Paralelización de Redes Neuronales Profundas

Neurocomputación - Ingeniería Informática

5 de mayo de 2021

Alejandro Santorum Varela
Sergio Galán Martín
David Cabornero Pascual

UAM

Ciudad Universitaria de Cantoblanco
28049 Madrid
Tlf.: +34 91 497 50 00
Fax: +34 91 497 50 00
informacion.general@uam.es
http://www.uam.es

Índice

1. Introducción

Las redes neuronales profundas son conocidas en muchas ocasiones por el gran coste temporal que supone el entrenamiento de sus modelos: algunos de ellos pueden llegar a alcanzar meses de entrenamiento. La paralelización y distribución de estos algoritmos surge como una solución a este problema, ya que muchas veces la serialización que realiza nuestra máquina para entrenar resulta ineficiente.

En este trabajo se muestra el aumento de eficiencia que supone trabajar con algunas de las técnicas de paralelización más habituales, como el uso de una GPU o procesamiento con varios núcleos. Para ello, se utilizará la conocida librería **TensorFlow** desde la plataforma de *Google Colab*.

2. Métodos de paralelización y distribución

Hay distintas formas de aprovechar el potencial los núcleos de un ordenador (paralelización) o de varios ordenadores simultáneamente (distribución). A continuación clasificamos los más habituales:

- Con **una sola máquina**, donde la memoria relacionada con el procesamiento y los datos se almacena en un único ordenador:
  - **Procesamiento *multi-core***. Con múltiples núcleos se pueden procesar distintos ejemplos en cada capa (casi no se considera paralelizar) o bien se puede utilizar *SGD* para que cada núcleo trabaje con pequeños lotes en paralelo.
  - **Uso de la GPU** para acciones computacionalmente caras pero con operaciones sencillas como multiplicación de matrices.
  - **Modelo híbrido** en el que se usan varios núcleos y todos ellos a su vez usan la GPU para cálculos exhaustivos. Es el que se usa en la realidad, ya que es mejor aprovechar las ventajas de los dos modelos.
- **Entrenamiento distribuido**, cuando no es posible guardar todos los datos o el modelo en una sola máquina:
  - Paralelización de **datos**, distribuyendo así los datos en distintas máquinas. Usado cuando no es posible almacenar todos los datos en una sola máquina o si se quiere entrenar más rápido el modelo.
  - Paralelización del **modelo**, cuando el modelo es demasiado grande para ser guardado en una única máquina. Por ejemplo, se puede guardar una capa de la red en cada máquina, y cualquier tipo de propagación requiere comunicación entre máquinas. Solo se utiliza si es estrictamente necesario, ya que es sumamente ineficiente.

Dada la naturaleza de nuestro trabajo y los recursos de los que disponemos, nos hemos centrado en valorar los modelos que solo necesitan una máquina. En especial, en el próximo capítulo se verá el papel que puede tomar la GPU en distintos problemas habituales en las redes neuronales profundas.

3. Paralelización usando GPUs

En esta sección procederemos a implementar la paralelización en **Google Colab** usando la librería **Tensorflow**, además de realizar una comparativa de rendimiento entre la ejecución de ciertas funciones en ambos entornos.

3.1. Uso de GPUs en Google Colab

Lo primero que debemos hacer (no es necesario pero sí es recomendable para comprobar que todo está funcionando correctamente) es cambiar el entorno de ejecución de Google Colab a **GPU**, y tras ello ejecutar una celda con el código 3.1

```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

Listing 3.1: Comprobación GPU

Una vez que nos aseguramos que el entorno está correctamente configurado, procedemos a implementar las dos funciones que compararemos en la sección 3.2, la multiplicación de matrices 3.2 y el entrenamiento de una red convolucional 3.3.

```
def matrix_mult_gpu(n):
  with tf.device('/device:GPU:0'):
    matrix_A = tf.random.normal((n, n))
    matrix_B = tf.random.normal((n, n))
    return tf.linalg.matmul(matrix_A, matrix_B)
```

Listing 3.2: Multiplicación de matrices GPU

```
def convolution_gpu(n):
  with tf.device('/device:GPU:0'):
    random_image = tf.random.normal((n, 128, 128, 3))
    net_gpu = tf.keras.layers.Conv2D(32, 8)(random_image)
    return tf.math.reduce_sum(net_gpu)
```

Listing 3.3: Red convolucional GPU

Como vemos, para indicar a Google Colab que queremos que la ejecución se lleve a cabo en la GPU tan solo tenemos que incluir la línea `with tf.device('/device:GPU:0')`. También destacamos que hemos implementado el entrenamiento de la red neuronal con n imágenes de 128×128 píxeles **aleatorias**, ya que solo nos interesa el tiempo dedicado al entrenamiento, no los resultados del mismo.

3.2. Comparativa de rendimiento CPU vs GPU

Con el código visto en la sección anterior y el equivalente ejecutándose en la **CPU** (cambiando la línea correspondiente por `with tf.device('/cpu:0')`), procedemos a realizar una comparativa del tiempo de ejecución de ambas implementaciones. Esta comparativa la obtenemos ejecutando los códigos que se pueden apreciar en 3.4, 3.5.

```
def run_simulation_matrix_mult(n_list=[50, 100, 500, 1000, 2000, 5000,
    7000, 10000], total_reps=10):
  cpu_times = []
  gpu_times = []
  for n in n_list:
    cpu_time = 0.0
    gpu_time = 0.0
    for _ in range(total_reps):
      t1 = time.time()
      matrix_mult_cpu(n)
      t2 = time.time()
      cpu_time += t2-t1

      t1 = time.time()
      matrix_mult_gpu(n)
      t2 = time.time()
      gpu_time += t2-t1

    cpu_times.append(cpu_time/total_reps)
    gpu_times.append(gpu_time/total_reps)
  plt.plot(n_list, cpu_times, label='cpu')
  plt.plot(n_list, gpu_times, label='gpu')
  plt.title('Tiempo de ejecución de multiplicación de matrices')
  plt.xlabel('Tamaño matriz')
  plt.ylabel('Tiempo (s)')
  plt.legend()
  plt.show()
```

Listing 3.4: Código comparativa multiplicación de matrices

```
def run_simulation_convolutional(n_list=[20, 40, 60, 80, 100, 200],
    total_reps=10):
  cpu_times = []
  gpu_times = []
  for n in n_list:
    cpu_time = 0.0
    gpu_time = 0.0
    for _ in range(total_reps):
      t1 = time.time()
      convolution_cpu(n)
      t2 = time.time()
      cpu_time += t2-t1

      t1 = time.time()
      convolution_gpu(n)
      t2 = time.time()
```

```
16        gpu_time += t2-t1
17
18      cpu_times.append(cpu_time/total_reps)
19      gpu_times.append(gpu_time/total_reps)
20    plt.plot(n_list, cpu_times, label='cpu')
21    plt.plot(n_list, gpu_times, label='gpu')
22    plt.title('Tiempo de ejecución de convolución')
23    plt.xlabel('Número de imágenes')
24    plt.ylabel('Tiempo (s)')
25    plt.legend()
26    plt.show()
```

Listing 3.5: Código comparativa entrenamiento red convolucional

Ahora, ejecutaremos ambas funciones con los valores que hemos establecido por defecto. Podemos ver las gráficas resultantes en 3.1 y 3.2.

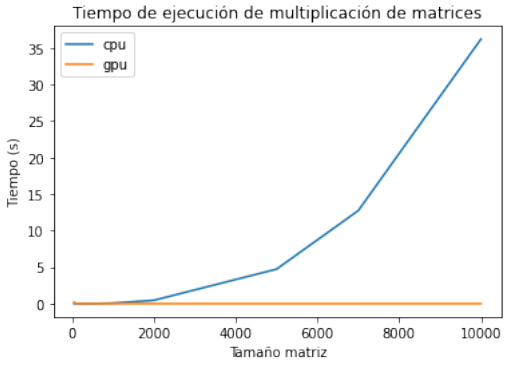

Figura 3.1: GPU frente CPU multiplicación de matrices

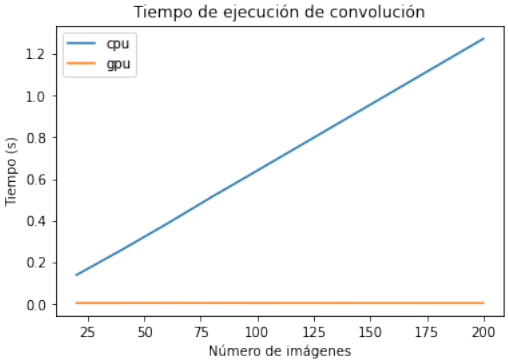

Figura 3.2: GPU frente CPU red convolucional

Como podemos ver, los resultados de la GPU son **abismalmente mejores** que los resultados de la CPU. Esto es debido al hecho de que la CPU tiene **pocos pero potentes** núcleos, mientras que la GPU dispone de una **gran cantidad de núcleos más simples**. De esta forma, y al poder estructurarse las operaciones de aprendizaje

de una forma que permite su fácil paralelización (por ejemplo, en forma de matrices para el perceptrón multicapa), tenemos que una gran cantidad de núcleos algo más lentos nos permiten entrenar en una cantidad muy inferior de tiempo. A fin de dar una comparativa más cuantitativa, las relaciones entre los tiempos de la multiplicación de una matriz de tamaño 10000×10000 es de unas **31500** veces más lenta en la CPU que en la GPU y el entrenamiento de una red convolucional con 200 imágenes de 128×128 píxeles es **210** veces más lento. Es importante también destacar que, aunque no se aprecie directamente en las gráficas, la complejidad computacional del problema **no cambia**, es decir, la multiplicación de matrices se mantiene en $\mathcal{O}(n^3)$ siendo n la dimensión de la matriz cuadrada y el entrenamiento de red convolucional se mantiene en $\mathcal{O}(n)$ siendo n el número de imágenes a procesar.

4. Entrenamiento distribuido

En este capítulo estudiaremos cómo podríamos nosotros paralelizar el entrenamiento de casi cualquier algoritmo de *machine learning* distribuyendo los datos entre todas las máquinas que dispongamos. En la sección 4.1 expondremos el algoritmo básico para **paralelizar y distribuir el entrenamiento del descenso del gradiente** (*Stochastic Gradient Descent, SGD*), y en la sección 4.2 detallaremos cómo podemos realizar este **entrenamiento distribuido en la librería Keras**, con un par de ejemplos.

4.1. Paralelización del descenso del gradiente

Una forma de **distribuir la computación** de ciertos programas en distintas máquinas es **dividir el conjunto de datos total** a utilizar en dichas máquinas para ejecutar algunos cálculos de forma simultánea.

A continuación vamos a detallar cómo sería el algoritmo paralelizado de actualización de pesos usando el descenso del gradiente (regla delta) para encontrar un mínimo local de la función de coste. La actualización de pesos se hará de forma síncrona, es decir, esperaremos a que todas las máquinas hayan computado su descenso del gradiente en su correspondiente porción de datos para actualizar los pesos utilizando la media ponderada de dichos gradientes parciales.

Algorithm 1 Paralelización de SGD

1: **procedure** ParalelizaSGD(parametros, datos, nMaquinas)
2: Mezclar conjunto de datos global
3: **for** $i \in \{1, ..., \text{n\_maquinas}\}$ **do**
4: $v_i \leftarrow$ SGD(parametros, porcionDatos$_i$)
5: **end for**
6: Media de gradientes: $v \leftarrow \frac{1}{nMaquinas} \sum_{i=1}^{nMaquinas} v_i$
7: **return** v
8: **end procedure**

El primer paso en el algoritmo 1 es **barajar** los datos para tener un subconjunto representativo del conjunto de datos completo en cada una de las máquinas. Después, como ya hemos comentado, simplemente **dividimos** en porciones el *dataset* y cada parte es entregada a **una máquina diferente** para que compute un descenso del gradiente parcial. Al final la actualización de pesos será **proporcional a la media de los gradientes** calculados.

4.2. Entrenamiento distribuido en Keras

Nos centramos ahora cómo podemos utilizar diferentes máquinas, en especial **GPUs**, a través de la librería de **Keras** para distribuir los datos de entrenamiento y paralelizar el mismo siguiendo el pseudocódigo 1.

La librería Keras permite configurar una **distribución síncrona de los datos** para entrenar diferentes porciones del conjunto de entrenamiento en diferentes máquinas (e.g GPUs). La sincronización permite mantener la convergencia del modelo idéntica a la habitual que observaríamos con una sola maquina entrenando.

A partir de ahora consideraremos que **una máquina** (ordenador) cuenta con **varias GPUs**, habitualmente entre 2 y 8. Cada GPU ejecutará una copia del modelo, llamada **réplica**, en una porción de datos diferente.

Como habíamos comentado en la sección anterior, en cada paso del entrenamiento se ejecutará:

1. El conjunto de entrenamiento total se dividirá en tantas porciones como GPUs dispongamos.

2. Cada una de las réplicas procesará, independientemente del resto, una propagación hacia adelante (*forward propagation*), hará una retroprapagación (*back-propagation*) y calculará el gradiente de los pesos con respecto a la porción de datos utilizada.

3. Los gradientes de los pesos correspondientes a las réplicas se fusionarán calculando la media aritmética (ponderada si las porciones no son iguales) y los pesos se actualizarán para la siguiente época de entrenamiento.

En Keras podemos crear la anterior rutina de entrenamiento utilizando la **API de Tensorflow** `tf.distribute.MirroredStrategy` y siguiendo el esquema expuesto en 4.1.

```
import tensorflor as tf
from tensorflow import keras

# Create a MirroredStrategy
strat = tf.distribute.MirroredStrategy()

# Open a strategy scope
with strat.scope():
```

```
    # Everything that creates variables should be under the scope
    # It's usually only model construction and compilation
    model = Model(...)
    model.compile(...)

# Train the model on all available devices
model.fit(train_data, validation_data=val_data, ...)

# Test the model on all available devices
model.evaluate(test_data)
```

Listing 4.1: Esquema de paralelización en varias GPUs con Keras

Para potenciar aún más el rendimiento de nuestra máquina podemos afinar aún más el apartado de carga de datos. Podemos ejecutar la función `dataset.cache` sobre un conjunto de entrenamiento para que los **datos estén en memoria RAM** ya alojados durante la primera iteración, y cada época posterior utilizará esta imagen del conjunto de datos sin necesitar acceder a memoria física, que es muy lenta en comparación con la memoria RAM. El uso de este método también está recomendado cuando los datos se alojan en un disco remoto o, en general, siempre y cuando los datos no se modifiquen entre iteraciones.

Adicionalmente podemos ejecutar `dataset.prefetch(buffer_size)` después de crear un conjunto de datos. Esto permitirá que la **canalización de datos se ejecute de forma asincrónica desde el modelo**, con nuevas muestras preprocesadas y almacenadas en un *buffer*, mientras que las muestras por lotes actuales se utilizan para entrenar el modelo. El siguiente lote se precargará en la memoria de la GPU cuando finalice el lote actual.

Un **ejemplo ejecutable** paralelizando el entrenamiento del conjunto de datos `MNIST` y que utiliza esta API lo podemos encontrar en la página oficial de Keras, en las guías para desarrolladores [2]. La guía ha sido escrita por el mismo creador de Keras y actual investigador de Google en *Deep Learning* François Chollet. El trozo de código que ilustra lo descrito hasta ahora se expone en 4.2.

```
import tensorflor as tf
from tensorflow import keras

def get_compiled_model():
    # Make a simple 2-layer densely-connected neural network.
    inputs = keras.Input(shape=(784,))
    x = keras.layers.Dense(256, activation="relu")(inputs)
    x = keras.layers.Dense(256, activation="relu")(x)
    outputs = keras.layers.Dense(10)(x)
    model = keras.Model(inputs, outputs)
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy()],
    )
    return model

```

```
18
19 def get_dataset():
20     batch_size = 32
21     num_val_samples = 10000
22
23     # Return the MNIST dataset in the form of a `tf.data.Dataset`.
24     (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
25
26     # Preprocess the data (these are Numpy arrays)
27     x_train = x_train.reshape(-1, 784).astype("float32") / 255
28     x_test = x_test.reshape(-1, 784).astype("float32") / 255
29     y_train = y_train.astype("float32")
30     y_test = y_test.astype("float32")
31
32     # Reserve num_val_samples samples for validation
33     x_val = x_train[-num_val_samples:]
34     y_val = y_train[-num_val_samples:]
35     x_train = x_train[:-num_val_samples]
36     y_train = y_train[:-num_val_samples]
37     return (
38         tf.data.Dataset.from_tensor_slices((x_train, y_train)).batch(
    batch_size),
39         tf.data.Dataset.from_tensor_slices((x_val, y_val)).batch(batch_size
    ),
40         tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(
    batch_size),
41     )
42
43
44 # Create a MirroredStrategy.
45 strategy = tf.distribute.MirroredStrategy()
46 print("Number of devices: {}".format(strategy.num_replicas_in_sync))
47
48 # Open a strategy scope.
49 with strategy.scope():
50     # Everything that creates variables should be under the strategy scope.
51     # In general this is only model construction & `compile()`.
52     model = get_compiled_model()
53
54 # Train the model on all available devices.
55 train_dataset, val_dataset, test_dataset = get_dataset()
56 model.fit(train_dataset, epochs=2, validation_data=val_dataset)
57
58 # Test the model on all available devices.
59 model.evaluate(test_dataset)
```

Listing 4.2: Ejemplo ejecutable de paralelización con GPUs en Keras

5. Conclusiones

A lo largo del trabajo hemos visto **diferentes formas de paralelizar y distribuir** los datos de un proyecto de *machine learning* para acelerar el proceso de entrenamiento. En especial, nos hemos centrado en cómo sacarle el máximo potencial a las Unidades de Procesamiento Gráfico o **GPUs**.

Las GPUs muestran un gran poder computacional en **tareas sencillas**, en especial son bien conocidos los problemas de **multiplicación de matrices**, y por tanto en problemas de redes neuronales profundas. Tal y como se ha visto, una red normal puede mejorarse en un factor mayor de 200, haciendo que si dicha red tardara en entrenarse varios meses pasara a entrenarse en unos pocos días.

Con esto, se quiere remarcar la importancia de un correcto uso de la GPU para operaciones **simples pero costosas**, ya que puede suponer una diferencia de coste computacional muy grande. La paralelización *multi-core* en CPU debe dejarse, por lo tanto, para **tareas más complejas que requieran un procesamiento general**.

Bibliografía

[1] Vishakh Hedge and Sheema Usmani. *Parallel and Distributed Deep Learning* (2016). Stanford University. Descargar

[2] François Chollet. Multi-GPU and distributed training (2020). Keras Developer Guides. Enlace