

## Tarea IV

### AdvNNs

17 de noviembre de 2025

#### Resumen

El objetivo de esta tarea es continuar trabajando en el proyecto final. Por favor, suba sus soluciones en un archivo comprimido a Classroom antes del 25 de noviembre.

#### Problema 1

Describa los siguientes puntos clave de su proyecto (puede agregar algunos gráficos a su respuesta):

a) ¿Su modelo presenta un alto sesgo (high bias)? Si es así, ¿cómo planea lidiar con ello?

Mi modelo logra una precisión del 75.82% en el conjunto de entrenamiento, lo que quiere decir que está aprendiendo patrones interesantes de los datos, sin embargo la complejidad de 22 clases hace del problema un problema que es complejo y difícil de entrenar lo que podría probar para mejorar es:

- Probar una arquitectura más compleja como un VGT, por ejemplo.
- Incrementar la cantidad de los datos haciendo un aumento de datos o buscando más imágenes de deportes en internet para el entendimiento de patrones mejorado para mi modelo.
- Mejorar el procesamiento, quizás aplicar más técnicas de aumento de datos.

b) ¿Su modelo presenta una alta varianza (high variance)? Si es así, ¿cómo planea lidiar con ello?

Entrenamiento - Pérdida: 0.7568, Precisión: 76.02% Prueba - Pérdida: 1.4026, Precisión: 66.82% Gap de precisión (train - test): 9.20%

Estos fueron los resultados obtenidos en la última del mejor checkpoint guardado, lo que indica que si hay una varianza moderada, para lidiar con ello podría probar:

- Hacer nuevas técnicas de aumento de datos.
- Probar más regularización, al momento solo estoy usando dropout puedo probar weight decay.
- Probar con arquitecturas más simples aunque eso contradice un poco con el punto anterior, vale la pena probar diferentes arquitecturas.

c) ¿Sus conjuntos de datos de entrenamiento y prueba muestran una distribución similar? Si no es así, ¿cómo procederá?

Ambas distribuciones en términos del target son muy similares, sin embargo en los diversos canales de color y textura se observa un drift considerable entre entrenamiento y prueba (por ejemplo, drift máximo de hasta 126% en la media del canal verde). Esto puede ser debido a las técnicas de aumento de datos que aplique, para proceder podría:

- Revisar y ajustar las técnicas de aumento de datos para que no generen tanta discrepancia entre ambos conjuntos.
- Ajustar de ser necesario la técnica de partición de los datos para que sean más similares.

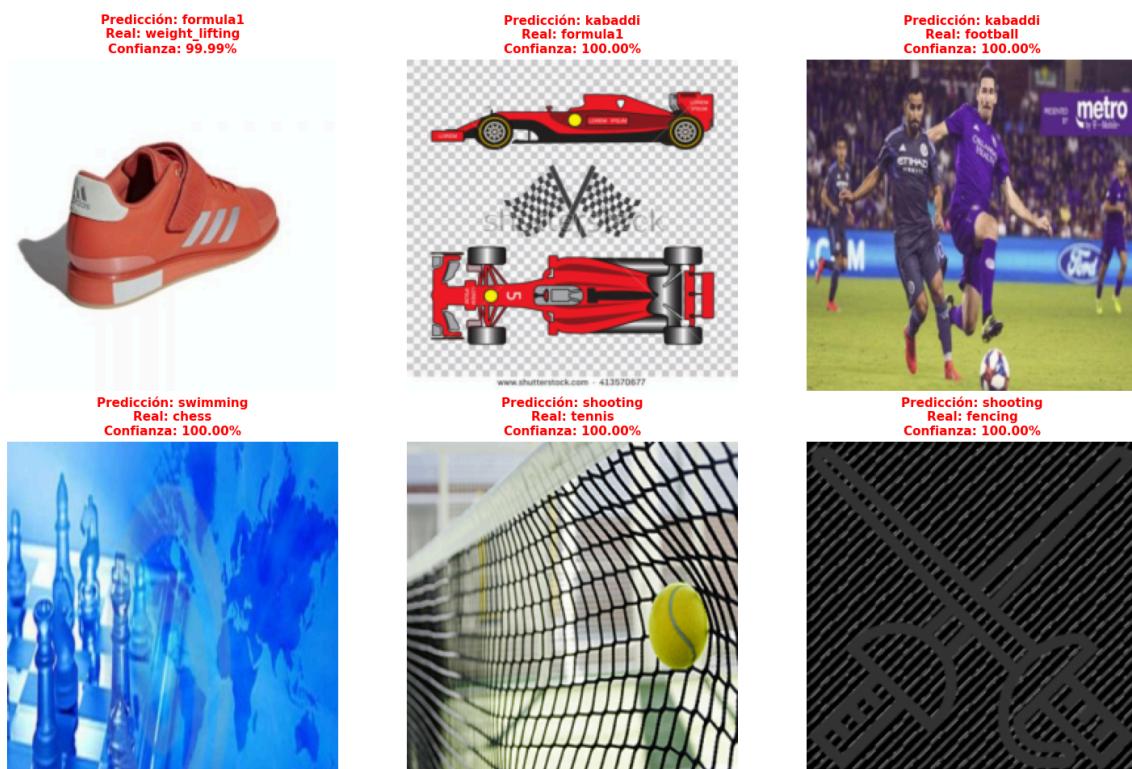
d) ¿Hay valores atípicos (outliers) en su conjunto de datos? Si es así, ¿cómo planea identificarlos y eliminarlos?

Generalmente no he identificado outliers evidentes en las imágenes, normalmente quizás es un mal etiquetado o mala representación de la clase por ejemplo hay una imagen que es un ajedrez pero com es muy azul se confunde con la etiqueta de natación, para proceder podría:

- Hacer revisión manuela
- Analizar estadísticamente su brillo, contraste y textura para identificar posibles outliers.
- Analizar imágenes que el modelo predice mal consistentemente para identificar posibles outliers.

e) Observe detenidamente los datos con una etiqueta de clasificación/pronóstico incorrecta. ¿Puede identificar algún patrón? Si es así, ¿cómo procederá?

#### Ejemplos de Desempeño POBRE - AlexNet



Claro que se ve un patron como el que mencione anteriormente, hay imagenes que por su coloracion o textura se confunden con otras clases, por ejemplo el kabbadi se puede confundir con futbol por la posicion de los jugadores, esa fotografía de lift weighting desde mi punto de vista no repsonda su etiqueta, es solo un tennis y como es rojo y blanco puede asociarlo con la F1 para proceder podria:

- Hacer una revisión manual de las imagenes mal clasificadas para ver si hay un mal etiquetado.
- Analizar estadisticamente las imagenes mal clasificadas para ver si hay algun patron en sus caracteristicas (brillo, contraste, color, textura, etc).
- Aumentar el conjunto de datos con imagenes similares a las mal clasificadas para mejorar el entendimiento del modelo.

f) ¿Ha identificado un repositorio con un modelo entrenado que pueda utilizar? Si es así, realice la implementación y muestre una comparación.

```
In [1]: 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 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_w
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_w

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=0.2),
    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_workers=4)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)

print('\n>> Data augmentation aplicado: train_transform definido con Resize, RandomResizedCrop, RandomHorizontalFlip y ColorJitter')
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\sports-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
imgs = np.clip(imgs, 0, 1)

```

```

fig, axes = plt.subplots(3, 4, figsize=(12, 9))
axes = axes.flatten()
for i, ax in enumerate(axes):
    if i >= imgs.shape[0]:
        ax.axis('off')
        continue
    ax.imshow(imgs[i])
    ax.set_title(f'Label: {labels[i].item()}')
    ax.axis('off')
plt.tight_layout()
plt.show()

```

== 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%

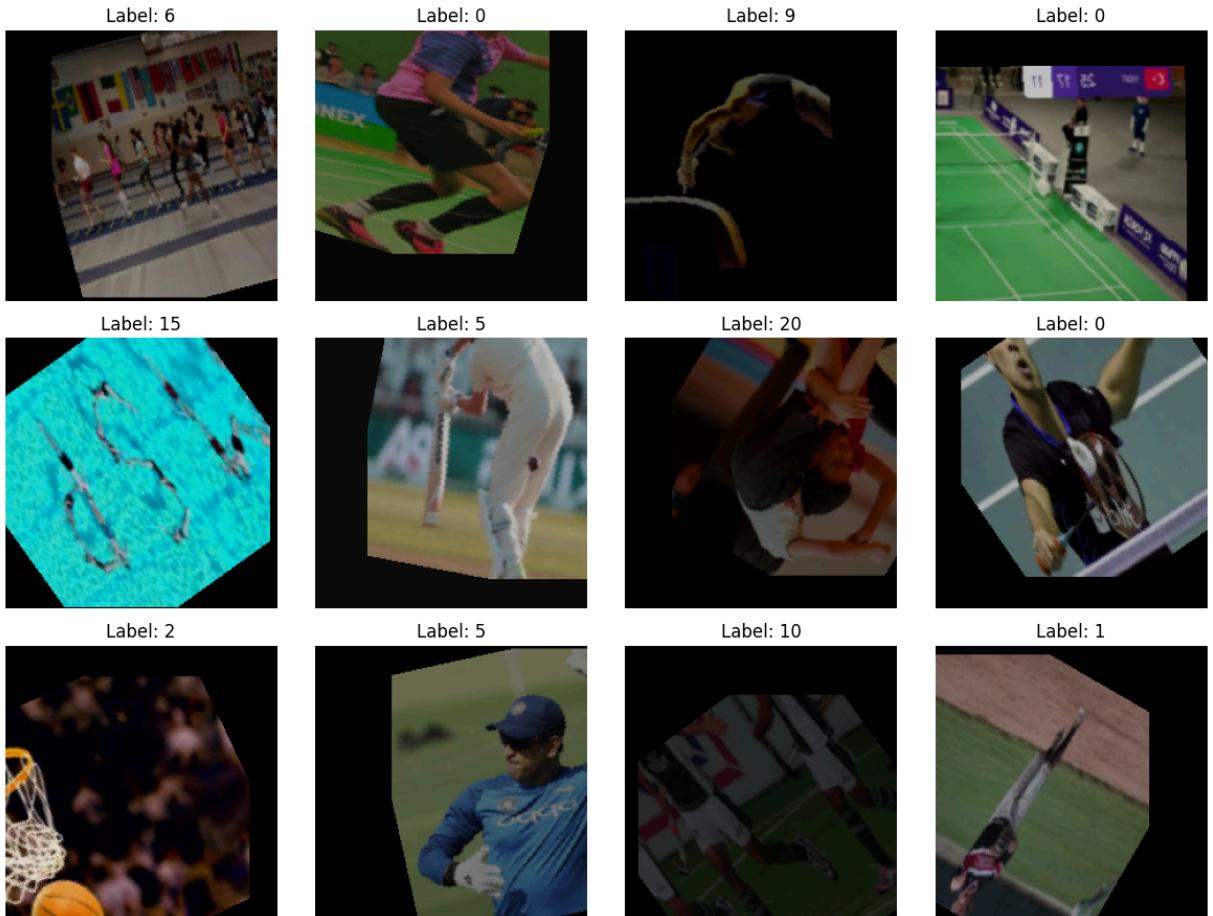
Mostrando 12 imágenes aumentadas de ejemplo (train)

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 [5]: 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}\n')

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)}')
ax2.legend()
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

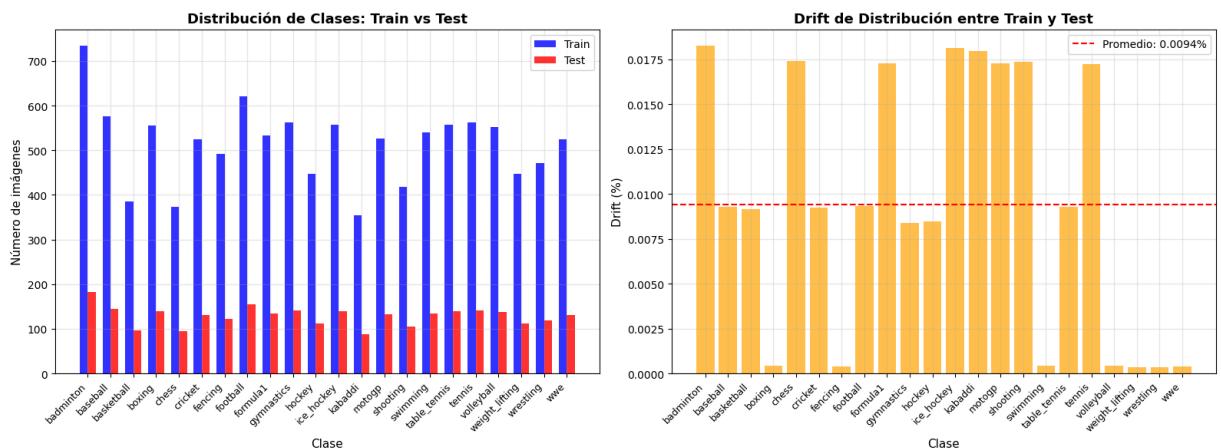
```

== Distribución de Clases ==

Clase	Train	Test	Train %	Test %	Drift %
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%



```
In [9]: MODEL_TYPE = 'vgg16'          # Opciones: 'vgg16' o 'alexnet'
USE_PRETRAINED = True               # True: Usar pesos de ImageNet | False: Entrenar desde cero
FREEZE_FIRST_N = 11                  # Número de capas a congelar (0 = todas entrenables, -1 =
                                         # para que solo las capas de la parte superior sean
                                         # entrenables)

import torch
import torch.nn as nn
import torchvision.models as models

print(f'\n== Configuración del Modelo ==')
print(f'Tipo de Modelo: {MODEL_TYPE.upper()}')
print(f'Pesos Pre-entrenados (ImageNet): {"SÍ" if USE_PRETRAINED else "NO"}')
print(f'Congelar primeras {FREEZE_FIRST_N} capas: {"SÍ" if FREEZE_FIRST_N > 0 else "No"}')
```

```

# 1. Cargar el modelo base
if MODEL_TYPE.lower() == 'vgg16':
    weights = models.VGG16_Weights.IMGNET1K_V1 if USE_PRETRAINED else None
    model = models.vgg16(weights=weights)

        # Reemplazar el clasificador completo con dropout personalizado (0.3 y 0.5)
    model.classifier = nn.Sequential(
        nn.Linear(512 * 7 * 7, 4096),
        nn.ReLU(True),
        nn.Dropout(0.3), # Dropout 1
        nn.Linear(4096, 4096),
        nn.ReLU(True),
        nn.Dropout(0.5), # Dropout 2
        nn.Linear(4096, len(dataset.classes))
    )

elif MODEL_TYPE.lower() == 'alexnet':
    weights = models.AlexNet_Weights.IMGNET1K_V1 if USE_PRETRAINED else None
    model = models.alexnet(weights=weights)

        # Reemplazar el clasificador con dropout personalizado
    model.classifier = nn.Sequential(
        nn.Dropout(0.3),
        nn.Linear(256 * 6 * 6, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(0.5),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, len(dataset.classes))
    )

else:
    raise ValueError(f"Modelo no soportado: {MODEL_TYPE}")

# 2. Congelar capas según configuración (Fine-tuning)
if USE_PRETRAINED and FREEZE_FIRST_N >= 0:
    # Listar todas las capas de features
    features_layers = list(model.features.children())

    if FREEZE_FIRST_N == -1:
        # Congelar todas las capas de features
        for param in model.features.parameters():
            param.requires_grad = False
        print(f">> Todas las capas de características congeladas.")

    elif FREEZE_FIRST_N > 0:
        # Congelar las primeras FREEZE_FIRST_N capas
        for i, layer in enumerate(features_layers):
            if i < FREEZE_FIRST_N:
                for param in layer.parameters():
                    param.requires_grad = False
        print(f">> Primeras {FREEZE_FIRST_N} capas congeladas (bloques 1-3).")
        print(f">> Capas {FREEZE_FIRST_N}+ entrenables (bloques 4-5 + clasificador)")

    # Mostrar estado de cada capa
    print(f'\n==== Estado de Capas (VGG16) ===')
    for i, layer in enumerate(model.features.children()):

```

```
trainable = any(p.requires_grad for p in layer.parameters()) if list(layer.parameters())
print(f'{i:2d} {layer.__class__.__name__:20s} Trainable: {trainable}')

model = model.to(device)

# 3. Resumen de parámetros
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==== Resumen de Parámetros ===')
print(f'Total: {total_params:,}')
print(f'Entrenables: {trainable_params:,}')
print(f'Congelados: {frozen_params:,}')
```

```
==== Configuración del Modelo ====
Tipo de Modelo: VGG16
Pesos Pre-entrenados (ImageNet): SÍ
Congelar primeras 11 capas: SÍ
>> Primeras 11 capas congeladas (bloques 1-3).
>> Capas 11+ entrenables (bloques 4-5 + clasificador).

==== Estado de Capas (VGG16) ====
0 Conv2d           Trainable: False
1 ReLU             Trainable: False
2 Conv2d           Trainable: False
3 ReLU             Trainable: False
4 MaxPool2d        Trainable: False
5 Conv2d           Trainable: False
6 ReLU             Trainable: False
7 Conv2d           Trainable: False
8 ReLU             Trainable: False
9 MaxPool2d        Trainable: False
10 Conv2d          Trainable: False
11 ReLU            Trainable: False
12 Conv2d          Trainable: True
13 ReLU            Trainable: False
14 Conv2d          Trainable: True
15 ReLU            Trainable: False
16 MaxPool2d        Trainable: False
17 Conv2d          Trainable: True
18 ReLU            Trainable: False
19 Conv2d          Trainable: True
20 ReLU            Trainable: False
21 Conv2d          Trainable: True
22 ReLU            Trainable: False
23 MaxPool2d        Trainable: False
24 Conv2d          Trainable: True
25 ReLU            Trainable: False
26 Conv2d          Trainable: True
27 ReLU            Trainable: False
28 Conv2d          Trainable: True
29 ReLU            Trainable: False
30 MaxPool2d        Trainable: False
>> Primeras 11 capas congeladas (bloques 1-3).
>> Capas 11+ entrenables (bloques 4-5 + clasificador).

==== Estado de Capas (VGG16) ====
0 Conv2d           Trainable: False
1 ReLU             Trainable: False
2 Conv2d           Trainable: False
3 ReLU             Trainable: False
4 MaxPool2d        Trainable: False
5 Conv2d           Trainable: False
6 ReLU             Trainable: False
7 Conv2d           Trainable: False
8 ReLU             Trainable: False
9 MaxPool2d        Trainable: False
10 Conv2d          Trainable: False
11 ReLU            Trainable: False
12 Conv2d          Trainable: True
```

```

13 ReLU           Trainable: False
14 Conv2d         Trainable: True
15 ReLU           Trainable: False
16 MaxPool2d      Trainable: False
17 Conv2d         Trainable: True
18 ReLU           Trainable: False
19 Conv2d         Trainable: True
20 ReLU           Trainable: False
21 Conv2d         Trainable: True
22 ReLU           Trainable: False
23 MaxPool2d      Trainable: False
24 Conv2d         Trainable: True
25 ReLU           Trainable: False
26 Conv2d         Trainable: True
27 ReLU           Trainable: False
28 Conv2d         Trainable: True
29 ReLU           Trainable: False
30 MaxPool2d      Trainable: False

```

==== Resumen de Parámetros ===

Total: 134,350,678  
 Entrenables: 133,795,350  
 Congelados: 555,328

==== Resumen de Parámetros ===

Total: 134,350,678  
 Entrenables: 133,795,350  
 Congelados: 555,328

In [12]: # Configuración de entrenamiento

```

# Hiperparámetros
BATCH_SIZE = 32
NUM_EPOCHS = 300
LEARNING_RATE = 1e-4

# Función de pérdida y optimizador con regularización L2
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE, weight_decay=1e-4)

# 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_weights_tensor = torch.tensor([class_weights_dict[i] for i in range(len(datas))

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

from torch.optim.lr_scheduler import ReduceLROnPlateau
import copy

# Learning Rate Scheduler: reduce LR cuando test loss no mejora

```

```

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_TYPE.upper()} (Pre-entrenado: {"SÍ" if USE_PRETRAINED else "No"})')
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: VGG16 (Pre-entrenado: SÍ)

Batch size: 32

Épocas: 300

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 vgg16\_best\_checkpoint.pth

✓ Early Stopping: patience=15 épocas

```
c:\Users\sergi\Documents\AdvNNs\.venv\Lib\site-packages\torch\optim\lr_scheduler.py:62: UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the learning rate.
  warnings.warn(
```

In [14]: device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')  
print(f'Usando dispositivo: {device}')

Usando dispositivo: cuda

In [15]: train\_losses = []  
train\_accuracies = []  
test\_losses = []  
test\_accuracies = []

```

print(f'\n==== Iniciando Entrenamiento de {MODEL_TYPE} ====')
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)}], 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} épocas')
        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 vgg16 ===

Total de épocas: 300

Entrenamiento en: cuda

Epoch [1/300], Step [50/354], Loss: 3.0044, Acc: 6.69%  
Epoch [1/300], Step [50/354], Loss: 3.0044, Acc: 6.69%  
Epoch [1/300], Step [100/354], Loss: 2.6118, Acc: 10.88%  
Epoch [1/300], Step [100/354], Loss: 2.6118, Acc: 10.88%  
Epoch [1/300], Step [150/354], Loss: 2.5090, Acc: 15.98%  
Epoch [1/300], Step [150/354], Loss: 2.5090, Acc: 15.98%  
Epoch [1/300], Step [200/354], Loss: 2.0134, Acc: 19.89%  
Epoch [1/300], Step [200/354], Loss: 2.0134, Acc: 19.89%  
Epoch [1/300], Step [250/354], Loss: 1.9103, Acc: 23.54%  
Epoch [1/300], Step [250/354], Loss: 1.9103, Acc: 23.54%  
Epoch [1/300], Step [300/354], Loss: 1.5192, Acc: 26.66%  
Epoch [1/300], Step [300/354], Loss: 1.5192, Acc: 26.66%  
Epoch [1/300], Step [350/354], Loss: 1.7679, Acc: 29.02%  
Epoch [1/300], Step [350/354], Loss: 1.7679, Acc: 29.02%  
✓ Nuevo mejor modelo guardado: Test Acc = 55.41% (época 1)

=====

Época [1/300] Completada

=====

Train Loss: 2.3274 | Train Acc: 29.24%  
Test Loss: 1.5286 | Test Acc: 55.41%  
Gap (Overfitting): -26.16%  
Mejor Test Acc hasta ahora: 55.41% (época 1)  
Épocas sin mejora: 0/15  
Learning Rate actual: 1.00e-04  
Tiempo de época: 239.38s

=====

✓ Nuevo mejor modelo guardado: Test Acc = 55.41% (época 1)

=====

Época [1/300] Completada

=====

Train Loss: 2.3274 | Train Acc: 29.24%  
Test Loss: 1.5286 | Test Acc: 55.41%  
Gap (Overfitting): -26.16%  
Mejor Test Acc hasta ahora: 55.41% (época 1)  
Épocas sin mejora: 0/15  
Learning Rate actual: 1.00e-04  
Tiempo de época: 239.38s

=====

Epoch [2/300], Step [50/354], Loss: 0.8909, Acc: 51.88%  
Epoch [2/300], Step [50/354], Loss: 0.8909, Acc: 51.88%  
Epoch [2/300], Step [100/354], Loss: 1.2196, Acc: 52.22%  
Epoch [2/300], Step [100/354], Loss: 1.2196, Acc: 52.22%  
Epoch [2/300], Step [150/354], Loss: 1.9938, Acc: 52.94%  
Epoch [2/300], Step [150/354], Loss: 1.9938, Acc: 52.94%  
Epoch [2/300], Step [200/354], Loss: 1.5257, Acc: 53.83%  
Epoch [2/300], Step [200/354], Loss: 1.5257, Acc: 53.83%  
Epoch [2/300], Step [250/354], Loss: 1.0831, Acc: 54.25%  
Epoch [2/300], Step [250/354], Loss: 1.0831, Acc: 54.25%

```
Epoch [2/300], Step [300/354], Loss: 1.7620, Acc: 54.61%
Epoch [2/300], Step [300/354], Loss: 1.7620, Acc: 54.61%
Epoch [2/300], Step [350/354], Loss: 1.1791, Acc: 55.33%
Epoch [2/300], Step [350/354], Loss: 1.1791, Acc: 55.33%
✓ Nuevo mejor modelo guardado: Test Acc = 70.74% (época 2)
```

=====

Época [2/300] Completada

=====

```
Train Loss: 1.5077 | Train Acc: 55.37%
Test Loss: 1.0136 | Test Acc: 70.74%
Gap (Overfitting): -15.37%
Mejor Test Acc hasta ahora: 70.74% (época 2)
Épocas sin mejora: 0/15
Learning Rate actual: 1.00e-04
Tiempo de época: 244.89s
```

=====

```
✓ Nuevo mejor modelo guardado: Test Acc = 70.74% (época 2)
```

=====

Época [2/300] Completada

=====

```
Train Loss: 1.5077 | Train Acc: 55.37%
Test Loss: 1.0136 | Test Acc: 70.74%
Gap (Overfitting): -15.37%
Mejor Test Acc hasta ahora: 70.74% (época 2)
Épocas sin mejora: 0/15
Learning Rate actual: 1.00e-04
Tiempo de época: 244.89s
```

=====

```
Epoch [3/300], Step [50/354], Loss: 1.4629, Acc: 63.94%
Epoch [3/300], Step [50/354], Loss: 1.4629, Acc: 63.94%
Epoch [3/300], Step [100/354], Loss: 1.6244, Acc: 62.47%
Epoch [3/300], Step [100/354], Loss: 1.6244, Acc: 62.47%
Epoch [3/300], Step [150/354], Loss: 0.7911, Acc: 62.38%
Epoch [3/300], Step [150/354], Loss: 0.7911, Acc: 62.38%
Epoch [3/300], Step [200/354], Loss: 0.9194, Acc: 62.75%
Epoch [3/300], Step [200/354], Loss: 0.9194, Acc: 62.75%
Epoch [3/300], Step [250/354], Loss: 0.7879, Acc: 62.89%
Epoch [3/300], Step [250/354], Loss: 0.7879, Acc: 62.89%
Epoch [3/300], Step [300/354], Loss: 1.5281, Acc: 62.94%
Epoch [3/300], Step [300/354], Loss: 1.5281, Acc: 62.94%
Epoch [3/300], Step [350/354], Loss: 1.3977, Acc: 63.55%
Epoch [3/300], Step [350/354], Loss: 1.3977, Acc: 63.55%
✓ Nuevo mejor modelo guardado: Test Acc = 74.45% (época 3)
```

=====

Época [3/300] Completada

=====

```
Train Loss: 1.2193 | Train Acc: 63.57%
Test Loss: 0.8965 | Test Acc: 74.45%
Gap (Overfitting): -10.89%
Mejor Test Acc hasta ahora: 74.45% (época 3)
Épocas sin mejora: 0/15
```

```
Epoch [85/300], Step [200/354], Loss: 0.0035, Acc: 98.72%
Epoch [85/300], Step [200/354], Loss: 0.0035, Acc: 98.72%
Epoch [85/300], Step [250/354], Loss: 0.0064, Acc: 98.72%
Epoch [85/300], Step [250/354], Loss: 0.0064, Acc: 98.72%
Epoch [85/300], Step [300/354], Loss: 0.0140, Acc: 98.65%
Epoch [85/300], Step [300/354], Loss: 0.0140, Acc: 98.65%
Epoch [85/300], Step [350/354], Loss: 0.0020, Acc: 98.67%
Epoch [85/300], Step [350/354], Loss: 0.0020, Acc: 98.67%
```

```
=====
Época [85/300] Completada
=====
```

```
Train Loss: 0.0422 | Train Acc: 98.67%
Test Loss: 0.6718 | Test Acc: 90.85%
Gap (Overfitting): 7.83%
Mejor Test Acc hasta ahora: 90.95% (época 71)
Épocas sin mejora: 14/15
Learning Rate actual: 7.81e-07
Tiempo de época: 2089.14s
```

```
=====
Época [85/300] Completada
=====
```

```
Train Loss: 0.0422 | Train Acc: 98.67%
Test Loss: 0.6718 | Test Acc: 90.85%
Gap (Overfitting): 7.83%
Mejor Test Acc hasta ahora: 90.95% (época 71)
Épocas sin mejora: 14/15
Learning Rate actual: 7.81e-07
Tiempo de época: 2089.14s
```

```
Epoch [86/300], Step [50/354], Loss: 0.0087, Acc: 99.31%
Epoch [86/300], Step [50/354], Loss: 0.0087, Acc: 99.31%
Epoch [86/300], Step [100/354], Loss: 0.0081, Acc: 98.91%
Epoch [86/300], Step [100/354], Loss: 0.0081, Acc: 98.91%
Epoch [86/300], Step [150/354], Loss: 0.0370, Acc: 98.88%
Epoch [86/300], Step [150/354], Loss: 0.0370, Acc: 98.88%
Epoch [86/300], Step [200/354], Loss: 0.1506, Acc: 98.81%
Epoch [86/300], Step [200/354], Loss: 0.1506, Acc: 98.81%
Epoch [86/300], Step [250/354], Loss: 0.0017, Acc: 98.78%
Epoch [86/300], Step [250/354], Loss: 0.0017, Acc: 98.78%
Epoch [86/300], Step [300/354], Loss: 0.0105, Acc: 98.75%
Epoch [86/300], Step [300/354], Loss: 0.0105, Acc: 98.75%
Epoch [86/300], Step [350/354], Loss: 0.0085, Acc: 98.76%
Epoch [86/300], Step [350/354], Loss: 0.0085, Acc: 98.76%
```

```
=====
Época [86/300] Completada
=====
```

```
Train Loss: 0.0416 | Train Acc: 98.76%
Test Loss: 0.6737 | Test Acc: 90.78%
Gap (Overfitting): 7.99%
Mejor Test Acc hasta ahora: 90.95% (época 71)
```

Épocas sin mejora: 15/15  
Learning Rate actual: 7.81e-07  
Tiempo de época: 2091.12s

```
=====
```

!!!!!!!!!!!!!!  
EARLY STOPPING ACTIVADO  
!!!!!!!!!!!!!!  
No se observó mejora en Test Accuracy durante 15 épocas consecutivas.  
Deteniendo entrenamiento en época 86.  
Mejor modelo: época 71 con Test Acc = 90.95%  
!!!!!!!!!

\*\*\*\*\*  
Cargando mejor modelo (época 71) para evaluación final...  
\*\*\*\*\*

#####  
ENTRENAMIENTO COMPLETADO  
#####  
Tiempo total: 1553.34 minutos (93200.15 segundos)  
Épocas completadas: 86  
Mejor Test Accuracy: 90.95% (época 71)  
Precisión final de entrenamiento: 98.76%  
Precisión final de prueba: 90.78%  
Gap final: 7.99%  
#####

=====  
Época [86/300] Completada  
=====  
Train Loss: 0.0416 | Train Acc: 98.76%  
Test Loss: 0.6737 | Test Acc: 90.78%  
Gap (Overfitting): 7.99%  
Mejor Test Acc hasta ahora: 90.95% (época 71)  
Épocas sin mejora: 15/15  
Learning Rate actual: 7.81e-07  
Tiempo de época: 2091.12s  
=====

!!!!!!!!!!!!!!  
EARLY STOPPING ACTIVADO  
!!!!!!!!!!!!!!  
No se observó mejora en Test Accuracy durante 15 épocas consecutivas.  
Deteniendo entrenamiento en época 86.  
Mejor modelo: época 71 con Test Acc = 90.95%  
!!!!!!!!!

\*\*\*\*\*  
Cargando mejor modelo (época 71) para evaluación final...  
\*\*\*\*\*

```
*****
#####
ENTRENAMIENTO COMPLETADO
#####
Tiempo total: 1553.34 minutos (93200.15 segundos)
Épocas completadas: 86
Mejor Test Accuracy: 90.95% (época 71)
Precisión final de entrenamiento: 98.76%
Precisión final de prueba: 90.78%
Gap final: 7.99%
#####
```

## 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 [16]: 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=2.5,
          xlabel('Época', fontsize=13, fontweight='bold'),
          ylabel('Pérdida (Loss)', fontsize=13, fontweight='bold'),
          title(f'Evolución de la Pérdida - {MODEL_TYPE}', fontsize=15, fontweight='bold'),
          legend(fontsize=11, loc='best'),
          grid(True, alpha=0.3, linestyle='--'),
          xlim(1, len(train_losses)))

# Gráfica de precisión
ax2.plot(epochs_range, train_accuracies, 'b-', label='Precisión de Entrenamiento',
          xlabel('Época', fontsize=13, fontweight='bold'),
          ylabel('Precisión (%)', fontsize=13, fontweight='bold'),
          title(f'Evolución de la Precisión - {MODEL_TYPE}', fontsize=15, fontweight='bold'),
          legend(fontsize=11, loc='best'),
          grid(True, alpha=0.3, linestyle='--'),
          xlim(1, len(train_accuracies)),
          ylim(0, 100))

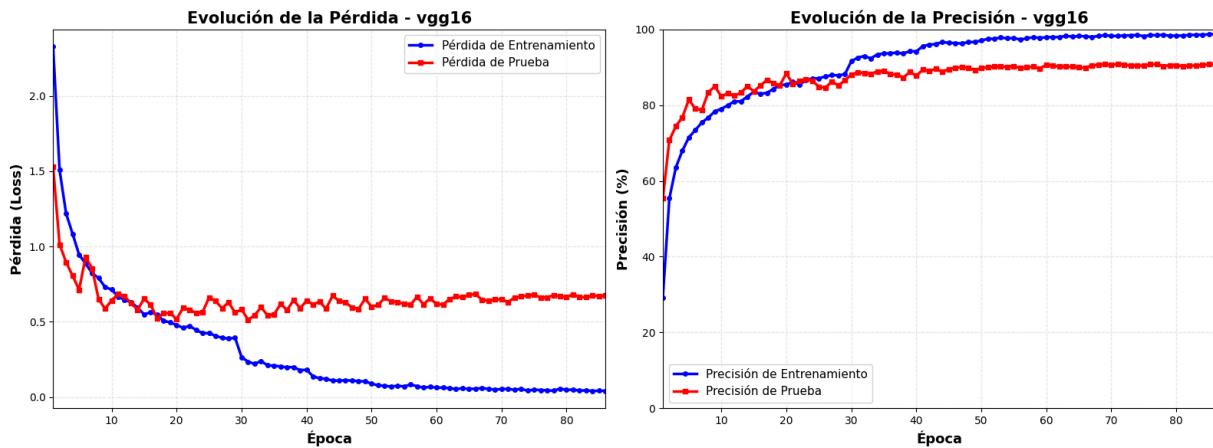
plt.tight_layout()
plt.show()

print(f'\n{"*80}')
print(f'{"RESUMEN DE MÉTRICAS FINALES":^80}')
print(f'{"*80}')
```

```

print(f'Modelo: {MODEL_TYPE}')
print(f'b) Precisión final de entrenamiento: {train_accuracies[-1]:.2f}%')
print(f'b) Precisión final de prueba: {test_accuracies[-1]:.2f}%')
print(f' 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'\nc) Tiempo total de simulación: {total_time/60:.2f} minutos ({total_time:.2f} segundos')
print(f' Tiempo promedio por época: {total_time/len(train_losses):.2f} segundos')
print(f'{"*80}\n')

```




---

#### RESUMEN DE MÉTRICAS FINALES

---

Modelo: vgg16  
 b) Precisión final de entrenamiento: 98.76%  
 b) Precisión final de prueba: 90.78%  
 Gap (Train - Test): 7.99%

Pérdida final de entrenamiento: 0.0416  
 Pérdida final de prueba: 0.6737

c) Tiempo total de simulación: 1553.34 minutos (93200.15 segundos)  
 Tiempo promedio por época: 1083.72 segundos

---

```

In [24]: import torch
import torchvision.models as models
import torch.nn as nn

MODEL_TYPE = 'vgg16'
save_best_path = f'{MODEL_TYPE}_best_checkpoint.pth'
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = models.vgg16()
# Personaliza el clasificador igual que al entrenar:
model.classifier = nn.Sequential(
    nn.Linear(512 * 7 * 7, 4096),
    nn.ReLU(True),
    nn.Dropout(0.3),
    nn.Linear(4096, 4096),
    nn.ReLU(True),
    nn.Dropout(0.5),
)

```

```

        nn.Linear(4096, 22) # Cambia 22 por el número de clases que usaste
    )

checkpoint = torch.load(save_best_path, map_location=device)
model.load_state_dict(checkpoint['model_state_dict'])
model = model.to(device)

```

In [25]:

```

import torch
import torch.nn.functional as F

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

with torch.no_grad():
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        train_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

avg_train_loss = train_loss / len(train_loader)
train_accuracy = 100 * correct / total

print(f'Pérdida promedio en entrenamiento: {avg_train_loss:.4f}')
print(f'Precisión en entrenamiento: {train_accuracy:.2f}%')

```

Pérdida promedio en entrenamiento: 0.0468  
 Precisión en entrenamiento: 98.43%

In [26]:

```

import torch
import torch.nn.functional as F

# Evaluar en entrenamiento
model.eval()
train_loss = 0.0
train_correct = 0
train_total = 0
with torch.no_grad():
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        train_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        train_total += labels.size(0)
        train_correct += (predicted == labels).sum().item()
avg_train_loss = train_loss / len(train_loader)
train_accuracy = 100 * train_correct / train_total

# Evaluar en prueba
test_loss = 0.0

```

```

test_correct = 0
test_total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = F.cross_entropy(outputs, labels)
        test_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()
avg_test_loss = test_loss / len(test_loader)
test_accuracy = 100 * test_correct / test_total

print(f'Entrenamiento - Pérdida: {avg_train_loss:.4f}, Precisión: {train_accuracy:.2f}%')
print(f'Prueba - Pérdida: {avg_test_loss:.4f}, Precisión: {test_accuracy:.2f}%')
print(f'Gap de precisión (train - test): {train_accuracy - test_accuracy:.2f}%')

```

Entrenamiento - Pérdida: 0.0431, Precisión: 98.66%

Prueba - Pérdida: 0.6225, Precisión: 90.95%

Gap de precisión (train - test): 7.70%

In [27]: # 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_TYPE}', 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%}', 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_TYPE}', 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: fencing      | Confianza: 1.0000 (100.00%)
2. Clase: kabaddi       | Confianza: 1.0000 (100.00%)
3. Clase: formula1     | Confianza: 1.0000 (100.00%)
4. Clase: ice_hockey    | Confianza: 1.0000 (100.00%)
5. Clase: football      | Confianza: 1.0000 (100.00%)
6. Clase: table_tennis   | Confianza: 1.0000 (100.00%)

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

1. Clase: fencing      | Confianza: 1.0000 (100.00%)
2. Clase: kabaddi       | Confianza: 1.0000 (100.00%)
3. Clase: formula1     | Confianza: 1.0000 (100.00%)
4. Clase: ice_hockey    | Confianza: 1.0000 (100.00%)
5. Clase: football      | Confianza: 1.0000 (100.00%)
6. Clase: table_tennis   | Confianza: 1.0000 (100.00%)
```

### Ejemplos de Desempeño EXCELENTE - vgg16

Predicción: fencing  
Real: fencing  
Confianza: 100.00%



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



Predicción: kabaddi  
Real: kabaddi  
Confianza: 100.00%



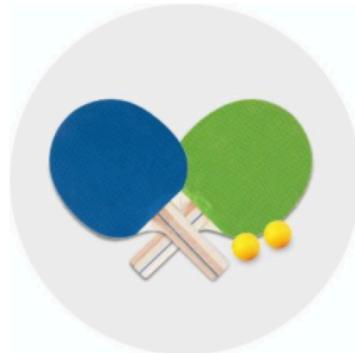
Predicción: football  
Real: football  
Confianza: 100.00%



Predicción: formula1  
Real: formula1  
Confianza: 100.00%



Predicción: table\_tennis  
Real: table\_tennis  
Confianza: 100.00%



---

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

---

1. Pred: badminton	Real: gymnastics	Confianza: 1.0000 (100.00%)
2. Pred: gymnastics	Real: weight_lifting	Confianza: 1.0000 (100.00%)
3. Pred: table_tennis	Real: boxing	Confianza: 1.0000 (100.00%)
4. Pred: tennis	Real: table_tennis	Confianza: 1.0000 (100.00%)
5. Pred: shooting	Real: fencing	Confianza: 1.0000 (100.00%)
6. Pred: fencing	Real: kabaddi	Confianza: 1.0000 (100.00%)

Ejemplos de Desempeño POBRE - vgg16

**Predicción: badminton**  
**Real: gymnastics**  
**Confianza: 100.00%**



**Predicción: tennis  
Real: table\_tennis  
Confianza: 100.00%**

**Predicción: gymnastics  
Real: weight\_lifting  
Confianza: 100,00%**



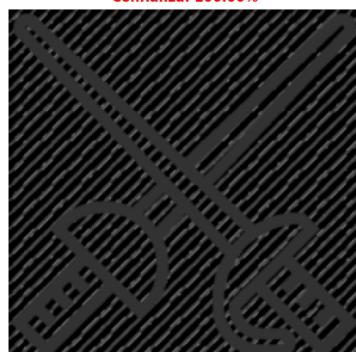
**Predicción: shooting  
Real: fencing  
Confianza: 100.00%**

**Predicción: table\_tennis  
Real: boxing  
Confianza: 100.00%**



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

A close-up photograph showing two bright yellow tennis balls and parts of three tennis rackets lying on a green grass surface. The rackets have red, purple, and white frames.



A group of students are playing basketball in a large, modern gymnasium. The court is marked with red and yellow lines. In the foreground, a player in a blue shirt and black shorts is dribbling the ball towards the basket. Another player in a white shirt and red shorts is positioned nearby. Other students are scattered across the court, some watching and others near the baseline. The gym has high ceilings with recessed lighting and large windows in the background.

Total de predicciones correctas: 2574 / 2830 (90.95%)  
Total de predicciones incorrectas: 256 / 2830 (9.05%)

```

    digits=4))

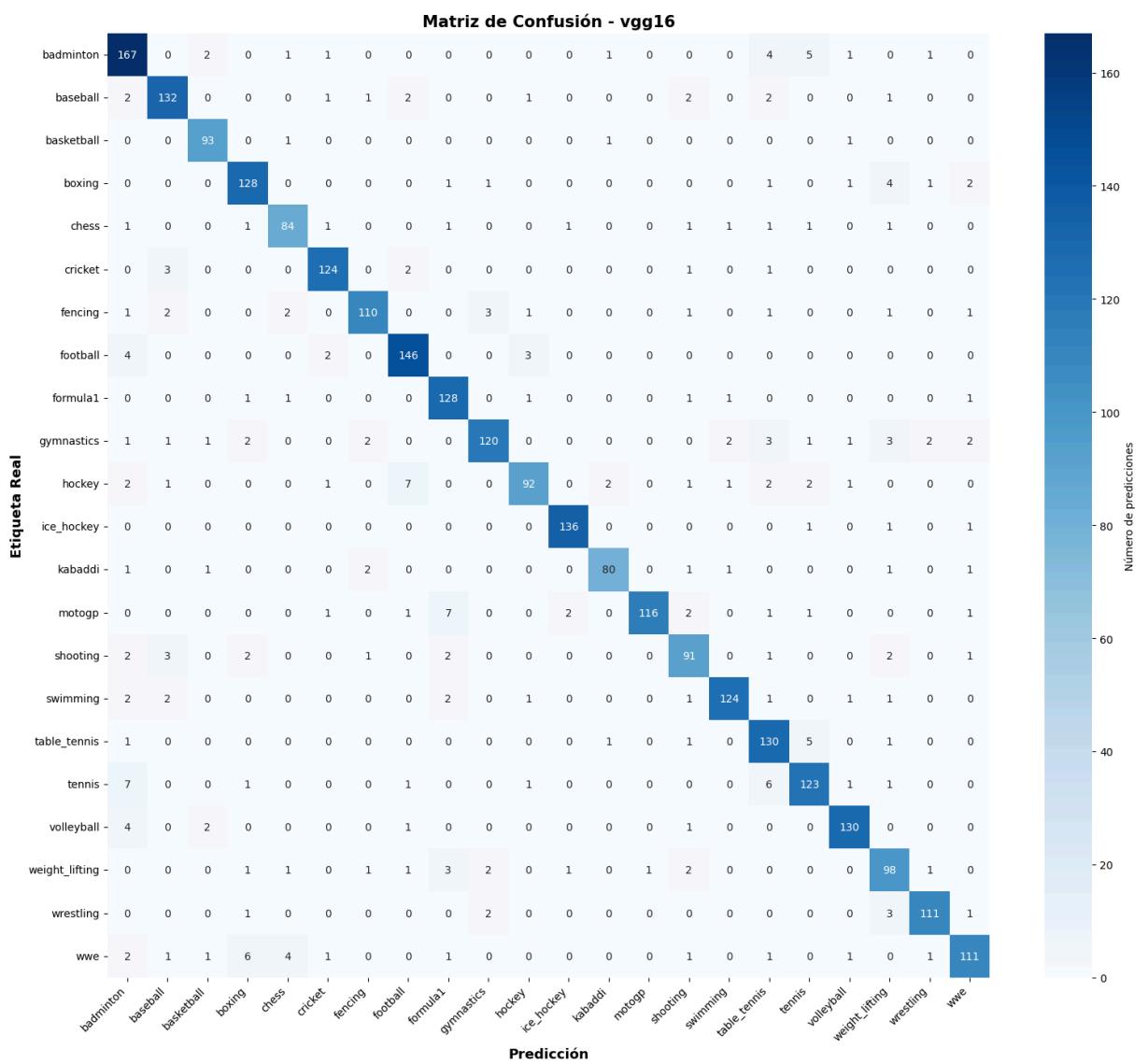
# Identificar clases con mejor y peor desempeño
from sklearn.metrics import precision_recall_fscore_support

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 (vgg16) ===

	precision	recall	f1-score	support
badminton	0.8477	0.9126	0.8789	183
baseball	0.9103	0.9167	0.9135	144
basketball	0.9300	0.9688	0.9490	96
boxing	0.8951	0.9209	0.9078	139
chess	0.8936	0.8936	0.8936	94
cricket	0.9394	0.9466	0.9430	131
fencing	0.9402	0.8943	0.9167	123
football	0.9068	0.9419	0.9241	155
formula1	0.8828	0.9552	0.9176	134
gymnastics	0.9375	0.8511	0.8922	141
hockey	0.9200	0.8214	0.8679	112
ice_hockey	0.9714	0.9784	0.9749	139
kabaddi	0.9412	0.9091	0.9249	88
motogp	0.9915	0.8788	0.9317	132
shooting	0.8505	0.8667	0.8585	105
swimming	0.9538	0.9185	0.9358	135
table_tennis	0.8387	0.9353	0.8844	139
tennis	0.8849	0.8723	0.8786	141
volleyball	0.9420	0.9420	0.9420	138
weight_lifting	0.8305	0.8750	0.8522	112
wrestling	0.9487	0.9407	0.9447	118
wwe	0.9098	0.8473	0.8775	131
accuracy			0.9095	2830
macro avg	0.9121	0.9085	0.9095	2830
weighted avg	0.9113	0.9095	0.9096	2830

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

ice_hockey	F1: 0.9749   Precision: 0.9714   Recall: 0.9784
basketball	F1: 0.9490   Precision: 0.9300   Recall: 0.9688
wrestling	F1: 0.9447   Precision: 0.9487   Recall: 0.9407
cricket	F1: 0.9430   Precision: 0.9394   Recall: 0.9466
volleyball	F1: 0.9420   Precision: 0.9420   Recall: 0.9420

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

weight_lifting	F1: 0.8522   Precision: 0.8305   Recall: 0.8750
shooting	F1: 0.8585   Precision: 0.8505   Recall: 0.8667
hockey	F1: 0.8679   Precision: 0.9200   Recall: 0.8214
wwe	F1: 0.8775   Precision: 0.9098   Recall: 0.8473
tennis	F1: 0.8786   Precision: 0.8849   Recall: 0.8723

```
In [29]: 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_TYPE}')
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: vgg16\_sports\_classifier.pth

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

## ARQUITECTURA

---

Modelo: vgg16

Parámetros totales: 134,350,678

Parámetros entrenables: 133,795,350

Parámetros congelados: 555,328

Bloques congelados: 1-3 (extracción de características básicas)

Bloques entrenables: 4-5 + clasificador personalizado

## HIPERPARÁMETROS

---

Épocas: 86

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: 98.76%

Precisión de prueba: 90.78%

Gap (Overfitting): 7.99%

Pérdida de entrenamiento: 0.0416

Pérdida de prueba: 0.6737

## TIEMPO DE ENTRENAMIENTO

---

Tiempo total: 1553.34 minutos (93200.15 segundos)

Tiempo promedio por época: 1083.72 segundos

## DATASET

---

Total de imágenes: 14149

Número de clases: 22

Imágenes de entrenamiento: 11319 (80.0%)

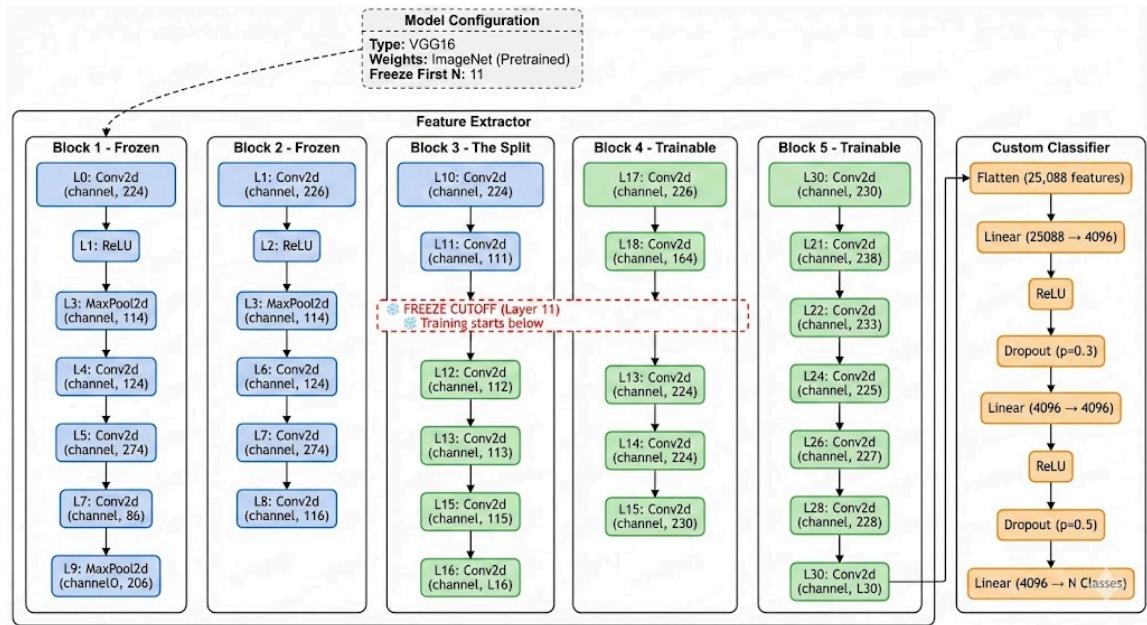
Imágenes de prueba: 2830 (20.0%)

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

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

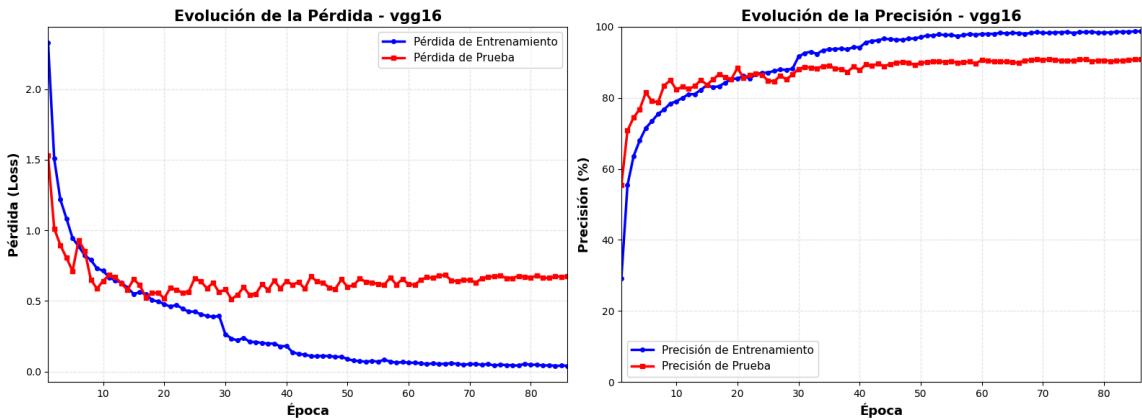
#####

En primer lugar el modelo se representa de la siguiente forma:

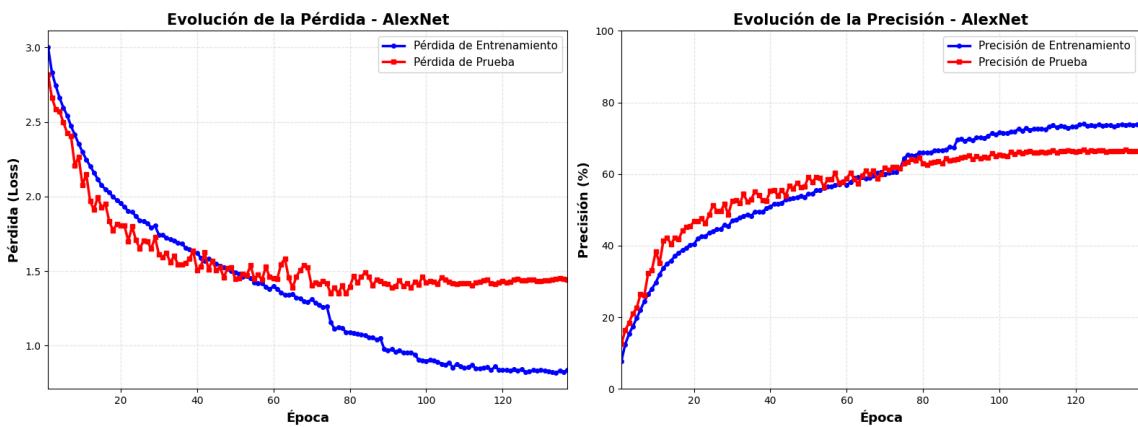


Es una vgg16 preentrenada en imagenet, con las capas convolucionales congeladas y solo entrenando las capas totalmente conectadas para evitar sobreajuste.

En cuanto a la fase de entrenamiento y validación se puede observar lo siguiente:

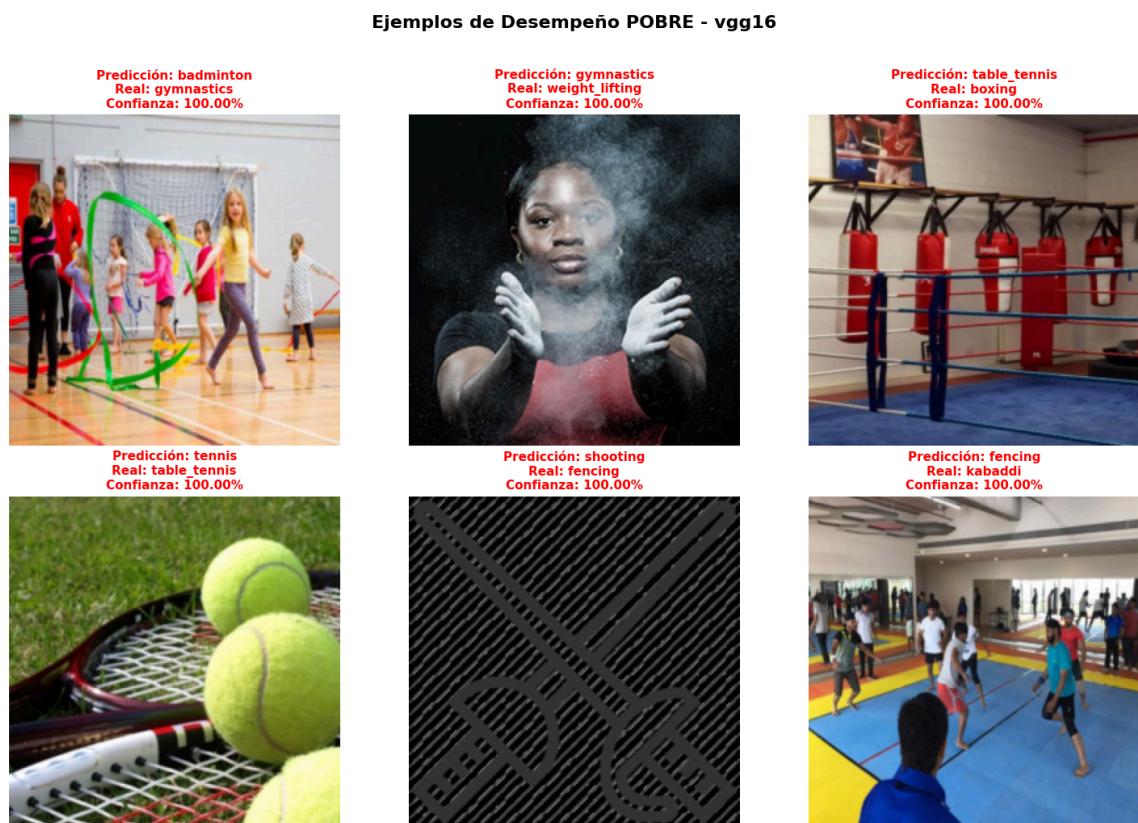


Comparado con la AlexNet si transfer learning:



Se logra ver claramente que la vgg16 pre-entrenada logra un accuracy impresionante del casi 91% mientras que la AlexNet sin preentrenamiento casi roza los 65% de accuracy en el test esto de bido a que la vgg tiene un conjunto de entrenamiento muy superior de IMAGENET, el cual ya reconoce muchisimos patrones en si misma, eso se puede ver en las primeras epochas muy claramente, la cantidad de accuracy en test con train sin siquiera haber pasado la epoca 5, eso no ocurre con la AlexNet.

Pasemos al **análisis de clasificación pobre de ambas redes**, muestro las de la vgg:



Y a continuación la de la AlexNet:

### Ejemplos de Desempeño POBRE - AlexNet



Podemos observar en la vgg por ejemplo puede llegar a confundir algunos deportes ya que cuenta con elementos similares como raquetas en el caso de el tennis de mesa o el tennis normal, e levantamiento de pesas y gimnasia el tema del talco y el uniforme que suelen usar es similar por lo que se puede confundir, se observa una predicción incorrecta en ambas redes en un par de espafas chocando en que en realidad es fencing sin embargo ambas la predicen como disparo, podría ser por los colores y formas que puede llegar a confundir a los modelos, quizá quitarla es buena idea.

Pasemos a las metricas:

VGG16 es inmensamente superior a AlexNet en esta tarea. No es una mejora marginal; es un salto cualitativo enorme.

- VGG16 alcanza una exactitud (Accuracy) del 91%.
- AlexNet se queda en un 67%.
- Diferencia: VGG16 acierta +24% más veces que AlexNet.

Clase	AlexNet (F1)	VGG16 (F1)	Mejora	Observación
Basketball	0.55	0.95	+0.40	Mejora Masiva. VGG16 entiende la cancha; AlexNet no.
Gymnastics	0.51	0.89	+0.38	De tirar una moneda a casi excelente.

Clase	AlexNet (F1)	VGG16 (F1)	Mejora	Observación
Weight Lifting	0.33	0.85	+0.52	La mayor diferencia. VGG16 rescató una clase perdida.
Wrestling	0.55	0.94	+0.39	AlexNet confundía los cuerpos entrelazados; VGG16 no.

Aunque VGG16 es mucho mejor, ambos modelos coinciden en cuál es la clase más difícil de clasificar.

La clase maldita: Weight Lifting (Levantamiento de pesas).

Es la peor clase de AlexNet (0.33).

Es la peor clase de VGG16 (0.85).

Conclusión: Las imágenes de levantamiento de pesas deben ser visualmente confusas o parecidas a otras categorías. Ambos modelos sufren aquí más que en cualquier otro deporte.

```
In [31]: !jupyter nbconvert --to html Tarea4.ipynb
```

```
[NbConvertApp] Converting notebook Tarea4.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 6 image(s).
[NbConvertApp] Writing 8872006 bytes to Tarea4.html
```