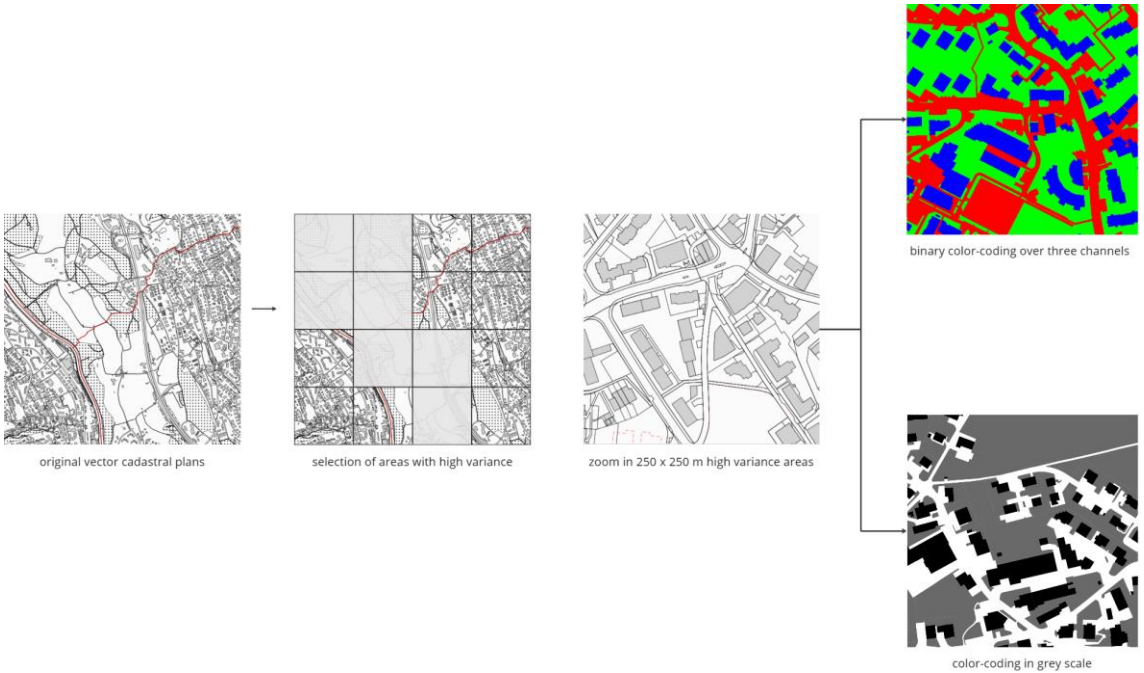
ASYNCHRONOUS MULTIPROCESS LARGE MODEL TRAINING ON PYTORCH FOR SYNTHETIC CITIES GENERATION

Pavel Sulimov, ZHAW
Furio Valerio Sordini, Implenia

Swiss Python Summit
21 September 2023

ABOUT SYNTHETIC CITIES

- The existing methods based on manual CAD drafting are inefficient and expensive: the number of possible drafts is limited by the skills and labor intensity of the expert.
- Using the dataset of cadastral maps of cantons of Switzerland, we suggest the model architecture that considers the city's morphology defined set of homogeneous entities: roads, buildings, and other areas.



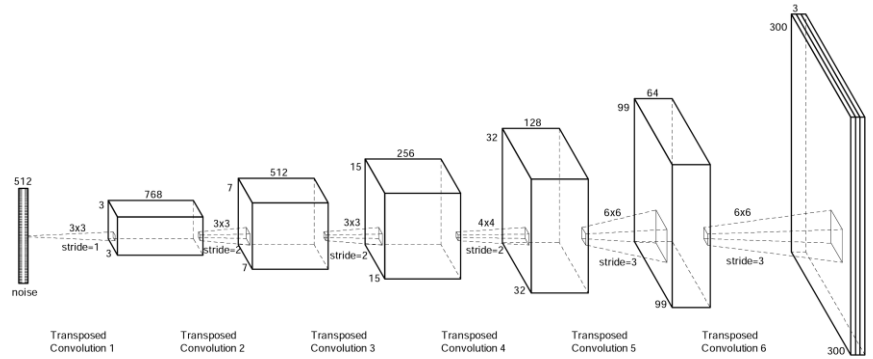
ABOUT THE SELECTED MODEL ARCHITECTURE

General Adversarial Networks

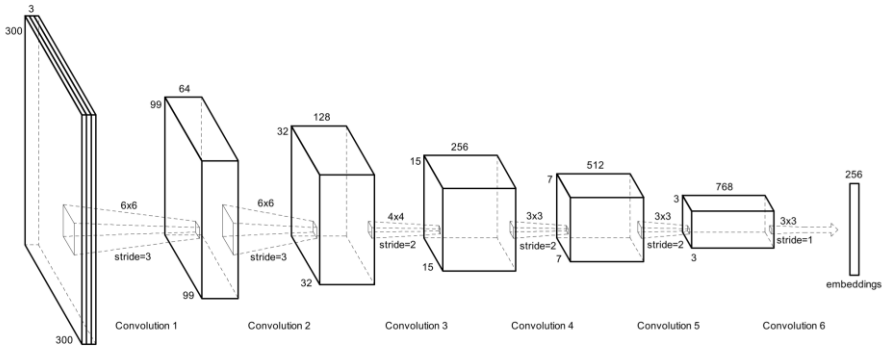
The solution is based on General Adversarial Networks, a method where two deep learning model compete against each other.

The first model, the generator, receives random noise as input and passes it through a series of hidden layer to generate an output, in this case a simulation of a urban cadastral plan.

The second network, the discriminators, receives both real images and simulations from the generator and aims to distinguish them.



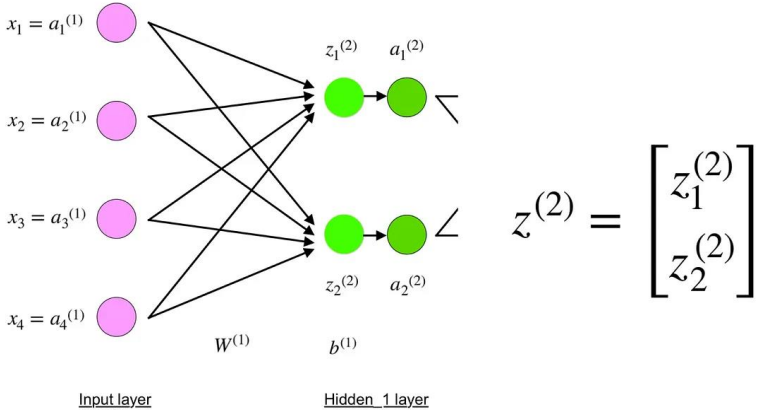
Generator architecture



Discriminator architecture

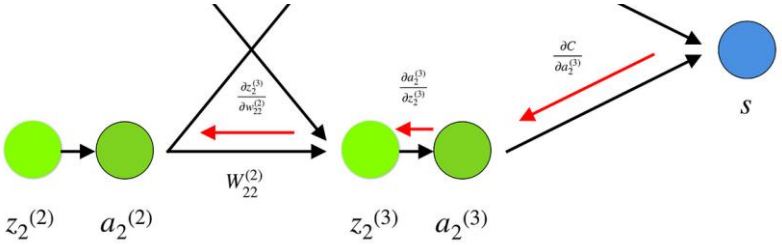
TRAINING A DEEP NEURAL NETWORK

1) Load the full model on the processor
2) Repeat the following process over more steps (epochs)
   A. Divide the dataset in batches (here group of images)
   B. In the forward pass the data points are passed through the model to train the discriminator. Calculate the loss function.
   C. In the backward pass the gradient of the loss with respect to each parameter is calculated.
   D. The model parameters are updated.



Forward step

Source: Understanding Backpropagation Algorithm



Backward propagation

Source: Understanding Backpropagation Algorithm

SO WHAT IS THE PROBLEM?

Some complex deep learning models (like ours) consist of millions and even billions of parameters. Therefore:

- The processors do not always satisfy the capacity requirements and can hardly store the entire model.
- Training in a single thread is extremely inefficient

SO WHAT IS THE PROBLEM?

Some complex deep learning models (like ours) consist of millions and even billions of parameters. Therefore:

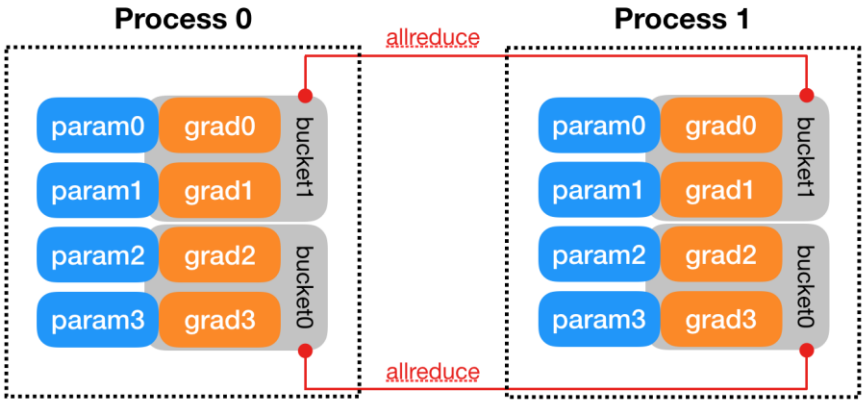
- The processors do not always satisfy the capacity requirements and can hardly store the entire model.
- Training in a single thread is extremely inefficient

Smells like a need for parallelisation? Let's use GPUs then!

- When given a task, a GPU will divide it into thousands of smaller subtasks and process them concurrently, instead of serially. E.g., in graphics rendering, GPUs handle complex mathematical and geometric calculations to create realistic visual effects and imagery. Instructions must be carried out simultaneously to draw and redraw images hundreds of times per second to create a smooth visual experience.
- GPUs are optimized for training deep learning models and can process multiple parallel tasks

DISTRIBUTED DATA PARALLEL

Data Distributed Parallel (DDP) acts on the training routine: divide data in random batches, forward pass, backward pass and model update.



Distributed Data Parallel
Source: [Distributed Data Parallel - PyTorch Documentation]

The batches are distributed over the processors. Depending on the number of batches and processors, each processor can repeat the process over more batches.

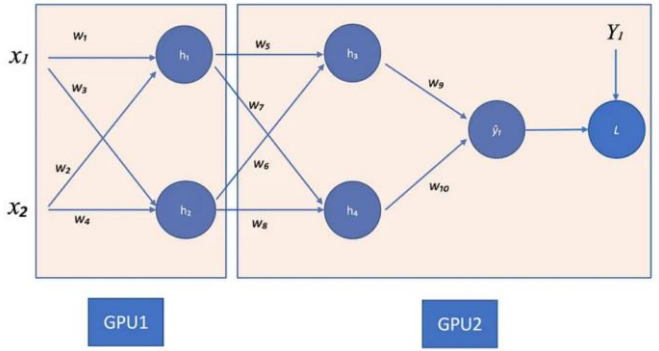
Each processors stores the entire model and passes the datapoints forward and backward.

The **`AllReduce`** GPU operator collects all the gradients calculated over the different batches and sums them to obtain the full gradient. The full gradient is used to update the model.

In DDP each GPU need to store the entire model to perform the forward and the backward pass!

DISTRIBUTED MODEL PARALLEL: VERTICAL CUT

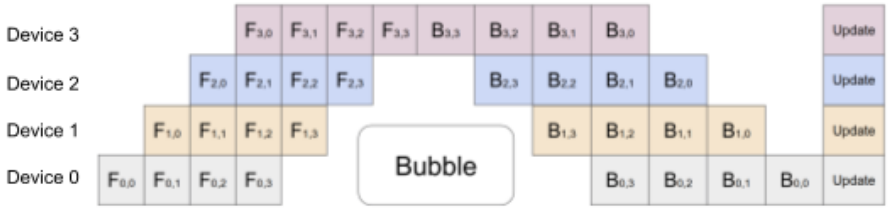
Naive Model Parallelism consists in distributing the layers of the neural network over the GPUs. Each time the GPU performs a step, the data needs to be passed to the other GPU: only one GPU is active, the others are idle.



Naïve Distributed Model Parallel

Source: [Distributed data parallel and distributed model parallel in PyTorch, by Wei Yi, Towards Data Science]

Pipeline Parallelism (PP) partially solves the problem by chunking the batches into minibatches and artificially creating a pipeline. The most advanced APIs introduce also the concept of interleaved pipelines, where the idle time is further reduced by prioritizing the backward passes.



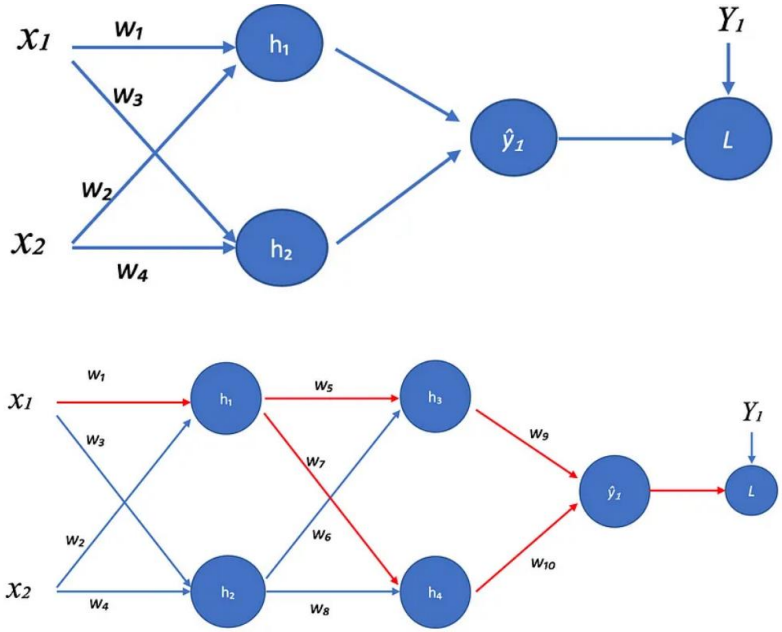
Pipeline Parallelism

Source: [Model Parallelism (huggingface.co)]

DISTRIBUTED MODEL PARALLEL: HORIZONTAL CUT

The model can be also distributed over the GPUs horizontally, separating each path instead of each layer as in the vertical cut. This process is also called **Tensor Parallelism (TP)**.

In the backward pass there are multiple routes from the model prediction to its inputs. The full gradient for a certain parameter needs to be consolidated over the different pass. The operation is done by the **`ReduceScatter`** operator: similar to the **`AllReduce`** operator it collects and consolidates the results over more GPUs, but it also scatters the results in equal blocks among GPUs, depending on a rank index.



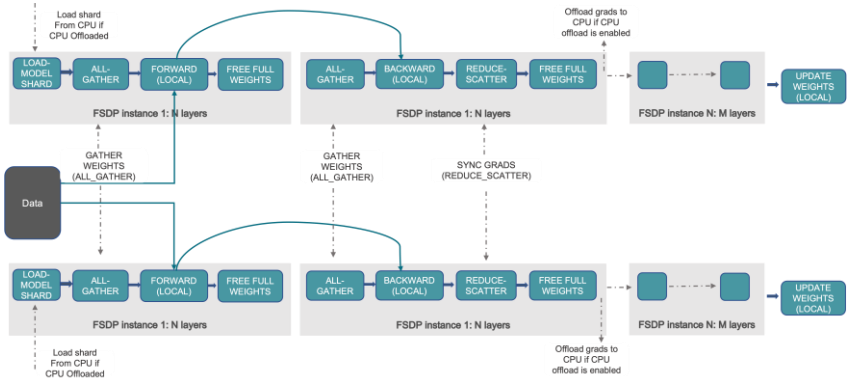
Horizontal Model Cut
Source: [Distributed data parallel and distributed model parallel in PyTorch, by Wei Yi, Towards Data Science]

AND THERE ARE MANY MORE OPTIONS FOR RICH …

Zero Redundancy Optimizer (ZeRO)

It is like to usual data parallel, but instead of replicating the whole model on each GPU, parameters, gradients and optimizer states are sharded over all the GPUs. The whole tensors gets reconstructed in time for backward or forward passes; therefore the model does not need any update. It supports offloading techniques to compensate for limited GPU memory.

Fully Shared Data Parallel (FSDP)



FSDP Workflow

Source: [Getting started with FSDP – PyTorch Tutorials]

WHAT CAN POOR PEOPLE DO? MULTIPROCESSING!

`torch.multiprocessing` is based on the `multiprocessing` (MP) module, available in native Python.

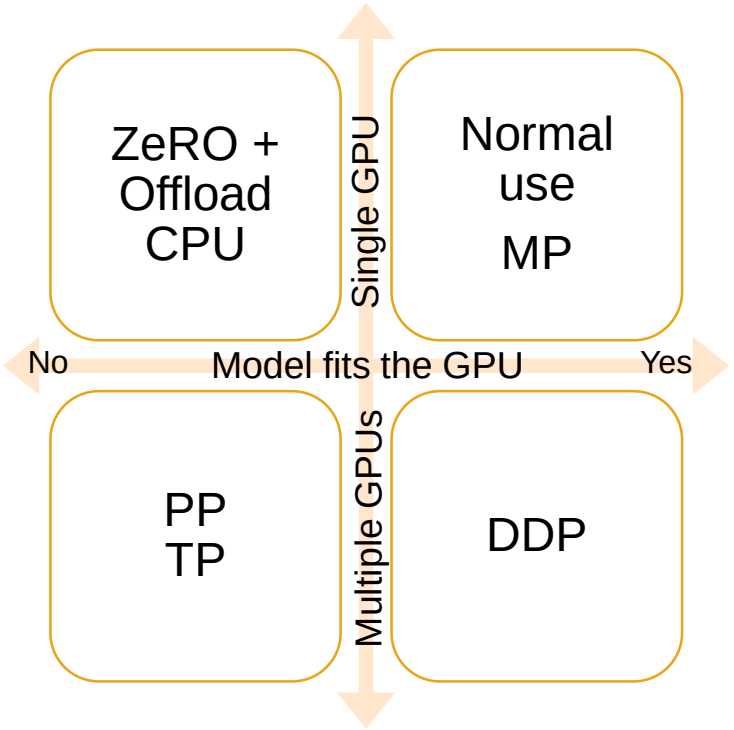
This module allows to queue, pool, manage processes and exchange variables (put or get) among them.

- subprocess can be executed on GPU (!)

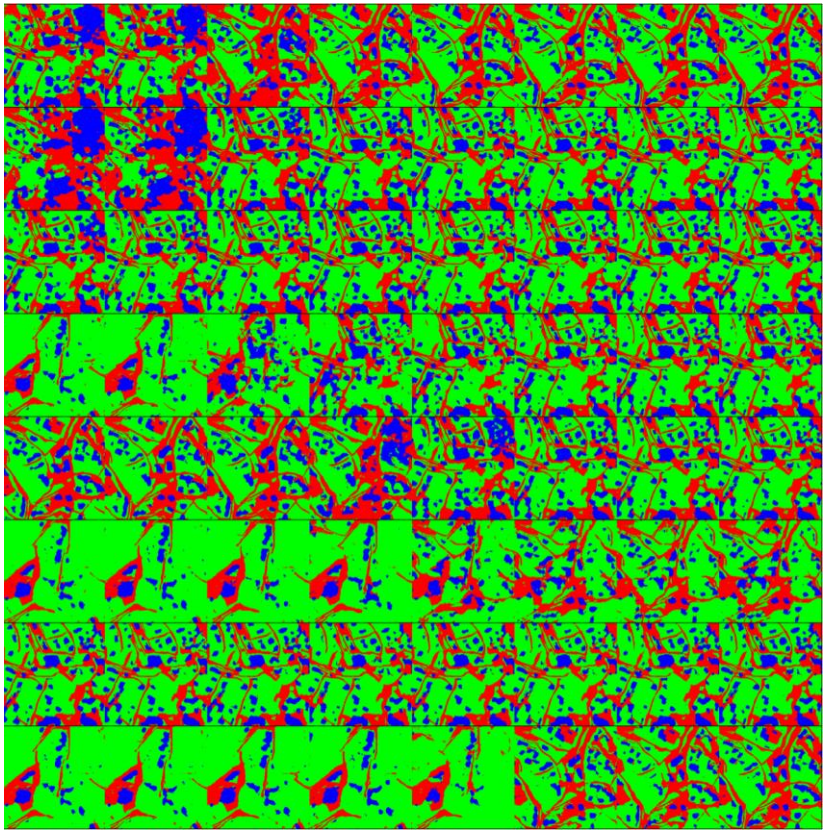
```python
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This will update the shared parameters

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```

Source: [Multiprocessing Best Practices – PyTorch]

RECOMMENDATIONS



RESULTS



The generator model used has 32.2M parameters and can be stored on a single GPU: distributed model parallel is therefore not necessary.

Still, multiprocessing can meaningfully speed up the training process, even using a single multi-core GPU.

Here below the performances with and without multiprocessing:

Training with Multiprocessing	~ 9.5 hrs
Traditional Training	~ 13 hrs

JOCKER: MOJO

FILE_NAME.🔥

```
struct MyPair:
  var first: Int
  var second: F32

  def __init__(self, first: Int, second: F32):
    self.first = first
    self.second = second
```

MOJO 🔥

MAKE_PLOT.🔥

```
def make_plot(m: Matrix):
  plt = Python.import_module("matplotlib.pyplot")
  fig = plt.figure(1, [10, 10 * yn // xn], 64)
  ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], False, 1)
  plt.imshow(image)
  plt.show()

make_plot(compute_mandelbrot())
```

matplotlib

FILE_NAME.🔥

```
def exp_buffer[dt: DType](data: ArraySlice[dt]):

  # Search for the best vector length
  alias vector_len = autotune(1, 4, 8, 16, 32)

  # Use it as the vectorization length
  vectorize[exp[dt, vector_len]](data)
```

THANK YOU FOR THE ATTENTION!

furiovalerio.sordini@implenia.com

pavel.sulimov@zhaw.ch