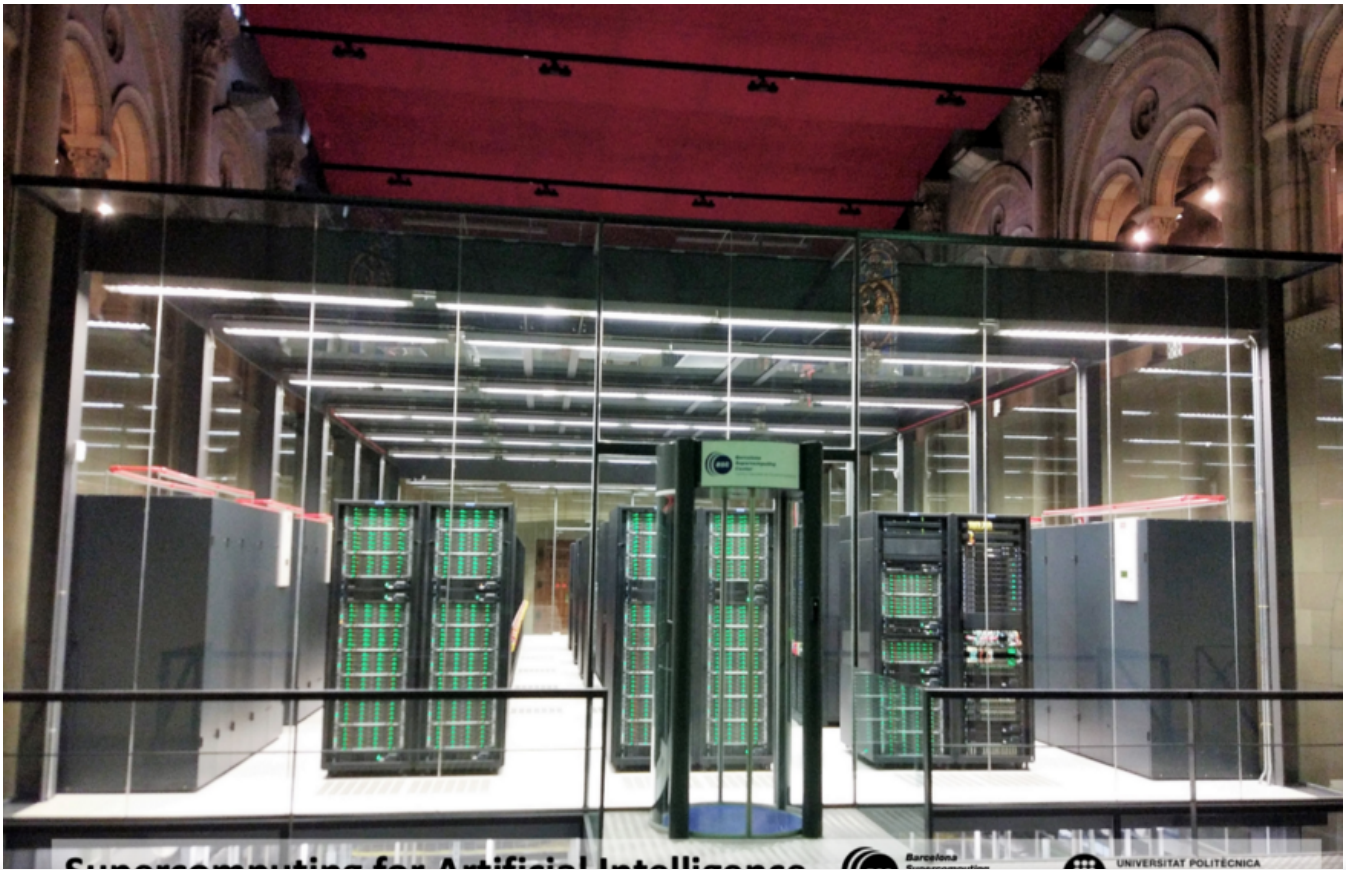
Jordi TORRES.AI Follow

Nov 23, 2020 · 13 min read · ✨ · Listen

SUPERCOMPUTING FOR ARTIFICIAL INTELLIGENCE — 03

Deep Learning Frameworks for Parallel and Distributed Infrastructures

Reduce training time for deep neural networks using Supercomputers



[This post will be used in the master course *Supercomputers Architecture* at UPC Barcelona Tech with the support of the BSC]

> "Methods that scale with computation are the future of Artificial Intelligence" — Rich Sutton, father of reinforcement learning (video 4:49)

This third post of this series will explore some fundamental concepts in distributed and parallel Deep Learning training and introduce current deep learning frameworks used by the community. We also introduce a sequential code of an image classification problem that we will use as a baseline to calculate the scalability performance that can be obtained with the proposed parallel and distributed frameworks in the next two posts.

1 Basic concepts in distributed and parallel training Deep Learning

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 by processing millions of examples so that the development of more accurate algorithms are usually limited by the throughput of the computing devices on which they are trained.

1.1 Performance metrics

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**.

The accuracy is independent of the platform, and it is the performance metric to compare multiple models. In contrast, the computation speed depends on the platform on which the model is deployed. This post will measure it by metrics such as **Speedup**, the ratio of solution time for the sequential (or 1 GPU) algorithms versus its parallel counterpart (with many GPUs). This is a prevalent concept in our daily argot in the Supercomputing community ;-) .

Another important metric is **Throughput**; for example, the number of images per unit time that can be processed. This can give us a good benchmark of performance

Finally, a concept that we usually 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 cluster configuration, the type of network used, or the framework's efficiency using the libraries and managing resources.

1.2 Parallel computer platforms

For this reason, an approach for parallel and distributed training is used. The main idea behind this computing paradigm is to run tasks in parallel instead of serially, as it would happen in a single machine.

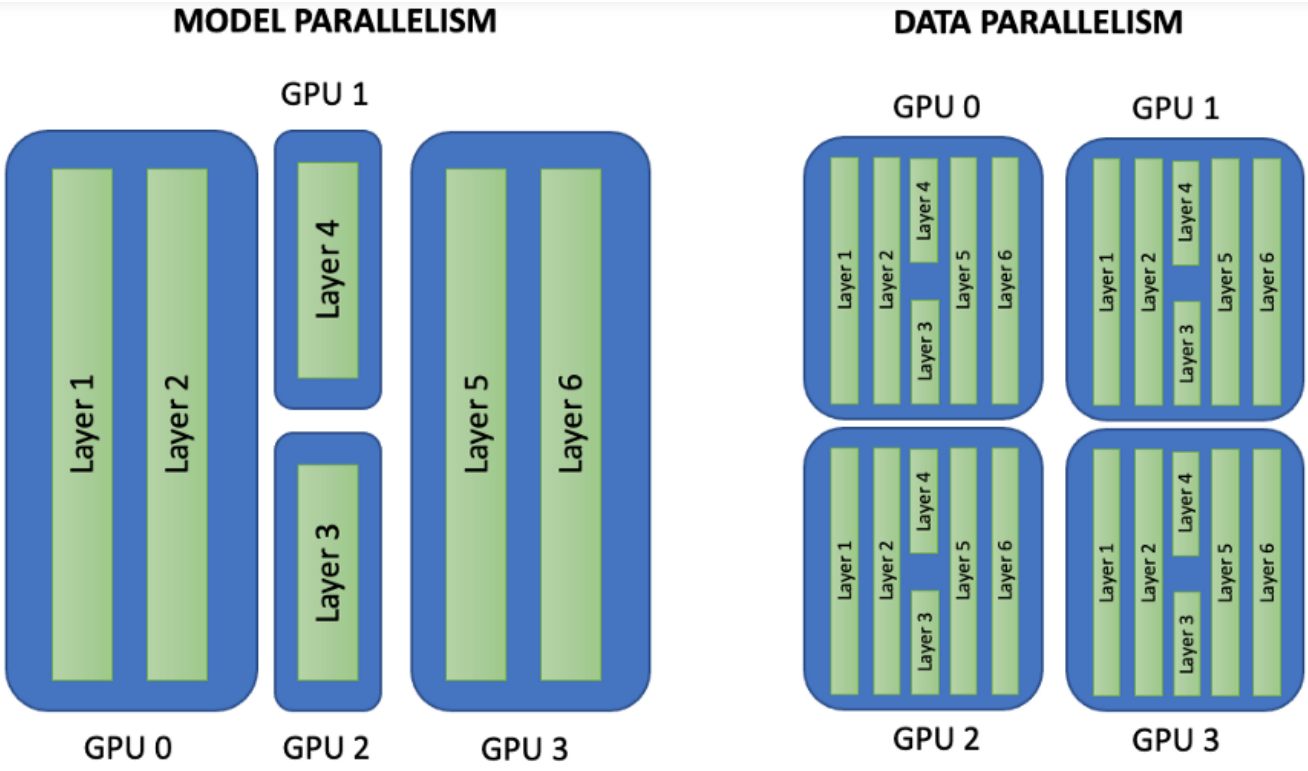
DNN are often compute-intensive, making them similar to traditional supercomputing (high-performance computing, HPC) applications. Thus, large learning workloads perform very well on accelerated systems such as general-purpose graphics processing units (GPU) that have been used in the Supercomputing field.

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 more detail about this.

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.



(Image by author)

1.3.1 Data parallelism

In this mode, the training data is divided into multiple subsets, and each one of them is run on the same replicated model in a different GPU (worker nodes). These will need to synchronize the model parameters (or its "gradients") at the end of the batch computation to ensure they are training a consistent model (just as if the algorithm run on a single processor) because each device will independently compute the errors between its predictions for its training samples and the labeled outputs (correct values for those training samples). Therefore, each device must send all of its changes to all of the models on all the other devices.

One interesting property of this setting is that it will scale with the amount of data available, and it speeds up the rate at which the entire dataset contributes to the optimization. Also, it requires less communication between nodes, as it benefits from a high amount of computations per weight. On the other hand, the model has to fit on each node entirely, and it is mainly used for speeding computation of convolutional neural networks with large datasets.

1.3.2 Model parallelism

into a particular layer, processes data across several subsequent layers in the neural network, and then sends the data to the next GPU.

In this case (also known as Network Parallelism), the model will be segmented into different parts that can run concurrently, and each one will run on the same data in different nodes. This method's scalability depends on the degree of task parallelization of the algorithm, and it is more complex to implement than the previous one. It may decrease the communication needs, as workers need only to synchronize the shared parameters (usually once for each forward or backward-propagation step) and works well for GPUs in a single server that shares a high-speed bus. It can be used with larger models as hardware constraints per node are no more a limitation.

In general, the parallelization of the algorithm is more complex to implement than run the same model in a different node with a subset of data.

In this series of posts, we will focus on the Data Parallelism approach.

2 Concurrency in data parallelism training

In distributed environments, there may be multiple instances of Stochastic gradient descent (SGD) running independently. Thus, the overall algorithm must be adapted and should consider different issues related to the **model consistency** or **parameters distribution**.

2.1 Synchronous versus asynchronous distributed training

Stochastic gradient descent (SGD) is an iterative algorithm that involves multiple rounds of training, where the results of each round are incorporated into the model in preparation for the next round. The rounds can be run on multiple devices, either synchronously or asynchronously.

Each SGD iteration runs on a mini-batch of training samples. In synchronous training, all the devices train their local model using different parts of data from a single (large) mini-batch. They then communicate their locally calculated gradients (directly or indirectly) to all devices. Only after all devices have successfully computed and sent their gradients th model is updated. The updated model is then sent to all nodes along with splits from the next mini-batch


through one or more central servers known as “parameter” servers. In the peer architecture, each device runs a loop that reads data, computes the gradients, sends them (directly or indirectly) to all devices, and updates the model to the latest version.

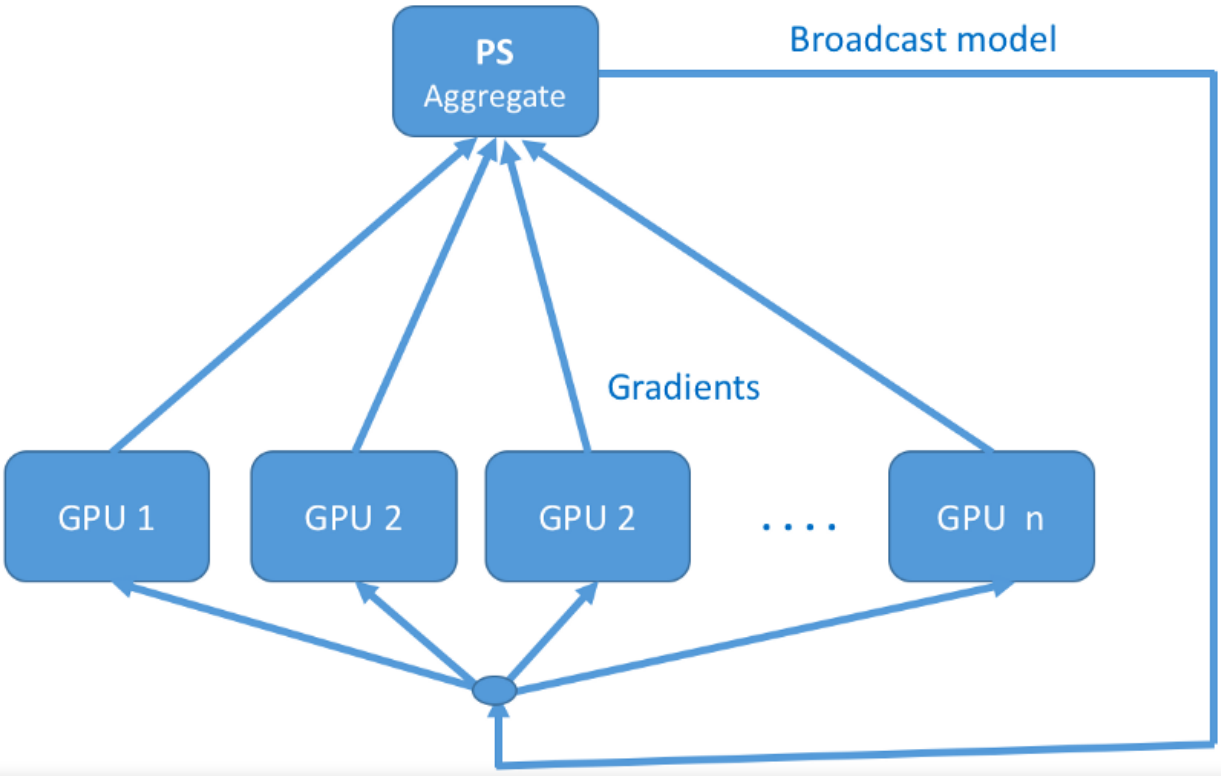
In practical implementations, the approaches are synchronous for up to 32–50 nodes and asynchronous for larger clusters and heterogeneous environments, according to this survey from ETH (Zurich). In this post, we will focus on a synchronous training approach.

2.2 Parameter distribution and communication in synchronous training

For synchronous training, we can choose between two main schemes: centralized or decentralized. The choice between designing a centralized and a decentralized scheme for DNN training depends on multiple factors, including the network topology, bandwidth, communication latency, parameter update frequency, or desired fault tolerance.

2.2.1 Centralized

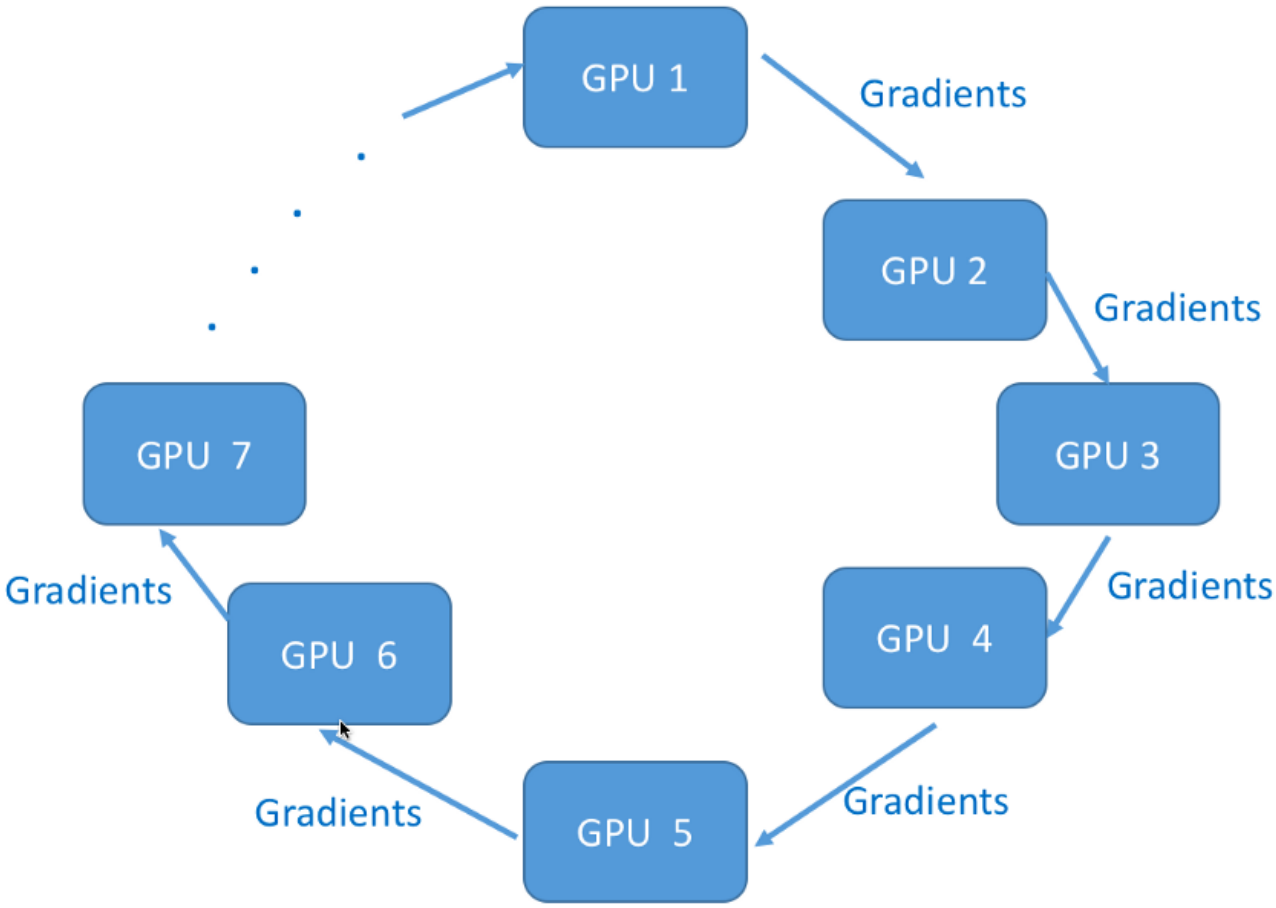
The centralized scheme would typically include a so-called Parameter Server strategy.



When parallel SGD uses parameter servers, the algorithm starts by broadcasting the model to the workers (servers). Each worker reads its own split from the mini-batch in each training iteration, computing its own gradients, and sending those gradients to one or more parameter servers. The parameter servers aggregate all the gradients from the workers and wait until all workers have completed before they calculate the new model for the next iteration, which is then broadcasted to all workers.

2.2.2 Decentralized

The decentralized scheme would rely on ring-allreduce to communicate parameter updates among the nodes. In the ring-allreduce architecture, there is no central server that aggregates gradients from workers. Instead, in a training iteration, each worker reads its own split for a mini-batch, calculates its gradients, sends its gradients to its successor neighbor on the ring, and receives gradients from its predecessor neighbor on the ring.



But luckily there are software libraries, known as DL Frameworks, that facilitate this parallelization or distribution that we saw in the previous section.

In this section, we will make a very brief introduction to the frameworks that we will use in the next two posts to speed up the training process using the techniques presented in the previous section.

3.1 Parallel training in one server

As we presented in the first post of this series, we can use frameworks as TensorFlow of Pytorch to program multi-GPU training in one server. To parallelize the training of the model, you only need to wrap the model with `torch.nn.parallel.DistributedDataParallel` in PyTorch and with `tf.distribute.MirroredStrategy` in TensorFlow.

3.2 Distributed training in multiple server

However, the number of GPUs that we can place in a server is very limited, and the solution goes through putting many of these servers together, as we did at the BSC, with the CTE-POWER supercomputer, where 54 servers are linked together with an InfiniBand network on optical fiber.

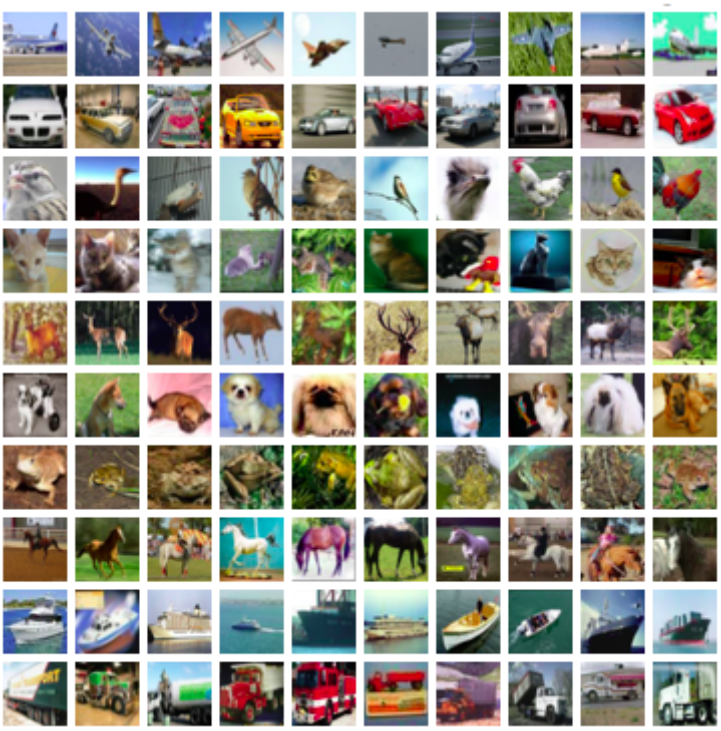
In this new scenario, we need an extension of the software stack to deal with multiple distributed GPUs in the neural network training process. There are other options, but in our research group at BSC, we decided to use Horovod, from Uber. Horovod Plugs into TensorFlow, PyTorch.

4 Case study: image classification

But before continuing, we take advantage of this post to introduce the sequential code of an image classification problem that we will use as a baseline to calculate the scalability performance that can be obtained with the proposed parallel and distributed frameworks.Specifically, we will use a neural network known as ResNet50V2 with 25,613,800 parameters. And as a dataset, we will use the popular CIFAR10.

4.1 Dataset: CIFAR10


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 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 being able to see larger training times for better comparison, we have applied a resize operation to make the images of 128x128 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
```

can be obtained from the course repository GitHub for readers that

4.2 Neural Networks architecture: ResNet

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 post, we will consider two architectures from the family of ResNet as a case study: ResNet50v2 and ResNet152v2.

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.ResNet101V2(
    include_top=True,
    weights="imagenet",
    input_tensor=None,
    input_shape=None,
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
```

```
        pooling=None,
        classes=1000,
        classifier_activation="softmax",
    )
```

The “50” and “152” stand for the number of weight layers in the network. The arguments for both networks 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.


> *centering our interest in Accuracy, we didn't download the file with the imagenet weights, therefore, it must be used* `weights=None`.

4.3 How to train a neural network with a DL framework

As an example of how to train neural networks, let's start with a sequential version of the ResNet50 and use TensorFlow as a framework.

The `ResNet50_seq.py` code could be the following one:

```
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/nct00002/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'])
```

To run it after being granted with the required resources with `salloc` and loading the corresponding modules with `module load`, we can do it like this:

```
python ResNet50_seq.py --epochs 5 --batch_size 256 2> err.txt
```

Note that we redirect the standard error (otherwise, it would go through mixed screen system information) in order to see the result of the output that gives us the Keras:

```
Epoch 1/5
196/196 - 42s - loss: 2.0179 - accuracy: 0.2585
Epoch 2/5
196/196 - 42s - loss: 1.7293 - accuracy: 0.3654
Epoch 3/5
196/196 - 42s - loss: 1.5707 - accuracy: 0.4237
Epoch 4/5
196/196 - 42s - loss: 1.4572 - accuracy: 0.4664
Epoch 5/5
196/196 - 42s - loss: 1.3707 - accuracy: 0.5001
```

For the purpose of this post, to calculate the time, which will be the metric we will use to compare, we will use the time that Keras himself tells us that it takes an epoch (sometimes we discard the first epoch as it is different from the rest since it has to create structures in memory and initialize them). Remember that we are in an academic example, and with this approximate measure of time, we have enough for the course goals.

For any other of the networks available in Keras, simply change in the code the piece that is in bold, resnet_v2.ResNet50V2, for the corresponding network.

5 It is time to learn by doing

If you want to learn more and consolidate the knowledge acquired, now it's your turn to get your hands dirty. The proposal is to reproduce the above results for the `ResNet152V2`, both with salloc and the Batch sbatch command. Your results based on `ResNet152V2` will be used in the next two posts.

This post has described an overview of the techniques that allow accelerating a Deep Neural Network training by distributing it over many GPUs. We have introduced two frameworks that implement these techniques, TensorFlow and Horovod, that we will use in the next two posts.

This post is part of the documentation used in my master course *Supercomputers Architecture* at UPC Barcelona Tech with BSC's support. I'm sharing it in case it can be useful for other readers.

See you in the next post where we present how to parallelize with the `tf.distributed.MirroredStrategy()` TensorFlow API.

Content of this series:

SUPERCOMPUTING FOR ARTIFICIAL INTELLIGENCE

1. **Artificial Intelligence is a Supercomputing problem**
2. **Using Supercomputers for Deep Learning Training**
3. **Deep Learning Frameworks for Parallel and Distributed Infrastructures**
4. **Train a Neural Network on multi-GPU with TensorFlow**
5. **Distributed Deep Learning with Horovod**

Acknowledgment: Many thanks to Juan Luis Domínguez and Oriol Aranda from BSC-CNS, who wrote the first version of the codes that appear in this series of posts, and to Carlos Tripiana for the essential support with the deployment of the software stack of POWER-CTE Supercomputer. Also many thanks to Alvaro Jover Alvarez, Miquel Escobar Castells and Raul Garcia Fuentes for their contributions to the proofreading of this document.