AlexNet

Introducción

El objetivo de este proyecto es recrear la arquitectura de AlexNet desde cero utilizando las bibliotecas de Python TensorFlow y Keras, y compararla con el rendimiento de modelos preentrenados.

AlexNet es una red neuronal convolucional (CNN) la cual tuvo un fuerte impacto, pues obtuvo resultados destacables en el desafío ImageNet 2012; en el presente trabajo, se implementó AlexNet, tal que, se entrenó utilizando el conjunto de datos CIFAR-10 y se evaluó su rendimiento. También se comparó con un modelo preentrenado, en este caso, VGG16, para evaluar la efectividad del entrenamiento desde cero frente a modelos que ya han sido entrenados.

Por su parte, el conjunto de datos CIFAR-10 contiene imágenes de 10 clases diferentes, lo que lo hace adecuado para el propósito de evaluar modelos de clasificación de imágenes, es decir, se entrenaron y evaluaron dos modelos: AlexNet desde cero y VGG16 preentrenado.

Metodología

Creación de AlexNet

En el código siguiente, se creó y entrenó el modelo de AlexNet desde cero utilizando la biblioteca TensorFlow y Keras. La red cuenta con cinco capas convolucionales, tres capas de max pooling y tres capas densas.

Para ello, se cargó y preprocesó el conjunto de datos CIFAR-10, que consiste en imágenes de 32x32 píxeles. Luego, se definió la arquitectura de AlexNet, comenzando con una capa convolucional seguida de la capa de max pooling,...

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10 # Cargar el conjunto de datos CIFAR-1
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0 # Normalización

# Convertir las etiquetas a formato one-hot
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

# Crear el modelo de AlexNet (desde cero)
def create_alexnet_model():
    model = models.Sequential()

    # Capa de entrada - convolución
    model.add(layers.Conv2D(96, (5, 5), strides=1, activation='relu', input_shape=(
    model.add(layers.MaxPooling2D((2, 2), strides=2))  # Del tamaño de pooling...

    # Capa 2
    model.add(layers.Conv2D(256, (5, 5), padding='same', activation='relu'))
    model.add(layers.MaxPooling2D((2, 2), strides=2))

    # Capa 3
    model.add(layers.Conv2D(384, (3, 3), padding='same', activation='relu'))

    # Capa 4
    model.add(layers.Conv2D(384, (3, 3), padding='same', activation='relu'))

    # Capa 5
    model.add(layers.Conv2D(256, (3, 3), padding='same', activation='relu'))
    model.add(layers.MaxPooling2D((2, 2), strides=2))  # Ajuste tamaño de pooling


    model.add(layers.Flatten()) # Aplanado

    # Capa densa 1
    model.add(layers.Dense(4096, activation='relu'))

    # Capa densa 2
    model.add(layers.Dense(4096, activation='relu'))

    # Capa de salida
    model.add(layers.Dense(10, activation='softmax'))

    return model
```

Entrenamiento de AlexNet

Consecuentemente, se creó el generador de aumento de datos para la rotación y desplazamiento de las imágenes durante el entrenamiento, pues ayuda a mejorar la capacidad de generalización del modelo. Después, se compiló y entrenó el modelo, utilizando un optimizador Adam con una tasa de aprendizaje pequeña y aplicando un esquema de ajuste de la tasa de aprendizaje (se observó que al aplicarlo, mejoró el accuracy de manera considerable).

Así mismo, se fue modificando el número de épocas hasta 50, pues se observó que de reducirlas, bajaba el accuracy de manera significativa, llegando a apenas 40-60% ; se compiló previamente con 10, 20,..., hasta llegar a 50 épocas.

```
# Generador aumento de datos
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

datagen.fit(x_train) # Ajustar el generador a los datos de entrenamiento

model_keras = create_alexnet_model()

# Optimizador
optimizer = tf.keras.optimizers.Adam(learning_rate=0.0001)

# Compilar el modelo
model_keras.compile(optimizer=optimizer,
                    loss='categorical_crossentropy',
                    metrics=['accuracy'])

# Configurar callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weigh
lr_scheduler = LearningRateScheduler(lambda epoch: 0.0001 * 10**(epoch / 20))

# Entrenar utilizando el generador de aumento de datos
history = model_keras.fit(datagen.flow(x_train, y_train, batch_size=64),
                          epochs=50,  # Aumentar el número de épocas
                          validation_data=(x_test, y_test),
                          callbacks=[early_stopping, lr_scheduler])

# Evaluar el modelo desde cero en el conjunto de prueba
test_loss, test_accuracy = model_keras.evaluate(x_test, y_test, verbose=2)
print(f'Pérdida en el conjunto de prueba (AlexNet desde cero): {test_loss}')
print(f'Precisión en el conjunto de prueba (AlexNet desde cero): {test_accuracy}'
```

Comparación con VGG16 preentrenado

Luego de entrenar AlexNet, se procedió a cargar un modelo preentrenado (VGG16). Este modelo ya ha sido entrenado en el conjunto de datos ImageNet, lo que resulta útil para la clasificación en CIFAR-10. Para este caso, VGG16 se ajustó tal que se adaptara a las 10 clases de CIFAR-10.

```
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Añadir nuevas capas para adaptarlo a CIFAR-10
model_pretrained = models.Sequential()
model_pretrained.add(base_model)
model_pretrained.add(layers.GlobalAveragePooling2D())  # Aplanado global
model_pretrained.add(layers.Dense(10, activation='softmax'))  # Capa de salida para

# Compilar (el preentrenado)
model_pretrained.compile(optimizer='adam', loss='categorical_crossentropy', metrics

# Entrenamiento del preentrenado (ajustando las capas superiores)
history_pretrained = model_pretrained.fit(x_train, y_train, epochs=10, batch_size=6

# Evaluar el preentrenado en el conjunto de prueba
test_loss_pretrained, test_accuracy_pretrained = model_pretrained.evaluate(x_test,
print(f'Modelo preentrenado (VGG16) - Pérdida: {test_loss_pretrained}, Precisión: {
```

```
Epoch 1/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 53s 60ms/step - accuracy: 0.2715 - loss: 1.9385 -
Epoch 2/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 73s 53ms/step - accuracy: 0.4729 - loss: 1.4397 -
Epoch 3/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 55ms/step - accuracy: 0.5482 - loss: 1.2445 -
Epoch 4/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.6038 - loss: 1.1110 -
Epoch 5/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.6425 - loss: 1.0109 -
Epoch 6/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.6660 - loss: 0.9491 -
Epoch 7/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.6931 - loss: 0.8701 -
Epoch 8/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 80s 50ms/step - accuracy: 0.7063 - loss: 0.8474 -
Epoch 9/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 38s 49ms/step - accuracy: 0.7202 - loss: 0.7968 -
Epoch 10/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 38s 49ms/step - accuracy: 0.7337 - loss: 0.7662 -
Epoch 11/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 48ms/step - accuracy: 0.7368 - loss: 0.7579 -
Epoch 12/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 50ms/step - accuracy: 0.7462 - loss: 0.7350 -
Epoch 13/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.7529 - loss: 0.7172 -
Epoch 14/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 54ms/step - accuracy: 0.7550 - loss: 0.7088 -
Epoch 15/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 43s 55ms/step - accuracy: 0.7561 - loss: 0.7014 -
```

```
Epoch 16/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.7578 - loss: 0.6998 -
Epoch 17/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.7596 - loss: 0.7033 -
Epoch 18/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.7590 - loss: 0.7062 -
Epoch 19/50
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 52ms/step - accuracy: 0.7573 - loss: 0.7133 -
313/313 - 2s - 7ms/step - accuracy: 0.7912 - loss: 0.6170
Pérdida en el conjunto de prueba (AlexNet desde cero): 0.6169846653938293
Precisión en el conjunto de prueba (AlexNet desde cero): 0.7911999821662903
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicat
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 63s 64ms/step - accuracy: 0.2154 - loss: 1.9971 -
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 65s 53ms/step - accuracy: 0.5206 - loss: 1.2766 -
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 53ms/step - accuracy: 0.6691 - loss: 0.9501 -
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 83s 54ms/step - accuracy: 0.7383 - loss: 0.7654 -
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 82s 54ms/step - accuracy: 0.7859 - loss: 0.6349 -
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 52ms/step - accuracy: 0.8187 - loss: 0.5465 -
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 51ms/step - accuracy: 0.8420 - loss: 0.4670 -
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 51ms/step - accuracy: 0.8642 - loss: 0.4171 -
Epoch 9/10
```

Conclusión

AlexNet y VGG16 muestran un notable desempeño en cuanto a accuracy y pérdida. En el caso de AlexNet, el modelo obtuvo una precisión en el conjunto de prueba de aproximadamente 79.12%, con una pérdida de 0.6170. Este desempeño mostró buen desempeño considerando que se entrenó desde cero. Alcanzó un rendimiento sólido tras 19 épocas.

Así mismo, el modelo VGG16 preentrenado alcanzó una precisión en el conjunto de prueba de 78.62% con una pérdida de 0.7258, es decir, el modelo preentrenado no mejoró significativamente la precisión respecto al modelo AlexNet.

Lamentablemente, debido al límite de Google Colab para utilizar GPU, no se pudo continuar con la modificación del código para realizar optimizaciones adicionales. Se intentó continuar el trabajo utilizando plataformas como Kaggle y Azure, pero ambas presentaron restricciones que no permitieron trabajar el código, por lo que se vió limitada la capacidad de afinar el modelo y por tanto mejorar su rendimiento.