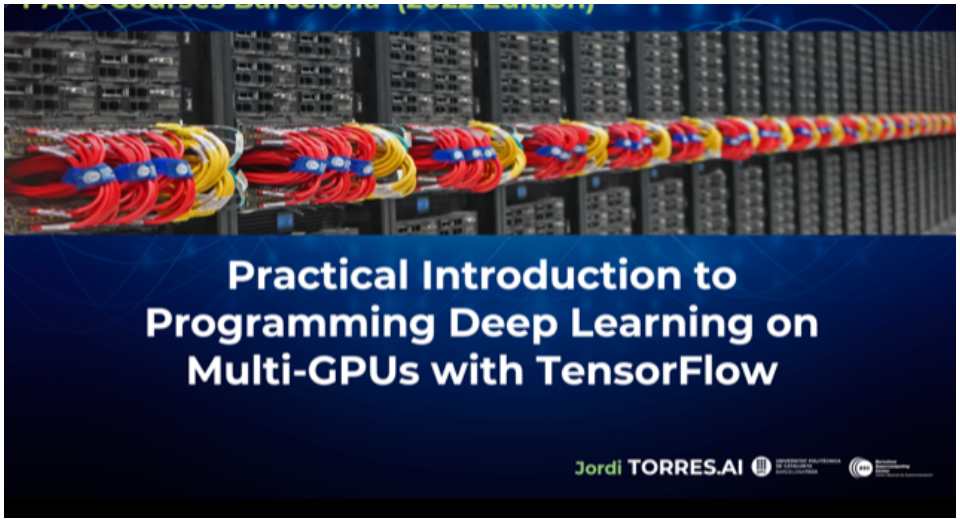


Fundamentals of Deep Learning for Multi-GPUs

February 10, 2022

Until now, we have been doing all the programming tasks on Jupyter notebooks. But how the same DL code can be parallelized on a supercomputer?

Content [hide]

PART 1: Using Supercomputers for Training Deep Learning Models

---

---

1 — BSC's CTE-POWER Cluster

This hands-on exercise uses the BSC's CTE-POWER cluster. Let's briefly review its characteristics. CTE-POWER is a cluster-based on IBM Power9 processors, with a Linux Operating System and an Infiniband interconnection network. CTE-POWER has 54 compute servers, each of them:

- 2 x IBM Power9 8335-GTG @ 3.00GHz (20 cores and 4 threads/core, total 160 threads per node)
- 512GB of main memory distributed in 16 DIMMs x 32GB @ 2666MHz
- 2 x SSD 1.9TB as local storage
- 2 x 3.2TB NVME
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2.
- Single Port Mellanox EDR
- GPFS via one fiber link 10 GBit
- The operating system is Red Hat Enterprise Linux Server 7.4.

One CTE-POWER computer server (Image from bsc.es)

More details of its characteristics can be found in the CTE-POWER user's guide and also in the information of the manufacturer of the AC922 servers.

The allocation of resources from the cluster for the execution of our code will start with a `ssh` login in the cluster using one of the login nodes using your account:

```
ssh -X nct01xxx@plogin1.bsc.es
```

Task 1:

Once you have a login username and its associated password, you can get into the CTE-POWER cluster (login node). Check that you have access to your home page.

> ***The code used in this post is based on the GitHub https://github.com/jorditorresBCN/Fundamentals-DL-CTE-POWER***

2 — Warm-up example: MNIST classification

For convenience, we will consider the same neural network that we used to classify MNIST digits in the previous part programmed in the Jupyter notebook.

In the following lines, there is the code of the TensorFlow version of the MNIST classifier described in class.

> *Note: For the following code lines, beware at COPY&PASTE!. Some symbols are "converted" by the HTML tranlator into non-standard. If a command does not work properly, repeat it by typing it.*
>
> *This will be the code* `MNIST.py` *(available at GitHub), which we will use as a first case study to show how to launch programs in the CTE-POWER supercomputing.*

```
import tensorflow as tf
from tensorflow import keras
```

```
import numpy as np
import matplotlib.pyplot as plt
print(tf.__version__)
```

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2
D
from tensorflow.keras.layers import MaxPo
oling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatt
en
```

```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='
relu',
          input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (5, 5), activation='
relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(10, activation='softmax')
)
model.summary()
```

```
from keras.utils import to_categorical
mnist = tf.keras.datasets.mnist
```

```
(train_images, train_labels), (test_image
s, test_labels) = mnist.load_data(path='/
gpfs/projects/nct00/nct00002/basics-utils
/mnist.npz')
```

```
train_images = train_images.reshape((6000
0, 28, 28, 1))
train_images = train_images.astype('float
32') / 255
```

```
test_images = test_images.reshape((10000,
 28, 28, 1))
test_images = test_images.astype('float32
') / 255
```

```
train_labels = to_categorical(train_label
s)
test_labels = to_categorical(test_labels)
```

```
model.compile(loss='categorical_crossentr
opy',
              optimizer='sgd',
              metrics=['accuracy'])
```

```
model.fit(train_images, train_labels, bat
ch_size=100,
          epochs=5, verbose=1)
```

```
test_loss, test_acc = model.evaluate(test
_images, test_labels)
```

```
print('Test accuracy:', test_acc)
```

3 — Software stack required for deep learning applications

It is important to know that before executing a DL application on a computer, it is required to load all the packages that build the application's software stack environment. At CTE-POWER supercomputer, it is done though `modules` , that can be done with the

command `module load` before running the corresponding `.py` code.

In our case study, we need the following modules that include the required libraries:

```
module load gcc/8.3.0 cuda/10.2 cudnn/7.6
.4 nccl/2.4.8 tensorrt/6.0.1 openmpi/4.0.
1 atlas/3.10.3 scalapack/2.0.2 fftw/3.3.8
 szip/2.1.1 ffmpeg/4.2.1 opencv/4.1.1 python
```

How to execute our `.py` code in the login node?

```
python MNIST.py
```

If we want to detach the *standard outputs* and the *standard error* messages, we can add this argument `2>err.txt`:

```
python MNIST.py 2>err.txt
```

Redirecting the *standard error* allows us to see the result of the training that gives us the Keras by the *standard output* without the information related to the execution environment:

```
Epoch 1/5
600/600 [======] - 2s 3ms/step - loss: 0.
9553 - accuracy: 0.7612
Epoch 2/5
600/600 [======] - 1s 2ms/step - loss: 0.
2631 - accuracy: 0.9235
Epoch 3/5
```

```
600/600 [======] - 2s 3ms/step - loss: 0.
1904 - accuracy: 0.9446
.
.
.
```

```
Test accuracy: 0.9671000242233276
```

> *Because our code is executed in the login node devoted to offering access to the users, not executing codes, the system probably will cancel the program before finishing due we are using too many resources. In this case, we could observe* `Killed` *in the standard output provoked by the execution environment system.*

Task 2:

Launch your `MNIST.py` sequential program in the CTE-POWER supercomputer.

Well, our code is executed in the login node shared with other jobs from other users, but what we really need is to allocate resources for our parallel code. How can we do it?

4 — How to allocate

computing resources with SLURM

Then, for executing a parallel code, the first thing to do is to allocate resources (in our case a node). At the CTE-POWER cluster, we use the SLURM workload manager. An excellent Quick Start User Guide can be found here.

The method for submitting jobs that we will center our today hands-on exercise will be using the SLURM `sbatch`command directly. `sbatch` submits a batch script to SLURM. The batch script may be given `sbatch` through a file name on the command line (`.sh` file). The batch script may contain options preceded with `#SBATCH` before any executable commands in the script. `sbatch` will stop processing further `#SBATCH` directives once the first non-comment or non-whitespace line has been reached in the script.

`sbatch` exits immediately after the script is successfully transferred to the SLURM controller and assigned a SLURM job ID. The batch script is not necessarily granted resources immediately, it may sit in the queue of pending jobs for some time before its required resources become available.

By default, both *standard output* and *standard error* are directed to the files indicated by `--output` and `--error` respectivelly:

```
#SBATCH --output=MNIST_%j.out
#SBATCH --error=MNIST_%j.err
```

where the "`%j`" is replaced by SLURM manager with the job allocation number. The file will be generated on the first node of the job allocation. When the job allocation is finally granted for the batch script, SLURM runs a single copy of the batch script on the first node in the

set of allocated nodes (in today's hands-on we will use only one node).

An example of a job script that allocates a node with 1 GPU for our case study looks like this (`MNIST.sh` file from github):

```
#!/bin/bash
#SBATCH --job-name="MNIST"
#SBATCH -D .
#SBATCH --output=MNIST_%j.out
#SBATCH --error=MNIST_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=40
#SBATCH --gres=gpu:1
#SBATCH --time=00:10:00
```

```
module load gcc/8.3.0 cuda/10.2 cudnn/7.6
.4 nccl/2.4.8 tensorrt/6.0.1 openmpi/4.0.
1 atlas/3.10.3 scalapack/2.0.2 fftw/3.3.8
 szip/2.1.1 ffmpeg/4.2.1 opencv/4.1.1 python
```

```
python MNIST.py
```

You can consult this official page documentation to know all the options we can use in the batch script preceded with`#SBATCH`.

These are the basic directives to submit and monitor jobs with SLURM that we will use in our case study:

- `sbatch <job_script>` submits a job script to the queue system.
- `squeue` shows all the submitted jobs with their

<job_id>.
- scancel <job_id> remove the job from the queue system, cancelling the execution of the processes, if they were still running.

In summary, this can be an example of a sequence of command lines, and the expected output of their execution will be:

```
[CTE-login-node ~]$ sbatch MNIST.sh
Submitted batch job 4910352
```

```
[CTE-login-node ~]$ squeue
JOBID    PARTITION  NAME    USER    ST TI
ME  NODES  NODELIST
4910352  main       MNIST   userid  R  0:
01  1      p9r1n16
```

```
[CTE-login-node ~]$ ls
MNIST.py
MNIST.sh
MNIST_4910352.err
MNIST_4910352.out
```

The *standard output* and *standard error* are directed to the files `MNIST_4910355.out` and `MNIST_4910355.err`, respectively. Here, the number `4910352`indicates the job id assigned to the job by SLURM.

BSC has made a special reservation of supercomputer nodes to be used by this PATC course. For using the reservations, you must add this line in the SLURM script:

```
#SBATCH --reservation=<ReservationName>
```

Task 3:

Execute your `MNIST.py` program with the SLURM workload manager system using a job script that allocates a node with 1 GPU in CTE-POWER. Inspect the `.out` and `.err` files obtained.

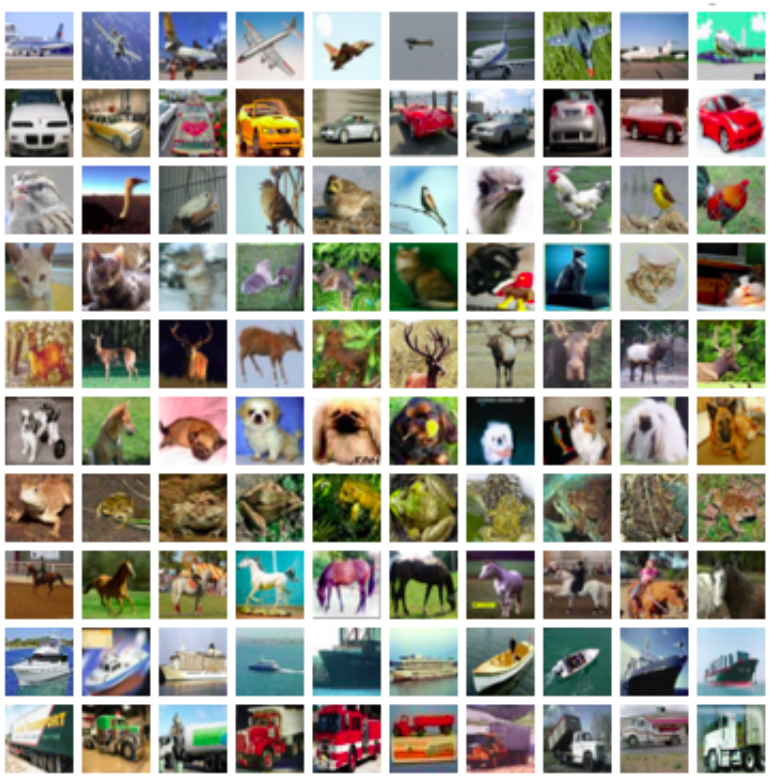
---

5 — Case Study

As we explained earlier, Deep Learning is a very mature area that offers many public datasets for beginners. Also, thanks to Transfer Learning techniques, we can use many well-known neural networks. We will use CIFAR10 as a dataset and a ResNet50 as a neural network in today's hands-on.

5.1 Dataset: CIFAR10

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32×32 colour images containing 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. There are 50,000 training images and 10,000 test images (Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009).



We have preloaded the CIFAR-10 dataset at CTE-POWER supercomputer in the directory `/gpfs/projects/nct00/nct00002/cifar-utils/cifar-10-batches-py` downloaded from http://www.cs.toronto.edu/~kriz/cifar.html.

For academic purposes, to make the training even harder and to be able to see larger training times for better comparison, we have applied a resize operation to make the images of 128×128 size. We created a custom load_data function (`/gpfs/projects/nct00/nct00002/cifar-utils/load_cifar.py`) that applies this resize operation and splits the data into training and test sets. We can use it as:

```
sys.path.append('/gpfs/projects/nct00/nct00002/cifar-utils')
```

```
from cifar import load_cifar
```

`load_cifar.py` can be obtained from GitHub for students that want to review it (for the students of this course it is not necessary to download and review it).

5.2 Neural Networks architecture: ResNet

Now we are going to use a neural network that has a specific architecture known as **ResNet**. In this scientific community, we find many networks with their own name. For instance, **AlexNet,** by Alex Krizhevsky, is the neural network architecture that won the ImageNet 2012 competition. **GoogleLeNet**, which with its inception module drastically reduces the parameters of the network (15 times less than AlexNet). Others, such as the **VGGnet**, helped to demonstrate that the depth of the network is a critical component for good results. The interesting thing about many of these networks is that we can find them already preloaded in most of the Deep Learning frameworks.

Keras Applications are prebuilt deep learning models that are made available. These models differ in architecture and the number of parameters; you can try some of them to see how the larger models train slower than the smaller ones and achieve different accuracy.

A list of all available models can be found here (the top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.). For this hands-on, we will consider one architecture from the family of ResNet as a case study: ResNet50v2. ResNet is a family of extremely deep neural network architectures showing compelling accuracy and nice convergence behaviors, introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition.* A few months later, the same authors published a new paper, *Identity Mapping in Deep Residual Network*, with a

new proposal for the basic component, the residual unit, which makes training easier and improves generalization. And this lets the V2 versions:

```
tf.keras.applications.ResNet50V2(
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
```

The “50” stand for the number of weight layers in the network. The arguments for the network are:

- **include_top**: whether to include the fully-connected layer at the top of the network.
- **weights**: one of `None` (random initialization), ‘imagenet’ (pre-training on ImageNet), or the path to the weights file to be loaded.
- **input_tensor**: optional Keras tensor (i.e. output of `layers.Input()`) to use as image input for the model.
- **input_shape**: optional shape tuple, only to be specified if `include_top` is False (otherwise the input shape has to be `(224, 224, 3)` (with `'channels_last'` data format) or `(3, 224, 224)` (with `'channels_first'` data format). It should have exactly 3 inputs channels, and width and height should be no smaller than 32. E.g. `(200, 200, 3)`would be one valid value.
- **pooling**: Optional pooling mode for feature extraction when `include_top` is `False`. (a)`None` means that the output of the model will be the 4D tensor output of the last convolutional block. (b) `avg` means that global average pooling will be applied to the output of the last convolutional

block, and thus the output of the model will be a 2D tensor. (c)`max` means that global max pooling will be applied.

- **classes**: optional number of classes to classify images into, only to be specified if `include_top`is True, and if no `weights` argument is specified.
- **classifier_activation**: A `str` or callable. The activation function to use on the "top" layer. Ignored unless `include_top=True`. Set `classifier_activation=None` to return the logits of the "top" layer.

> *Note that if `weights="imagenet"`, Tensorflow middleware requires a connection to the internet to download the imagenet weights (pre-training on ImageNet). Due we are not centering our interest in Accuracy, we didn't download the file with the imagenet weights; therefore, it must be used `weights=None`.*

Task 4:

Have a look at the ResNet50v2 and ResNET152V2 neural networks and glimpse the main differences.

6—How to use a GPU to accelerate the training

Before showing how to train a neural network in parallel, let's start with a sequential version using only

one GPU in order to get familiarized with the neural network classifier.

6.1 Python code

The sequential code to train the previously described problem of classification of the CIFAR10 dataset using a ResNet50 neural network could be the following (we will refer to it as `ResNet50_seq.py`):

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
```

```
import numpy as np
import argparse
import time
import sys
```

```
sys.path.append('/gpfs/projects/nct00/nct
00002/cifar-utils')
from cifar import load_cifar
```

```
parser = argparse.ArgumentParser()
parser.add_argument(' — epochs', type=int
, default=5)
parser.add_argument(' — batch_size', type
=int, default=2048)
```

```
args = parser.parse_args()
batch_size = args.batch_size
```

```
epochs = args.epochs
```

```
train_ds, test_ds = load_cifar(batch_size)
```

```
model = tf.keras.applications.resnet_v2.ResNet50V2(
        include_top=True,
        weights=None,
        input_shape=(128, 128, 3),
        classes=10)
```

```
opt = tf.keras.optimizers.SGD(0.01)
```

```
model.compile(loss='sparse_categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
```

```
model.fit(train_ds, epochs=epochs, verbose=2)
```

> `ResNet50_seq.py` *file can be downloaded from the course repository GitHub.*

6.2 SLURM script

To run this python code using the SLURM system, as you know, it can be done using the following SLURM script (we will refer to it as `ResNet50_seq.sh`):

```
#!/bin/bash
#SBATCH --job-name="ResNet50_seq"
#SBATCH -D .
#SBATCH --output=RESNET50_seq_%j.out
#SBATCH --error=RESNET50_seq_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=160
#SBATCH --time=00:05:00
```

```
module load gcc/8.3.0 cuda/10.2 cudnn/7.6
.4 nccl/2.4.8 tensorrt/6.0.1 openmpi/4.0.
1 atlas/3.10.3 scalapack/2.0.2 fftw/3.3.8
 szip/2.1.1 ffmpeg/4.2.1 opencv/4.1.1 python
```

```
python ResNet50_seq.py --epochs 5 --batch
_size 256
```

> `ResNet50_seq.sh` *is not included in the course repository GitHub. Create your* `ResNet50_seq.sh` *script based in the previous explanation.*

Task 5:

Write your `ResNet50_seq.py` program and execute it **with 5 epochs** in CTE-POWER with the SLURM workload manager system using the job script `ResNet50_seq.sh` presented in this section (**with 5 minutes as a maximum time**) What is the result? (Review the relevant part of the `.out` and `.err` files).

As a hint, in my executions appeared the following error:

```
slurmstepd: error: *** JOB <job_id> ON p9
r2n12 CANCELLED AT 202X-11-19T09:48:59 DU
E TO TIME LIMIT ***
```

It means that we exhausted the time indicated in the SLURM time flag `--time`.

> *If you observe that SLURM management system does not control the time limit correctly (sometimes it happens) , I propose to cancel the job after 10 minutes.*

6.3 Using a GPU for training

Unlike the MNIST problem, in general, we cannot train a neural network with a single CPU. It is clear that we need more computing power for training this problem. In this case, we can add this line to the SLURM script to use a GPU:

```
#SBATCH --gres=gpu:1
```

and we need to adjust the cores too:

```
#SBATCH --cpus-per-task=40
#SBATCH --gres=gpu:1
```

Task 6:

Execute the same `ResNet50_seq.py` program using the previous job script `ResNet50_seq.sh` , but now, **including the allocation of one GPU** (and remember to adjust the `cpus-per-task` flag). What is the result? (Review the relevant part of the `.out` file to observe that the time required for one Epoch is lower).

Remember that in the `.out` file we can see the result of the output that gives us the Keras that specify the time required for each epoch, and the loss value and de accuracy achieved with this new epoch:

```
Epoch 1/5
196/196 - 41s - loss: 2.0176 - accuracy:
0.2584
```

Analyzing the `.out` file, what is the Accuracy obtained for this problem in the execution in Task 6? What can we do to improve the Accuracy?

6.4 Improving the Accuracy

From the results of Tasks 6, you can conclude that within 5 minutes you can execute a few epochs, and therefore, the Accuracy obtained is not good. What we can do is to increase the number of Epochs, right?. In this case, it is required to increase the time demanded

in the SLURM time flag `--time`and indicate in the program argument `--epochs` the desired number of epochs.

Task A [optional]:

Execute the same `ResNet50_seq.py` program using the previous job script `ResNet50_seq.sh` , but now adjusting the program argument `--epochs` with the proper value. But what is the appropriate value? What is the obtained Accuracy?

Task B [optional]:

Compare the results of using ResNet50v2 vs ResNET152V2 neural networks in terms of Accuracy obtained and time required for executing one epoch.

In summary, we have seen that using one GPU can accelerate the training process. But, how can we use a distributed strategy in order to use more than one GPU?

PART 2: Accelerate the Learning with Parallel Training using a Multi-GPU Parallel Server

---

7 — Overview of a Parallel Training with TensorFlow

Deep Neural Networks (DNN) base their success on building high learning capacity models with millions of parameters that are tuned in a data-driven fashion. These models are trained in parallel.

7.1 Basics Concepts

7.1.1 Performance metrics: Speedup, Throughput, and Scalability

In order to make the training process faster, we are going to need some performance metrics to measure it. The term *performance* in these systems has a double interpretation. On the one hand, it refers to the **predictive accuracy of the model.** On the other, to the **computational speed of the process**.

Accuracy is independent of the computational resources, and it is the performance metric to compare different DNN models.

In contrast, the computation speed depends on the platform on which the model is deployed. We will measure it by metrics such as **Speedup**, the ratio of solution time for the sequential algorithms (using one GPU in our hands-on exercises) versus its parallel counterpart (using many GPUs). This is a prevalent concept in our daily argot in the supercomputing community.

Another important metric is **Throughput.** In general terms, throughput is the rate of production or the rate at which something is processed; for example, the number of images per unit time that can be processed. This can give us a good benchmark of performance

(although it depends on the neural network type).

Finally, a concept that we sometimes use is **Scalability**. It is a more generic concept that refers to the ability of a system to handle a growing amount of work efficiently. These metrics will be highly dependent on the computer cluster configuration, the type of network used, or the framework's efficiency using the libraries and managing resources.

7.1.2 Parallel computer platforms

The parallel and distributed training approach is broadly used by Deep Learning practitioners. This is because DNNs are compute-intensive, making them similar to traditional supercomputing (high-performance computing, HPC) applications. Thus, large learning workloads perform very well on accelerated systems such as general-purpose graphics processing units (GPU) that have been used in the Supercomputing field.

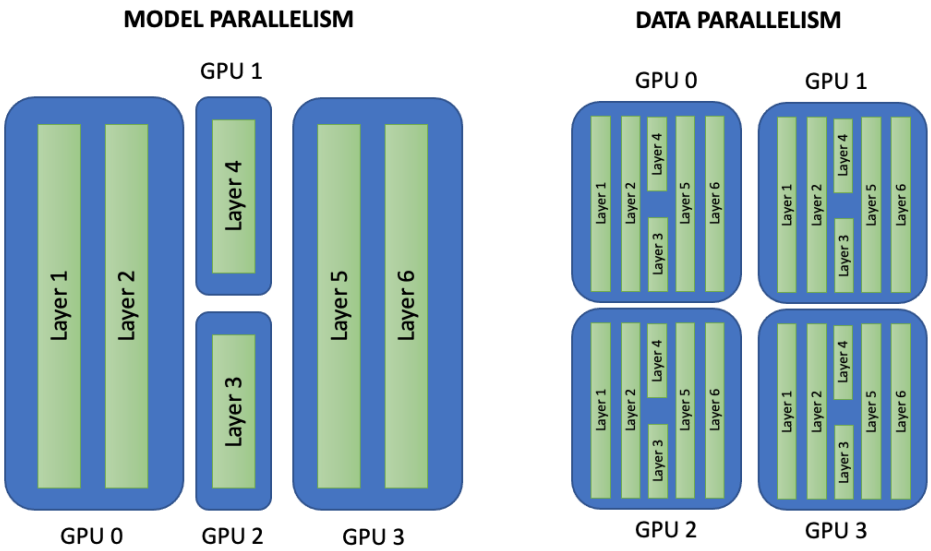
The main idea behind this computing paradigm is to run tasks in parallel instead of serially, as it would happen in a single machine (or single GPU). Multiple GPUs increase both memory and compute available for training a DNN. In a nutshell, we have several choices, given a minibatch of training data that we want to classify. In the next subsection, we will go into an introduction of the main options.

7.1.3 Types of parallelism

To achieve the distribution of the training step, there are two principal implementations, and it will depend on the needs of the application to know which one will perform better, or even if a mix of both approaches can increase the performance.

For example, different layers in a Deep Learning model may be trained in parallel on different GPUs. This

training procedure is commonly known as **Model parallelism**. Another approach is **Data parallelism**, where we use the same model for every execution unit, but train the model in each computing device using different training samples.



> *In this hands-on, we will focus on the Data Parallelism approach.*

7.1.4 Scalable Deep Learning Frameworks

In distributed/parallel environments, there may be multiple instances of stochastic gradient descent (SGD) running independently. Thus, to parallelise the SGD training algorithm, the overall algorithm must be adapted and consider different **model consistency** or **parameters distribution** issues. As you can imagine, these are no easy tasks. But luckily there are software libraries, known as DL frameworks, that facilitate this parallelization or distribution. **We can use frameworks like TensorFlow to program multi-GPU training in one server. Let's see how we can take advantage of a server with 4 GPUs.**

7.2 TensorFlow for multiple GPUs

If our server/node has more than one GPU, in TensorFlow the GPU with the lowest ID will be selected by default. However, TensorFlow does not place operations into multiple GPUs automatically and we need to add some code using a specific API.

`tf.distribute.Strategy` is a TensorFlow API to distribute training across multiple GPU or TPUs with minimal code changes (from the sequential version presented in the previous section). This API can be used with a high-level API like Keras, and can also be used to distribute custom training loops.

`tf.distribute.Strategy` intends to cover a number of distribution strategies use cases along different axes. The official web page of this feature presents all the currently supported combinations, however, in this course we will focus our attention on `tf.distribute.MirroredStrategy` one of the strategies included in `tf.distribute.Strategy.`

`tf.distribute.MirroredStrategy` supports the training process on multiple GPUs (**multiple devices**) on one server (**single host**). It creates one replica per GPU device. Each variable in the model is mirrored across all the replicas. These variables are kept in sync with each other by applying identical updates.

Let's assume we are on a *single machine* that has *multiple GPUs* and we want to use more than one GPUs for training. We can accomplish this by creating our `MirroredStrategy`:

```
mirrored_strategy = tf.distribute.MirroredS
trategy()
```

This will create a `MirroredStrategy` instance that will use all the GPUs visible to TensorFlow. It is possible to see the list of available GPU devices doing the

following:

```
devices = tf.config.experimental.list_phy
sical_devices(“GPU”)
```

It is also possible to use a subset of the available GPUs in the system by doing the following:

```
mirrored_strategy = tf.distribute.MirroredS
trategy(devices=["/gpu:0", "/gpu:1"])
```

We then need to declare our model architecture and compile it within the scope of the `MirroredStrategy`. To build the model and compile it inside the `MirroredStrategy`scope we can do it in the following way:

```
with mirrored_strategy.scope():
     model = tf.keras.applications.resnet_v2
             include_top=True, weights=None,
             input_shape=(128, 128, 3), clas

     opt = tf.keras.optimizers.SGD(learning_

     model.compile(loss='sparse_categorical_
                   optimizer=opt, metrics=[‘
```

This allows us to create distributed variables instead of regular variables: each variable is mirrored across all the replicas and is kept in sync with each other by applying identical updates. It is important during the coding phase, that the creation of variables should be under the strategy scope. In general, this is only during the model construction step and the compile step. Training can be done as usual outside the strategy scope with:

```
dataset = load_data(batch_size)
model.fit(dataset, epochs=5, verbose=2)
```

8—Parallelization of the Case Study

In this section, we will show how we can **parallelize the training step** on the CTE-POWER cluster using the same case study that classifies the CIFAR10 dataset using the ResNet50 neural network.

8.1 Parallel code for ResNet50 neural network

Following the steps presented in the above section on how to apply `MirroredStrategy`, below we present the resulting parallel code for the ResNet50:

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
```

```
import numpy as np
import argparse
import time
import sys
```

```
sys.path.append('/gpfs/projects/nct00/nct
00002/cifar-utils')
from cifar import load_cifar
```

```
parser = argparse.ArgumentParser()
parser.add_argument(‘ -- epochs’, type=in
t, default=5)
parser.add_argument(‘ -- batch_size’, typ
e=int, default=2048)
parser.add_argument(‘ -- n_gpus’, type=in
t, default=1)
```

```
args = parser.parse_args()
batch_size = args.batch_size
epochs = args.epochs
n_gpus = args.n_gpus
```

```
train_ds, test_ds = load_cifar(batch_size)
```

```
device_type = ‘GPU’
devices = tf.config.experimental.list_phy
sical_devices(
          device_type)
devices_names = [d.name.split(“e:”)[1] fo
r d in devices]
```

```
strategy = tf.distribute.MirroredStrategy
(
           devices=devices_names[:n_gpus])
```

```
with strategy.scope():
     model = tf.keras.applications.resnet_v2
             include_top=True, weights=None,
```

```
                input_shape=(128, 128, 3), clas
        opt = tf.keras.optimizers.SGD(0.01*n_gp
        model.compile(loss='sparse_categorical_
                      optimizer=opt, metrics=['
```

```
model.fit(train_ds, epochs=epochs, verbos
e=2)
```

8.2 Choose the Batch Size and Learning Rate

When training, it is required to allocate memory to store samples for training the model and the model itself. We have to keep in mind this in order to avoid an out-of-memory error.

Remember that the `batch_size` is the number of samples that the model will see at each training step, and in general, **we want to have this number as biggest as possible** (powers of 2). We can calculate it by try and error approach testing different values until an error related to the memory capacity appears:

```
python ResNet50.py -- epoch 1 -- batch_si
ze 16
python ResNet50.py -- epoch 1 -- batch_si
ze 32
python ResNet50.py -- epoch 1 -- batch_si
ze 64
.
.
.
```

> *Frim know on, you can assume that*

> *in this case study the maximum* `batch_size` *is* `256`

When using `MirroredStrategy` with multiple GPUs, the batch size indicated is divided by the number of replicas. Therefore the `batch_size` **that we should specify to TensorFlow is equal to the maximum value for one GPU multiplied by the number of GPUs we are using**. This is, in our example, use these flags in the python program:

```
python ResNet50.py -- epochs 5 -- batch_s
ize 256  -- n_gpus 1
python ResNet50.py -- epochs 5 -- batch_s
ize 512  -- n_gpus 2
python ResNet50.py -- epochs 5 -- batch_s
ize 1024 -- n_gpus 4
```

Accordingly, with the `batch_size`, if we are using `MirroredStrategy` with multiple GPUs, we change the `learning_rate`to `learning_rate*num_GPUs`:

```
learning_rate = learning_rate_base*number
_of_gpus
opt = tf.keras.optimizers.SGD(learning_ra
te)
```

We do this update of the `learning_rate`, due to the researchers say that because of a larger batch_size, we can also take bigger steps in the direction of the minimum to preserve the number of epochs to converge.

8.3 SLURM script

The SLURM script that allocates resources and executes the model for different numbers of GPUs can be as the following one (`ResNet50.sh`):

```
#!/bin/bash
#SBATCH --job-name="ResNet50"
#SBATCH --D .
#SBATCH --output=ResNet50_%j.output
#SBATCH --error=ResNet50_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=160
#SBATCH --gres=gpu:4
#SBATCH --time=00:30:00
```

```
module purge; module load gcc/8.3.0 cuda/
10.2 cudnn/7.6.4 nccl/2.4.8 tensorrt/6.0.
1 openmpi/4.0.1 atlas/3.10.3 scalapack/2.
0.2 fftw/3.3.8 szip/2.1.1 ffmpeg/4.2.1 op
encv/4.1.1 python/3.7.4_ML
```

```
python ResNet50.py -- epochs 5 -- batch_s
ize 256 -- n_gpus 1
python ResNet50.py -- epochs 5 -- batch_s
ize 512 -- n_gpus 2
python ResNet50.py -- epochs 5 -- batch_s
ize 1024 -- n_gpus 4
```

If we use the same SLURM script for all three executions, pay attention to indicate the maximum number of GPUs required with `--gres=gpu:4` .

Once we run the script, in the file that has stored the *standard output* we find the following execution times:

```
python ResNet50.py --epochs 5 --batch_size 256 --n_gpus 1
Epoch 1/5
196/196 - 49s - loss: 2.0408 - accuracy: 0.2506
Epoch 2/5
196/196 - 45s - loss: 1.7626 - accuracy: 0.3536
Epoch 3/5
196/196 - 45s - loss: 1.5863 - accuracy: 0.4164
Epoch 4/5
196/196 - 45s - loss: 1.4550 - accuracy: 0.4668
Epoch 5/5
196/196 - 45s - loss: 1.3539 - accuracy: 0.5070
```

```
python ResNet50.py --epochs 5 --batch_size 512 --n_gpus 2
Epoch 1/5
98/98 - 26s - loss: 2.0314 - accuracy: 0.2498
Epoch 2/5
98/98 - 24s - loss: 1.7187 - accuracy: 0.3641
Epoch 3/5
98/98 - 24s - loss: 1.5731 - accuracy: 0.4207
Epoch 4/5
98/98 - 24s - loss: 1.4543 - accuracy: 0.4686
Epoch 5/5
98/98 - 24s - loss: 1.3609 - accuracy: 0.5049
```

```
python ResNet50.py --epochs 5 --batch_siz
e 1024 --n_gpus 4
Epoch 1/5
49/49 - 14s - loss: 2.0557 - accuracy: 0.
2409
Epoch 2/5
49/49 - 12s - loss: 1.7348 - accuracy: 0.
3577
Epoch 3/5
49/49 - 12s - loss: 1.5696 - accuracy: 0.
4180
Epoch 4/5
49/49 - 12s - loss: 1.4609 - accuracy: 0.
4625
Epoch 5/5
49/49 - 12s - loss: 1.3689 - accuracy: 0.
5010
```

It is important to note that we center our interest on the computational speed of the process rather than the model's accuracy. For this reason, we will only execute a few epochs during the training, due as we can see, training times per epoch are constant (approx.), and with 5 epochs for each experiment, we achieve the same accuracy in all three cases. That means that only 5 epochs allow comparing the three options.

```
1 GPU: 45 seconds
2 GPU: 24 seconds
4 GPU: 12 seconds
```

> *In this hands-on exercise,* ***we will consider the epoch time as a measure of the computation time*** *for training a Distributed Neural*

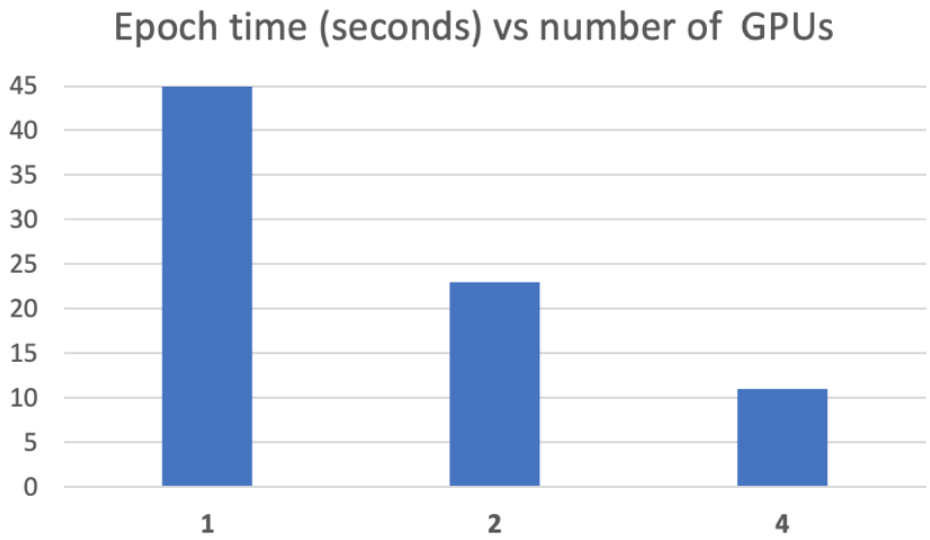
> *Network (DNN). This approximated measure in seconds, provided by Keras by the* `.fit` *method,* ***is enough for the purpose of this academic exercise****. In our case, we suggest discarding the first time epoch as it includes creating and initializing structures. Obviously, for certain types of performance studies, it is necessary to go into more detail, differentiating the loading data, feeds forward time, loss function time, backpropagation time, etc., but it falls outside the scope of this academic case study that we described in this hands-on exercise.*

Task 7:

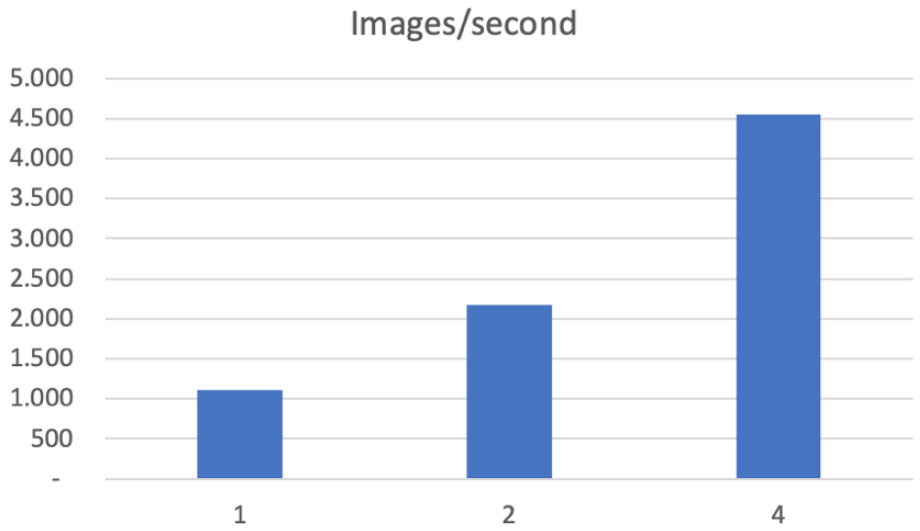
Execute the parallel `ResNet50.py` program for 1,2, and 4 GPUs in CTE-POWER cluster using the SLURM job script `ResNet50.sh` presented in GitHub. Inspect the `.out` and `.err` files.

9—Analysis of the results

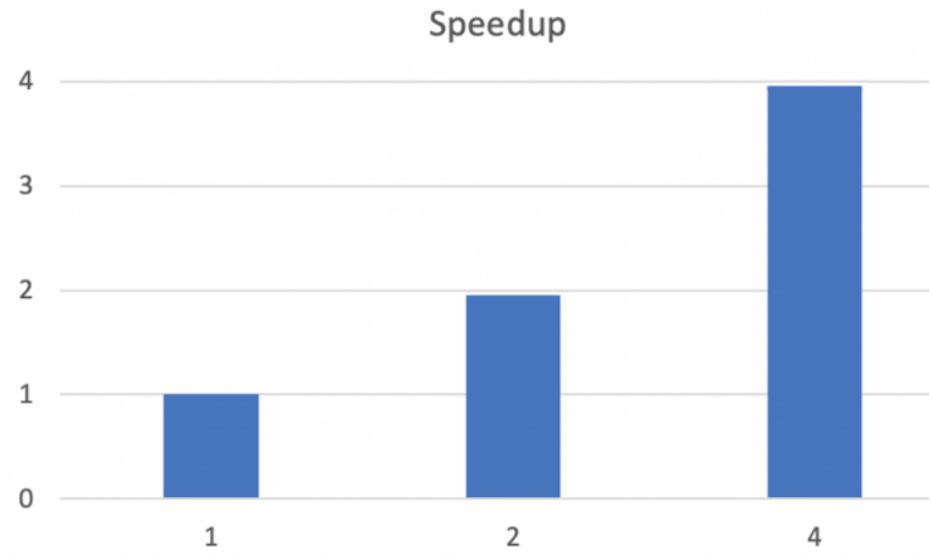
It is time to analyse the results obtained in Task 7. We expect that on average the **time** required for doing one epoch should be similar to the values shown in the following plot (results of Task 7 done by the teacher):



We can translate this to images per second (since we know there are 50,000 images), which gives us the **throughput**:



Finally, as we said, the **speedup** is a relevant metric:



It's an almost linear speedup!. We refer to linear

speedup when the workload is equally divided between the number of GPUs.

Task C (optional):

Obtain the data from your `.out` file and generate the plots of epoch time, image/second, and speedup for your `ResNET50V2` classifier.

Now it's your turn to **get your hands really dirty** and reproduce the above results for the `ResNET152V2` classifier.

Task D (optional):

Create the new `.py` to do the experiments with this `ResNET152V2`new classifier. Include the file `.py`in the answer to this Task.

Remember that the first step is to find the best `batch_size`.

Task E (optional):

Determine the maximum `batch_size` that we can use for training this classifier (following the approach presented in section 3.2). Include in your answer the relevant information from in the `.err` file.

Task F (optional):

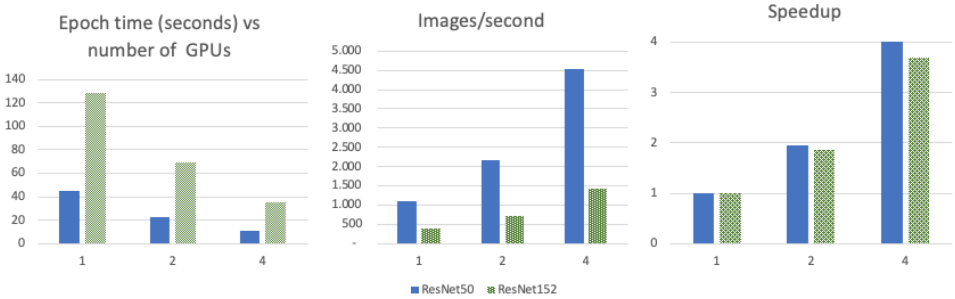
Execute the parallel `ResNET152V2.py` program for 1,2, and 4 GPUs in CTE-POWER cluster using your job script `ResNET152V2.sh` based on the previously `ResNet50.sh` used. Inspect the `.out` and `.err` files. Include in the answer the codes `ResNET152V2.py` , `ResNET152V2.sh`used and the relevant part of the `.out` and `.err` files that demonstrate your results.

(*) Warning with the `learning_rate` requirements presented in the previous section.

Task G (optional):

Generate the plots of epoch time, image/second, and Speedup and compare them with those presented in the previous section. Include in the answer the relevant part of the `.out` and `.err` files that justify your results (if not included in Task 9).

As a hint, if you plot the results of both case studies together, you should find results comparable to those shown below:



A couple of relevant things are observed. The first is that the time to run a ResNet152 epoch is much longer, and therefore the throughput in images per second is much lower than on the ResNet50 network. Why is this happening? ResNet152 network is deeper, meaning that it has more layers, therefore the training time will be higher.

It can be seen that the speedup for the ResNet152 is no

longer linear; could you try to give a possible answer as to why this is happening? Obviously, it depends on many things and it is required a detailed analysis; however, due to the biggest size of the network, it is adding additional latency for synchronization.

10—Conclusions

As we can see in the results, the accuracy achieved by our model is more or less constant independently of the number of GPUs. **In summary, using a distributed strategy, our model can learn the same but faster, thanks to parallelization!**

Acknowledgement: Many thanks to Juan Luis Domínguez and Oriol Aranda, who helped with the first version of the codes that appear in this hands-on, and to the support team at BSC Operations Department for the essential support using the CTE-POWER cluster. Also, many thanks to Alvaro Jover Alvarez, Miquel Escobar Castells, and Raul Garcia Fuentes for their contributions to the proofreading of previous versions of this hands-on.