```
import torch
import torch.nn as nn
import torchvision  #Para conjuntos de datos y operaciones con imágenes
import torchvision.transforms as transforms  #Para transformaciones de datos
import matplotlib.pyplot as plt  #Para visualización
import numpy as np
from sklearn.metrics import classification_report
import random as rd
import torch.optim as optim
```

1. Modelo AlexNet en PyTorch

1.1 Creación de los bloques Convolucionales y Densos

Vamos a definir el modelo AlexNet en PyTorch, para ello debemos de definir:

1. Bloque Convolucional (tendrá una capa convolucional y una función de activación).
2. Bloque de denso (serán neuronas conectadas con una salida y una funcion de activación).

Para el **Bloque Convoucional**, agregamos una capa convolucional que se encargara de extraer las caracteristicas de las imagenes proporcionadas (como los bordes) con canales de entrada (para imagenes a colo son 3 RGB), los canales de salida (numero de caracteristicas a extarer), un tamaño del kernel (se encarga de recorrer la imagen, por ejemplo, de 3x3 pixeles recorre la imagen y es menos costo computacional), el pading relleno (al ser igual a 1 mantiene la imagen en su tamaño original) y al final lo que se obtiene se pasa por la finción de activación ReLU.

El **Bloque denso** se encargara de recibir caracteristicas, para nuestro fin seran las caracteristicas extraidas por la CNN. El hecho de que sea lineal quiere decir que todas las neuronas estan conectadas. Se reciben las caracteristicas y produce una salida que en nuestro caso será 10 para el dataset de *CIFAR 10* y al final le aplicamos a la salida la función de activación ReLU.

```
#Capa de convolución
class BloqueConvolucional(nn.Module):
    def __init__(self, canales_entrada, canales_salida, tamaño_kernel=3, relleno=1):
        super(BloqueConvolucional, self).__init__()
        #Capa convolucional que extrae características de la imagen
        self.convolucion = nn.Conv2d(canales_entrada, canales_salida, tamaño_kernel, padding=relleno)
        #Función de activación ReLU
        self.activacion = nn.ReLU(inplace=True)

    def forward(self, x):
        return self.activacion(self.convolucion(x))#Aplicar convolución seguida de ReLU

#Capa densa
class BloqueDenso(nn.Module):
    def __init__(self, caracteristicas_entrada, caracteristicas_salida):
        super(BloqueDenso, self).__init__()
        self.lineal = nn.Linear(caracteristicas_entrada, caracteristicas_salida)
        self.activacion = nn.ReLU(inplace=True)

    def forward(self, x):
        return self.activacion(self.lineal(x))
```

1.2 Definición del modelo AlexNet

Para el modelo, debemos de usar 5 capas convolucionales, 3 capas max poooling y 3 capas densas. Primero extaremos las caracteristicas mediante la primera capa convolucional con 3 canales de entrada para los datos RGB que van aumentando mientras se avanza por las 5 capas convolucionales pero con las caps Maxpooling tambien vamos reduciendo el tamaño de las imagenes quedandonos con varias caracteristicas en menos pixeles. Hasta este punto tenemos 5 capas convolucionales + 3 capas Maxpooling = 8 capas.

Para el segundo bloque de capas, usamos las capas densas para recibir los tensores los cuales los aplana hacia la capa densa 2 donde llegan solo vectores y depues se pasan a la última capa para clasificación, que en el caso de la base de datos, donde son 10 clases donde se puede clasificar las imagenes. Todo eso lo juntamos en el ***forward***.

```
#Red AlexNet(PyTorch)
class AlexNet(nn.Module):
    def __init__(self, numero_clases=10):
```

```
        super(AlexNet, self).__init__()

        #Capas convolucionales para extracción de características
        self.caracteristicas = nn.Sequential(
            BloqueConvolucional(3, 96), #Conv1: entrada RGB (3 canales)
            nn.MaxPool2d(kernel_size=3, stride=2), #Reducción dimensional con MaxPool1
            BloqueConvolucional(96, 256), #Conv2
            nn.MaxPool2d(kernel_size=3, stride=2), #Reducción dimensional con MaxPool2
            BloqueConvolucional(256, 384), #Conv3
            BloqueConvolucional(384, 384), #Conv4
            BloqueConvolucional(384, 256), #Conv5
            nn.MaxPool2d(kernel_size=3, stride=2) #Reducción dimensional con MaxPool3
        )

        #Capas para clasificación
        self.clasificador = nn.Sequential(
            BloqueDenso(256 * 3 * 3, 4096), #Capa densa 1 (aplanar salidas anteriores)
            BloqueDenso(4096, 4096),        #Capa densa 2
            nn.Linear(4096, numero_clases)  #Capa densa 3 (salida)
        )

    def forward(self, x):
        x = self.caracteristicas(x) #Procesar con capas convolucionales
        x = torch.flatten(x, 1)     #Aplanar tensor
        x = self.clasificador(x)    #Procesar con capas densas
        return x
```

1.3 Cargar la base de datos

Descargamos la base y esta se guardara en la ruta indicada, separamos en conjuntos de entrenamiento (para entrenar el modelo de Red) y en conjunto de prueba (se utiliara para ver como se desempeña el modelo ante datos nuevos). Cuando los utilicemos debemos de normalizar los pixeles y vamos a utilizar un batch de 30 (son los datos que se van pasando al modelo, en este caso de 30 en 30).

```
#Cargar CIFAR-10
transformar = transforms.Compose([
    transforms.ToTensor(), #Convertir imágenes a tensores
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) #Normalizar píxeles a [-1, 1]
])

#Conjunto de entrenamiento
conjunto_entrenamiento = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transformar)
#Cargador de datos en lotes de 30
cargador_entrenamiento = torch.utils.data.DataLoader(conjunto_entrenamiento, batch_size=30, shuffle=True)

#Conjunto de prueba
conjunto_prueba = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transformar)
#Cargador de datos en lotes de 30
cargador_prueba = torch.utils.data.DataLoader(conjunto_prueba, batch_size=30, shuffle=False)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:03<00:00, 43.7MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

1.4 Entrenar el modelo

Definimos como **modelo1** el modelo de AlexNet y usamos:

- Función de perdida: Entropia cruzada (usualmente empleada en modelos de clasificación)
- Optimizador de la función de perdida: Descenso del Gradiente Estocastico (es más lento que adam pero es más preciso con más iteraciones que adam) con una tasa de aprendizaje de 0.001. Se prefirio SGD ya que, si bien se tenia un menor error de entrenamiento, en el error de prueba se obteneian peores resultados.
- 20 épocas o iteraciones para entrenar el modelo.

Además, vamos a usar el CPU o GPU, dependiendo de nuestra PC.

```
#Inicializar modelo, función de pérdida y optimizador
modelo1 = AlexNet()
criterio = nn.CrossEntropyLoss() #Función de pérdida
optimizador = torch.optim.SGD(modelo1.parameters(), lr=0.001, momentum=0.9) #Optimizador SGD
```

```
#Configurar dispositivo (GPU si está disponible)
dispositivo = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
modelo1.to(dispositivo)  # Mover modelo al dispositivo

#Bucle de entrenamiento
epocas_modelo = 20
for epoca in range(epocas_modelo):
    modelo1.train()
    perdida_acumulada = 0.0
    #Iterar sobre lotes de entrenamiento
    for i, (entradas, etiquetas) in enumerate(cargador_entrenamiento):
        entradas, etiquetas = entradas.to(dispositivo), etiquetas.to(dispositivo)

        optimizador.zero_grad() #Reiniciar gradientes
        salidas = modelo1(entradas) #Calcular predicciones
        perdida = criterio(salidas, etiquetas) #Calcular perdida
        perdida.backward() #Retropropagación
        optimizador.step() #Actualizar pesos

        perdida_acumulada += perdida.item() #Acumular perdida

    #Mostrar pérdida por época
    print(f'Época {epoca+1}, Pérdida: {perdida_acumulada/len(cargador_entrenamiento)}')
```

```
Época 1, Pérdida: 2.3006620897671817
Época 2, Pérdida: 1.9944575370681021
Época 3, Pérdida: 1.6269077512412708
Época 4, Pérdida: 1.4343157799285404
Época 5, Pérdida: 1.2623996114973974
Época 6, Pérdida: 1.1275993689277892
Época 7, Pérdida: 1.0140475594110667
Época 8, Pérdida: 0.9188935680523845
Época 9, Pérdida: 0.8282509428087508
Época 10, Pérdida: 0.7558683384551784
Época 11, Pérdida: 0.6924303247317913
Época 12, Pérdida: 0.6414471424548441
Época 13, Pérdida: 0.5909679847460488
Época 14, Pérdida: 0.5442672680256677
Época 15, Pérdida: 0.5012322283284566
Época 16, Pérdida: 0.45854267941471627
Época 17, Pérdida: 0.4216360343194466
Época 18, Pérdida: 0.3847499661143125
Época 19, Pérdida: 0.34767415529160567
Época 20, Pérdida: 0.3159668788483705
```

1.4 Evaluación del desempeño del modelo

Usamos el modelo entrenado para usarlo en el conjunto de prueba y obtener la precisión de las imagenes.

```
#Evaluar el modelo en el conjunto de prueba
modelo1.eval()
correctas = 0
total = 0

with torch.no_grad(): #Desactiva calculo de gradientes para evaluación
    for datos, etiquetas in cargador_prueba:
        datos = datos.to(dispositivo)
        etiquetas = etiquetas.to(dispositivo)

        #Obtener predicciones
        salidas = modelo1(datos)
        _, predichas = torch.max(salidas.data, 1)

        #Calcular metricas
        total += etiquetas.size(0)
        correctas += (predichas == etiquetas).sum().item()

#Calculamos y mostramos la precisión en el conjunto de prueba
print(f'\nPrecisión en el conjunto de prueba: {100 * correctas / total:.2f}%')
```

```
Precisión en el conjunto de prueba: 79.22%
```

```
#Análisis de clasificación
todas_etiquetas = []
```

```
todas_predicciones = []

modelo1.eval()
with torch.no_grad():
    for datos, etiquetas in cargador_prueba:
        datos = datos.to(dispositivo)
        salidas = modelo1(datos)
        _, predichas = torch.max(salidas, 1)

        todas_etiquetas.extend(etiquetas.cpu().numpy())
        todas_predicciones.extend(predichas.cpu().numpy())

# Nombres de las clases de CIFAR-10
clases = ['avión', 'auto', 'pájaro', 'gato', 'ciervo',
          'perro', 'rana', 'caballo', 'barco', 'camión']

print(classification_report(
    todas_etiquetas,
    todas_predicciones,
    target_names=clases,
    digits=4
))
```

```
              precision    recall  f1-score   support

       avión     0.8623    0.7950    0.8273      1000
        auto     0.8973    0.9090    0.9031      1000
      pájaro     0.8505    0.6260    0.7212      1000
        gato     0.6155    0.5810    0.5977      1000
      ciervo     0.8085    0.7390    0.7722      1000
       perro     0.6008    0.8550    0.7057      1000
        rana     0.9495    0.7150    0.8157      1000
     caballo     0.7810    0.9020    0.8371      1000
       barco     0.8495    0.9090    0.8783      1000
       camión     0.8327    0.8910    0.8609      1000

    accuracy                         0.7922     10000
   macro avg     0.8048    0.7922    0.7919     10000
weighted avg     0.8048    0.7922    0.7919     10000
```

```
modelo1.eval()

im = rd.randint(0, 29) #Imagenes aleatorias en el conjunto de prueba

#Obtener un lote de prueba
dataiter = iter(cargador_prueba)
imagenes, etiquetas = next(dataiter)

#Seleccionar la imagen a predecir
imagen = imagenes[im].unsqueeze(0).to(dispositivo)  # Añadir dimensión de batch
etiqueta_real = etiquetas[im].item()

#Realizar la predicción
with torch.no_grad():
    salidas = modelo1(imagen)
    probabilidades = torch.nn.functional.softmax(salidas, dim=1)
    prob_predicha, clase_predicha = torch.max(probabilidades, 1)

#Preparar la imagen para visualización
imagen_mostrar = imagen.squeeze().cpu().numpy().transpose((1, 2, 0))  # Convertir a HWC
imagen_mostrar = imagen_mostrar * 0.5 + 0.5  # Desnormalizar ([-1,1] -> [0,1])

#Nombres de las clases
nombres_clases = ['avion', 'automovil', 'pajaro', 'gato', 'ciervo',
                  'perro', 'rana', 'caballo', 'barco', 'camion']

#Crear figura con prediccion
plt.figure(figsize=(6, 6))
plt.imshow(imagen_mostrar)
plt.title(f"Predicción: {nombres_clases[clase_predicha.item()]}\n"
          f"Probabilidad: {prob_predicha.item()*100:.1f}%\n"
          f"Real: {nombres_clases[etiqueta_real]}")
plt.axis('off')
plt.show()
```

2. Modelo preentrenado AlexNet

2.1 Cargar modelo y datos

Para usar el modelo preentrenado lo debemos de cargar y vamos a solo modificar la ultima capa para ajustarlo a la clasificación deseada, en este caso 10 clases.

```
dispositivo = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {dispositivo}")
# Cargar AlexNet preentrenado y modificarm la ultima capa para las 10 clasificaciones
modelo_pretrain = torchvision.models.alexnet(pretrained=True)
modelo_pretrain.classifier[6] = nn.Linear(4096, 10) #10clases para CIFAR-10
modelo_pretrain = modelo_pretrain.to(dispositivo)

#Preparación de datos especificos para modelo preentrenado
transformaciones_pretrain = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

#Cargar datos
cifar_entrenamiento_pretrain = torchvision.datasets.CIFAR10(root='./datos', train=True, download=True, transform=transformaciones_pretrain)
cifar_prueba_pretrain = torchvision.datasets.CIFAR10(root='./datos', train=False, download=True, transform=transformaciones_pretrain)

#Cargadores
cargador_entrenamiento_pretrain = torch.utils.data.DataLoader(cifar_entrenamiento_pretrain, batch_size=30, shuffle=True)
cargador_prueba_pretrain = torch.utils.data.DataLoader(cifar_prueba_pretrain, batch_size=30, shuffle=False)
```

```
Usando dispositivo: cuda
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.
  warnings.warn(
/usr/local/lib/python3.11/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or `None` for
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to /root/.cache/torch/hub/checkpoints/alexnet-owt-7be5be79.p
100%|██████████| 233M/233M [00:01<00:00, 182MB/s]
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./datos/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:05<00:00, 31.4MB/s]
Extracting ./datos/cifar-10-python.tar.gz to ./datos
Files already downloaded and verified
```

2.2 Entrenar modelo

Usamos los mismos hiperparámetros y funciones de perdida, el optimizador y las epocas que en el modelo anterior para ser justos.

```
criterio_pretrain = nn.CrossEntropyLoss()
optimizador_pretrain = optim.SGD(modelo_pretrain.parameters(), lr=0.001, momentum=0.9)

#Entrenar el modelo
epocas_pretrain = 20
for epoca in range(epocas_pretrain):
    modelo_pretrain.train()
    perdida_acumulada = 0.0

    for entradas, etiquetas in cargador_entrenamiento_pretrain:
        entradas = entradas.to(dispositivo)
        etiquetas = etiquetas.to(dispositivo)

        optimizador_pretrain.zero_grad()
        salidas = modelo_pretrain(entradas)
        perdida = criterio_pretrain(salidas, etiquetas)
        perdida.backward()
        optimizador_pretrain.step()

        perdida_acumulada += perdida.item()

    print(f'Época {epoca+1}, Pérdida: {perdida_acumulada/len(cargador_entrenamiento_pretrain):.4f}')
```

```
Época 1, Pérdida: 0.6339
Época 2, Pérdida: 0.4018
Época 3, Pérdida: 0.3134
Época 4, Pérdida: 0.2524
Época 5, Pérdida: 0.2093
Época 6, Pérdida: 0.1730
Época 7, Pérdida: 0.1421
Época 8, Pérdida: 0.1171
Época 9, Pérdida: 0.0987
Época 10, Pérdida: 0.0842
Época 11, Pérdida: 0.0732
Época 12, Pérdida: 0.0642
Época 13, Pérdida: 0.0566
Época 14, Pérdida: 0.0496
Época 15, Pérdida: 0.0433
Época 16, Pérdida: 0.0423
Época 17, Pérdida: 0.0339
Época 18, Pérdida: 0.0350
Época 19, Pérdida: 0.0311
Época 20, Pérdida: 0.0276
```

2.3 Evaluación del desempeño del modelo

Probemos como se desempeña ante datos nuevos el modelo entrenado con las imagenes de *CIFAR 10*

```
#Evaluar el modelo en el conjunto de prueba
modelo_pretrain.eval()
correctas = 0
total = 0

with torch.no_grad():#Desactiva calculo de gradientes para evaluación
    for datos, etiquetas in cargador_prueba_pretrain:
        datos = datos.to(dispositivo)
        etiquetas = etiquetas.to(dispositivo)

        #Obtener predicciones
        salidas = modelo_pretrain(datos)
        _, predichas = torch.max(salidas.data, 1)

        #Calcular metricas
        total += etiquetas.size(0)
        correctas += (predichas == etiquetas).sum().item()

#Calculamos y mostramos la precisión en el conjunto de prueba
```

```
print(f'\nPrecisión en el conjunto de prueba: {100 * correctas / total:.2f}%')
```

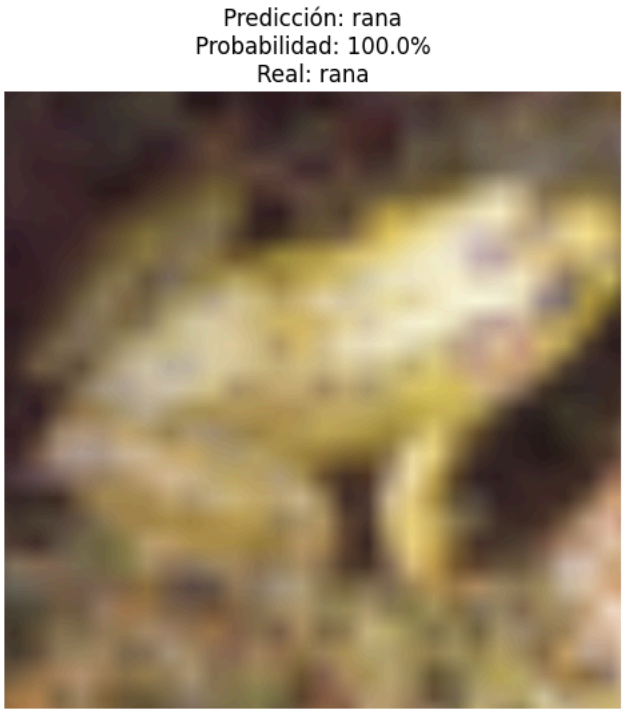
```
Precisión en el conjunto de prueba: 91.16%
```

```
#Análisis de clasificación
todas_etiquetas = []
todas_predicciones = []

with torch.no_grad():
    for datos, etiquetas in cargador_prueba_pretrain:
        datos = datos.to(dispositivo)
        salidas = modelo_pretrain(datos)
        _, predichas = torch.max(salidas, 1)

        todas_etiquetas.extend(etiquetas.cpu().numpy())
        todas_predicciones.extend(predichas.cpu().numpy())

clases = ['avión', 'auto', 'pájaro', 'gato', 'ciervo',
          'perro', 'rana', 'caballo', 'barco', 'camión']

print("\nReporte de clasificación para modelo preentrenado:")
print(classification_report(
    todas_etiquetas,
    todas_predicciones,
    target_names=clases,
    digits=4
))
```

```
Reporte de clasificación para modelo preentrenado:
              precision    recall  f1-score   support

           avión     0.8974    0.9530    0.9243      1000
            auto     0.9273    0.9700    0.9482      1000
          pájaro     0.8843    0.9020    0.8931      1000
            gato     0.8690    0.7760    0.8199      1000
          ciervo     0.9135    0.8980    0.9057      1000
           perro     0.8737    0.8720    0.8729      1000
            rana     0.9402    0.9430    0.9416      1000
         caballo     0.9036    0.9470    0.9248      1000
           barco     0.9549    0.9310    0.9428      1000
          camión     0.9506    0.9240    0.9371      1000

        accuracy                         0.9116     10000
       macro avg     0.9115    0.9116    0.9110     10000
    weighted avg     0.9115    0.9116    0.9110     10000
```

```
modelo_pretrain.eval()

im = rd.randint(0, 29) #Imagenes aleatorias en el conjunto de prueba

#Obtener un lote de prueba
dataiter_pretrain = iter(cargador_prueba_pretrain)
imagenes_pretrain, etiquetas_pretrain = next(dataiter_pretrain)

#Seleccionar la imagen a predecir
imagen_pretrain = imagenes_pretrain[im].unsqueeze(0).to(dispositivo)
etiqueta_real_pretrain = etiquetas_pretrain[im].item()

#Realizar predicción
with torch.no_grad():
    salidas_pretrain = modelo_pretrain(imagen_pretrain)
    probabilidades_pretrain = torch.nn.functional.softmax(salidas_pretrain, dim=1)
    prob_predicha_pretrain, clase_predicha_pretrain = torch.max(probabilidades_pretrain, 1)

# Preparar imagen para visualización (ajustando desnormalización)
imagen_mostrar_pretrain = imagen_pretrain.squeeze().cpu().numpy().transpose((1, 2, 0))
imagen_mostrar_pretrain = imagen_mostrar_pretrain * np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456, 0.406])
imagen_mostrar_pretrain = np.clip(imagen_mostrar_pretrain, 0, 1)  # Asegurar valores válidos

# Crear figura
plt.figure(figsize=(6, 6))
plt.imshow(imagen_mostrar_pretrain)
plt.title(f"Predicción: {clases[clase_predicha_pretrain.item()]}\n"
          f"Probabilidad: {prob_predicha_pretrain.item()*100:.1f}%\n"
          f"Real: {clases[etiqueta_real_pretrain]}")
```

```
plt.axis('off')
plt.show()
```



3. Conclusiones

De los modelos probados, destaca el modelo original de AlexNet por obvias razones, siendo superior. Sin embargo, el modelo creado fue muy bueno considerando las condiciones de tecnologia y recursos computacionales. Para la ejecución se uso la GPU que ofrece Google Colab y se obtuvieron los siguientes resultados.

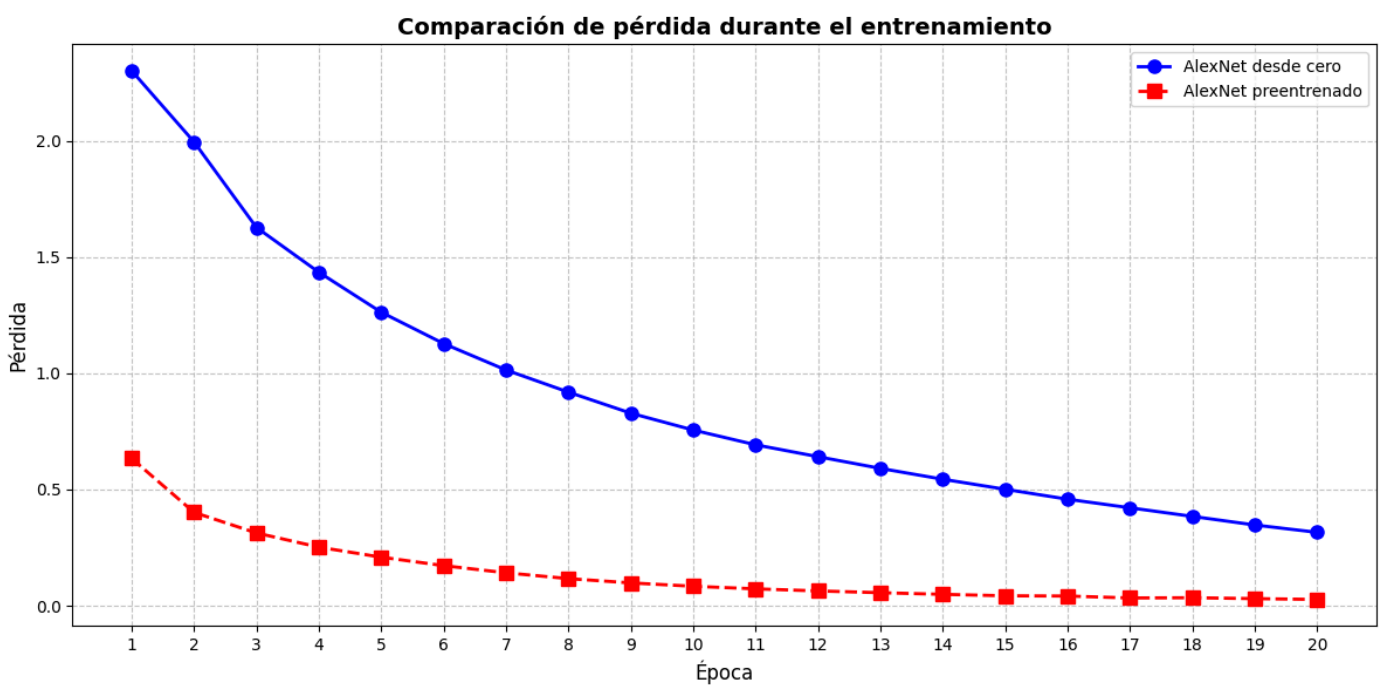
3.1 Comparación de Modelos

Métrica	AlexNet creado desde cero	AlexNet preentrenado
Tiempo de entrenamiento	10 min	42 min
Pérdida (Época 1)	2.30066	0.6339
Pérdida (Época 20)	0.31597	0.0276
Precisión en conjunto de prueba	**79.22%**	**91.16%**

Podemos notar como con un menor tiempo se obtuvieron buenos resultados con modelo construido desde cero ya que le tomo poco menos de 1/4 de tiempo de lo que le toma realizar 20 epocas al modelo preentrenado de AlexNet. Una gran diferencia es la ventaja de tener pesos y valores en las capas de convolución del modelo original ya que su perdida inicia en valores muy bajos, como es el caso en esta ejecución donde comenzo en 0.6339. Para datos nuevos, la precisión fue buena, en otras ejecuciones la precisión llego a 86%.

```
perdida_desde_cero = [
    2.3007, 1.9945, 1.6269, 1.4343, 1.2624,
    1.1276, 1.0140, 0.9189, 0.8283, 0.7559,
    0.6924, 0.6414, 0.5910, 0.5443, 0.5012,
    0.4585, 0.4216, 0.3847, 0.3477, 0.3160
]

perdida_preentrenado = [
    0.6339, 0.4018, 0.3134, 0.2524, 0.2093,
    0.1730, 0.1421, 0.1171, 0.0987, 0.0842,
    0.0732, 0.0642, 0.0566, 0.0496, 0.0433,
    0.0423, 0.0339, 0.0350, 0.0311, 0.0276
]

epocas = range(1, 21)

plt.figure(figsize=(12, 6))
```

```
#modelo desde cero
plt.plot(epocas, perdida_desde_cero, 'b-o', label='AlexNet desde cero', linewidth=2, markersize=8)

#modelo preentrenado
plt.plot(epocas, perdida_preentrenado, 'r--s', label='AlexNet preentrenado', linewidth=2, markersize=8)

plt.title('Comparación de pérdida durante el entrenamiento', fontsize=14, fontweight='bold')
plt.xlabel('Época', fontsize=12)
plt.ylabel('Pérdida', fontsize=12)
plt.xticks(np.arange(1, 21, step=1))
plt.legend()
plt.grid(True, linestyle='--', alpha=0.7)
#plt.yscale('log')
plt.tight_layout()

plt.show()
```



Metricas de rendimiento

AlexNet desde cero

Clase	Precision	Recall	F1-score	Support
avión	0.8623	0.7950	0.8273	1000
auto	0.8973	0.9090	0.9031	1000
pájaro	0.8505	0.6260	0.7212	1000
gato	0.6155	0.5810	0.5977	1000
ciervo	0.8085	0.7390	0.7722	1000
perro	0.6008	0.8550	0.7057	1000
rana	0.9495	0.7150	0.8157	1000
caballo	0.7810	0.9020	0.8371	1000
barco	0.8495	0.9090	0.8783	1000
camión	0.8327	0.8910	0.8609	1000
accuracy	**0.7922**			10000
macro avg	**0.8048**	**0.7922**	**0.7919**	10000
weighted avg	**0.8048**	**0.7922**	**0.7919**	10000

AlexNet Preentrenado

Clase	Precision	Recall	F1-score	Support
avión	0.8974	0.9530	0.9243	1000
auto	0.9273	0.9700	0.9482	1000
pájaro	0.8843	0.9020	0.8931	1000
gato	0.8690	0.7760	0.8199	1000
ciervo	0.9135	0.8980	0.9057	1000
perro	0.8737	0.8720	0.8729	1000
rana	0.9402	0.9430	0.9416	1000
caballo	0.9036	0.9470	0.9248	1000
barco	0.9549	0.9310	0.9428	1000
camión	0.9506	0.9240	0.9371	1000
accuracy	**0.9116**			10000
macro avg	**0.9115**	**0.9116**	**0.9110**	10000
weighted avg	**0.9115**	**0.9116**	**0.9110**	10000

Podemos notar algunas métricas de cómo clasifica el modelo hecho desde cero contra el modelo original. Por ejemplo:

- La precisión indica cuántas de las predicciones que el modelo hizo para una clase específica fueron correctas, siendo superior por 0.11.
- El recall o sensibilidad indica cuántos de los ejemplos reales de una clase fueron correctamente identificados por el modelo, siendo superior en casi la misma medida anterior.

Finalmente, podemos concluir que el modelo en PyTorch es "más rápido de entrenar" para obtener "buenos resultados" con pocos recursos computacionales, ya que incluso se llegó a correr en una PC con CPU y tardó alrededor de 40 a 60 min. Sin embargo, aunque es más tardado el modelo preentrenado para ajustarlo a los datos dados, es excelente para predecir. Debemos notar que el modelo AlexNet tiene varios años y aun así es eficiente; sin embargo, con más poder computacional se podría mejorar más el modelo, incluso creándolo desde cero.

```
import tensorflow as tf
from tensorflow.keras import Model, layers
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
import random as rd
import tensorflow_hub as hub
from tensorflow.keras.applications import VGG16
```

4. Modelo AlexNet desde cerp en TensorFlow

4.1 Carga de los datos

Cargamos los datos de la base CIFAR-10, que contiene imágenes clasificadas en 10 categorías. Importamos los conjuntos de entrenamiento para la red y el conjunto de prueba para ver como de desempeña el modelo ante datos nuevos. Debemos de ajustar los valores de los píxeles para normalizarlos ya que originalmente los píxeles van de 0 a 255, pero al dividirlos entre 2 resulta 127.5 y al restar 1 quedan dentro del rango -1 a 1.

Después, partimos los datos en lotes para el entrenamiento. Al agrupar las imágenes en conjuntos de 30 y mezclarlas aleatoriamente (con shuffle), se optimiza el proceso de aprendizaje, esto permite que el modelo generalice mejor y no dependa del orden de las muestras. En el conjunto de prueba, solo se divide en lotes sin mezclar, ya que estos datos no los ve el modelo cuando se esta entrenado y no tiene caso cambiar el orden de las imagenes.

```
#Cargar y preparar los datos de la base de CIFAR 10
(x_entren, y_entren), (x_prueba, y_prueba) = tf.keras.datasets.cifar10.load_data()

# Normalizar imagenes entre -1 y 1, donde l valor maximo de un pixel y al divirlo ente 2 = 127.5 quedando en un rango de 0 a 2 y al restar e
x_entren = (x_entren.astype('float32') / 127.5) - 1.0
x_prueba = (x_prueba.astype('float32') / 127.5) - 1.0

#Crear datasets
lote = 30
ds_entren = tf.data.Dataset.from_tensor_slices((x_entren, y_entren))
ds_entren = ds_entren.shuffle(1000).batch(lote)

ds_prueba = tf.data.Dataset.from_tensor_slices((x_prueba, y_prueba)).batch(lote)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
```

4.2 Creación del modelo AlexNet

De igual manera que en PyTorch, vamos a crear AlexNet usando las 11 capas mencionadas: 5 capas convolucionales, 3 capas Maxpooling y 3 capas densasa. Nuevamente extraemos las caracteristicas con las capas convolucionales y usamos ReLU al final de la capa para despues aplicar maxpooling. Finalmente, aplanamos los tensores producidos en el bloque convolucional y defininimos las capas densas en donde llegaran las caracteristicas extraidas y tendran una salida de 10 neuronas que son las clases a clasificar. Lo que resalta al definir el modelo en TesorFlow, es la facilidad de agregar capas, es más sencillo que PyTorch y la estructura es más entendible a primera vista.

```
#Definir el modelo AlexNet con TensorFlow
class AlexNet(Model):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()
        #Capas Convolucionales y Maxpooling
        self.caracteristicas = tf.keras.Sequential([
            layers.Conv2D(96, 3, padding='same', activation='relu', input_shape=(32, 32, 3)), #Conv1: recibe las imagenes
            layers.MaxPooling2D(3, strides=2, padding='valid'), #Maxpool1
            layers.Conv2D(256, 3, padding='same', activation='relu'), #Conv2
            layers.MaxPooling2D(3, strides=2, padding='valid'), #Maxpool2
            layers.Conv2D(384, 3, padding='same', activation='relu'), #Conv3
            layers.Conv2D(384, 3, padding='same', activation='relu'), #Conv4
            layers.Conv2D(256, 3, padding='same', activation='relu'), #Conv5
            layers.MaxPooling2D(3, strides=2, padding='valid') #Maxpool3
        ])

        #Capas Densas
```

```
        self.clasificador = tf.keras.Sequential([
            layers.Flatten(), #Aplanar las entradas qu
            layers.Dense(4096, activation='relu'),
            layers.Dense(4096, activation='relu'),
            layers.Dense(num_classes)
        ])

    def call(self, x):
        x = self.caracteristicas(x)
        x = self.clasificador(x)
        return x
```

4.3 Entrenamiento del modelo

Para el entrenamiento, es muy sencillo compilar y entrenar el modelo, ya que al compilarlo solo se define el optimizador, que en este caso fue el mismo que en los demás modelo (Descenso del Gradiente Estocástico), y la función de perdida tambien fue la misma (Entropia Cruzada). Con el metodo *fit* entrenamos el modelo y usamos 20 epocas, esto es bastante sencillo si comporamos la cantidad de codigo empleado en PyTorch.

```
#Compilar modelo
modelo = AlexNet()
modelo.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)

#Entrenamiento
historial = modelo.fit(ds_entren, epochs=20)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 38s 18ms/step - accuracy: 0.2421 - loss: 2.0359
Epoch 2/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 34s 17ms/step - accuracy: 0.5005 - loss: 1.3786
Epoch 3/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 43s 18ms/step - accuracy: 0.6012 - loss: 1.1203
Epoch 4/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.6694 - loss: 0.9389
Epoch 5/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.7178 - loss: 0.8055
Epoch 6/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.7592 - loss: 0.6924
Epoch 7/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 41s 18ms/step - accuracy: 0.7887 - loss: 0.6062
Epoch 8/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.8114 - loss: 0.5369
Epoch 9/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 40s 17ms/step - accuracy: 0.8390 - loss: 0.4635
Epoch 10/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 42s 18ms/step - accuracy: 0.8624 - loss: 0.3972
Epoch 11/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 41s 18ms/step - accuracy: 0.8848 - loss: 0.3293
Epoch 12/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 41s 18ms/step - accuracy: 0.9058 - loss: 0.2718
Epoch 13/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 41s 18ms/step - accuracy: 0.9270 - loss: 0.2132
Epoch 14/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.9425 - loss: 0.1675
Epoch 15/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.9570 - loss: 0.1273
Epoch 16/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 30s 18ms/step - accuracy: 0.9648 - loss: 0.1025
Epoch 17/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 40s 17ms/step - accuracy: 0.9701 - loss: 0.0847
Epoch 18/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 29s 17ms/step - accuracy: 0.9778 - loss: 0.0683
Epoch 19/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 41s 17ms/step - accuracy: 0.9830 - loss: 0.0500
Epoch 20/20
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 29s 17ms/step - accuracy: 0.9850 - loss: 0.0473
```

Al finalizar el entrenamiento, el acuracy fue del 98% y una perdida de 0.0468 tardando 10 min aproximadamente pero a diferencia de PyTorch, el desempeño fue mejor en el entrenamiento.Sin emnargo, en la precisión del conjunto de prueba fue casi la misma, obteniendo un 80.4%.

```
#Evaluación del modelo
pérdida_prueba, precisión_prueba = modelo.evaluate(ds_prueba)
print(f'\nPrecisión en conjuto de prueba: {precisión_prueba*100:.2f}%')
```

```
334/334 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.8126 - loss: 0.9261

    Precisión en conjuto de prueba: 80.76%
```

```
#Reporte de clasificación
predichos = []
reales = []
for imgs, etiq in ds_prueba:
    logits = modelo.predict(imgs)
    predichos.extend(tf.argmax(logits, axis=1).numpy())
    reales.extend(etiq.numpy().flatten())

clases = ['avión', 'auto', 'pájaro', 'gato', 'ciervo',
          'perro', 'rana', 'caballo', 'barco', 'camión']

print("\nReporte de clasificación:")
print(classification_report(reales, predichos, target_names=clases, digits=4))
```

```
       macro avg     0.8150    0.8076    0.8090     10000
    weighted avg     0.8150    0.8076    0.8090     10000
```

```
#Visualización de predicciones
lote_prueba = next(iter(ds_prueba))
imagenes, etiquetas = lote_prueba
índice = rd.randint(0, imagenes.shape[0] - 1)

#Obtener predicción
imagen = imagenes[índice]
etiqueta_real = etiquetas[índice].numpy().item()
logits = modelo.predict(tf.expand_dims(imagen, axis=0))
probabilidades = tf.nn.softmax(logits).numpy()
confianza = np.max(probabilidades) * 100
clase_predicha = np.argmax(probabilidades)

#Imagen para visualización
imagen_mostrar = (imagen * 0.5 + 0.5).numpy()

plt.figure(figsize=(6, 6))
plt.imshow(imagen_mostrar)
plt.title(f"Predicción: {clases[clase_predicha]}\n"
          f"Probailidad: {confianza:.1f}%\n"
          f"Real: {clases[etiqueta_real]}")
plt.axis('off')
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 919ms/step
```

4. AlexNet preentrenado

Dado que no se pueden inicalizar los pesos como en AlexNet de Pytorch, se opto po usar VGG16.

4.1 Cargar el modelo

De nuevo, se cargan los dastos de entrenamiento y prueba pero se carga el modelo VGG16 con sus respectivos pesos y los "congelamos", solo entrenaremos la capa final.

```
#Cargar datos CIFAR-10
(x_entren, y_entren), (x_prueba, y_prueba) = tf.keras.datasets.cifar10.load_data()

#Preprocesamiento para VGG16
```

```
def preprocesar_vgg(imagenes, etiquetas):
    imagenes = tf.image.resize(imagenes, [224, 224]) #VGG16 requiere imagenes de  224x224
    imagenes = tf.keras.applications.vgg16.preprocess_input(imagenes) #Normalización
    return imagenes, etiquetas

# Configurar datasets
lote = 30
ds_entren_vgg = tf.data.Dataset.from_tensor_slices((x_entren, y_entren))
ds_entren_vgg = ds_entren_vgg.map(preprocesar_vgg).shuffle(1000).batch(lote)

ds_prueba_vgg = tf.data.Dataset.from_tensor_slices((x_prueba, y_prueba))
ds_prueba_vgg = ds_prueba_vgg.map(preprocesar_vgg).batch(lote)

#Cargar VGG16 preentrenado
base_vgg = VGG16(
    weights='imagenet',
    include_top=False, #No incluimos las capas densas finales
    input_shape=(224, 224, 3)
)
base_vgg.trainable = False #Congelar pesos preentrenados

#Definir modelo personalizado para CIFAR 10
modelo_vgg = tf.keras.Sequential([
    base_vgg,
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(512, activation='relu'),
    layers.Dense(10, activation='softmax')
])


#Compilar el modelo
modelo_vgg.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=['accuracy']
)

#Entrenar el modelo
historial_vgg = modelo_vgg.fit(
    ds_entren_vgg,
    epochs=10,
    validation_data=ds_prueba_vgg
)
```

```
Epoch 1/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 359s 212ms/step - accuracy: 0.7461 - loss: 0.9890 - val_accuracy: 0.8624 - val_loss: 0.4079
Epoch 2/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 354s 211ms/step - accuracy: 0.9272 - loss: 0.2142 - val_accuracy: 0.8721 - val_loss: 0.4259
Epoch 3/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 354s 211ms/step - accuracy: 0.9727 - loss: 0.0803 - val_accuracy: 0.8818 - val_loss: 0.4778
Epoch 4/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 354s 212ms/step - accuracy: 0.9933 - loss: 0.0236 - val_accuracy: 0.8816 - val_loss: 0.5601
Epoch 5/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 382s 211ms/step - accuracy: 0.9987 - loss: 0.0079 - val_accuracy: 0.8837 - val_loss: 0.6117
Epoch 6/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 382s 212ms/step - accuracy: 0.9995 - loss: 0.0030 - val_accuracy: 0.8871 - val_loss: 0.6444
Epoch 7/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 353s 211ms/step - accuracy: 1.0000 - loss: 8.5145e-04 - val_accuracy: 0.8886 - val_loss: 0.6721
Epoch 8/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 350s 192ms/step - accuracy: 1.0000 - loss: 4.4437e-04 - val_accuracy: 0.8893 - val_loss: 0.6910
Epoch 9/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 354s 211ms/step - accuracy: 1.0000 - loss: 3.2362e-04 - val_accuracy: 0.8896 - val_loss: 0.7059
Epoch 10/10
1667/1667 ━━━━━━━━━━━━━━━━━━━━ 349s 191ms/step - accuracy: 1.0000 - loss: 2.6002e-04 - val_accuracy: 0.8893 - val_loss: 0.7171
```

4.2 Analisis del modelo entrenado.

Notemos que es más lento, usando el mismo optimizador.

```
# Evaluaar el modelo
perdida_vgg, precision_vgg = modelo_vgg.evaluate(ds_prueba_vgg)
print(f'\nPrecisión VGG16: {precision_vgg*100:.2f}%')
```

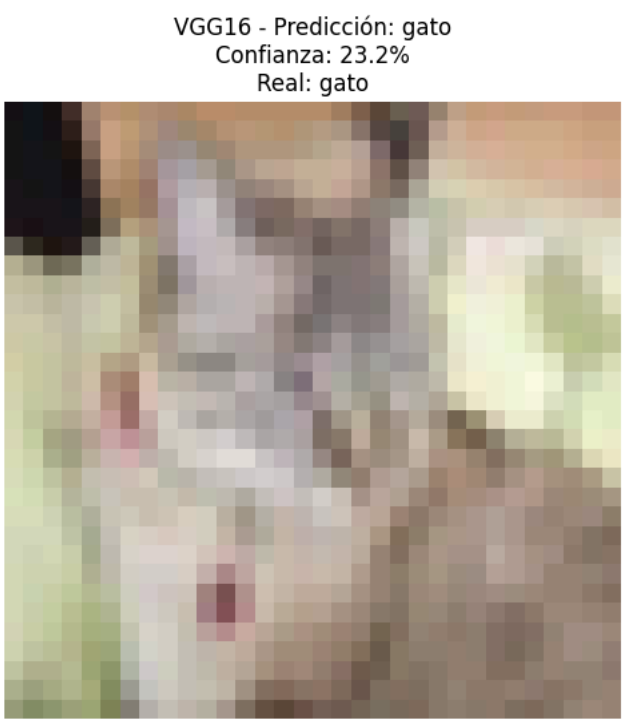
```
334/334 ━━━━━━━━━━━━━━━━━━━━ 54s 160ms/step - accuracy: 0.8866 - loss: 0.7426
```

```
Precisión VGG16: 88.93%
```

```
# Reporte de clasificación VGG16
predichos_vgg = []
reales_vgg = []
for imgs, etiq in ds_prueba_vgg:
    logits = modelo_vgg.predict(imgs, verbose=0)
    predichos_vgg.extend(tf.argmax(logits, axis=1).numpy())
    reales_vgg.extend(etiq.numpy().flatten())

clases = ['avión', 'auto', 'pájaro', 'gato', 'ciervo',
          'perro', 'rana', 'caballo', 'barco', 'camión']

print("\nReporte de clasificación VGG16:")
print(classification_report(reales_vgg, predichos_vgg, target_names=clases, digits=4))
```

```
Reporte de clasificación VGG16:
              precision    recall  f1-score   support

       avión     0.9094    0.9230    0.9161      1000
        auto     0.9529    0.9310    0.9418      1000
      pájaro     0.8647    0.8370    0.8506      1000
        gato     0.7684    0.7930    0.7805      1000
      ciervo     0.8563    0.8700    0.8631      1000
       perro     0.8599    0.8160    0.8374      1000
        rana     0.8943    0.9310    0.9123      1000
      caballo     0.9268    0.9120    0.9194      1000
       barco     0.9240    0.9480    0.9358      1000
      camión     0.9395    0.9320    0.9357      1000

    accuracy                         0.8893     10000
   macro avg     0.8896    0.8893    0.8893     10000
weighted avg     0.8896    0.8893    0.8893     10000
```

```
#Visualización de algunas predicciones de VGG16
lote_prueba_vgg = next(iter(ds_prueba_vgg))
imagenes_vgg, etiquetas_vgg = lote_prueba_vgg
índice = rd.randint(0, imagenes_vgg.shape[0] - 1)

#Obtener imagen original sin preprocesar
imagen_original = x_prueba[índice]
imagen_procesada = imagenes_vgg[índice] #Imagen procesada para VGG de 224x224

#Hacer la predicción
logits_vgg = modelo_vgg.predict(tf.expand_dims(imagen_procesada, axis=0), verbose=0)
probabilidades_vgg = tf.nn.softmax(logits_vgg).numpy()
confianza_vgg = np.max(probabilidades_vgg) * 100
clase_predicha_vgg = np.argmax(probabilidades_vgg)
etiqueta_real_vgg = etiquetas_vgg[índice].numpy().item()

#Mostrar imagen original
plt.figure(figsize=(6, 6))
plt.imshow(imagen_original/255.0)
plt.title(f"VGG16 - Predicción: {clases[clase_predicha_vgg]}\n"
         f"Confianza: {confianza_vgg:.1f}%\n"
         f"Real: {clases[etiqueta_real_vgg]}")
plt.axis('off')
plt.show()
```



5. Conclusiones Finales

En base a los resultados obtenidos, podemos notar que la implementación en TensorFlow es más sencilla. Sin embargo, la precisión del modelo es casi la misma. Comaparado con los modelos preentreados, el entrenamiento con los datos es más tardado, en particular, VGG16 es más lento. En PyTorch, AlexNet es superior, se arriesga tiempo de entrenamiento pero se gana mucha precisión como se ve en las pruebas realizadas y en las métricas usadas. Como conclusión. me quedo con la implementación en TensorFlow pero si quisiera hacer algo desde cero para entender la estructura de la red neuronal usaría PyTorch, pues en este ejercicio pude entender el modelo.shas

Métrica	AlexNet creado desde cero (PyTorch)	AlexNet preentrenado (PyTorch)	AlexNet creado desde cero (Keras)	VGG6 preentrenado (Keras)
Tiempo de entrenamiento	10 min	42 min	10–12 min	1h y 20 min
Pérdida (Época 1)	2.30066	0.6339	2.0359	0.9890
Pérdida (Época final)	0.31597 (Época 20)	0.0276 (Época 20)	0.0473 (Época 20)	2.6002e-04 (Época 10)
Precisión en conjunto de prueba	**79.22%**	**91.16%**	**80.76%**	**88.93%**

Resultados usando la GPU de Google Colab.