I. Các thành phần của CuDNN

1. Tensor Operations

- **Vai trò:** Là thành phần nền tảng để quản lý và xử lý dữ liệu trong mạng nơ-ron.
- **Tại sao cần thiết:** Tất cả các phép toán trong Deep Learning (Convolution, Pooling, RNN, v.v.) đều hoạt động trên tensor.
- **Ví dụ ứng dụng:**
    - Định nghĩa đầu vào và đầu ra của mô hình.
    - Biến đổi hình dạng tensor phù hợp với cấu trúc mạng.

2. Convolutional Neural Networks (CNNs) - Nếu dùng mạng CNN

- **Vai trò:** Thực hiện các phép toán tích chập (convolution), một thành phần không thể thiếu trong mạng CNN.
- **Tại sao cần thiết:** Các mạng xử lý hình ảnh và video (ResNet, VGG, YOLO) dựa trên phép tích chập.
- **Ví dụ ứng dụng:**
    - Dự đoán hình ảnh, nhận diện khuôn mặt.
    - Xử lý ảnh trong mô hình tự động hóa.

3. Activation Functions

- **Vai trò:** Tạo ra phi tuyến tính trong mô hình, giúp mạng học được các mẫu phức tạp.
- **Tại sao cần thiết:** Hầu hết các mô hình đều cần một hàm kích hoạt như ReLU, Sigmoid hoặc Tanh ở mỗi lớp.
- **Ví dụ ứng dụng:**
    - ReLU trong mạng nơ-ron sâu.
    - Sigmoid ở đầu ra của mô hình phân loại.

4. Pooling Layers - Nếu dùng giảm kích thước đầu ra

- **Vai trò:** Giảm kích thước tensor, làm nhẹ dữ liệu mà vẫn giữ được thông tin quan trọng.
- **Tại sao cần thiết:** Giúp giảm chi phí tính toán trong các tầng mạng nơ-ron.
- **Ví dụ ứng dụng:**
    - Max Pooling trong CNN để giảm số lượng feature maps.
    -

5. Recurrent Neural Networks (RNNs) - Nếu dùng mạng RNN

- **Vai trò:** Xử lý dữ liệu tuần tự, như chuỗi thời gian, văn bản hoặc âm thanh.

- **Tại sao cần thiết:** Là thành phần chính của LSTM và GRU, được dùng rộng rãi trong xử lý ngôn ngữ tự nhiên (NLP).
- **Ví dụ ứng dụng:**
    - Dịch máy, phân tích cảm xúc.
    - Mô hình dự đoán chuỗi thời gian.

6. Memory Management

- **Vai trò:** Quản lý tài nguyên GPU hiệu quả, giảm overhead và tăng hiệu năng.
- **Tại sao cần thiết:** Các ứng dụng cần khởi tạo và giải phóng bộ nhớ GPU đúng cách.
- **Ví dụ ứng dụng:**
    - cudnnCreate để khởi tạo handle.
    - cudnnDestroy để giải phóng tài nguyên.

II. Các hàm cơ bản

1. Memory Management (Quản lý bộ nhớ)

Các hàm này được sử dụng để khởi tạo, giải phóng tài nguyên và quản lý ngữ cảnh của CuDNN.

- **cudnnCreate**
    - **Mô tả:** Khởi tạo một handle cho CuDNN.
    - **Cách dùng:**

```
cudnnHandle_t handle;
cudnnCreate(&handle);
```

    - **Ý nghĩa:** Tất cả các tác vụ CuDNN yêu cầu handle này để thực thi.
- **cudnnDestroy**
    - **Mô tả:** Giải phóng tài nguyên CuDNN.
    - **Cách dùng:**

```
cudnnDestroy(handle);
```

2. Tensor Operations (Phép toán trên Tensor)

Định nghĩa tensor là cơ sở dữ liệu của CuDNN.

- **cudnnCreateTensorDescriptor**
    - **Mô tả:** Tạo một tensor descriptor để mô tả cấu trúc của tensor (batch size, channels, height, width).

- **Cách dùng:**

```
cudnnTensorDescriptor_t tensorDesc;
cudnnCreateTensorDescriptor(&tensorDesc);
```

- **cudnnSetTensor4dDescriptor**
  - **Mô tả:** Thiết lập tensor với cấu trúc 4D (batch, channel, height, width).
  - **Cách dùng:**
    cudnnSetTensor4dDescriptor(tensorDesc,CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, batchSize, channels, height, width);

- **cudnnDestroyTensorDescriptor**
  - **Mô tả:** Giải phóng descriptor sau khi sử dụng.
  - **Cách dùng:**
    cudnnDestroyTensorDescriptor(tensorDesc);

3. Convolution Operations (Phép toán tích chập)

Các hàm thực hiện tích chập trong mạng CNN.

- **cudnnCreateConvolutionDescriptor**
  - **Mô tả:** Tạo descriptor cho phép tích chập.
  - **Cách dùng:**

```
cudnnConvolutionDescriptor_t convDesc;
cudnnCreateConvolutionDescriptor(&convDesc);
```

- **cudnnSetConvolution2dDescriptor**
  - **Mô tả:** Thiết lập thông số cho phép tích chập (stride, padding, dilation).
  - **Cách dùng:**
    cudnnSetConvolution2dDescriptor(convDesc, padHeight, padWidth, strideHeight, strideWidth, dilationHeight, dilationWidth, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT);
- **cudnnConvolutionForward**
  - **Mô tả:** Thực hiện phép tích chập tiến (forward pass).
  - **Cách dùng:**
    cudnnConvolutionForward(handle, &alpha, inputTensorDesc, inputData, filterDesc, filterData, convDesc, algo, workspace, workspaceSize, &beta, outputTensorDesc, outputData);

4. Activation Functions (Hàm kích hoạt)

Xử lý các hàm kích hoạt như ReLU, Sigmoid, Tanh.

- **cudnnCreateActivationDescriptor**
  - **Mô tả:** Tạo descriptor cho hàm kích hoạt.
  - **Cách dùng:**

```
cudnnActivationDescriptor_t activationDesc;
cudnnCreateActivationDescriptor(&activationDesc);
```

- **cudnnSetActivationDescriptor**
  - **Mô tả:** Thiết lập loại hàm kích hoạt và các tham số.
  - **Cách dùng:**
    cudnnSetActivationDescriptor(activationDesc,CUDNN_ACTIVATION_RELU, CUDNN_PROPAGATE_NAN, 0.0);
- **cudnnActivationForward**
  - **Mô tả:** Thực hiện hàm kích hoạt trong pass tiến.
  - **Cách dùng:**
    cudnnActivationForward(handle, activationDesc, &alpha, inputTensorDesc, inputData, &beta, outputTensorDesc, outputData);

5. Pooling Operations (Phép pooling)

Thực hiện giảm kích thước đầu ra bằng max pooling hoặc average pooling.

- **cudnnCreatePoolingDescriptor**
  - **Mô tả:** Tạo descriptor cho phép pooling.
  - **Cách dùng:**

```
cudnnPoolingDescriptor_t poolingDesc;
cudnnCreatePoolingDescriptor(&poolingDesc);
```

- **cudnnSetPooling2dDescriptor**
  - **Mô tả:** Thiết lập thông số pooling (loại pooling, kích thước, stride).
  - **Cách dùng:**
    cudnnSetPooling2dDescriptor(poolingDesc, CUDNN_POOLING_MAX, CUDNN_PROPAGATE_NAN, windowHeight, windowWidth, padHeight, padWidth, strideHeight, strideWidth);

- **cudnnPoolingForward**
    - **Mô tả:** Thực hiện phép pooling trong pass tiến.
    - **Cách dùng:**
      cudnnPoolingForward(handle, poolingDesc, &alpha, inputTensorDesc, inputData, &beta, outputTensorDesc, outputData);

III. Cách cài đặt CuDNN

1. Trên máy có GPU

1. Cài đặt CUDA Toolkit:

- Tải và cài đặt phiên bản CUDA tương thích từ CUDA Toolkit.
- Sau khi cài đặt, kiểm tra phiên bản bằng lệnh: nvcc --version

2. Tải CuDNN

- Tạo tài khoản NVIDIA Developer:
    - Đăng ký tại NVIDIA Developer Program.
- Tải CuDNN:
    - Truy cập trang tải CuDNN.
    - Chọn phiên bản CuDNN phù hợp với phiên bản CUDA đã cài đặt.
    - Tải file CuDNN dạng .zip dành cho Windows.

---

3. Cài đặt CuDNN

- Giải nén CuDNN:
    - Giải nén file .zip bạn vừa tải về.
- Sao chép các file vào thư mục CUDA:
    - Vào thư mục CuDNN đã giải nén, bạn sẽ thấy các thư mục con: include, lib, và bin.
    - Sao chép nội dung từ các thư mục này vào thư mục cài đặt CUDA trên hệ thống, mặc định là:
        - cuda/include/ → C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vX.Y\include
        - cuda/lib/x64/ → C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vX.Y\lib\x64
        - cuda/bin/ → C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vX.Y\bin

4. Kiểm tra cài đặt

- Kiểm tra phiên bản CuDNN:
    - Mở Command Prompt, chuyển đến thư mục include của CUDA, ví dụ:
    - **cd "C:\Program Files\NVIDIA GPU ComputingToolkit\CUDA\vX.Y\include"**
- Sử dụng lệnh sau để kiểm tra phiên bản CuDNN:
    - **findstr "CUDNN_MAJOR" cudnn_version.h**
- Đầu ra ví dụ:
  #define CUDNN_MAJOR 8
  #define CUDNN_MINOR 9
  #define CUDNN_PATCHLEVEL 6

- Kiểm tra bằng TensorFlow hoặc PyTorch:
    - Cài đặt TensorFlow hoặc PyTorch trong Python.
    - Chạy đoạn mã sau để kiểm tra GPU và CuDNN:
      import tensorflow as tf
      print("CuDNN Enabled:", tf.test.is_built_with_cuda())
      print("GPU Available:", tf.config.list_physical_devices('GPU'))

2. Trên Colab

Google Colab tích hợp sẵn CuDNN, vì vậy bạn không cần phải cài đặt thủ công. Tuy nhiên, nếu muốn xác minh phiên bản hoặc thay đổi phiên bản CuDNN, bạn có thể thực hiện như sau:

- **Kiểm tra phiên bản CUDA trên Colab**
  !nvcc --version

  Đầu ra ví dụ:
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2023 NVIDIA Corporation
  Built on Tue_Aug_15_22:02:13_PDT_2023
  Cuda compilation tools, release 12.2, V12.2.140

- **Kiểm tra phiên bản CUDNN trên Colab**
  !cat /usr/include/cudnn_version.h | grep CUDNN_MAJOR -A 2

  Đầu ra ví dụ:
  #define CUDNN_MAJOR 8
  #define CUDNN_MINOR 9
  #define CUDNN_PATCHLEVEL 6

IV. Các khái niệm tóm tắt về GPU

1. Thread (Luồng)

- Thread là đơn vị xử lý nhỏ nhất trong một GPU. Mỗi thread thực hiện một phần nhỏ của công việc, ví dụ như tính toán một phép toán.
- GPU có thể chạy hàng nghìn, thậm chí hàng triệu threads song song.
- Mỗi thread có một ID riêng để phân biệt với các threads khác.

2. Block (Khối)

- Block là một nhóm các threads. Các threads trong một block có thể giao tiếp và chia sẻ dữ liệu với nhau qua bộ nhớ chung.
- Một block có thể chứa hàng trăm hoặc hàng nghìn threads, tùy thuộc vào cấu trúc của GPU.
- Threads trong cùng một block có thể đồng bộ hóa với nhau khi cần thiết.

3. Grid (Lưới)

- Grid là tập hợp các blocks. Mỗi block trong grid có thể chạy độc lập, và mỗi thread trong block thực hiện một tác vụ nhỏ của công việc.
- Grid đại diện cho một bài toán lớn, được chia thành các phần nhỏ hơn để xử lý song song.

4. Warp

- Warp là nhóm gồm 32 threads chạy cùng lúc trên một đơn vị xử lý của GPU (SM). Các threads trong một warp thực hiện cùng một lệnh tại cùng một thời điểm.
- Nếu các threads trong một warp không thể thực thi đồng thời (ví dụ vì điều kiện kiểm tra không khớp), GPU phải sử dụng SIMT (Single Instruction, Multiple Threads) để xử lý các tình huống này, nghĩa là GPU sẽ xử lý từng thread một trong warp khi cần thiết.

5. SM (Streaming Multiprocessor)

- SM (Streaming Multiprocessor) là một đơn vị xử lý quan trọng trong kiến trúc của GPU. Một GPU có thể có nhiều SMs, mỗi SM có thể xử lý nhiều warps cùng lúc.
- SM là nơi thực thi các lệnh và điều phối việc thực hiện của các threads và warps.
- Mỗi SM có bộ nhớ riêng, và có thể truy cập bộ nhớ chia sẻ (shared memory) để các threads trong một block có thể giao tiếp với nhau.

6. SMTM (Simultaneous Multi-Threading)

- SMTM là một tính năng cho phép mỗi SM có thể thực hiện nhiều thread cùng lúc. Điều này giúp tận dụng tối đa tài nguyên của SM, cải thiện hiệu suất xử lý song song.
- Với SMTM, GPU có thể xử lý nhiều thread trên mỗi đơn vị tính toán, tương tự như công nghệ Hyper-Threading trên CPU.

V, Kiến trúc bộ nhớ phân cấp trong GPU

GPU được thiết kế với kiến trúc bộ nhớ phân cấp để tối ưu hóa tốc độ truy cập dữ liệu và hiệu quả tính toán. Các cấp bộ nhớ chính bao gồm:

a. Bộ nhớ trên GPU (Device Memory)

- **Registers**:
  - Dung lượng: nhỏ (vài KB/thread).
  - Tốc độ: truy cập nhanh nhất.
  - Dùng để lưu biến cục bộ trong từng luồng (thread).
- **Shared Memory**:
  - Dung lượng: vài chục KB/block.
  - Tốc độ: nhanh hơn Global Memory (cận RAM cache).
  - Chia sẻ giữa các luồng trong cùng một block.
  - Tối ưu khi các luồng trong block cần truy cập dữ liệu chung.
- **Global Memory**:
  - Dung lượng: vài GB.
  - Tốc độ: chậm nhất trên GPU.
  - Tất cả các luồng trong tất cả các block có thể truy cập.
  - Yêu cầu truy cập đồng bộ để tăng hiệu suất (coalesced memory access).
- **Texture/Constant Memory**:
  - Bộ nhớ chỉ đọc, tối ưu cho truy cập có tính không gian hoặc thời gian cục bộ.
  - Sử dụng trong các ứng dụng yêu cầu truy cập dữ liệu cố định (lookup table).

b. Bộ nhớ trên Host (CPU Memory)

- **Pinned Memory**:
  - Bộ nhớ cố định trên host, cho phép truyền tải dữ liệu giữa CPU và GPU nhanh hơn so với bộ nhớ thông thường.

VI. Giao tiếp giữa Host và Device

a. Host và Device

- **Host (CPU)**:
  - Điều khiển các tác vụ, phân phối công việc, truyền dữ liệu.
  - Chạy các chương trình chính (host code).
- **Device (GPU)**:
  - Xử lý các tác vụ song song thông qua các kernel được khởi chạy từ host.
  - Sử dụng bộ nhớ riêng biệt.

b. Quá trình truyền dữ liệu

- Dữ liệu phải được truyền từ bộ nhớ CPU (host) sang GPU (device) trước khi tính toán.
- Sau khi GPU hoàn thành, dữ liệu sẽ được trả về CPU nếu cần.

c. Các hàm truyền dữ liệu

Trong CUDA, các hàm chính dùng để truyền dữ liệu bao gồm:

1. cudaMemcpy():

Cú pháp:

```
cudaMemcpy(destination, source, size, cudaMemcpyKind);
```

- **destination**: con trỏ đích.
- **source**: con trỏ nguồn.
- **size**: số byte cần sao chép.
- **cudaMemcpyKind**: loại truyền:
  - `cudaMemcpyHostToDevice` (H2D): từ host sang device.
  - `cudaMemcpyDeviceToHost` (D2H): từ device sang host.
  - `cudaMemcpyDeviceToDevice` (D2D): giữa các vùng bộ nhớ trên GPU.
  - `cudaMemcpyHostToHost` (H2H): giữa các vùng bộ nhớ trên CPU.

2. cudaMalloc():

- Cấp phát bộ nhớ trên GPU.

Ví dụ:

```
cudaMalloc((void**)&device_ptr, size);
```

3. cudaFree():

- Giải phóng bộ nhớ trên GPU.

Ví dụ:

```
cudaFree(device_ptr);
```

4. cudaMemcpyAsync():

- Truyền dữ liệu không đồng bộ, cho phép chồng chéo với tính toán.

5. cudaHostAlloc():

- Cấp phát bộ nhớ cố định (pinned memory) trên host.

VII, Cấu trúc chương trình CUDA

Một chương trình CUDA thường bao gồm ba phần chính:

a. Host Code

- Chạy trên CPU, điều khiển toàn bộ chương trình, bao gồm:
  - Khởi tạo bộ nhớ (host và device).
  - Truyền dữ liệu giữa CPU và GPU.
  - Khởi chạy kernel.
  - Đồng bộ hóa (sử dụng `cudaDeviceSynchronize()`).

b. Kernel Code

- Chạy trên GPU.
- Cấu trúc cơ bản:

```
__global__ void kernel_function(arguments) {
    // Code chạy trên GPU
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    // Tính toán...
}
```

c. Các thành phần quan trọng

1. **Thread Hierarchy**:
   - Threads: đơn vị tính toán nhỏ nhất.
   - Block: nhóm các threads.

    - Grid: nhóm các blocks.
2. **Launch Kernel**:
    - Khởi chạy kernel từ host:

      kernel_function<<<num_blocks, threads_per_block>>>(arguments);

        - **num_blocks**: số lượng block trong grid.
        - **threads_per_block**: số thread trong mỗi block.
2. **Đồng bộ hóa**:
    - Giữa các threads: `__syncthreads()`.
    - Giữa GPU và CPU: `cudaDeviceSynchronize()`.

Tóm tắt

- GPU có kiến trúc bộ nhớ phân cấp để tối ưu hóa hiệu suất.
- Truyền dữ liệu giữa host và device sử dụng các hàm như `cudaMemcpy`, `cudaMalloc`.
- Cấu trúc chương trình CUDA bao gồm mã host (CPU), mã kernel (GPU), và các thao tác truyền dữ liệu.
- Tối ưu hóa bộ nhớ và tổ chức threads/block/grid là yếu tố quyết định hiệu suất trong CUDA.

VIII. Chương trình ví dụ sử dụng CuDNN

Một ví dụ đơn giản về cách sử dụng CuDNN để tạo và huấn luyện một mô hình học sâu với TensorFlow (một framework hỗ trợ CuDNN).

Cấu trúc chương trình với CuDNN trên Colab

1. Chuẩn bị môi trường: Cài đặt các thư viện cần thiết (TensorFlow, CuDNN).
2. Tạo mô hình học sâu: Dùng TensorFlow để tạo một mô hình học sâu đơn giản (ví dụ, mạng neural đơn giản cho phân loại ảnh).
3. Huấn luyện mô hình: Huấn luyện mô hình với các dữ liệu đầu vào (chẳng hạn như MNIST).
4. Kiểm tra kết quả: Đánh giá mô hình sau khi huấn luyện

Bước 1: Khởi tạo môi trường trên Google Colab

Mặc dù Google Colab đã tích hợp CuDNN và TensorFlow sẵn, nhưng bạn vẫn cần kiểm tra xem môi trường có GPU và CuDNN hay không.

Để làm điều này, bạn có thể sử dụng các đoạn mã sau:

```
import tensorflow as tf

# Kiểm tra xem có GPU không
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))

# Kiểm tra phiên bản của TensorFlow và CuDNN
print("TensorFlow version:", tf.__version__)
print("CuDNN version:", tf.sysconfig.get_build_info()['cudnn_version'])
```

Kết quả:

```
Num GPUs Available:  0
TensorFlow version: 2.17.1
CuDNN version: 8
```

Bước 2: Xây dựng và huấn luyện mô hình học sâu với CuDNN

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Tải dữ liệu MNIST
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Tiền xử lý dữ liệu
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

train_images, test_images = train_images / 255.0, test_images / 255.0

# Tạo mô hình sử dụng CuDNN
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Biên dịch mô hình
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Huấn luyện mô hình
history = model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_data=(test_images, test_labels))

# Đánh giá mô hình
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

Kết quả:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `inp
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 64s 66ms/step - accuracy: 0.8767 - loss: 0.4165 - val_accuracy: 0.9816 - val_loss: 0.0551
Epoch 2/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 76s 61ms/step - accuracy: 0.9853 - loss: 0.0473 - val_accuracy: 0.9877 - val_loss: 0.0379
Epoch 3/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 80s 59ms/step - accuracy: 0.9889 - loss: 0.0346 - val_accuracy: 0.9901 - val_loss: 0.0315
Epoch 4/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 82s 59ms/step - accuracy: 0.9919 - loss: 0.0256 - val_accuracy: 0.9901 - val_loss: 0.0286
Epoch 5/5
938/938 ━━━━━━━━━━━━━━━━━━━━ 57s 60ms/step - accuracy: 0.9941 - loss: 0.0197 - val_accuracy: 0.9890 - val_loss: 0.0340
313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9855 - loss: 0.0426
Test accuracy: 0.9890000224113464
```

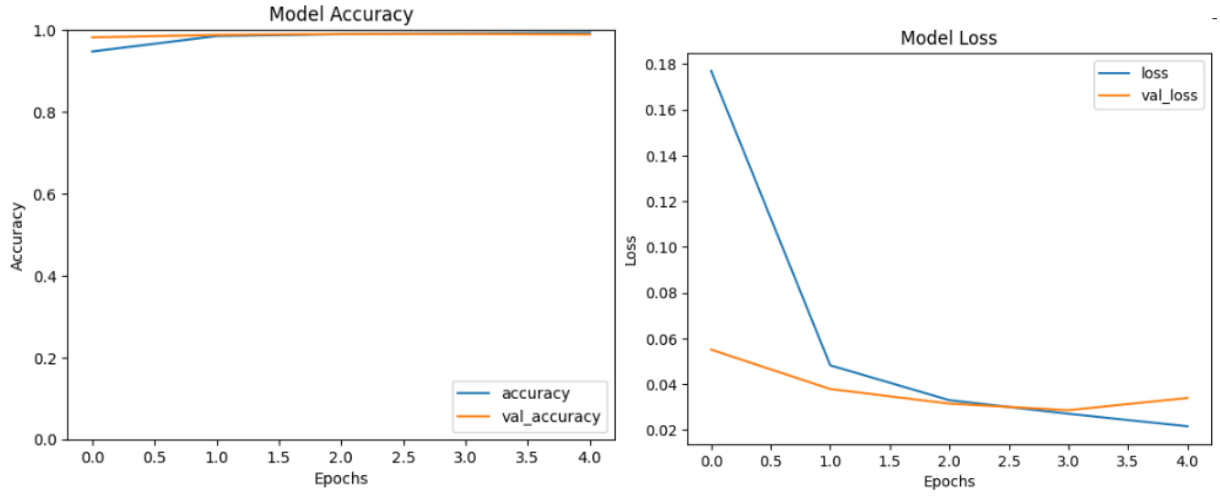
Bước 3: Đánh giá kết quả

Sau khi huấn luyện xong, bạn có thể vẽ biểu đồ mô tả quá trình huấn luyện và độ chính xác của mô hình qua các epoch.

```python
# Vẽ đồ thị độ chính xác và mất mát trong quá trình huấn luyện
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.ylim([0, 1])
plt.legend(loc='lower right')
plt.title('Model Accuracy')
plt.show()

# Vẽ đồ thị mất mát trong quá trình huấn luyện
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label = 'val_loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('Model Loss')
plt.show()
```

Kết quả:



Giải thích code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
```

- **import tensorflow as tf**: Import thư viện TensorFlow, nền tảng phần mềm để xây dựng và huấn luyện các mô hình học sâu.
- **from tensorflow.keras import layers, models**: Import các lớp (layers) và mô hình (models) từ **Keras**, một API cấp cao của TensorFlow giúp xây dựng và huấn luyện các mô hình học máy.
- **from tensorflow.keras.datasets import mnist**: Import bộ dữ liệu MNIST (bao gồm hình ảnh chữ số viết tay) từ Keras.
- **import matplotlib.pyplot as plt**: Import thư viện để vẽ đồ thị, hỗ trợ việc hiển thị các kết quả, hình ảnh trong quá trình huấn luyện.

```
# Tải dữ liệu MNIST
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

mnist.load_data(): Tải bộ dữ liệu **MNIST**, bao gồm hình ảnh và nhãn của chúng.

- **train_images**: Hình ảnh trong bộ dữ liệu huấn luyện (60.000 ảnh).
- **train_labels**: Nhãn tương ứng với các hình ảnh huấn luyện.
- **test_images**: Hình ảnh trong bộ dữ liệu kiểm tra (10.000 ảnh).
- **test_labels**: Nhãn tương ứng với các hình ảnh kiểm tra.

```
# Tiền xử lý dữ liệu
train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))
test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

train_images, test_images = train_images / 255.0, test_images / 255.0
```

reshape(): Chuyển đổi hình ảnh từ dạng mảng 2 chiều (28x28) thành dạng mảng 4 chiều (samples, height, width, channels) để có thể sử dụng trong các lớp của Keras.

- train_images.shape[0]: Số lượng mẫu trong bộ huấn luyện.
- 28, 28: Kích thước của mỗi ảnh (28x28 pixel).
- 1: Số kênh màu (1 kênh cho ảnh grayscale).

train_images / 255.0, test_images / 255.0: Chuẩn hóa giá trị pixel của hình ảnh về khoảng [0, 1] bằng cách chia cho 255, giúp cải thiện quá trình huấn luyện.

```python
# Tạo mô hình sử dụng CuDNN
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

- **models.Sequential()**: Tạo một mô hình tuần tự, nghĩa là các lớp sẽ được xếp chồng lên nhau từ đầu đến cuối.
- **layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1))**:
    - Lớp **Conv2D** (Convolutional 2D) thực hiện phép toán tích chập (convolution). Ở đây, mô hình sử dụng 32 bộ lọc (filters) kích thước (3, 3) và hàm kích hoạt **ReLU**.
    - **input_shape=(28, 28, 1)**: Xác định kích thước đầu vào của ảnh là (28, 28, 1), tức là mỗi ảnh có kích thước 28x28 và 1 kênh màu.
- **layers.MaxPooling2D((2, 2))**: Lớp **MaxPooling2D** thực hiện phép giảm kích thước ảnh (max pooling) với cửa sổ (2, 2), giúp giảm số lượng tham số và làm giảm độ phức tạp tính toán.
- Các lớp **Conv2D** và **MaxPooling2D** này được lặp lại hai lần, với bộ lọc 64 ở các lớp tiếp theo.
- **layers.Flatten()**: Lớp này biến đổi kết quả đầu ra của các lớp convolution và pooling thành một mảng 1 chiều để có thể đưa vào các lớp Dense.
- **layers.Dense(64, activation='relu')**: Lớp Fully Connected (Dense) với 64 neuron và hàm kích hoạt **ReLU**.
- **layers.Dense(10, activation='softmax')**: Lớp đầu ra với 10 neuron, tương ứng với 10 chữ số (0-9), và hàm kích hoạt **Softmax** giúp xác suất hóa đầu ra.

```python
# Biên dịch mô hình
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Huấn luyện mô hình
history = model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_data=(test_images, test_labels))

# Đánh giá mô hình
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

- **optimizer='adam'**: Chọn **Adam** làm bộ tối ưu hóa, một trong những thuật toán tối ưu hóa phổ biến trong học sâu.

- **loss='sparse_categorical_crossentropy'**: Hàm mất mát (loss function) cho bài toán phân loại nhiều lớp. Vì nhãn là số nguyên, ta sử dụng sparse_categorical_crossentropy.
- **metrics=['accuracy']**: Chỉ số đánh giá mô hình trong quá trình huấn luyện và kiểm tra. Ở đây, ta sử dụng **accuracy** (độ chính xác) để đánh giá hiệu suất.

- **model.fit()**: Huấn luyện mô hình trên bộ dữ liệu huấn luyện.
    - **train_images, train_labels**: Dữ liệu huấn luyện.
    - **epochs=5**: Số lần huấn luyện qua toàn bộ bộ dữ liệu.
    - **batch_size=64**: Kích thước của một lô (batch) trong mỗi lần huấn luyện.
    - **validation_data=(test_images, test_labels)**: Dữ liệu kiểm tra, được sử dụng để đánh giá mô hình sau mỗi epoch.

- **model.evaluate()**: Đánh giá mô hình trên dữ liệu kiểm tra để tính toán giá trị **loss** và **accuracy**.
- **print(f"Test accuracy: {test_acc}")**: In ra độ chính xác của mô hình trên bộ dữ liệu kiểm tra.

Tổng kết

Trong ví dụ trên đã:

- Tạo và huấn luyện một mô hình học sâu sử dụng TensorFlow và CuDNN trên Google Colab.
- Kiểm tra GPU và CuDNN có sẵn trong môi trường Colab.
- Sử dụng các bước huấn luyện mô hình cơ bản với dữ liệu MNIST.
- Đánh giá và vẽ đồ thị về độ chính xác và mất mát của mô hình.

CuDNN tự động được tích hợp trong TensorFlow trên Colab, vì vậy không cần phải cài đặt CuDNN thủ công. Khi sử dụng TensorFlow trên Colab, tận dụng CuDNN cho các phép toán học sâu, như tích chập và tối ưu hóa hiệu suất huấn luyện.