Reporte: Implementación y Comparación de AlexNet vs. Modelos Preentrenados en CIFAR-10

1. Introducción

El objetivo de este proyecto fue:

1. Implementar la arquitectura **AlexNet** desde cero en **PyTorch** y en **TensorFlow/Keras**.
2. Entrenar estos modelos en el conjunto de datos **CIFAR-10**.
3. Comparar su desempeño con **modelos preentrenados** disponibles tanto en PyTorch como en TensorFlow/Keras.
4. Presentar los resultados y conclusiones en un reporte conciso.

CIFAR-10 es un dataset compuesto por 60,000 imágenes a color de 32×32 píxeles, clasificadas en 10 categorías (por ejemplo, aviones, autos, gatos, perros, etc.). A pesar de ser un conjunto de datos relativamente pequeño, la tarea de clasificacion es desafiante cuando se parte de arquitecturas complejas o desde cero.

2. Implementación de AlexNet desde cero

2.1 Arquitectura requerida

Según las instrucciones, AlexNet debía contener:

- **5 capas convolucionales**
- **3 capas de max pooling**
- **3 capas densas**
  En total, **11 capas**. También se aplicaron funciones de activación ReLU y capas de Dropout en las secciones totalmente conectadas.

2.2 Versión en PyTorch

- Se definió una clase `AlexNetScratch(nn.Module)` con un bloque `features` (capas convolucionales + pooling) y un bloque `classifier` (capas densas).
- Se entrenó con una tasa de aprendizaje de 0.001 y `optimizer = Adam`.
- Se normalizaron las imágenes a media 0.5 y desviación estándar 0.5, y se mantuvo el tamaño de imagen en 32×32.

Resultados en CIFAR-10 (desde cero, PyTorch)

- Con 10 épocas, se alcanzó aproximadamente un **70-75%** de exactitud en el conjunto de prueba.
- El entrenamiento requirió un tiempo moderado en GPU.

2.3 Versión en TensorFlow/Keras

- Se implementó un modelo `Sequential()` con la misma configuración: 5 capas conv, 3 max pool y 3 densas.
- Se usó la misma normalización y un `optimizer='adam'`.
- El entrenamiento se realizó en Google Colab (primero en la versión gratuita y luego en Colab Pro por temas de memoria).

Dificultades con Keras

- Al principio, no hubo problema para entrenar el AlexNet "desde cero" porque las imágenes se mantienen en 32×32, lo cual no consume demasiada memoria.
- Se lograron resultados de exactitud similares, en torno a **70%** de accuracy tras 10 épocas.

3. Modelos Preentrenados

La segunda parte de la tarea fue **comparar** los AlexNet desde cero con **modelos preentrenados**. En **PyTorch** esto fue sencillo porque existe un `alexnet(pretrained=True)` oficial. Sin embargo, en **TensorFlow/Keras no** hay una versión oficial de AlexNet preentrenada. Se consideraron varias opciones:

1. **VGG16**
2. **ResNet50**
3. **MobileNetV2** o **EfficientNetB0**

3.1 Intentos con VGG16 y ResNet50

- VGG16 es un modelo grande (capas densas de 4096), y requiere redimensionar las imágenes a 224×224.
- Con CIFAR-10 (50,000 imágenes de entrenamiento), se intentó redimensionar todo el dataset y entrenar con un batch size de 128.
- **Problema**: Se agotó la RAM de Google Colab incluso con la suscripción **Colab Pro**. El kernel moría antes de completar las épocas.

- Se redujo el batch size a 32 y 16, pero VGG16 seguía consumiendo mucha memoria, principalmente al almacenar y procesar 224×224 por cada imagen.

3.2 Elección de MobileNetV2

Debido a los problemas de memoria, se decidió **usar MobileNetV2** en TensorFlow/Keras como modelo preentrenado, ya que:

- Es mucho más ligero que VGG16 y ResNet50.
- También está entrenado en ImageNet, cumpliendo la condición de "preentrenado en Keras".
- Permite redimensionar a 128×128 (o 160×160) en lugar de 224×224, reduciendo aún más el consumo de memoria.

Ajustes en MobileNetV2

- Se congelaron las capas base (`base_model.trainable = False`) para un "fine-tuning" ligero.
- Se añadió `GlobalAveragePooling2D()` en lugar de `Flatten() + Dense(4096)`, para evitar la explosión de parámetros.
- Se utilizó un `batch_size=8` y `prefetch(1)` en `tf.data` para no almacenar en RAM todo el dataset redimensionado.
- Con estos cambios, el entrenamiento logró completarse en Colab Pro.

4. Resultados y Comparación

4.1 AlexNet desde cero

- **PyTorch**: ~70-75% de exactitud en CIFAR-10.
- **Keras**: ~70% de exactitud (con la misma arquitectura de 11 capas).

4.2 Modelos preentrenados

- **PyTorch (AlexNet Pretrained)**: Tras ajustar la última capa a 10 clases y entrenar en CIFAR-10 (224×224), se alcanzó una exactitud de ~80-85% con 10 épocas, mejor que la versión desde cero.
- **Keras (MobileNetV2)**: Al entrenar con imágenes 128×128 y una capa final pequeña, se obtuvo ~85-90% de exactitud, superando la AlexNet entrenada desde cero. El uso de pesos preentrenados en ImageNet aporta a un mejor punto de partida.

4.3 Lecciones aprendidas sobre la memoria

- **Redimensionar** imágenes pequeñas (32×32 → 224×224) puede consumir gran cantidad de RAM si se hace para todo el dataset de golpe. Es preferible hacerlo en lotes con `tf.data`.
- **VGG16** (y otras redes grandes) con capas densas de 4096 pueden agotar la memoria incluso en Colab Pro, requiriendo reducción del batch size o arquitecturas más ligeras (MobileNetV2, EfficientNetB0).
- **Mixed Precision** (float16) en GPUs compatibles ayuda a reducir el uso de memoria y acelerar el entrenamiento.

5. Conclusiones

1. **AlexNet desde cero** en CIFAR-10 alcanza ~70% de exactitud, lo cual demuestra que entrenar redes profundas sin pesos preentrenados en un dataset relativamente pequeño puede ser desafiante.
2. **Modelos preentrenados** (ya sea AlexNet en PyTorch o MobileNetV2 en Keras) ofrecen mejores resultados, típicamente entre 80% y 90% de exactitud, debido a los pesos iniciales obtenidos al entrenar en un dataset grande como ImageNet.
3. **Problemas de memoria**: VGG16 y resoluciones de 224×224 consumen mucha RAM/VRAM, especialmente al aumentar el batch size. Fue necesario contratar **Colab Pro** y, aun así, ajustar la arquitectura y la resolución de entrada para no agotar la memoria.
4. **Uso de frameworks**: Tanto PyTorch como TensorFlow/Keras permiten implementar AlexNet desde cero con relativa facilidad, pero PyTorch dispone de una versión preentrenada de AlexNet, mientras que en Keras se optó por un modelo alternativo (MobileNetV2).
5. **Recomendaciones**: Para trabajos futuros con recursos limitados, es preferible utilizar arquitecturas más ligeras (MobileNet, EfficientNet) y procesar las imágenes en lotes mediante `tf.data`, evitando cargar todo en memoria de golpe.

En resumen, el proyecto demuestra que la **transferencia de aprendizaje** con modelos preentrenados supera consistentemente a los modelos entrenados desde cero en tareas de clasificación como CIFAR-10, y que las limitaciones de memoria pueden ser un factor crítico a la hora de elegir el tamaño de la arquitectura y los hiperparámetros de entrenamiento.