Rate actual: 7.81e-07
```

```
Tiempo de época: 215.00s
```

```
=====
!!!!!!EARLY STOPPING ACTIVADO!!!!!!
=====
```

```
! No se observó mejora en Test Accuracy durante 15 épocas consecutivas.
```

```
Deteniendo entrenamiento en época 137.
```

```
Mejor modelo: época 122 con Test Acc = 66.82%
```

```
=====
*****
```

```
*****Cargando mejor modelo (época 122) para evaluación final...*****
```

```
#####
ENTRENAMIENTO COMPLETADO
#####
```

```
Tiempo total: 578.96 minutos (34737.67 segundos)
```

```
Épocas completadas: 137
```

```
Mejor Test Accuracy: 66.82% (época 122)
```

```
Precisión final de entrenamiento: 73.50%
```

```
Precisión final de prueba: 66.43%
```

```
Gap final: 7.07%
```

```
#####
*****
```

## a) Gráficas de Error y Precisión a lo Largo de las Iteraciones

Visualización del progreso del entrenamiento para identificar:

- Convergencia del modelo
- Presencia de sobreajuste (gap entre train y test)
- Estabilidad del entrenamiento

```
In [ ]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

epochs_range = range(1, len(train_losses) + 1)

# Gráfica de pérdida
ax1.plot(epochs_range, train_losses, 'b-', label='Pérdida de Entrenamiento', linewidth=2)
ax1.plot(epochs_range, test_losses, 'r-', label='Pérdida de Prueba', linewidth=2.5,
         ax1.set_xlabel('Época', fontsize=13, fontweight='bold')
         ax1.set_ylabel('Pérdida (Loss)', fontsize=13, fontweight='bold')
         ax1.set_title(f'Evolución de la Pérdida - {model_name}', fontsize=15, fontweight='bold')
         ax1.legend(fontsize=11, loc='best')
         ax1.grid(True, alpha=0.3, linestyle='--')
         ax1.set_xlim(1, len(train_losses))

# Gráfica de precisión
```

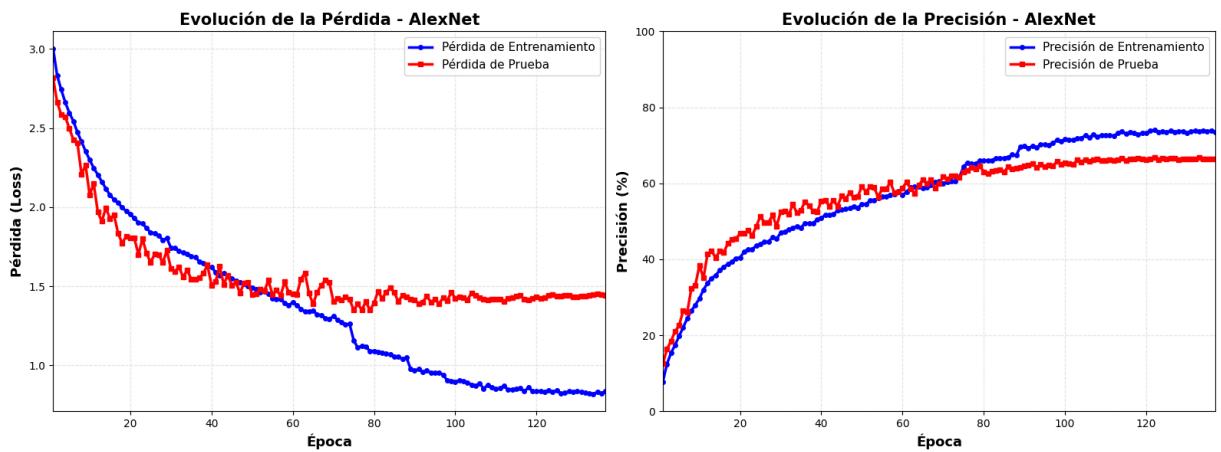
```

ax2.plot(epochs_range, train_accuracies, 'b-', label='Precisión de Entrenamiento',
ax2.plot(epochs_range, test_accuracies, 'r-', label='Precisión de Prueba', linewidth=2)
ax2.set_xlabel('Época', fontsize=13, fontweight='bold')
ax2.set_ylabel('Precisión (%)', fontsize=13, fontweight='bold')
ax2.set_title(f'Evolución de la Precisión - {model_name}', fontsize=15, fontweight='bold')
ax2.legend(fontsize=11, loc='best')
ax2.grid(True, alpha=0.3, linestyle='--')
ax2.set_xlim(1, len(train_accuracies))
ax2.set_ylim(0, 100)

plt.tight_layout()
plt.show()

print(f'\n{"*80}')
print(f'{"RESUMEN DE MÉTRICAS FINALES":^80}')
print(f'{"\n*80"}')
print(f'Modelo: {model_name}')
print(f'b) Precisión final de entrenamiento: {train_accuracies[-1]:.2f}%')
print(f'b) Precisión final de prueba: {test_accuracies[-1]:.2f}%')
print(f'c) Gap (Train - Test): {train_accuracies[-1] - test_accuracies[-1]:.2f}%')
print(f'\n    Pérdida final de entrenamiento: {train_losses[-1]:.4f}')
print(f'    Pérdida final de prueba: {test_losses[-1]:.4f}')
print(f'c) Tiempo total de simulación: {total_time/60:.2f} minutos ({total_time:.2f} segundos)')
print(f'c) Tiempo promedio por época: {total_time/len(train_losses):.2f} segundos')
print(f'{"\n*80"}\n')

```




---

#### RESUMEN DE MÉTRICAS FINALES

---

- Modelo: AlexNet  
 b) Precisión final de entrenamiento: 73.50%  
 b) Precisión final de prueba: 66.43%  
 Gap (Train - Test): 7.07%

Pérdida final de entrenamiento: 0.8362  
 Pérdida final de prueba: 1.4432

- c) Tiempo total de simulación: 578.96 minutos (34737.67 segundos)  
 Tiempo promedio por época: 253.56 segundos
-

```
In [ ]: # d) y e) Análisis de predicciones: Desempeño excelente vs pobre

print('\n==== Evaluando modelo en conjunto de prueba ===')
model.eval()

# Almacenar todas las predicciones
all_predictions = []
all_labels = []
all_confidences = []
all_images = []

with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        probs = torch.nn.functional.softmax(outputs, dim=1)
        confidences, predictions = torch.max(probs, 1)

        all_predictions.extend(predictions.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
        all_confidences.extend(confidences.cpu().numpy())
        all_images.extend(images.cpu())

all_predictions = np.array(all_predictions)
all_labels = np.array(all_labels)
all_confidences = np.array(all_confidences)

correct_mask = all_predictions == all_labels
incorrect_mask = ~correct_mask

correct_indices = np.where(correct_mask)[0]
incorrect_indices = np.where(incorrect_mask)[0]

# d) Mejores predicciones (alta confianza + correctas)
if len(correct_indices) > 0:
    correct_confidences = all_confidences[correct_indices]
    best_indices = correct_indices[np.argsort(correct_confidences)[-6:]] # Top 6

    print(f'\n{"*80"}')
    print(f'd) DESEMPEÑO EXCELENTE - Predicciones con alta confianza (correctas)')
    print(f'{{"*80"}')

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    axes = axes.flatten()

    for idx, img_idx in enumerate(best_indices):
        img = all_images[img_idx].numpy().transpose(1, 2, 0)
        img = img * np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456, 0.406])
        img = np.clip(img, 0, 1)

        pred_class = dataset.classes[all_predictions[img_idx]]
        true_class = dataset.classes[all_labels[img_idx]]
        confidence = all_confidences[img_idx]

        axes[idx].imshow(img)
```

```

        axes[idx].set_title(f'Predicción: {pred_class}\n'
                            f'Real: {true_class}\n'
                            f'Confianza: {confidence:.2%}' ,
                            fontsize=11, color='green', fontweight='bold')
        axes[idx].axis('off')

        print(f'{idx+1}. Clase: {pred_class:15s} | Confianza: {confidence:.4f} ({co

plt.suptitle(f'Ejemplos de Desempeño EXCELENTE - {model_name}', fontsize=16, fo
plt.tight_layout()
plt.show()

# e) Peores predicciones (incorrectas con alta confianza en error)
if len(incorrect_indices) > 0:
    incorrect_confidences = all_confidences[incorrect_indices]
    worst_indices = incorrect_indices[np.argsort(incorrect_confidences)[-6:]] # Er

    print(f'\n{"*80}')
    print(f'e) DESEMPEÑO POBRE - Predicciones incorrectas con alta confianza')
    print(f'{"*80}')

    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    axes = axes.flatten()

    for idx, img_idx in enumerate(worst_indices):
        img = all_images[img_idx].numpy().transpose(1, 2, 0)
        img = img * np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456, 0.406
        img = np.clip(img, 0, 1)

        pred_class = dataset.classes[all_predictions[img_idx]]
        true_class = dataset.classes[all_labels[img_idx]]
        confidence = all_confidences[img_idx]

        axes[idx].imshow(img)
        axes[idx].set_title(f'Predicción: {pred_class}\n'
                            f'Real: {true_class}\n'
                            f'Confianza: {confidence:.2%}' ,
                            fontsize=11, color='red', fontweight='bold')
        axes[idx].axis('off')

        print(f'{idx+1}. Pred: {pred_class:15s} | Real: {true_class:15s} | Confianz

plt.suptitle(f'Ejemplos de Desempeño POBRE - {model_name}', fontsize=16, fontwe
plt.tight_layout()
plt.show()

print(f'\n{"*80}')
print(f'Total de predicciones correctas: {correct_mask.sum()} / {len(all_labels)} ('
print(f'Total de predicciones incorrectas: {incorrect_mask.sum()} / {len(all_labels
print(f'{"*80}\n')

```

==== Evaluando modelo en conjunto de prueba ===

=====  
d) DESEMPEÑO EXCELENTE - Predicciones con alta confianza (correctas)  
=====

1. Clase: ice_hockey	Confianza: 1.0000 (100.00%)
2. Clase: kabaddi	Confianza: 1.0000 (100.00%)
3. Clase: swimming	Confianza: 1.0000 (100.00%)
4. Clase: kabaddi	Confianza: 1.0000 (100.00%)
5. Clase: kabaddi	Confianza: 1.0000 (100.00%)
6. Clase: baseball	Confianza: 1.0000 (100.00%)

=====  
d) DESEMPEÑO EXCELENTE - Predicciones con alta confianza (correctas)  
=====

1. Clase: ice_hockey	Confianza: 1.0000 (100.00%)
2. Clase: kabaddi	Confianza: 1.0000 (100.00%)
3. Clase: swimming	Confianza: 1.0000 (100.00%)
4. Clase: kabaddi	Confianza: 1.0000 (100.00%)
5. Clase: kabaddi	Confianza: 1.0000 (100.00%)
6. Clase: baseball	Confianza: 1.0000 (100.00%)

Ejemplos de Desempeño EXCELENTE - AlexNet

Predicción: ice\_hockey  
Real: ice\_hockey  
Confianza: 100.00%



Predicción: kabaddi  
Real: kabaddi  
Confianza: 100.00%

Predicción: kabaddi  
Real: kabaddi  
Confianza: 100.00%



Predicción: kabaddi  
Real: kabaddi  
Confianza: 100.00%

Predicción: swimming  
Real: swimming  
Confianza: 100.00%



Predicción: baseball  
Real: baseball  
Confianza: 100.00%



=====  
e) DESEMPEÑO POBRE - Predicciones incorrectas con alta confianza  
=====

1. Pred: formula1	Real: weight_lifting	Confianza: 0.9999 (99.99%)
2. Pred: kabaddi	Real: formula1	Confianza: 1.0000 (100.00%)
3. Pred: kabaddi	Real: football	Confianza: 1.0000 (100.00%)
4. Pred: swimming	Real: chess	Confianza: 1.0000 (100.00%)
5. Pred: shooting	Real: tennis	Confianza: 1.0000 (100.00%)
6. Pred: shooting	Real: fencing	Confianza: 1.0000 (100.00%)

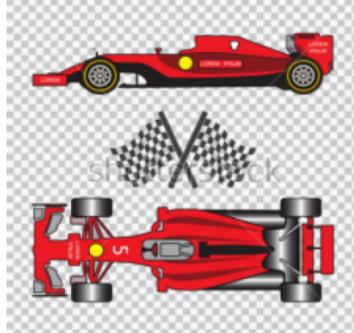
Ejemplos de Desempeño POBRE - AlexNet

**Predicción: formula1  
Real: weight\_lifting  
Confianza: 99.99%**



**Predicción: swimming**  
**Real: chess**  
**Confianza: 100.00%**

**Predicción: kabaddi  
Real: formula1  
Confianza: 100.00%**



**Predicción: shooting  
Real: tennis  
Confianza: 100.00%**

**Predicción: kabaddi**  
**Real: football**  
**Confianza: 100.00%**



**Predicción: shooting  
Real: fencing  
Confianza: 100.00%**

A collection of blue glass chess pieces (King, Queen, Rook, Bishop, Knight, and Pawn) arranged on a light-colored wooden chessboard. The pieces are translucent and have a slightly aged appearance.



Total de predicciones correctas: 1891 / 2830 (66.82%)  
Total de predicciones incorrectas: 939 / 2830 (33.18%)

```

    digits=4))

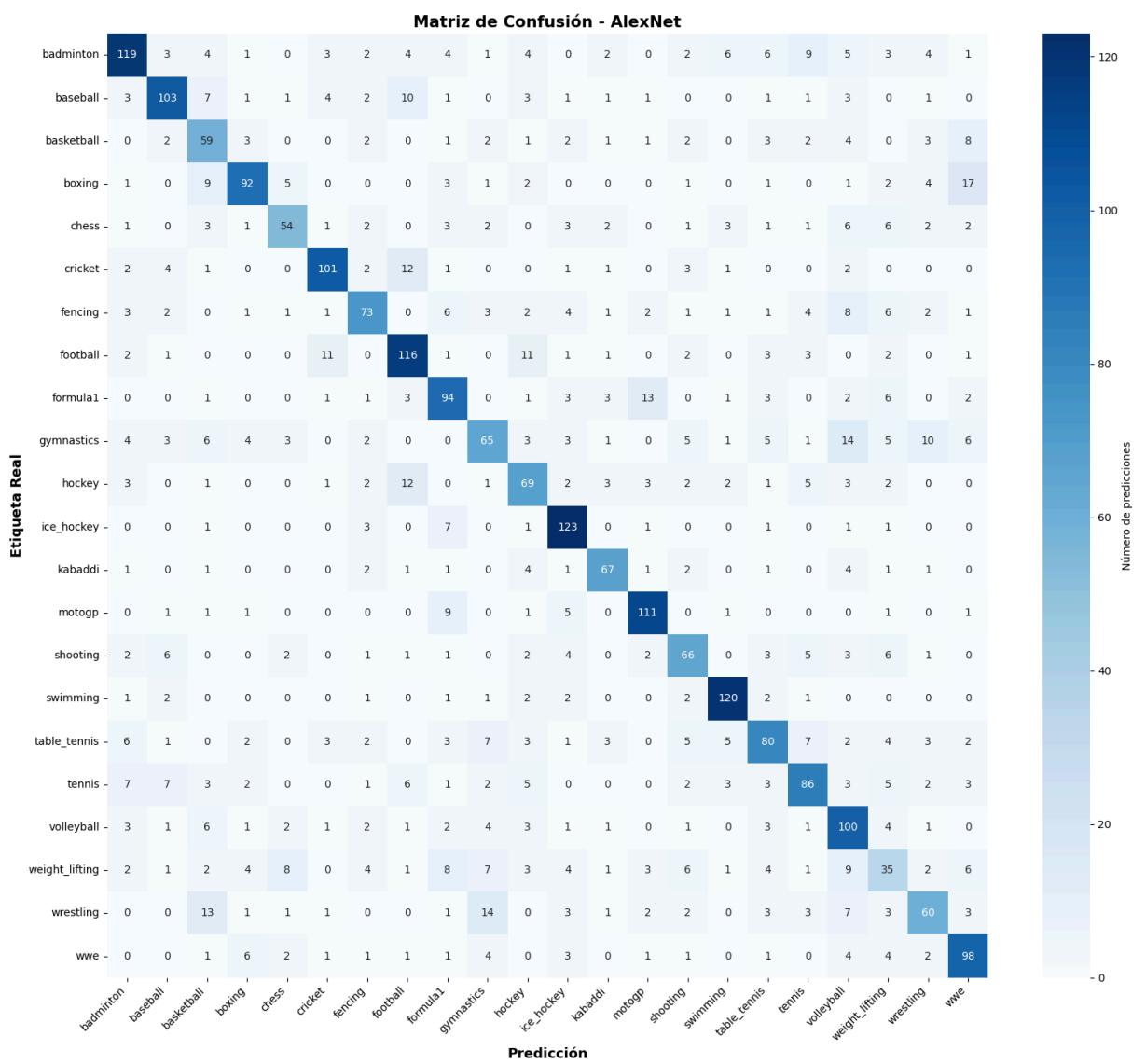
# Identificar clases con mejor y peor desempeño
from sklearn.metrics import precision_recall_fscore_support

precision, recall, f1, support = precision_recall_fscore_support(all_labels, all_pr

print('\n== TOP 5 CLASES CON MEJOR F1-SCORE ==')
best_classes = np.argsort(f1)[-5:][::-1]
for idx in best_classes:
    print(f'{dataset.classes[idx]:20s} | F1: {f1[idx]:.4f} | Precision: {precision[

print('\n== TOP 5 CLASES CON PEOR F1-SCORE ==')
worst_classes = np.argsort(f1)[:5]
for idx in worst_classes:
    print(f'{dataset.classes[idx]:20s} | F1: {f1[idx]:.4f} | Precision: {precision[

```



==== REPORTE DE CLASIFICACIÓN POR CLASE (AlexNet) ===

	precision	recall	f1-score	support
badminton	0.7438	0.6503	0.6939	183
baseball	0.7518	0.7153	0.7331	144
basketball	0.4958	0.6146	0.5488	96
boxing	0.7667	0.6619	0.7104	139
chess	0.6835	0.5745	0.6243	94
cricket	0.7829	0.7710	0.7769	131
fencing	0.6952	0.5935	0.6404	123
football	0.6905	0.7484	0.7183	155
formula1	0.6309	0.7015	0.6643	134
gymnastics	0.5702	0.4610	0.5098	141
hockey	0.5750	0.6161	0.5948	112
ice_hockey	0.7365	0.8849	0.8039	139
kabaddi	0.7528	0.7614	0.7571	88
motogp	0.7872	0.8409	0.8132	132
shooting	0.6226	0.6286	0.6256	105
swimming	0.8276	0.8889	0.8571	135
table_tennis	0.6349	0.5755	0.6038	139
tennis	0.6615	0.6099	0.6347	141
volleyball	0.5525	0.7246	0.6270	138
weight_lifting	0.3646	0.3125	0.3365	112
wrestling	0.6122	0.5085	0.5556	118
wwe	0.6490	0.7481	0.6950	131
accuracy			0.6682	2830
macro avg	0.6631	0.6633	0.6602	2830
weighted avg	0.6688	0.6682	0.6655	2830

==== TOP 5 CLASES CON MEJOR F1-SCORE ===

swimming	F1: 0.8571   Precision: 0.8276   Recall: 0.8889
motogp	F1: 0.8132   Precision: 0.7872   Recall: 0.8409
ice_hockey	F1: 0.8039   Precision: 0.7365   Recall: 0.8849
cricket	F1: 0.7769   Precision: 0.7829   Recall: 0.7710
kabaddi	F1: 0.7571   Precision: 0.7528   Recall: 0.7614

==== TOP 5 CLASES CON PEOR F1-SCORE ===

weight_lifting	F1: 0.3365   Precision: 0.3646   Recall: 0.3125
gymnastics	F1: 0.5098   Precision: 0.5702   Recall: 0.4610
basketball	F1: 0.5488   Precision: 0.4958   Recall: 0.6146
wrestling	F1: 0.5556   Precision: 0.6122   Recall: 0.5085
hockey	F1: 0.5948   Precision: 0.5750   Recall: 0.6161

```
In [ ]: model_save_path = f'{MODEL_TYPE}_sports_classifier.pth'
torch.save({
    'epoch': len(train_losses),
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'train_loss': train_losses[-1],
    'test_loss': test_losses[-1],
    'train_accuracy': train_accuracies[-1],
    'test_accuracy': test_accuracies[-1],
    'class_names': dataset.classes,
```

```

        'model_type': MODEL_TYPE
    }, model_save_path)

print(f'✓ Modelo guardado en: {model_save_path}')

# Resumen final completo
print(f'\n{"#"*100}')
print(f'{"RESUMEN FINAL DEL PROYECTO":^100}')
print(f'{"#"*100}')
print(f'\n{"ARQUITECTURA":^100}')
print(f'{"-*100}')
print(f'Modelo: {model_name}')
print(f'Parámetros totales: {total_params:,}')
print(f'Parámetros entrenables: {trainable_params:,}')
print(f'Parámetros congelados: {frozen_params:,}')
if MODEL_TYPE.lower() == 'vgg16':
    print(f'Bloques congelados: 1-3 (extracción de características básicas)')
    print(f'Bloques entrenables: 4-5 + clasificador personalizado')
else:
    print(f'Capas congeladas: 1-10 (extracción de características básicas)')
    print(f'Capas entrenables: 11+ + clasificador personalizado')

print(f'\n{"HIPERPARÁMETROS":^100}')
print(f'{"-*100}')
print(f'Épocas: {len(train_losses)}')
print(f'Batch size: {BATCH_SIZE}')
print(f'Learning rate: {LEARNING_RATE}')
print(f'Optimizador: Adam')
print(f'Función de pérdida: CrossEntropyLoss con class weights')
print(f'Regularización: Dropout(0.3) + Dropout(0.5)')

print(f'\n{"MÉTRICAS FINALES":^100}')
print(f'{"-*100}')
print(f'Precisión de entrenamiento: {train_accuracies[-1]:.2f}%')
print(f'Precisión de prueba: {test_accuracies[-1]:.2f}%')
print(f'Gap (Overfitting): {train_accuracies[-1] - test_accuracies[-1]:.2f}%')
print(f'Pérdida de entrenamiento: {train_losses[-1]:.4f}')
print(f'Pérdida de prueba: {test_losses[-1]:.4f}')

print(f'\n{"TIEMPO DE ENTRENAMIENTO":^100}')
print(f'{"-*100}')
print(f'Tiempo total: {total_time/60:.2f} minutos ({total_time:.2f} segundos)')
print(f'Tiempo promedio por época: {total_time/len(train_losses):.2f} segundos')

print(f'\n{"DATASET":^100}')
print(f'{"-*100}')
print(f'Total de imágenes: {len(dataset)}')
print(f'Número de clases: {len(dataset.classes)}')
print(f'Imágenes de entrenamiento: {len(train_indices)} ({100*len(train_indices)/len(dataset)}%)')
print(f'Imágenes de prueba: {len(test_indices)} ({100*len(test_indices)/len(dataset)}%)')
print(f'Clases: {", ".join(dataset.classes)}')
print(f'\n{"#"*100}\n')

```

✓ Modelo guardado en: alexnet\_sports\_classifier.pth

```
#####
##### RESUMEN FINAL DEL PROYECTO #####
#####
```

## ARQUITECTURA

---

Modelo: AlexNet

Parámetros totales: 57,093,974

Parámetros entrenables: 57,093,974

Parámetros congelados: 0

Capas congeladas: 1-10 (extracción de características básicas)

Capas entrenables: 11+ + clasificador personalizado

## HIPERPARÁMETROS

---

Épocas: 137

Batch size: 32

Learning rate: 0.0001

Optimizador: Adam

Función de pérdida: CrossEntropyLoss con class weights

Regularización: Dropout(0.3) + Dropout(0.5)

## MÉTRICAS FINALES

---

Precisión de entrenamiento: 73.50%

Precisión de prueba: 66.43%

Gap (Overfitting): 7.07%

Pérdida de entrenamiento: 0.8362

Pérdida de prueba: 1.4432

## TIEMPO DE ENTRENAMIENTO

---

Tiempo total: 578.96 minutos (34737.67 segundos)

Tiempo promedio por época: 253.56 segundos

## DATASET

---

Total de imágenes: 14149

Número de clases: 22

Imágenes de entrenamiento: 11319 (80.0%)

Imágenes de prueba: 2830 (20.0%)

Clases: badminton, baseball, basketball, boxing, chess, cricket, fencing, football, formula1, gymnastics, hockey, ice\_hockey, kabaddi, motogp, shooting, swimming, table\_tennis, tennis, volleyball, weight\_lifting, wrestling, wwe

```
#####
#####
```

```
#####
```

Observo que el problema principalmente es el sobreajuste, ya que la precisión en el conjunto de entrenamiento es significativamente mayor que en el conjunto de prueba. Esto indica que el modelo está aprendiendo demasiado bien los datos de entrenamiento, pero no generaliza bien a datos no vistos.

Es complicado entrenar la red neuoronal con un conjunto de datos relativamente pequeño y diverso como este, lo que puede llevar a que el modelo memorice los datos de entrenamiento en lugar de aprender patrones generales.

```
In [ ]: !jupyter nbconvert --to html Tarea3.ipynb
```