Implementación de AlexNet desde Cero en PyTorch y TensorFlow/Keras con CIFAR-10

Introducción

AlexNet es una red neuronal convolucional (CNN) que revolucionó el campo del aprendizaje profundo al ganar el concurso ImageNet en 2012. Su arquitectura, compuesta por múltiples capas convolucionales, de pooling y densas, demostró que las redes profundas podían superar significativamente a los métodos tradicionales de visión por computadora. Desde entonces, ha sido una referencia clave en tareas de clasificación de imágenes.

El objetivo de este trabajo es implementar AlexNet desde cero en PyTorch y TensorFlow/Keras, entrenarlo con el conjunto de datos CIFAR-10 y comparar su rendimiento con modelos preentrenados disponibles en ambas plataformas. A través de esta comparación, buscamos evaluar las diferencias en precisión, tiempo de entrenamiento y eficiencia entre los modelos entrenados desde cero y aquellos que aprovechan el aprendizaje por transferencia (transfer learning).

AlexNet

AlexNet es una red neuronal convolucional profunda que fue desarrollada inicialmente por Alex Krizhevsky y sus colegas en 2012. Fue diseñada para clasificar imágenes para la competencia ImageNet LSVRC-2010, donde logró resultados de vanguardia.

AlexNet se compone de **11 capas**, organizadas en dos bloques principales:

1. **Extractor de Características:** Compuesto por **5 capas convolucionales** con activaciones ReLU y **3 capas de max pooling**, diseñadas para extraer patrones de la imagen de entrada, capturando estructuras de bajo y alto nivel.
2. **Clasificador:** Formado por **3 capas densas (fully connected, FC)** que transforman las características extraídas en probabilidades de clase. Para evitar el sobreajuste, AlexNet emplea **dropout** en las capas densas.

Conjunto de Datos CIFAR-10

CIFAR-10 es un conjunto de datos ampliamente utilizado en problemas de clasificación de imágenes. Contiene **60,000 imágenes en color de 32 × 32 píxeles**, divididas en **10 clases**: avión, automóvil, pájaro, gato, ciervo, perro, rana, caballo, barco y camión. Cada clase tiene **6,000 imágenes**, con una división de **50,000 para entrenamiento** y **10,000 para prueba**.

Implementación en PyTorch

Este modelo se entrenó utilizando una GPU en Google Colab por lo que también se importaron las librerías necesarias junto con la definición de una variable device, para que la máquina sepa que debe usar una GPU para entrenar el modelo si está disponible.

Carga y Preprocesamiento de los datos CIFAR-10

AlexNet espera que los datos de entrada tengan un tamaño de 224 × 224. Sin embargo, los conjuntos de datos como CIFAR-10 incluyen imágenes de 32 × 32. Por lo tanto, se cambió el tamaño de las imágenes durante el preprocesamiento. Además de cambiar el tamaño, también se normalizaron para garantizar

que los valores de los píxeles se encuentren dentro de un rango que acelere la convergencia durante el entrenamiento.

Definición del modelo AlexNet

Se definió la clase AlexNet con las capas convolucionales, de pooling y densas, siguiendo la arquitectura original

1. Extractor de Características (Capas Convolucionales y Max Pooling)

- **Conv1:** Aplica 64 filtros de tamaño **11 × 11** con un stride de 4, lo que reduce significativamente la dimensión espacial.
- **ReLU:** Introduce no linealidad, permitiendo que la red capture patrones complejos.
- **MaxPool:** Reduce el tamaño espacial mientras conserva las características más importantes.
- **Conv2-Conv5:** Refinan la extracción de características con filtros más pequeños, aprendiendo representaciones progresivamente más abstractas.

2. Clasificador (Capas Densas y Dropout)

- **FC1 y FC2:** Expanden la representación de las características a un espacio de **4096 dimensiones**, permitiendo que el modelo aprenda combinaciones complejas de características.
- **Dropout:** Apagado aleatorio de neuronas para reducir el sobreajuste.
- **FC3:** Asigna la representación final a las **10 clases** de CIFAR-10.

Función de pérdida : Se utilizó CrossEntropyLoss, que es perfecta para problemas de clasificación de múltiples clases como CIFAR-10.

Optimizador : El descenso de gradiente estocástico (SGD) con caída de peso y momentum es una opción recomendada para entrenar AlexNet. El momentum ayuda a que el modelo escape de los mínimos locales y la caída de peso lo regulariza.

Entrenamiento del Modelo

El modelo se entrenó durante 20 épocas utilizando el optimizador SGD con momentum. La función de pérdida utilizada fue CrossEntropyLoss.

Evaluación final del Modelo desde cero:

--- Accuracy por Clase ---

Avión: 88.30% Automóvil: 93.30%

Pájaro: 71.90%	Gato: 60.10%
Ciervo: 80.30%	Perro: 71.90%
Rana: 85.80%	Caballo: 80.70%
Barco: 83.80%	Camión: 81.10%

El modelo alcanzó una precisión del 79.72 % en el conjunto de prueba de CIFAR-10.

Implementación de modelo preentrenado PyTorch

Se importó el modelo preentrenado AlexNet de la librería torchvision.

Se reemplazó la última capa completamente conectada (model.classifier[6]), con ésto indicamos a AlexNet que ajuste sus predicciones al conjunto de datos. Las capas anteriores permanecen intactas y conservan las características aprendidas, mientras que la nueva capa aprende a clasificar las categorías específicas de CIFAR-10.

El modelo prentrenado alcanzó una precisión del 84 % en el conjunto de prueba de CIFAR-10.

Implementación en TensorFlow/Keras

Para mejorar la precisión del modelo se aumentó el conjunto de datos aplicando **Data Augmentation**, la cual es una técnica que aplica transformaciones aleatoria como:

- Volteo horizontal
- Rotación
- Zoom

Con esto el modelo puede mejorar la generalización

A continuación se definió el modelo utilizando la API secuencial de Keras, incluyendo capas convolucionales, de pooling y densas.

En esta red se incluye una capa de normalización por lotes (**BatchNormalization()**)

El modelo se compiló con el optimizador Adam y se entrenó durante 20 épocas.

El modelo alcanzó una precisión del 79% en el conjunto de prueba de CIFAR-10.

Implementación de modelo preentrenado para TensorFlow/Keras

Ya que TensorFlow no tiene AlexNet preentrenado oficialmente, se optó por utilizar los pesos entrenados proporcionados por el siguiente enlace de la **Universidad de Toronto**: [AlexNet implementation + weights in TensorFlow]

Desde el sitio web se descargó el archivo `bvlc_alexnet.npy` que contiene los pesos de AlexNet y a continuación se cargaron

Asignación de pesos

Los pesos fueron entrenados en **Caffe** por lo que las dimensiones de los pesos son diferentes a la arquitectura planteada en este trabajo. Esto sucede porque en Caffe, AlexNet fue entrenada con 2 GPUs, lo que dividía los canales en las capas convolucionales.

Ya que Caffe dividía las convoluciones en 2 GPUs, la mitad de los canales estaban en un grupo y la otra mitad en otro. En Keras, necesitamos concatenarlos para formar la arquitectura correcta y a continuación asignamos los pesos a las capas correspondientes.

Ya que AlexNet fue entrenada para clasificación de 1000 clases diferentes, procedímos cambiar la última capa densa por una capa de 10 salidas para coincidir con las clases de CIFAR-10 y a continuación se entrenaron únicamente las capas densas.

El modelo alcanzó una precisión del 70% en el conjunto de prueba de CIFAR-10.

Resultados

Comparación de Precisión

Los resultados obtenidos en este trabajo evidencian una diferencia notable entre los modelos implementados desde cero y los modelos pre entrenados. A continuación, se presenta un resumen de la precisión alcanzada por cada modelo en el conjunto de prueba de CIFAR-10:

Modelo	**Precisión (PyTorch)**	**Precisión (TensorFlow/Keras)**
AlexNet desde cero	79%	79%
AlexNet preentrenado	84%	70%

Análisis de los Resultados

- **Modelos desde cero:** Tanto en PyTorch como en TensorFlow/Keras, los modelos implementados desde cero lograron precisiones similares, cercanas al 79-80%. Esto

indica que las implementaciones personalizadas pueden aprender características relevantes, aunque su rendimiento sigue siendo inferior al de los modelos preentrenados.

- **Modelos preentrenados:** En PyTorch, el modelo preentrenado superó claramente al modelo desde cero, alcanzando un 84% de precisión. Sin embargo, en TensorFlow/Keras, la precisión del modelo preentrenado fue menor (70%). Esta diferencia puede deberse a que el modelo preentrenado en TensorFlow fue originalmente entrenado con **Caffe**, que utiliza una arquitectura optimizada para entrenar con **dos GPU** y con imágenes de **227x227 píxeles**, mientras que en este trabajo se empleó la arquitectura original de AlexNet. Estas diferencias en la forma de entrenamiento y en la arquitectura utilizada podrían haber afectado el desempeño del modelo en TensorFlow/Keras. Tal vez entrenando durante más tiempo o cambiando los hiperparámetros podría mejorar.

Conclusión

Principales Aprendizajes

- **Diferencias entre modelos desde cero y preentrenados:**
    - Entrenar un modelo desde cero permite comprender mejor la arquitectura y el proceso de optimización.
    - Los modelos preentrenados pueden ofrecer ventajas en términos de precisión, pero su desempeño depende de cómo fueron entrenados originalmente y de su compatibilidad con la tarea actual.
- **Impacto del framework en el desempeño:**
    - En PyTorch, el modelo preentrenado superó al modelo desde cero, lo que muestra que el uso de pesos preentrenados correctamente ajustados puede mejorar el rendimiento.
    - En TensorFlow/Keras, el modelo preentrenado tuvo menor precisión que el modelo desde cero, probablemente debido a diferencias en la arquitectura utilizada durante su entrenamiento original en Caffe. Esto destaca la importancia de evaluar la compatibilidad de un modelo preentrenado antes de implementarlo en una nueva tarea.

- **Factores a considerar al elegir un modelo preentrenado:**
    - Es crucial analizar la arquitectura original del modelo y sus condiciones de entrenamiento antes de aplicarlo a un nuevo conjunto de datos.
    - La resolución de las imágenes, la configuración del hardware y el framework utilizado pueden influir significativamente en el rendimiento final del modelo.

- **Comparación de Frameworks:**
    - **PyTorch** se destaca por su flexibilidad y facilidad para la investigación, lo que lo hace ideal para el desarrollo de nuevos modelos y ajustes personalizados.
    - **TensorFlow/Keras** es más conveniente para implementaciones optimizadas y producción, aunque el desempeño de modelos preentrenados puede verse afectado por su configuración original.