Deep Learning Optimisé - Jean Zay

Les parallélismes des gros modèles

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

DLO-JZ

IDRIS course, Part 6

Commented slides

Authors: Bertrand Cabot, Nathan Cassereau

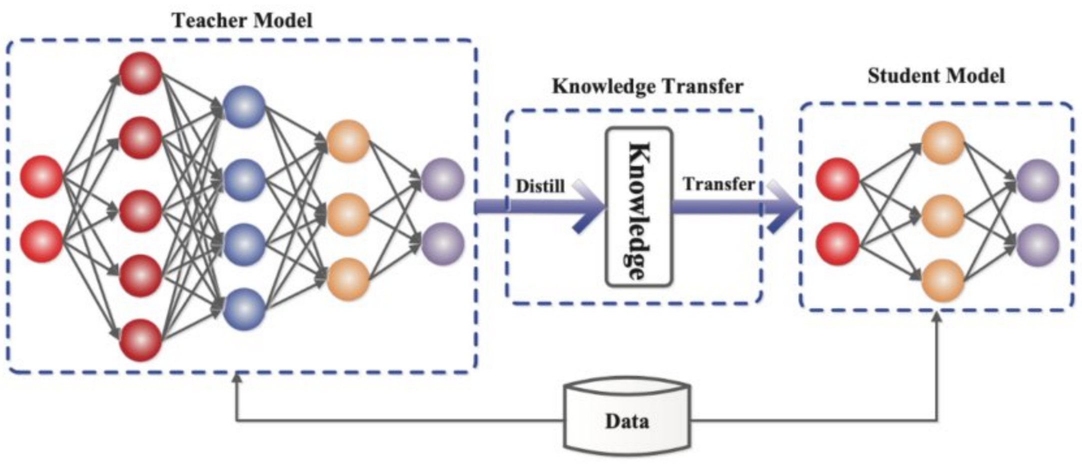
May 2022, updated October 2023

Chapters:

Inférence et fine-tuning

The goal of this section is to briefly present a few methods for inference and fine-tuning large models.

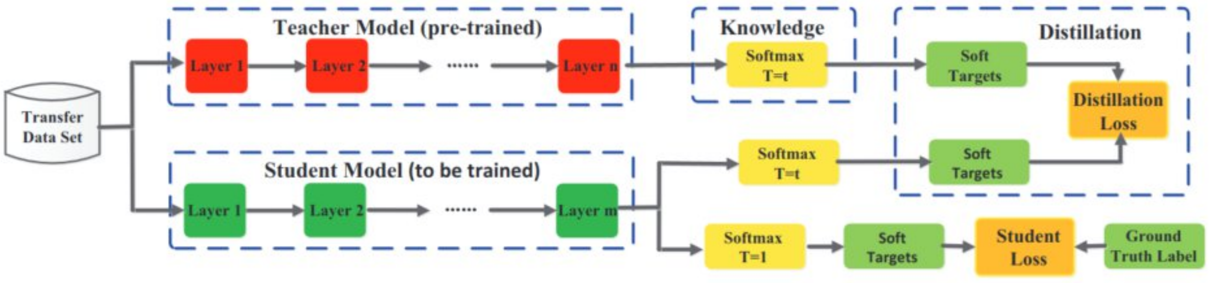
Distillation



[18]

Distillation enables solving a complex task with a relatively small model. It trains a smaller network and packs densely into it the knowledge acquired by a larger network. We will then have an insignificant loss of performance as well as a potentially great compute speed-up.

Distillation



$$\mathcal{L}_{\text{tot}} = \mathcal{L}_{\text{distil}}\left(y_{\text{teacher}}, y_{\text{student}}\right) + \lambda \mathcal{L}_{\text{CE}}\left(y_{\text{target}}, y_{\text{student}}\right)$$

Cross-entropy, Divergence KL, Wasserstein, ...

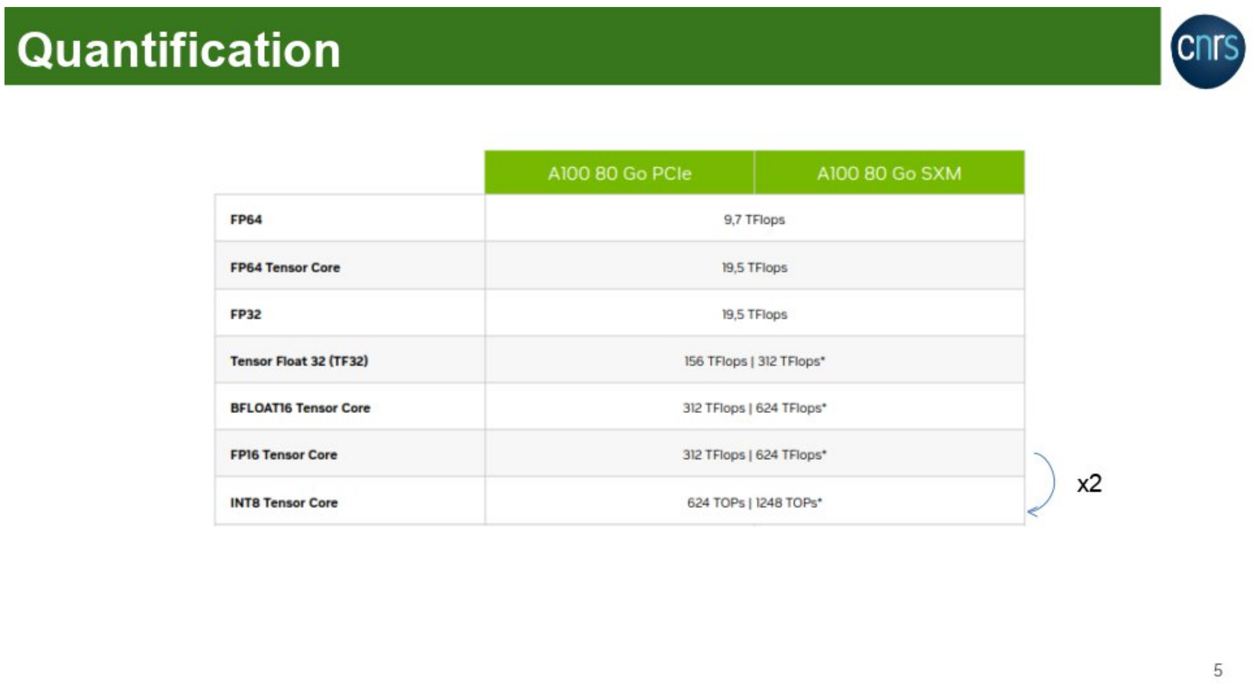
[18, 27]

DistilBERT a 40% moins de paramètres, mais ne perd que 3% de performance.

In order to perform distillation, one can train a small network exploiting the predictions of a pretrained larger network. It is also possible to use the actual labels to do a standard training step, provided that they are still available.

Possible loss functions between the teacher's prediction and the student's output include the mean square error, the cross-entropy, the Kullback-Leibler divergence, etc.

With this method, DistilBERT reached 97% of BERT performance with only 60% of its size.



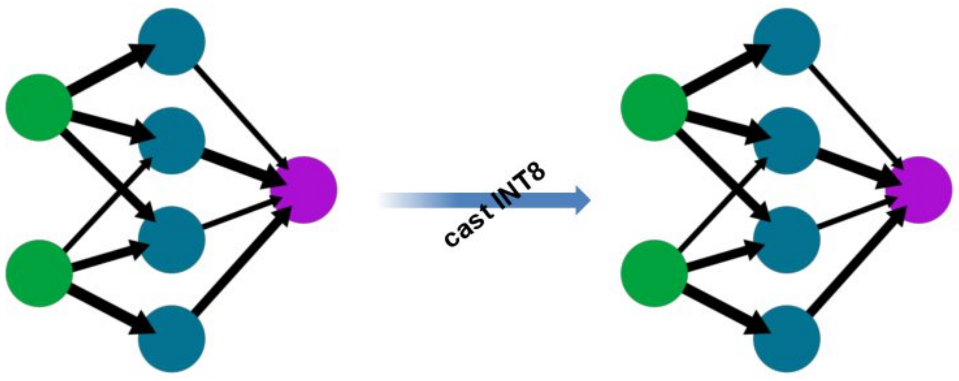
Quantification

	A100 80 Go PCIe	A100 80 Go SXM
FP64	9,7 TFlops	
FP64 Tensor Core	19,5 TFlops	
FP32	19,5 TFlops	
Tensor Float 32 (TF32)	156 TFlops | 312 TFlops*	
BFLOAT16 Tensor Core	312 TFlops | 624 TFlops*	
FP16 Tensor Core	312 TFlops | 624 TFlops*	
INT8 Tensor Core	624 TOPs | 1248 TOPs*	

x2

Quantization allows us to reduce the memory footprint of the model and reach the highest throughput which can be achieved by our hardware accelerator (namely a GPU most of the time) by using 8 bits integer instead of floating-point numbers.

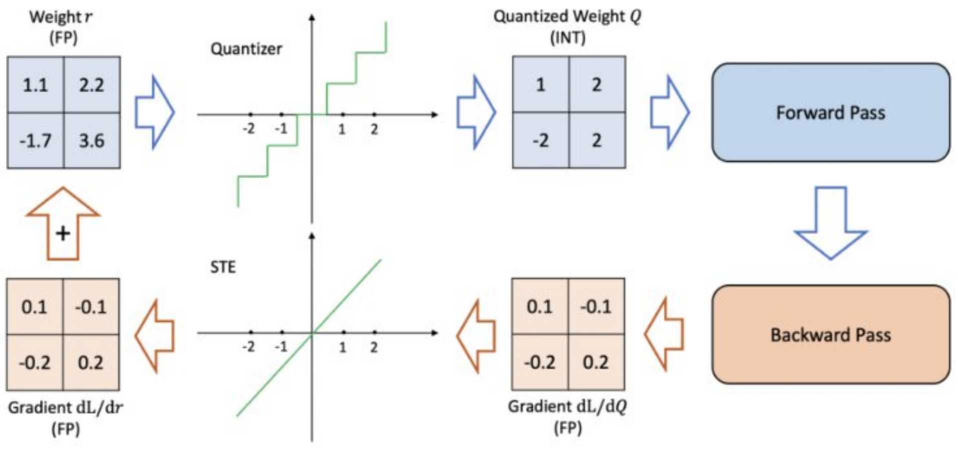
Quantification



Il est possible de rencontrer une perte en performance

A quantized network is mostly just the regular network whose weights have been casted to another format, for instance 8 bits integers. However, we need to be careful as we may degrade performance significantly. To compensate this effect, we may need to add an additional fine-tuning step in order to recover the performance we lost.

Quantification

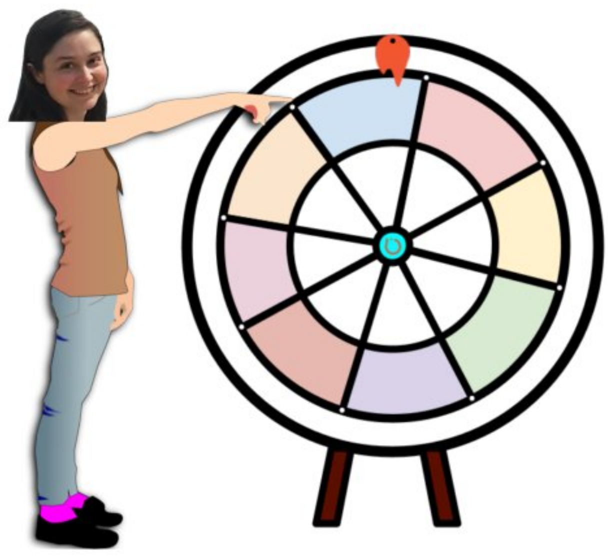

Weight r (FP)
1.1
2.2
-1.7
3.6
Quantizer
-2
-1
1
2
Quantized Weight Q (INT)
1
2
-2
2
Forward Pass
Backward Pass
+
0.1
-0.1
-0.2
0.2
STE
Gradient $\mathrm{d}L/\mathrm{d}r$ (FP)
Gradient $\mathrm{d}L/\mathrm{d}Q$ (FP)

Exemple d'entraînement post-quantisation

[19, 23]

Training with integers weights is impossible because we cannot differentiate our loss with respect to integers. To overcome this issue, we keep a copy of the floating-point weights before casting them to integers. We can navigate from the floating-point domain to the integer domain with a step function. We can do the opposite with the *Straight-Through Estimator* [23] which ignores the rounding step, and estimates floating-point gradients with an identity operator.

Pruning

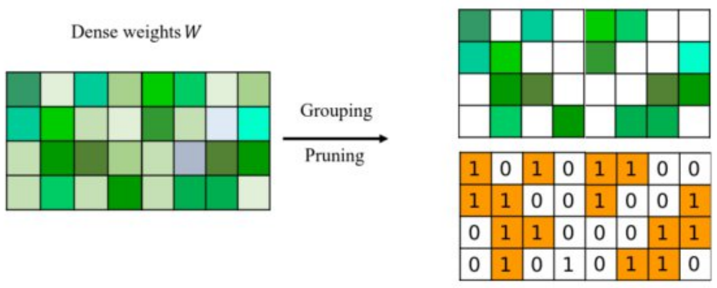


The Lottery Ticket Hypothesis. A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.

[24, 25, 26]

The Lottery Ticket Hypothesis is the idea that a randomly-initialized neural network is actually made of a set of subnetworks, some of which have similar performances than the bigger network. It means that we could delete some weights. Computation would be accelerated without much performance loss. Research [26] shows that a large sparse network is much more powerful than a smaller network.

Pruning



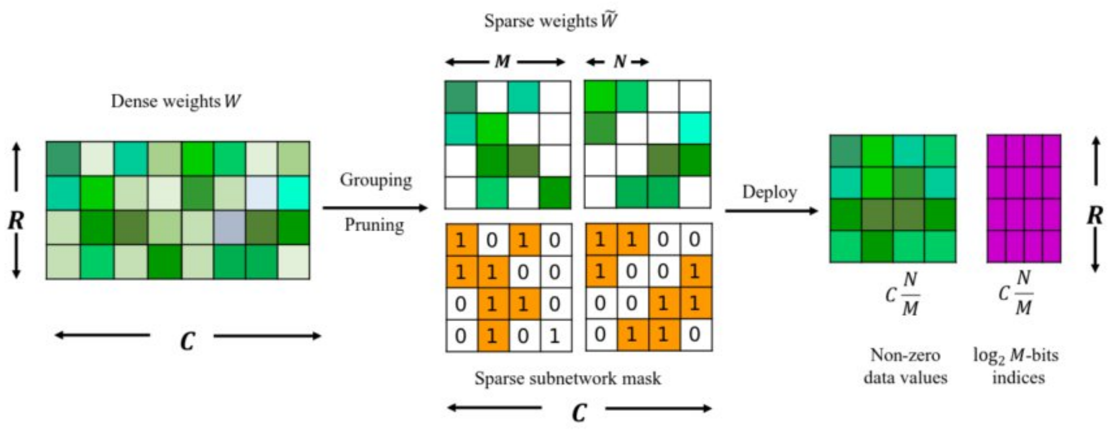
Les poids les plus petits sont mis à 0. Mais combien ?
Quel impact sur le temps de calcul ?

[20]

There are multiple ways to choose which weights to keep and which to drop. But research [25] shows that a simple magnitude-based choice is just as efficient than more sophisticated methods.

We can then validate the weights which were kept and recover the potential performance loss thanks to an additional fine-tuning step. We can use an extension of *Straight-Through Estimators* [20] from quantization methods to train the smaller network with a copy of the bigger model.

Pruning



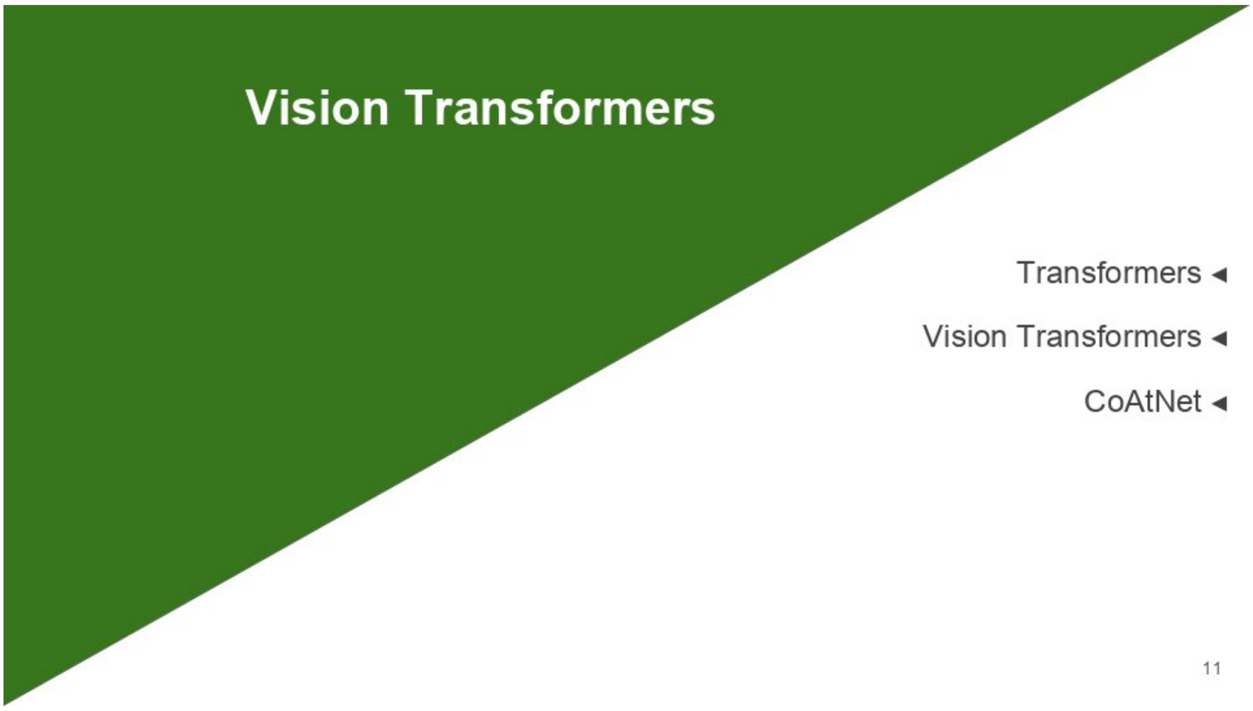
Les Tensor Cores des NVIDIA A100 supportent une dispersion 2:4.

[20, 21]

10

Dropping some weights allows (in theory) a speed-up in computation because the workload is smaller. Nonetheless it goes against modern hardware architecture which cannot adapt to this sparse layout. So in reality, the execution time is not really decreased. Some of the most recent GPU do take sparsity into account however. For instance A100 GPU from NVIDIA support 2:4 sparsity with their tensor cores. It means that every four consecutive weights, two can be dropped.

If we want to maximize the usage of the sparsity support of our GPU, we may want to swap two columns. If these swaps are performed wisely, the overall operation may remain unchanged. For instance with a Self-Attention Layer, the attention matrix is unchanged when swapping two columns of Q, provided that the two corresponding columns of K are also swapped, because the attention matrix depends on QK^T. The same note can be made with dense layers if we consider them by packs of two consecutive dense layers.



Vision Transformers

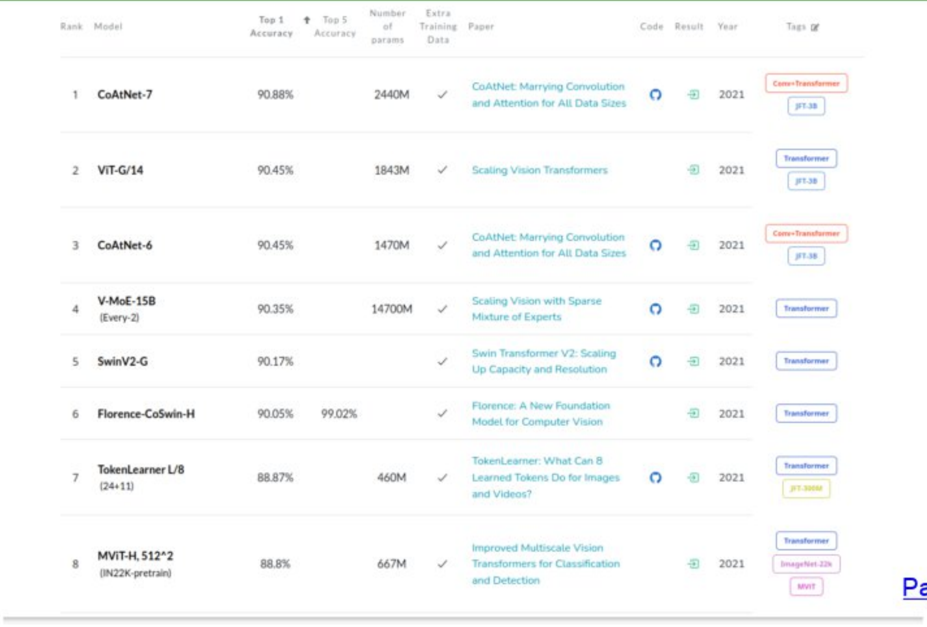
Transformers ◂

Vision Transformers ◂

CoAtNet ◂

In this section, we will present Transformers, VisionTransformers, and then CoAtNet, which will serve as a large model for the practice exercises in Model Parallelism.

Vision Transformers >> Resnet-50: 25M



Rank	Model	Top 1 Accuracy	Top 5 Accuracy	Number of params	Extra Training Data	Paper	Code	Result	Year	Tags
1	CoAtNet-7	90.88%		2440M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes	●	⇥	2021	Conv+Transformer JFT-3B
2	ViT-G/14	90.45%		1843M	✓	Scaling Vision Transformers		⇥	2021	Transformer JFT-3B
3	CoAtNet-6	90.45%		1470M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes	●	⇥	2021	Conv+Transformer JFT-3B
4	V-MoE-15B (Every-2)	90.35%		14700M	✓	Scaling Vision with Sparse Mixture of Experts	●	⇥	2021	Transformer
5	SwinV2-G	90.17%			✓	Swin Transformer V2: Scaling Up Capacity and Resolution	●	⇥	2021	Transformer
6	Florence-CoSwin-H	90.05%	99.02%		✓	Florence: A New Foundation Model for Computer Vision		⇥	2021	Transformer
7	TokenLearner L/8 (24+11)	88.87%		460M	✓	TokenLearner: What Can 8 Learned Tokens Do for Images and Videos?	●	⇥	2021	Transformer JFT-300M
8	MViT-H, 512^2 (IN22K-pretrain)	88.8%		667M	✓	Improved Multiscale Vision Transformers for Classification and Detection		⇥	2021	Transformer ImageNet-22k MViT

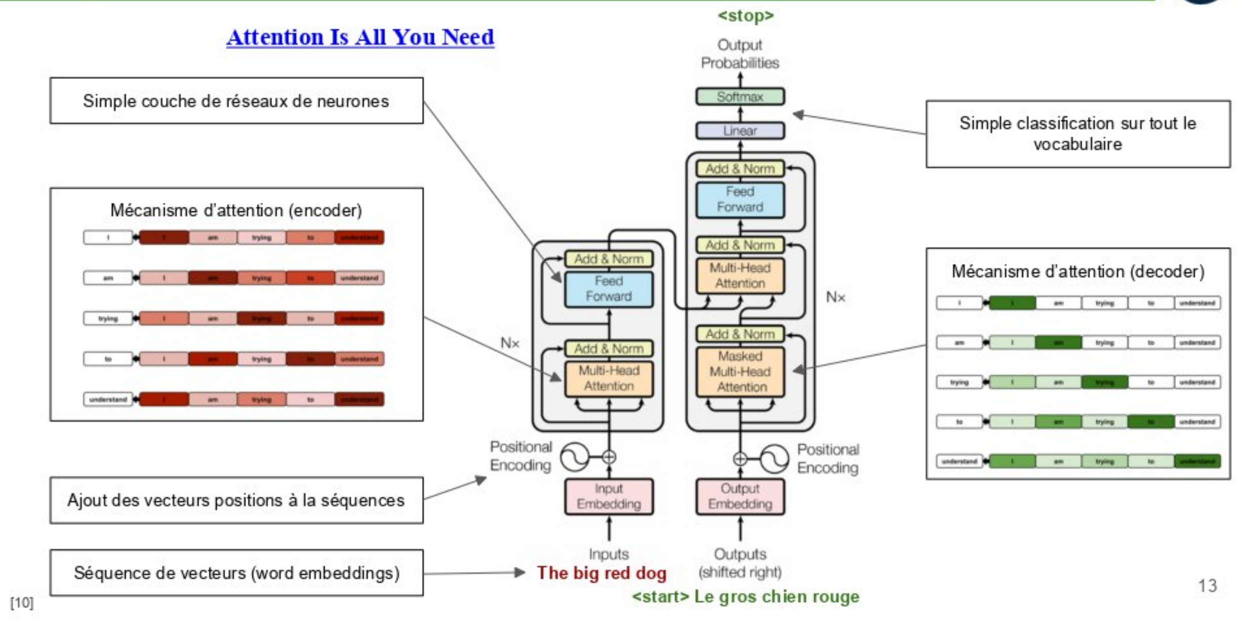
Paperwithcode

On PapersWithCode, we can see that the Vision Transformers,with more than a billion parameters, are the state-of-the-art models on ImageNet. These are large models which have the problematics we discussed previously.

Note that CoAtNet, which we will use during the practice exercises, has first place in the ranking.

Le premier Transformer

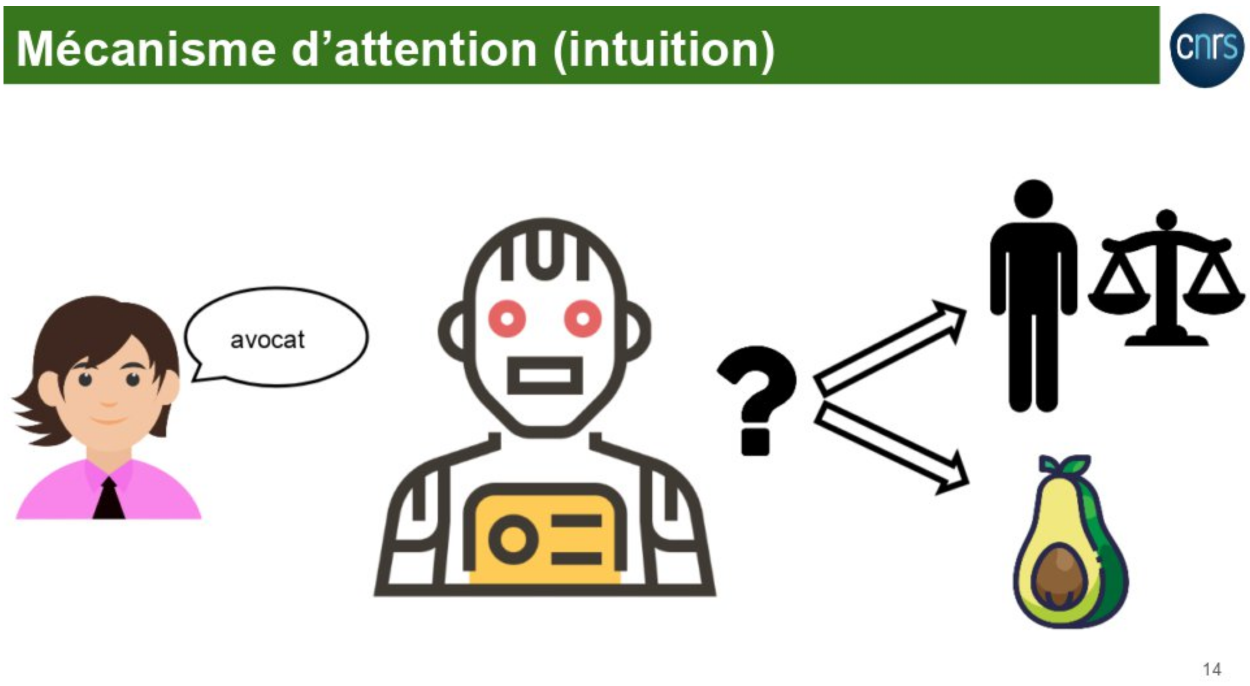
Attention Is All You Need



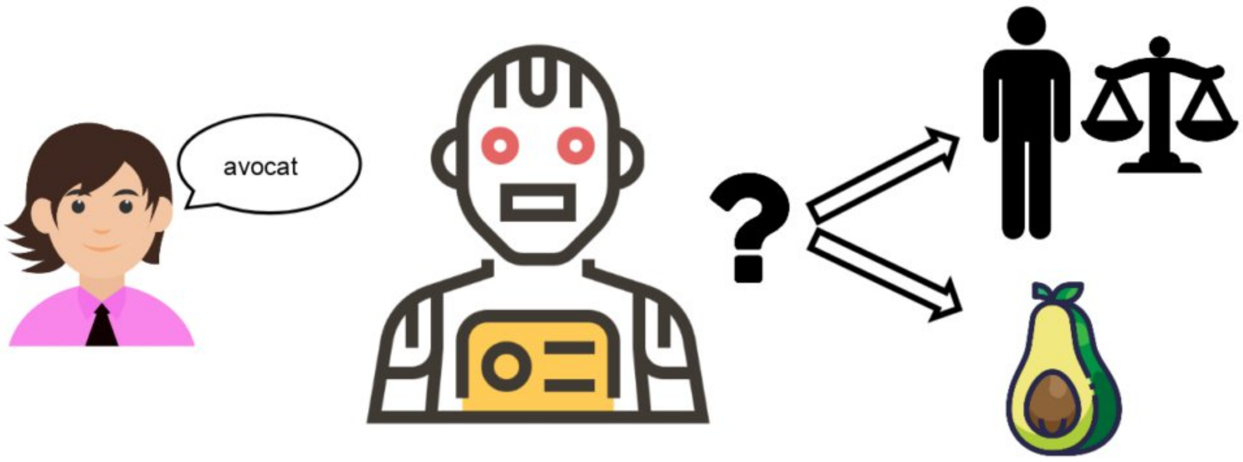
The first Transformer described in the paper, “Attention Is All You Need”, was a new type of architecture enabling text translation from English to French. This first transformer came from and improved the RNNs used for this same translation task. It contains an encoder and a decoder.

The most sophisticated RNNs used an attention mechanism to resolve the problem of sentence representation between the encoder and the decoder. The Transformer exclusively uses this attention mechanism without having recurrent layers.

Instead of encoding the sentence word by word as the RNNs do, it takes the whole sequence to encode by adding information about the position of the word in the sentence, and transforms the sequence into a latent sequence by using the Self-Attention mechanism.

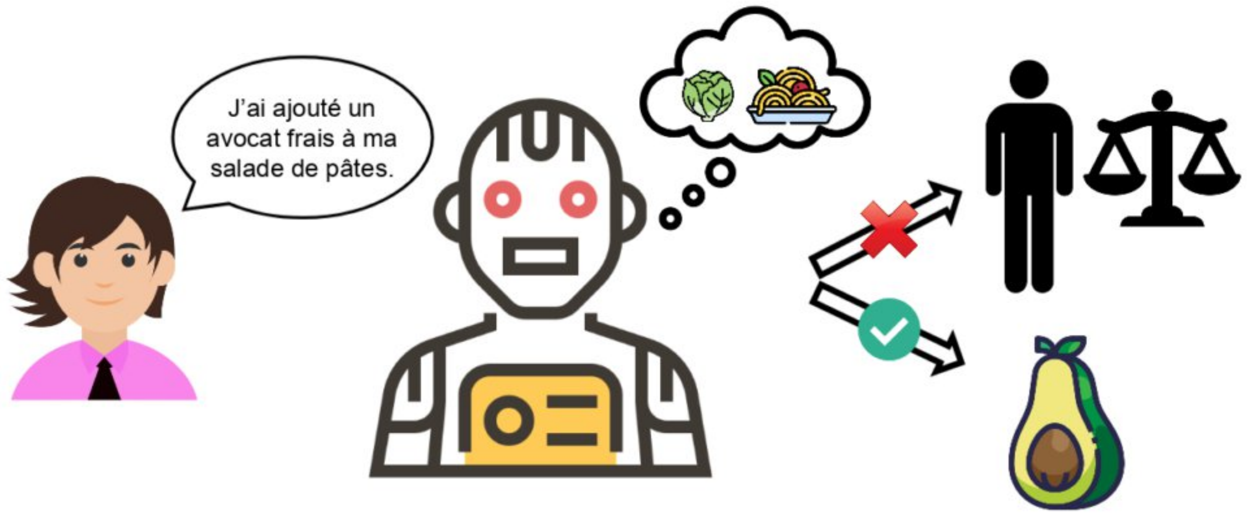


Mécanisme d’attention (intuition)



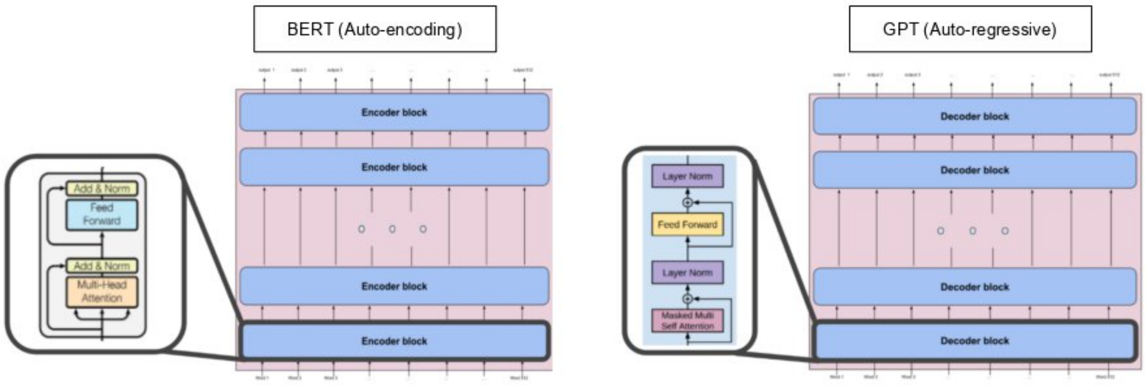
In English and most languages, a same word may have two different meanings. The corresponding vector will then carry information about both meanings. For instance in English, the word “bat” may refer to an animal or an object.

Mécanisme d'attention (intuition)



The transformer can solve this ambiguity thanks to the context. If the sentence contains words such as « flying » or « night » then we may clarify the meaning in one direction ; in the other if the sentence has words such as « trunk » or « ball ». The corresponding vector will then be enriched such that there is no longer any ambiguity.

BERT vs GPT



The first transformer was made of a stack of encoders as well as stack of decoders. The most famous transformer with such architecture is T5. Other famous transformers have adopted a different architecture.

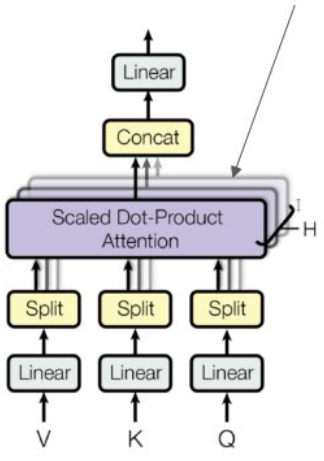
BERT is the transformer which started the « transformer mania ». It is simply made of a stack of encoders. It is very good for classification, whether it is sentence classification or token classification.

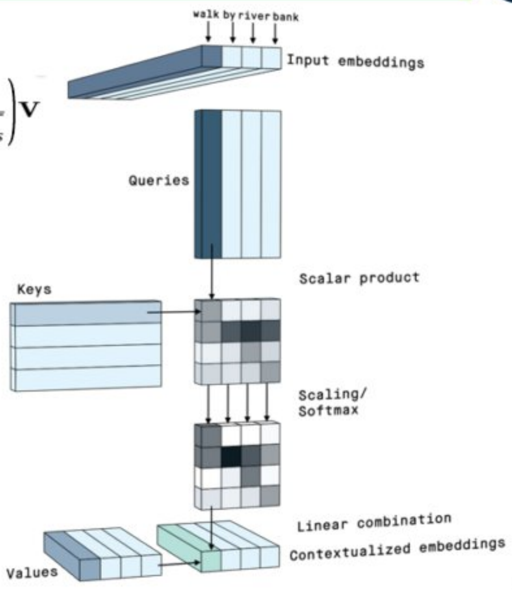
GPT-like transformers are made of a stack of decoders. It make them very good for text generation. A recent discovery with LLM (Large Language Models) is that every task can be rephrased as a generation task, including classification task.

Le mécanisme de Self-Attention

Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{QK}^{\mathsf{T}}}{\sqrt{d_{keys}}}\right)\mathbf{V}$$





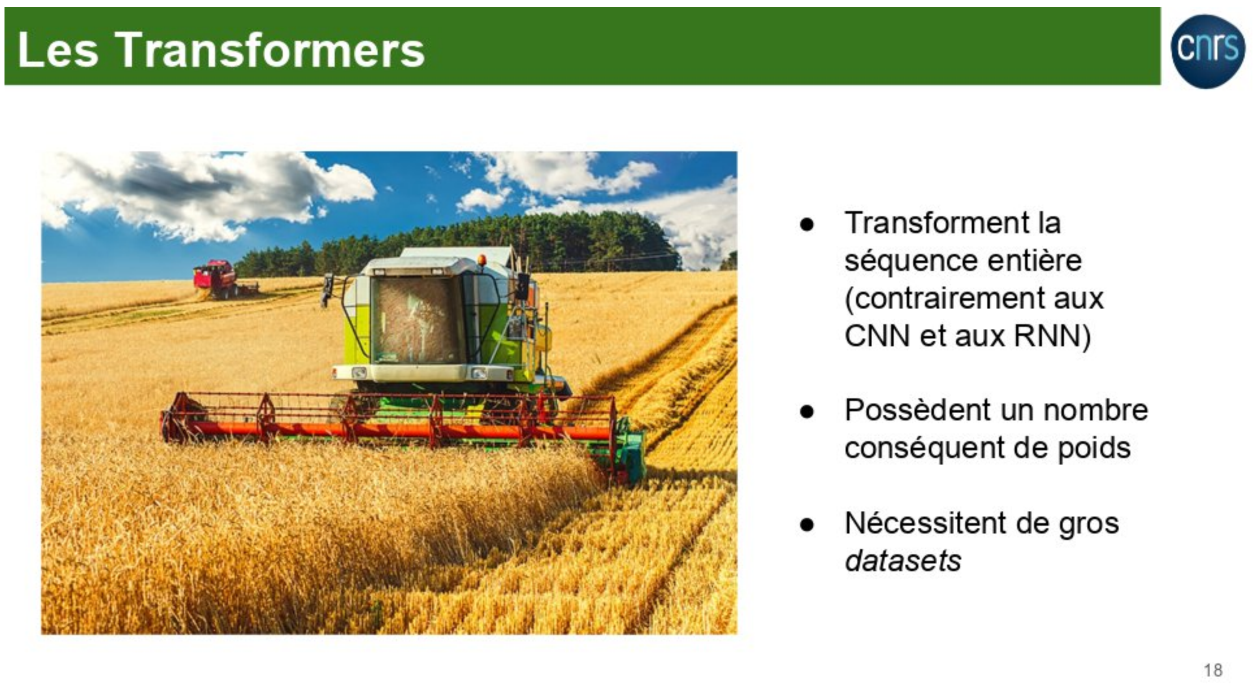
[10, 11]

The Self-Attention mechanism applies the Scaled Dot-Product Attention to a sequence of Input embeddings (which has already undergone a linear transformation).

The Query, Key and Value tensors are identical in the Self-Attention mechanism enabling the transformation of a sequence into a latent sequence which takes into account the dependency of each word compared to each other word.

QK^{T} corresponds to the dependency matrix.

Multi-head Attention divides the sequence of input embeddings to pass it into several Scaled Dot-Product heads. This allows taking into account several levels of dependency links at the same time and several representations of a same vector. This also becomes a completely parallelizable and accelerated process. Moreover, all the linear transformations allow having a total liberty over the tensor dimensions.



Les Transformers

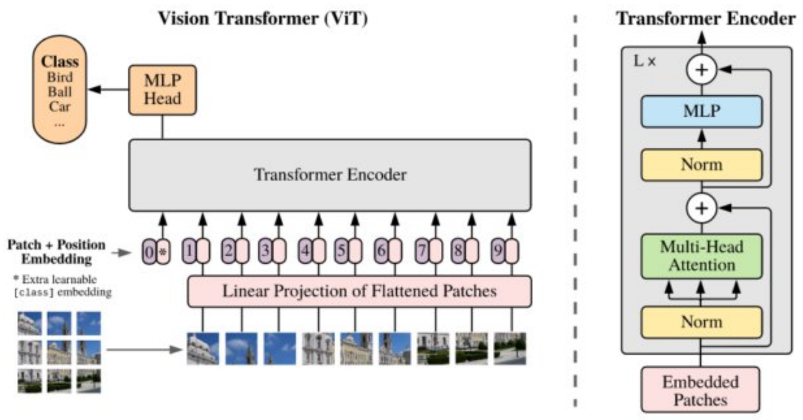
- Transforment la séquence entière (contrairement aux CNN et aux RNN)
- Possèdent un nombre conséquent de poids
- Nécessitent de gros *datasets*

18

In summary, the Transformers:

- Transform the entire sequence (unlike the CNNs and RNNs).
- Possess, in consequence, a significant number of weights.
- Require large datasets.

Vision Transformer (ViT)



- Images découpées en *patch*
- *Patchs* séquencés avec un *Position embedding*
- Ajout d'un "classification token" pour réaliser la classification finale

[12]

The first Vision Transformer applied to imagery is described in the paper, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale".

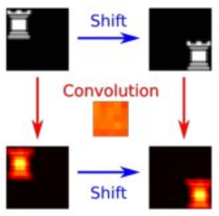
Instead of treating a word sequence, the ViT treats an image sequence. To do this, it cuts images into patches of sub-images represented in pixel vectors. After a linear transformation, these patches represent the sequence of input embeddings to which we add position information: exactly as for the first Transformer in NLP.

However, as what is described for BERT in NLP, we add a patch 0 to the sequence, a class token which will serve as the classification output after transformation.

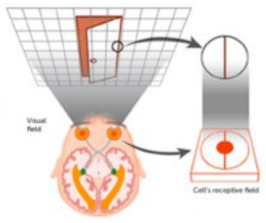
CoAtNet

"Marrying Convolution and Attention for All Data Sizes"

ConvNet
Translation Equivariance



Self-Attention Net
Global Receptive Field
Context comprehension



No patching !!

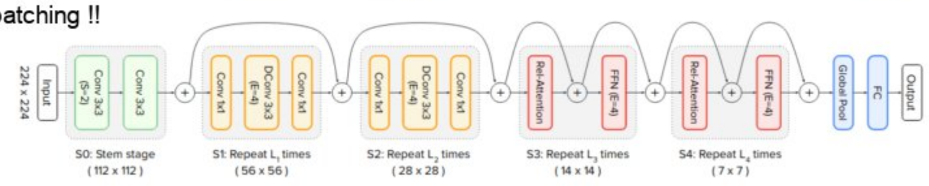

224 x 224
Input
Conv 3x3 (S=2)
Conv 3x3
Conv 1x1
DConv 3x3 (E=4)
Conv 1x1
Rel-Attention
FFN (E=4)
Global Pool
FC
Output
S0: Stem stage (112 x 112)
S1: Repeat L1 times (56 x 56)
S2: Repeat L2 times (28 x 28)
S3: Repeat L3 times (14 x 14)
S4: Repeat L4 times (7 x 7)

Figure 4: Overview of the proposed CoAtNet.

[13, 14, 15]

Finally, CoAtNet, which we will use for the practice exercises on Model Parallelism, is described in the paper, "CoAtNet: Marrying Convolution and Attention for All Data Sizes".

CoAtNet is combining a classic CNN, first, followed by a Vision Transformer. Since the model begins in CNN, there is no image patching to do as input. Therefore, we can easily replace a CNN in a code by a CoAtNet. The transition between CNN and Vision Transformer is done by patching the output of the CNN part.

This has the benefit of taking advantage of the CNN translational equivariance and the overall understanding of the image context of the Vision Transformers, at the same time.

CoAtNet - Résultats

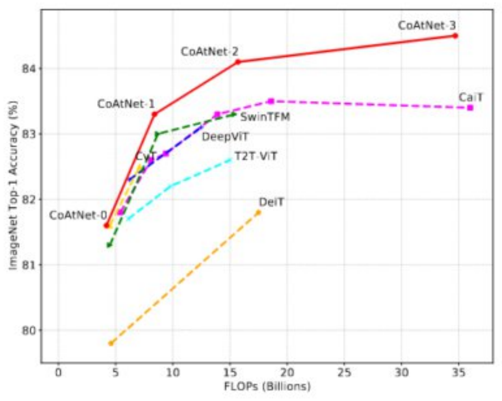


Figure 2: Accuracy-to-FLOPs scaling curve under ImageNet-1K only setting at 224x224.

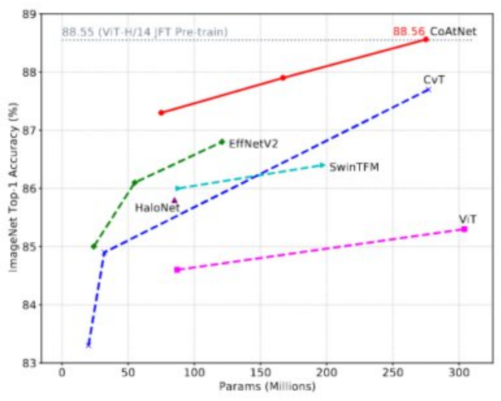
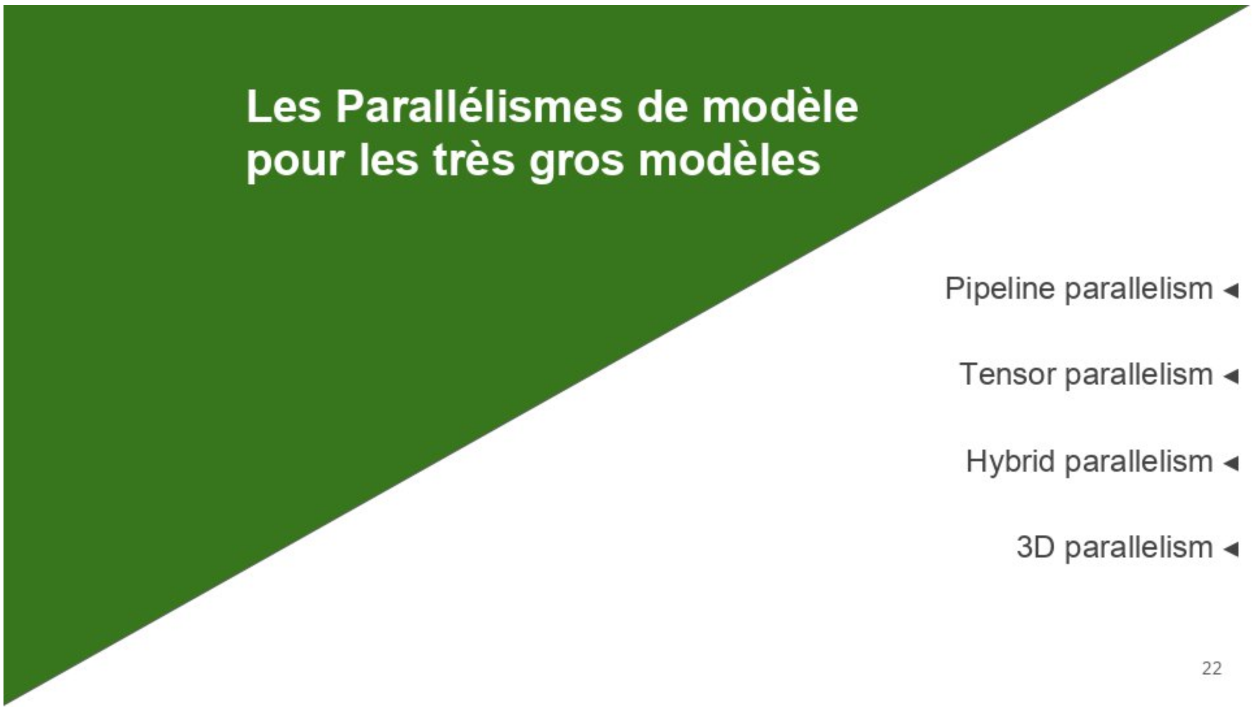


Figure 3: Accuracy-to-Params scaling curve under ImageNet-21K ⇒ ImageNet-1K setting.

[13]

In comparison, the results of the 7 levels of CoAtNet models surpass all the current Vision Transformers and current CNNs which have an equal number of parameters or equal computing cost.

It is interesting to note that there are extensions for increasing the accuracy score of Vision Transformers on ImageNet-1k.For example, ImageNet-21k is 100 times larger for an augmented training.



Les Parallélismes de modèle pour les très gros modèles

Pipeline parallelism ◂

Tensor parallelism ◂

Hybrid parallelism ◂

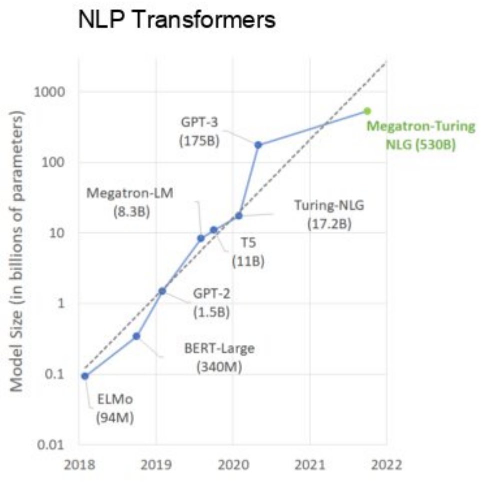
3D parallelism ◂

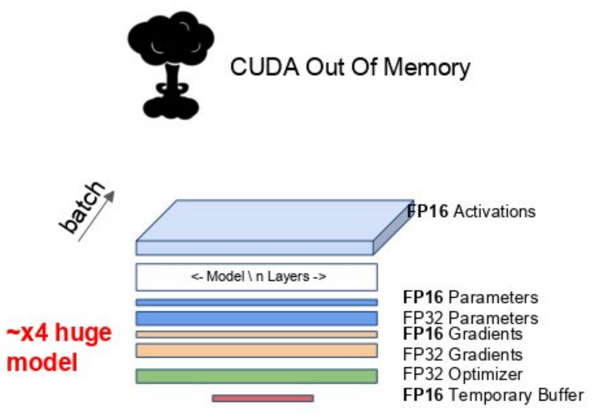
The goal of this section is to describe the different types of Model Parallelism possible and their advantages.

Due to its complexity, Model Parallelism is only used for very large models.

Each parallelism must be seen as functioning along a specific axis. We will be describing Pipeline Parallelism, Hybrid Parallelism, Tensor Parallelism and 3D Parallelism.

Énormes modèles > 1 Milliard de paramètres

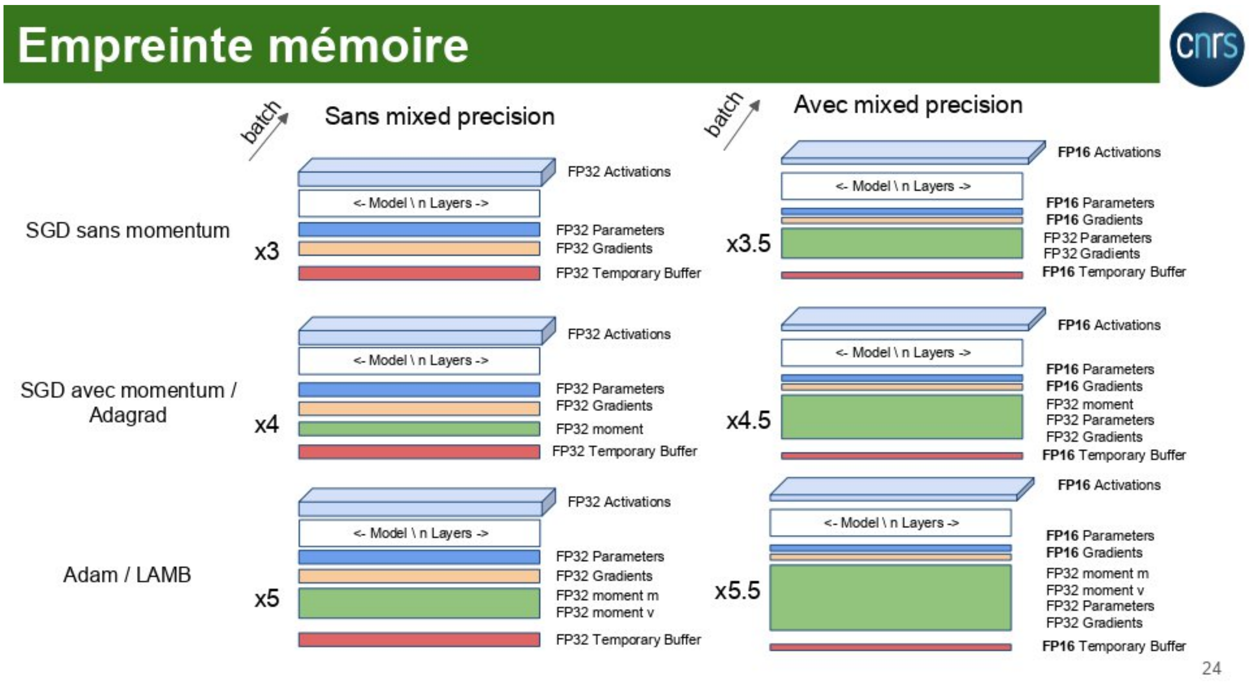




[1]

Since the arrival of NLP Transformers and Vision Transformers in Computer Vision,neural network models have entered into another dimension. We will speak about large models of more than a billion parameters which bring new problematics, even when using a supercomputer, and are completely beyond the capacity of a classical computer.

As we have seen previously, the memory footprint directly linked to a model was negligible for models of a standard size. For large models, it becomes critical.

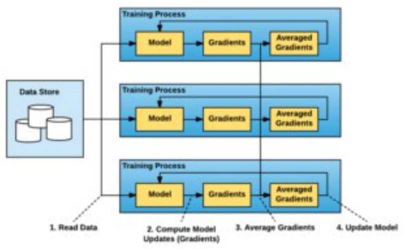


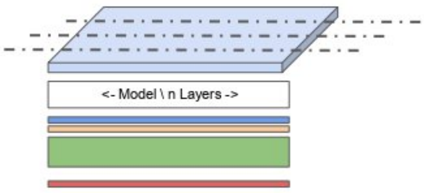
Empreinte mémoire

The memory footprint linked to the model obviously increases in relation to the model size but also when we use Mixed Precision, the optimizer which is used and the number of associated momentums.

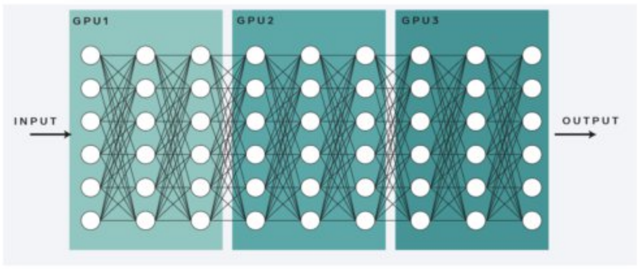
Les différents parallélismes

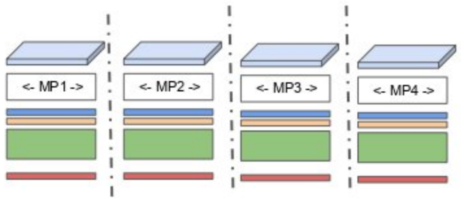
Data Parallelism
- Meilleur Throughput
- Seule l'empreinte mémoire des activations est distribuée
- Multi Processing





Pipeline Model Parallelism
- Empreinte mémoire distribuée
- Mono ou multi-processing



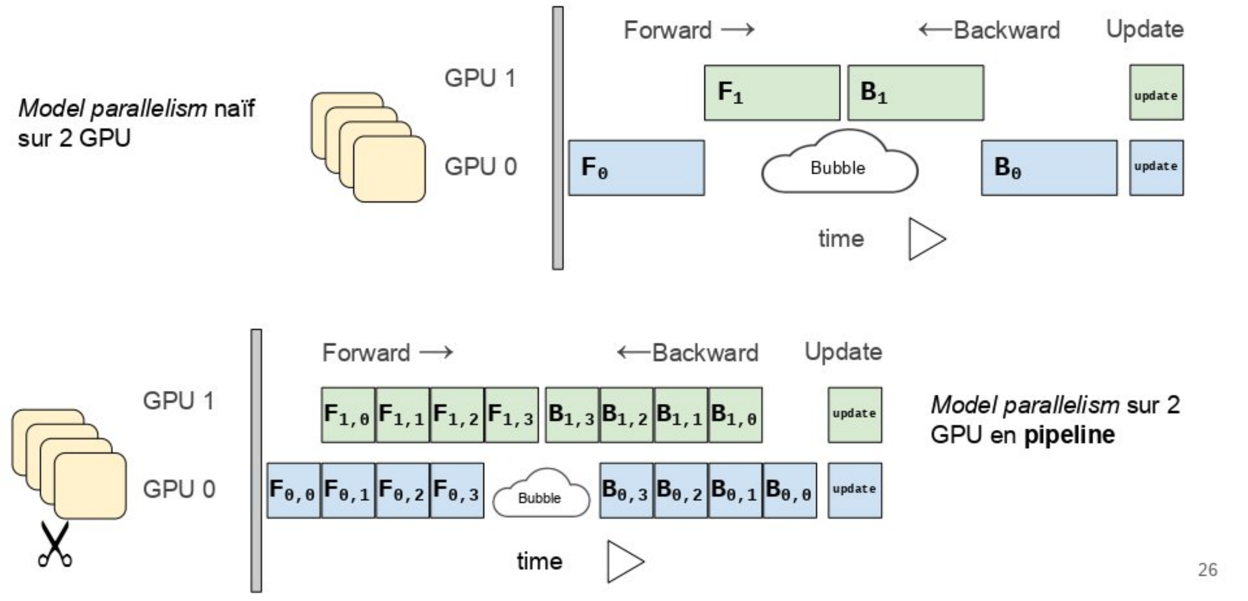


[2, 3]

Data parallelism, which we previously discussed, is the best solution for the throughput and for ease of implementation. However, only the memory footprint of the activations is distributed and this is a problem for large models. Data parallelism distributes the batches through the GPUs.

For large models, therefore, we are adding the principle of model parallelism (notably, pipeline parallelism) which consists of distributing the model using the GPUs. In this way, the complete memory footprint is distributed. Moreover, the batch size is not augmented which is an advantage when we use a very large number of GPUs for the distribution when compared to Distributed Data Parallelism (DDP).

Pipeline Parallelism


Model parallelism naïf sur 2 GPU
GPU 1
GPU 0
Forward →
←Backward
Update
F_1
B_1
update
F_0
Bubble
B_0
update
time
GPU 1
GPU 0
Forward →
←Backward
Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$
update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$
Bubble
$B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$
update
time
Model parallelism sur 2 GPU en **pipeline**

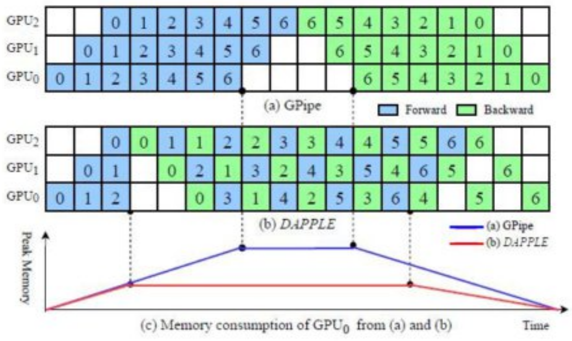
The naive way to implement model parallelism is to split the model between 2 layers and to use the GPUs sequentially during the training loop. However, because there isn’t any parallelization, there will not be any acceleration. Only distribution of the memory footprint will be done.

Pipeline parallelism, by splitting the model in the same way between 2 layers, enables in addition, the acceleration of the process. For this, it subdivides the batch into micro-batches. The training iteration is complete when all the micro-batches are passed in forward and in backward propagation.

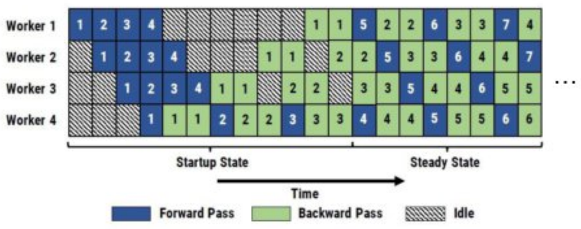
Thus, the bubble corresponding to GPU inactivity time is largely reduced. In practice, pipeline parallelism is used when we split the model going in the direction of the layers.

Pipeline Parallelism

Synchronous pipeline :
GPipe, DAPPLE
- - Throughput
- + Memory consumption
- + Convergence



Asynchronous pipeline :
PipeDream, PipeMare
- + Throughput
- - Memory consumption
- - Convergence

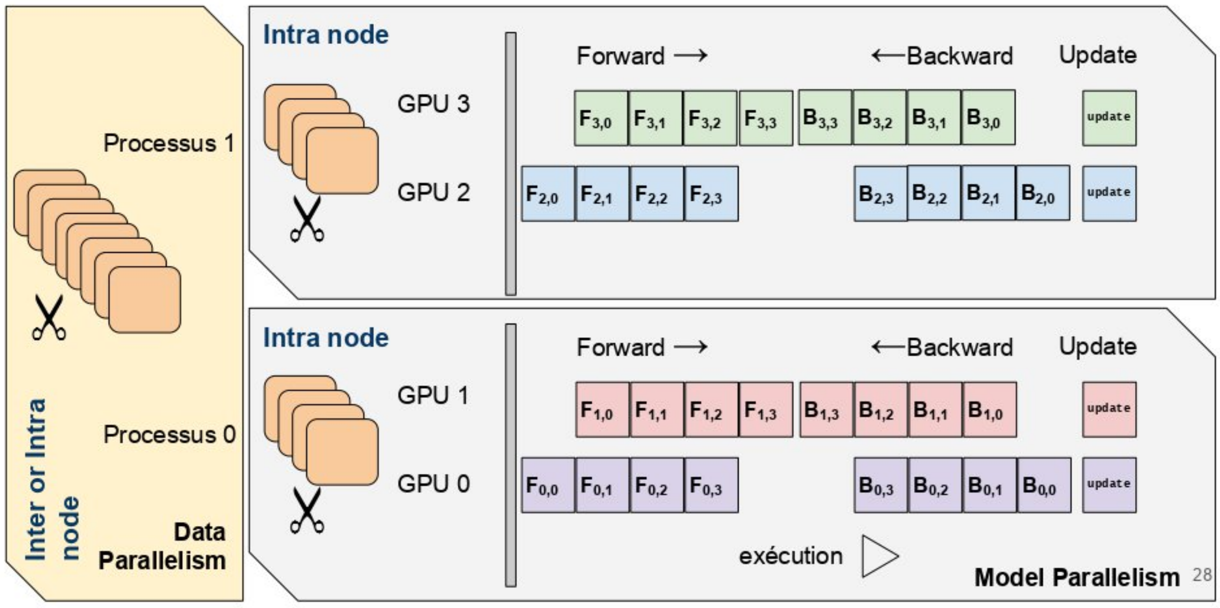


[4, 5]

The possible optimizations of pipeline parallelism are organized according to the 2 following methods:

- **Synchronous** pipeline parallelism corresponds to the pipeline parallelism described in the previous slide. One possible optimization is to change the execution order of the micro-batches in order to reduce the memory footprint. When the backward propagation is executed, the memory footprint of the activations of the micro-batch can be freed.
- **Asynchronous** pipeline parallelism does not permit waiting for the end of a training iteration before beginning a new one. This results in a loss calculation, sometimes not with the N-1 version of the model but with its N-2 version. Thus, the descent gradient process loses in training quality but the parallelism is maximally accelerated and there is no longer a bubble. The acceleration is comparable with that of data parallelism. However, the asynchronous pipeline is rarely used.

Hybrid Parallelism


Processus 1
Processus 0
Inter or Intra node
Data Parallelism
Intra node
GPU 3
GPU 2
Intra node
GPU 1
GPU 0
Forward ⟶
⟵Backward
Update
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ update
exécution
Model Parallelism
28

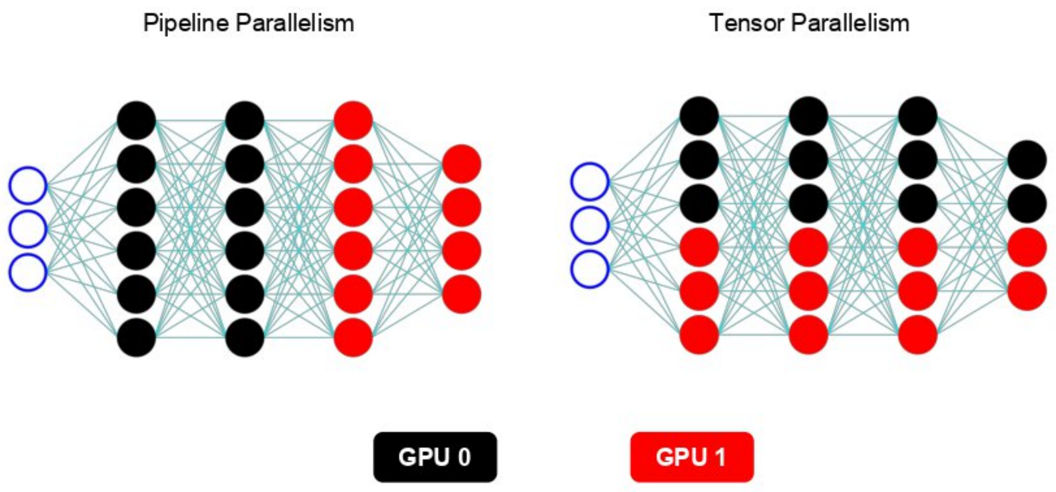
Hybrid parallelism refers to using data parallelism and pipeline parallelism at the same time.

Data parallelism is more effective in accelerating the training. However, a certain degree of pipeline parallelism is necessary for large models.

A subtle mix of both of these enables the best optimization of the process.

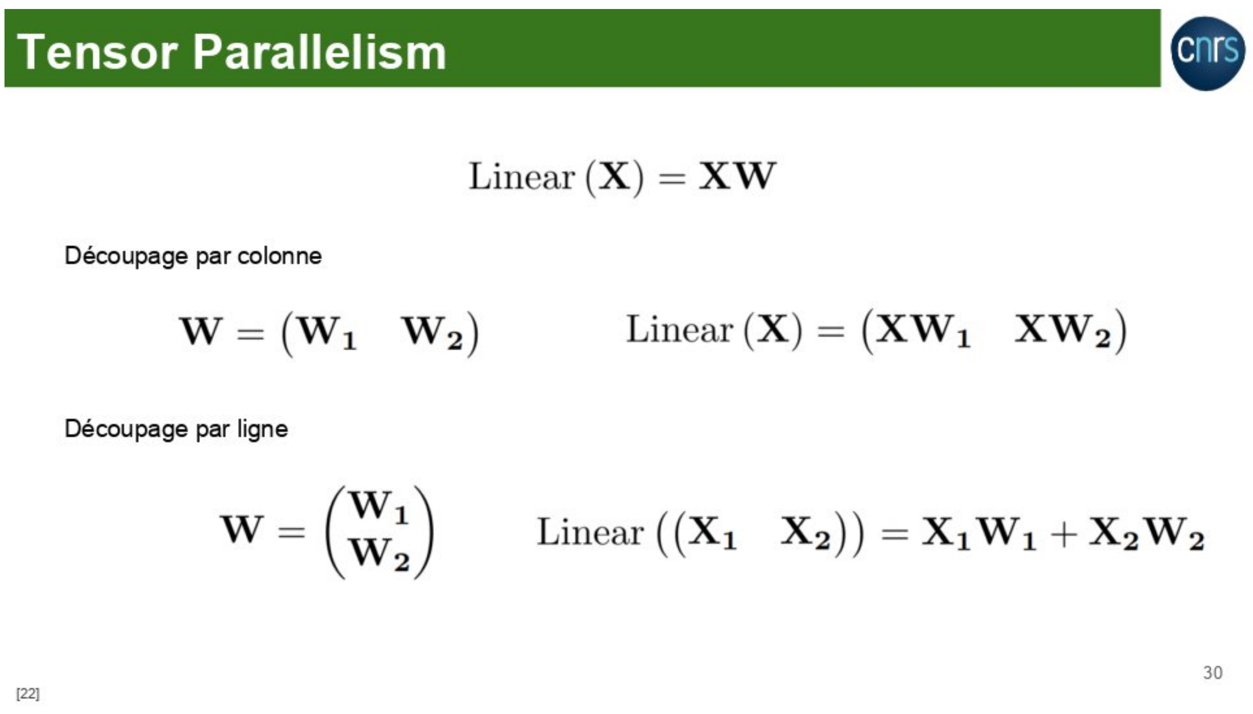
In addition, if we use a large number of GPUs, data parallelism could lead to batch sizes which are too large. Using pipeline parallelism allows moderating the batch size.

Two paradigms for model parallelism



There are two ways of doing model parallelism. The aforementioned *Pipeline* method splits the model by partitioning layers across all GPUs.

The *Tensor* method splits each layer separately into multiple GPUs.



Tensor Parallelism

$$\text{Linear}(\mathbf{X}) = \mathbf{XW}$$

Découpage par colonne

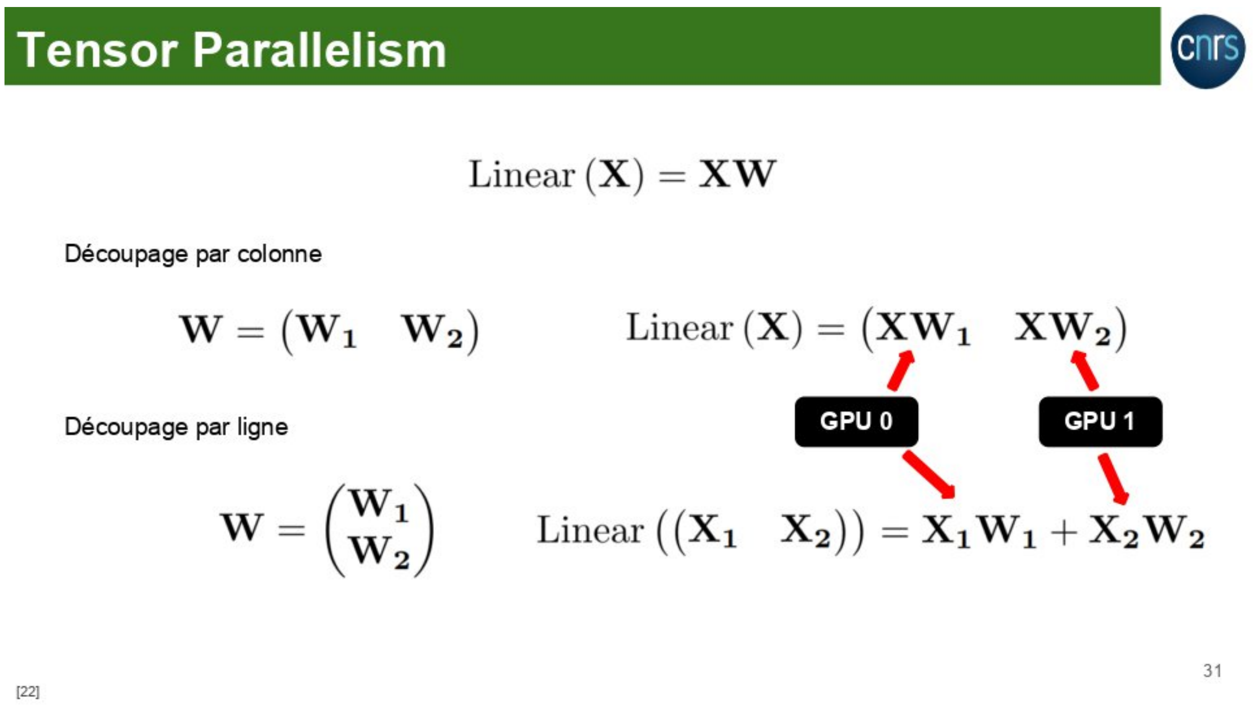
$$\mathbf{W} = \begin{pmatrix} \mathbf{W_1} & \mathbf{W_2} \end{pmatrix} \qquad \text{Linear}(\mathbf{X}) = \begin{pmatrix} \mathbf{XW_1} & \mathbf{XW_2} \end{pmatrix}$$

Découpage par ligne

$$\mathbf{W} = \begin{pmatrix} \mathbf{W_1} \\ \mathbf{W_2} \end{pmatrix} \qquad \text{Linear}\left(\begin{pmatrix} \mathbf{X_1} & \mathbf{X_2} \end{pmatrix}\right) = \mathbf{X_1 W_1} + \mathbf{X_2 W_2}$$

[22]

With Tensor Parallelism, we can split a dense layer along two different directions: along columns or along rows.



Tensor Parallelism

$$\mathrm{Linear}(\mathbf{X}) = \mathbf{XW}$$

Découpage par colonne

$$\mathbf{W} = \begin{pmatrix}\mathbf{W_1} & \mathbf{W_2}\end{pmatrix} \qquad \mathrm{Linear}(\mathbf{X}) = \begin{pmatrix}\mathbf{XW_1} & \mathbf{XW_2}\end{pmatrix}$$

GPU 0

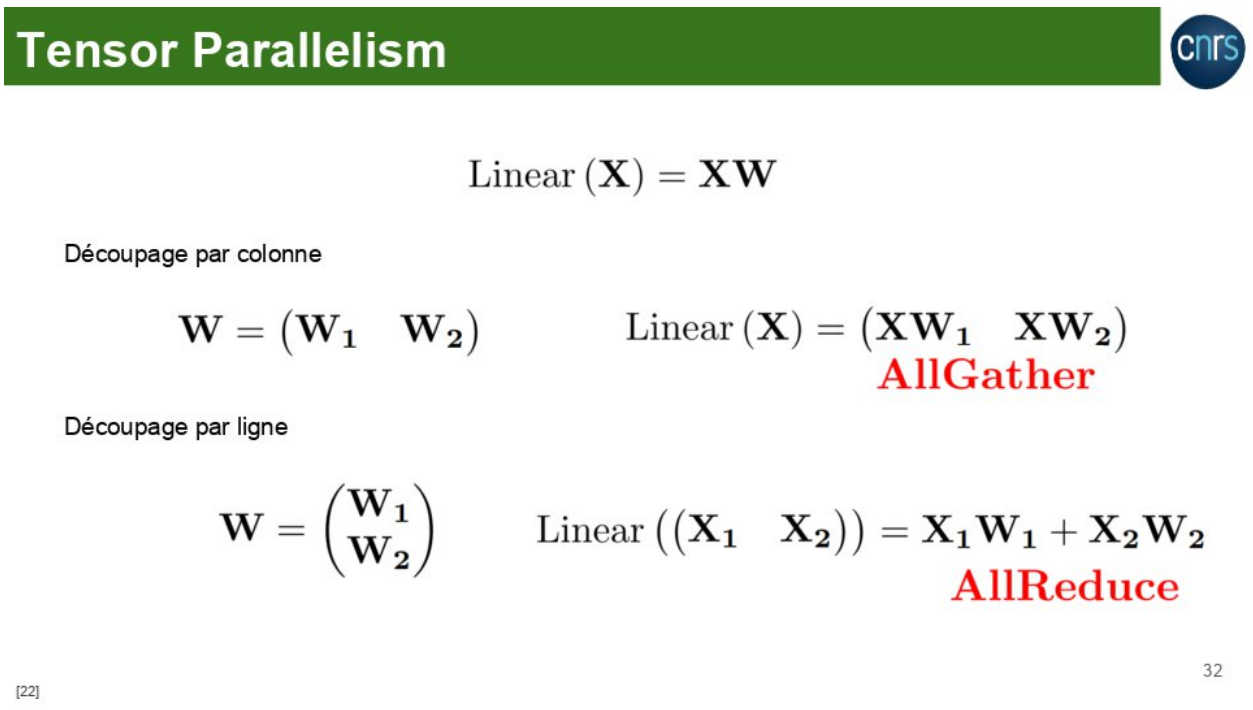
GPU 1

Découpage par ligne

$$\mathbf{W} = \begin{pmatrix}\mathbf{W_1} \\ \mathbf{W_2}\end{pmatrix} \qquad \mathrm{Linear}\left(\begin{pmatrix}\mathbf{X_1} & \mathbf{X_2}\end{pmatrix}\right) = \mathbf{X_1W_1} + \mathbf{X_2W_2}$$

[22]

This way of splitting the weight tensor highlights multiple subcomputation which can be performed simultaneously on multiple GPUs. Since the output of the layer is the input of the next layer, this next layer will need the full vector. This implies a communication between both GPUs to pool their result.



Tensor Parallelism

$$\text{Linear}(\mathbf{X}) = \mathbf{XW}$$

Découpage par colonne

$$\mathbf{W} = \begin{pmatrix}\mathbf{W_1} & \mathbf{W_2}\end{pmatrix} \qquad \text{Linear}(\mathbf{X}) = \begin{pmatrix}\mathbf{XW_1} & \mathbf{XW_2}\end{pmatrix}$$

AllGather

Découpage par ligne

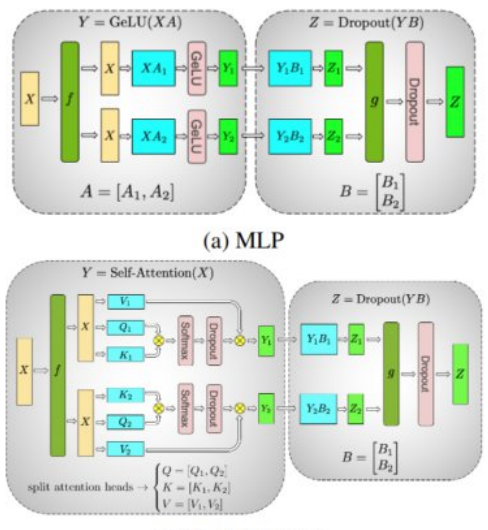
$$\mathbf{W} = \begin{pmatrix}\mathbf{W_1} \\ \mathbf{W_2}\end{pmatrix} \qquad \text{Linear}\left(\begin{pmatrix}\mathbf{X_1} & \mathbf{X_2}\end{pmatrix}\right) = \mathbf{X_1W_1} + \mathbf{X_2W_2}$$

AllReduce

[22]

With the column-wise split, we need to perform an AllGather communication to concatenate outputs of all GPUs. With the row-wise split, we need to perform an AllReduce communication to do the sum of all results. It is necessary to do these communications steps at every layer.

Tensor Parallelism



$Y = \mathrm{GeLU}(XA)$
$Z = \mathrm{Dropout}(YB)$
X
f
XA_1
XA_2
GeLU
Y_1
Y_2
Y_1B_1
Y_2B_2
Z_1
Z_2
g
Dropout
Z
$A = [A_1, A_2]$
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
$Y = \mathrm{Self\text{-}Attention}(X)$
V_1
Q_1
K_1
K_2
Q_2
V_2
Softmax
split attention heads → $\begin{cases} Q = [Q_1, Q_2] \\ K = [K_1, K_2] \\ V = [V_1, V_2] \end{cases}$

(a) MLP

(b) Self-Attention

Par défaut, le tensor parallelism exige des synchronisations à **chaque** couche.

En alternant coupure en lignes et coupure en colonnes, on peut se permettre de ne communiquer qu'une fois toutes les **deux** couches denses.

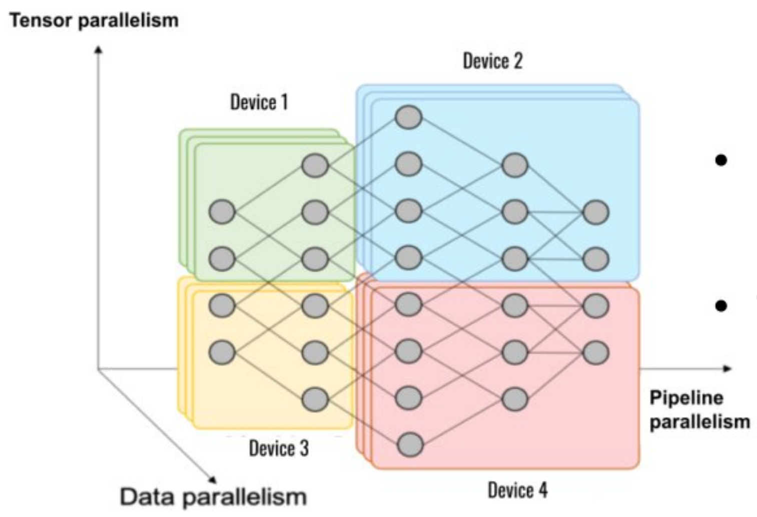
[22]

33

Since with row-wise splitting, each GPU only requires a slice of the input vector, and since column-wise splitting gives a sliced output, then putting a row-wise split layer after a column-wise split layer allows us to only do the communication step after the second layer. It removes entirely the need to communicate after the first layer. Therefore the volume of communication required by Tensor Parallelism is halved.

The same reasoning can be done with Self-Attention mechanism, which is actually even easier since more fundamentally parallel.

3D Parallelism



- **Data Parallelism**
  - Simple à implémenter
  - Meilleure performance
  - Augmente la taille du batch (problème de convergence)
- **Pipeline Parallelism**
  - Effort d'implémentation.
  - Équilibre entre mémoire, performance et convergence.
- **Tensor Parallelism**
  - Effort important d'implémentation
  - Bonne accélération des calculs
  - **Bande passante très sollicitée** (implémentation Intra-nœud)

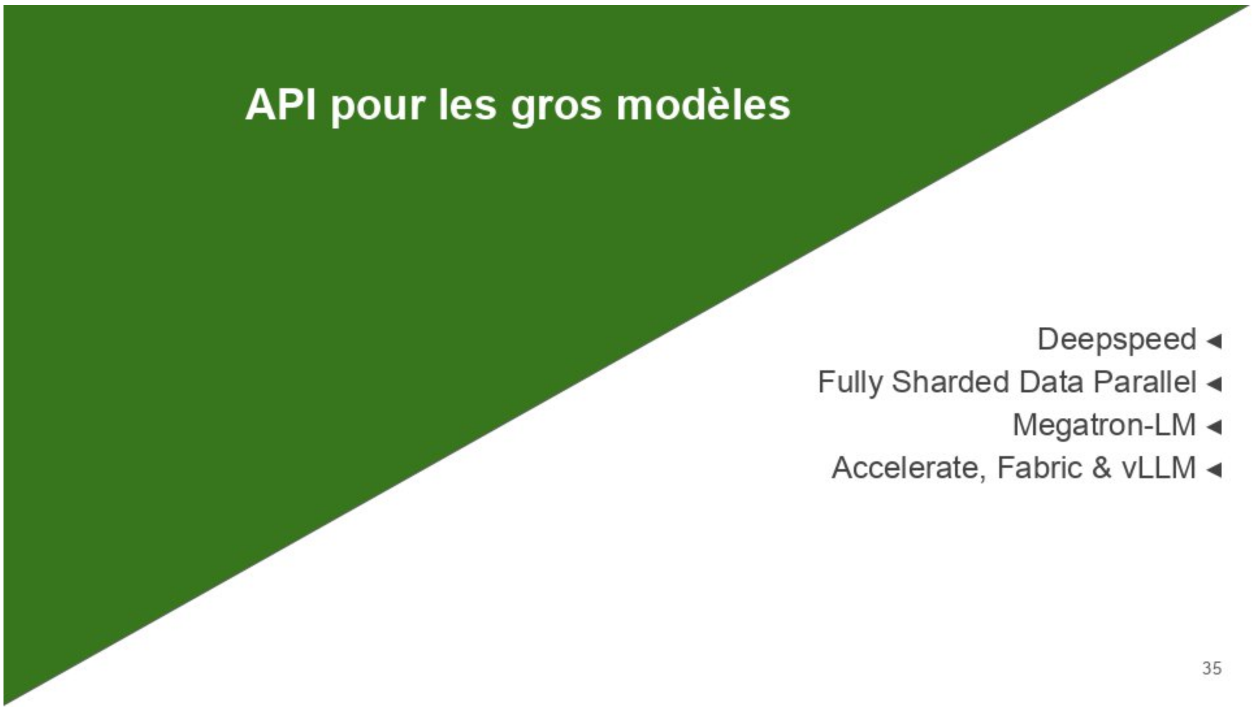
3D parallelism distributes along 3 axes :

- Data parallelism along the batch axis
- Pipeline parallelism along the layer axis
- Tensor parallelism along the axis of the layer nodes or inside the tensor computations specific to each layer

We have seen that data parallelism is simple to implement because today, all the Deep Learning frameworks integrate a data parallelism solution which is the best solution for accelerating the code. However, there is a trend towards larger and larger batches.

We have also seen that pipeline parallelism enables both accelerating the code and distributing the memory footprint linked to the model. It requires a certain effort to be implemented but it can be accessed via a group of libraries dedicated to model parallelism.

Tensor parallelism, only accessible with an important implementation effort or through very specialized libraries(Megatron-LM), enables a net computation acceleration and sharing of the memory footprint. However, it generates a greater flow in volume of communications between the GPUs, so it will only be used between GPUs of the same compute node.



API pour les gros modèles

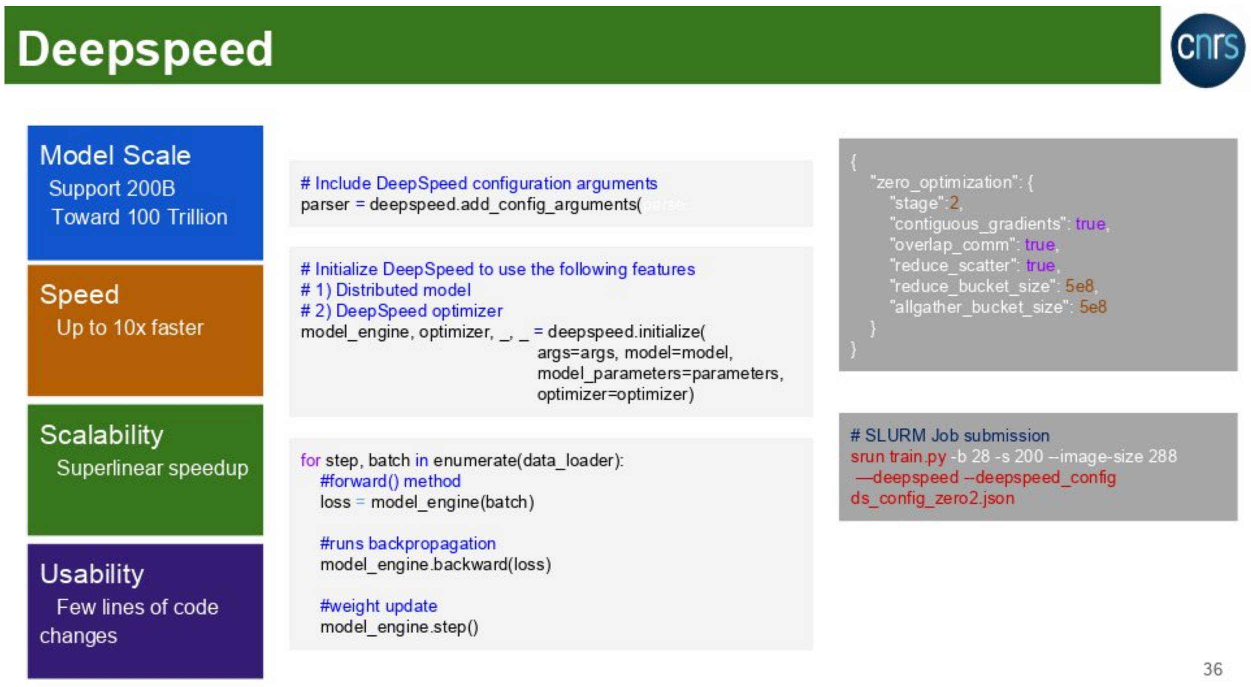
Deepspeed ◂
Fully Sharded Data Parallel ◂
Megatron-LM ◂
Accelerate, Fabric & vLLM ◂

35

The implementation of model parallelism and the training of large models, or acceleration on a very large scale as we have just described, is accessible through certain dedicated libraries.

This section describes the following dedicated libraries:

- *Deepspeed* from Microsoft
- *Fully Sharded Data Parallel* from FairScale
- *Megatron-LM* from NVIDIA
- *Accelerate*, *Lightning Fabric* & *vLLM*



Deepspeed

Model Scale
Support 200B
Toward 100 Trillion

Speed
Up to 10x faster

Scalability
Superlinear speedup

Usability
Few lines of code changes

```
# Include DeepSpeed configuration arguments
parser = deepspeed.add_config_arguments(parser)
```

```
# Initialize DeepSpeed to use the following features
# 1) Distributed model
# 2) DeepSpeed optimizer
model_engine, optimizer, _, _ = deepspeed.initialize(
                                args=args, model=model,
                                model_parameters=parameters,
                                optimizer=optimizer)
```

```
for step, batch in enumerate(data_loader):
    #forward() method
    loss = model_engine(batch)

    #runs backpropagation
    model_engine.backward(loss)

    #weight update
    model_engine.step()
```

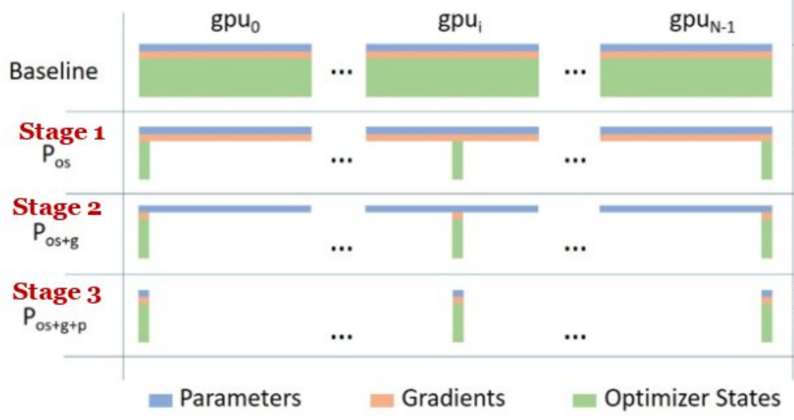
```
{
  "zero_optimization": {
    "stage":2,
    "contiguous_gradients": true,
    "overlap_comm": true,
    "reduce_scatter": true,
    "reduce_bucket_size": 5e8,
    "allgather_bucket_size": 5e8
  }
}
```

```
# SLURM Job submission
srun train.py -b 28 -s 200 --image-size 288
 —deepspeed --deepspeed_config
ds_config_zero2.json
```

Microsoft’s Deepspeed, dedicated to accelerating large models and very large models, offers a group of techniques in PyTorch.

Deepspeed is fairly easy to integrate in a PyTorch code and contains a multitude of acceleration and parallelism functions.

ZeRO — Optimisation du data parallelism



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
Stage 1 P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
Stage 2 P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
Stage 3 P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

[6]

The most interesting functionality of Deepspeed is the Zero Redundancy Optimizer (ZeRO). ZeRO is an optimization of data parallelism for large models and it enables reducing the memory footprint linked to the model by using data parallelism. Data parallelism completely copies the variables linked to the model (weights, gradients, the optimizer historic) into each GPU.
ZeRO enables sharing the memory footprint linked to the model. Each GPU keeps a different portion of the model footprint. When a GPU needs a part of the model which it doesn't have, the GPU which holds this portion communicates it to the GPU in question. In this way, ZeRO functions exactly like data parallelism by distributing along the batch axis while at the same time distributing the stored information linked to the model.

ZeRO has 3 stages:

- Stage 1 - distribution of the optimizer part, equivalent to data parallelism in terms of inter-GPU communication.
- Stage 2 - distribution of the optimizer and gradient parts, equivalent to data parallelism in terms of inter-GPU communication.
- Stage 3 - distribution of the optimizer, gradient and model weights parts, which increases the cost up to 50% in terms of communication.

There are also optimizations of ZeRO: ZeRO Offload and ZeRO Infinity. These use CPU memory to distribute even larger models which have more than 1000 billion parameters, corresponding to models of the future.

Fused optimizers

Implémentations présentent dans *APEX*

Fusionne des ***kernels* GPU** pour économiser les opérations de lecture / écriture de mémoire

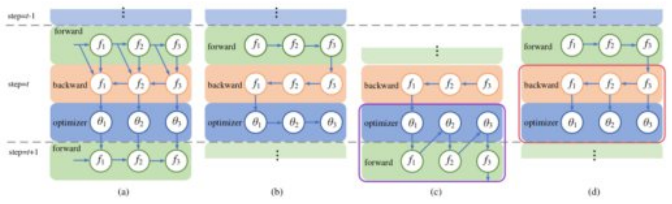
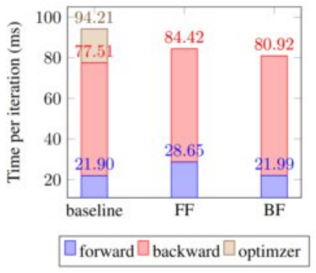


Figure 1: (a) Data dependency graph. (b) Baseline method. (c) `Forward-fusion`. (d) `Backward-fusion`. θ_i represents the trainable parameters in the layer f_i.



But : accélère l'étape des optimiseurs sur GPU.

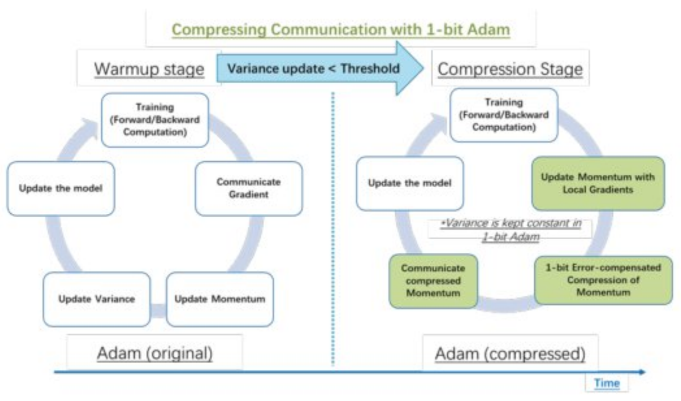
38

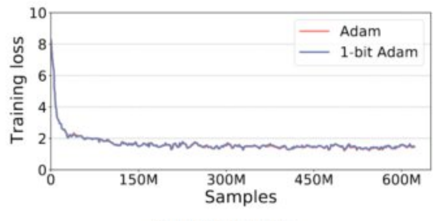
[7]

Deepspeed also integrates the « fused » optimizers offered by APEX.

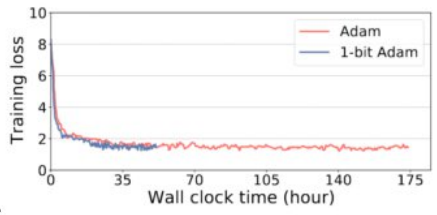
It fuses the **GPU kernels** in order to economize the initialization operations and the memory reading/writing. This causes a slight acceleration of the optimizers within the GPUs.

1-bit optimizers


Compressing Communication with 1-bit Adam
Warmup stage
Variance update < Threshold
Compression Stage
Training (Forward/Backward Computation)
Update the model
Communicate Gradient
Update Variance
Update Momentum
Training (Forward/Backward Computation)
Update the model
Update Momentum with Local Gradients
*Variance is kept constant in 1-bit Adam
Communicate compressed Momentum
1-bit Error-compensated Compression of Momentum
Adam (original)
Adam (compressed)
Time



(a) Sample-wise



(b) Time-wise

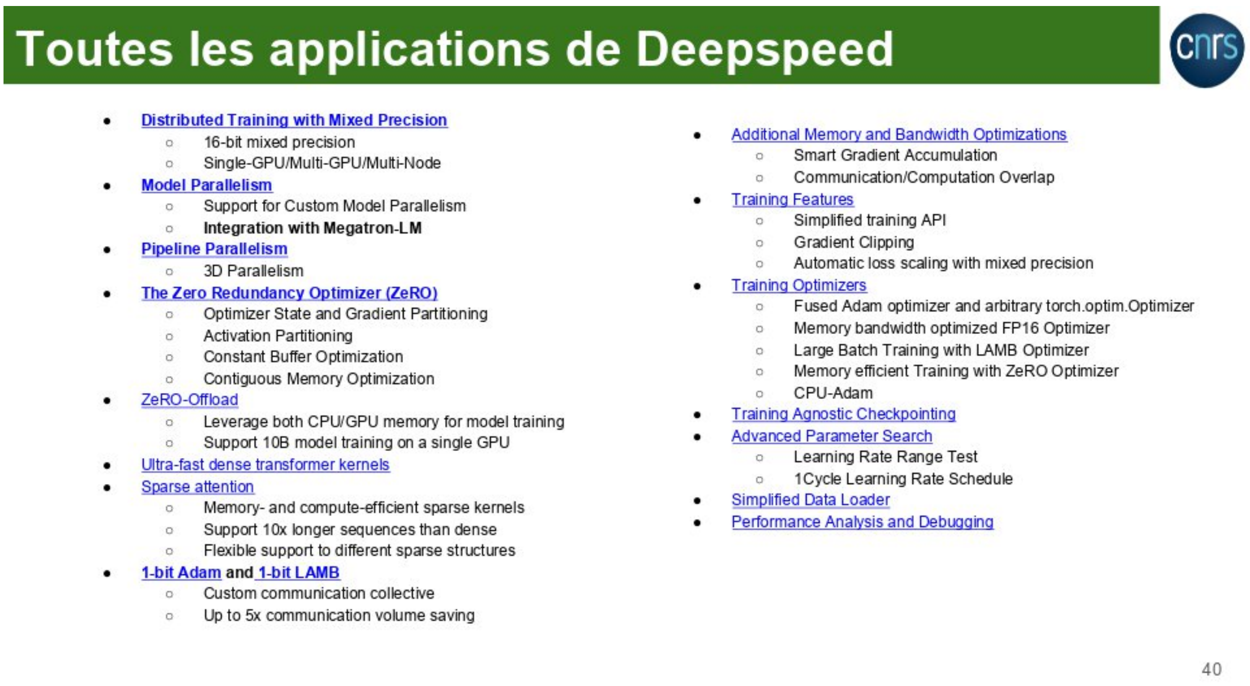
But : diminuent les communications nécessaires et donc accélère l'étape des optimiseurs pour un modèle distribué.

[8, 9]

Deepspeed also offers one-bit optimizers which decrease the necessary communication volume,thereby accelerating the optimizer step for a model distributed in data parallelism.

The priniciple of the 1-bit Adam optimizer is to compute the momentum only locally and then, to exchange only a divergence information coded on 1-bit Adam between the local momentums. The variance, being non-linear, cannot be processed in the same way. On the empirical observation that the variance rapidly becomes nearly stable, the 1-bit Adam, after a warm-up stage corresponding to a classic Adam during a few epochs, switches into its compressed mode where the variance is set at the last value computed.

This enables an approximate 97% reduction in communication volume. For a large model deployed on a large number of GPUs in data parallelism, we observe a time reduction by a factor of 3 with practically identical accuracy.

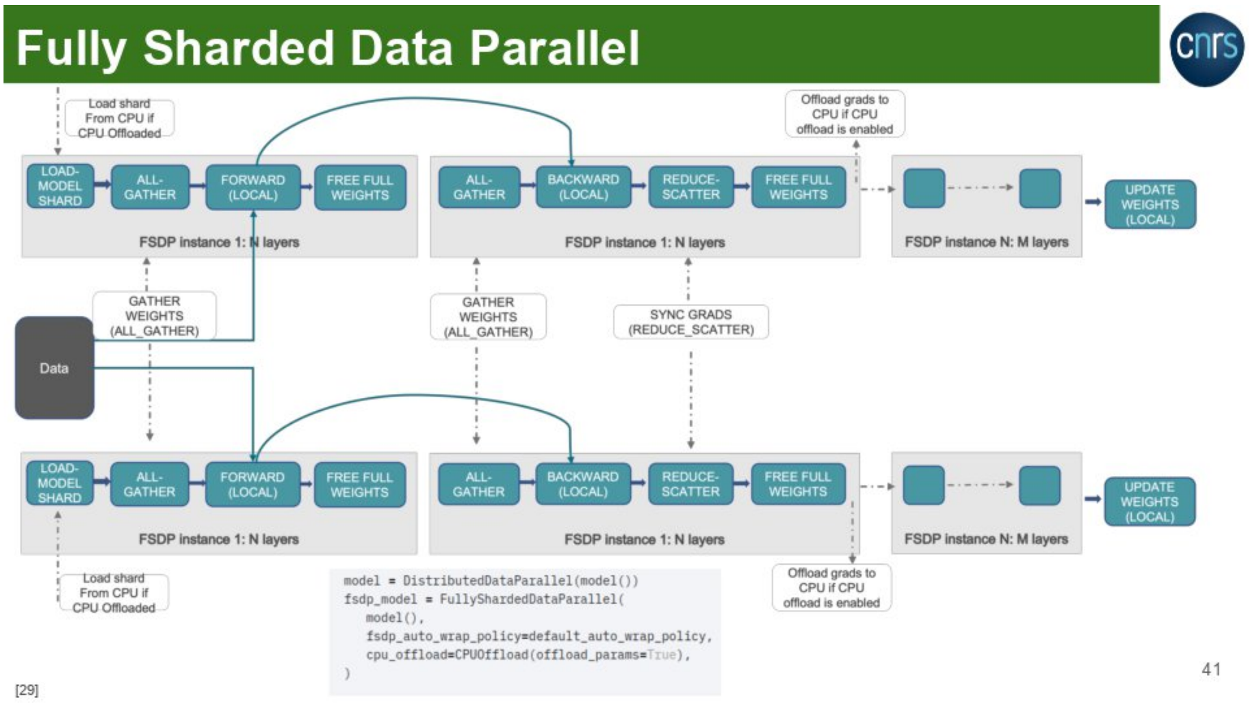

Toutes les applications de Deepspeed

- Distributed Training with Mixed Precision
  - 16-bit mixed precision
  - Single-GPU/Multi-GPU/Multi-Node
- Model Parallelism
  - Support for Custom Model Parallelism
  - **Integration with Megatron-LM**
- Pipeline Parallelism
  - 3D Parallelism
- The Zero Redundancy Optimizer (ZeRO)
  - Optimizer State and Gradient Partitioning
  - Activation Partitioning
  - Constant Buffer Optimization
  - Contiguous Memory Optimization
- ZeRO-Offload
  - Leverage both CPU/GPU memory for model training
  - Support 10B model training on a single GPU
- Ultra-fast dense transformer kernels
- Sparse attention
  - Memory- and compute-efficient sparse kernels
  - Support 10x longer sequences than dense
  - Flexible support to different sparse structures
- 1-bit Adam and 1-bit LAMB
  - Custom communication collective
  - Up to 5x communication volume saving
- Additional Memory and Bandwidth Optimizations
  - Smart Gradient Accumulation
  - Communication/Computation Overlap
- Training Features
  - Simplified training API
  - Gradient Clipping
  - Automatic loss scaling with mixed precision
- Training Optimizers
  - Fused Adam optimizer and arbitrary torch.optim.Optimizer
  - Memory bandwidth optimized FP16 Optimizer
  - Large Batch Training with LAMB Optimizer
  - Memory efficient Training with ZeRO Optimizer
  - CPU-Adam
- Training Agnostic Checkpointing
- Advanced Parameter Search
  - Learning Rate Range Test
  - 1Cycle Learning Rate Schedule
- Simplified Data Loader
- Performance Analysis and Debugging

40

In addition to ZeRO, Deepspeed offers Fused kernels, 1-bit optimizers, and a number of other important applications for acceleration:

- Pipeline Parallelism
- Integration of Megatron-LM enabling the use of 3D Parallelism coupled with ZeRO
- 0/1 Adam which is an optimization of 1-bit Adam
- Memory and buffer configuration
- Sparse Attention
- etc.



Fully Sharded Data Parallel

Load shard From CPU if CPU Offloaded
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
Offload grads to CPU if CPU offload is enabled
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
GATHER WEIGHTS (ALL_GATHER)
GATHER WEIGHTS (ALL_GATHER)
SYNC GRADS (REDUCE_SCATTER)
Data
LOAD-MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
FSDP instance N: M layers
UPDATE WEIGHTS (LOCAL)
Load shard From CPU if CPU Offloaded
Offload grads to CPU if CPU offload is enabled

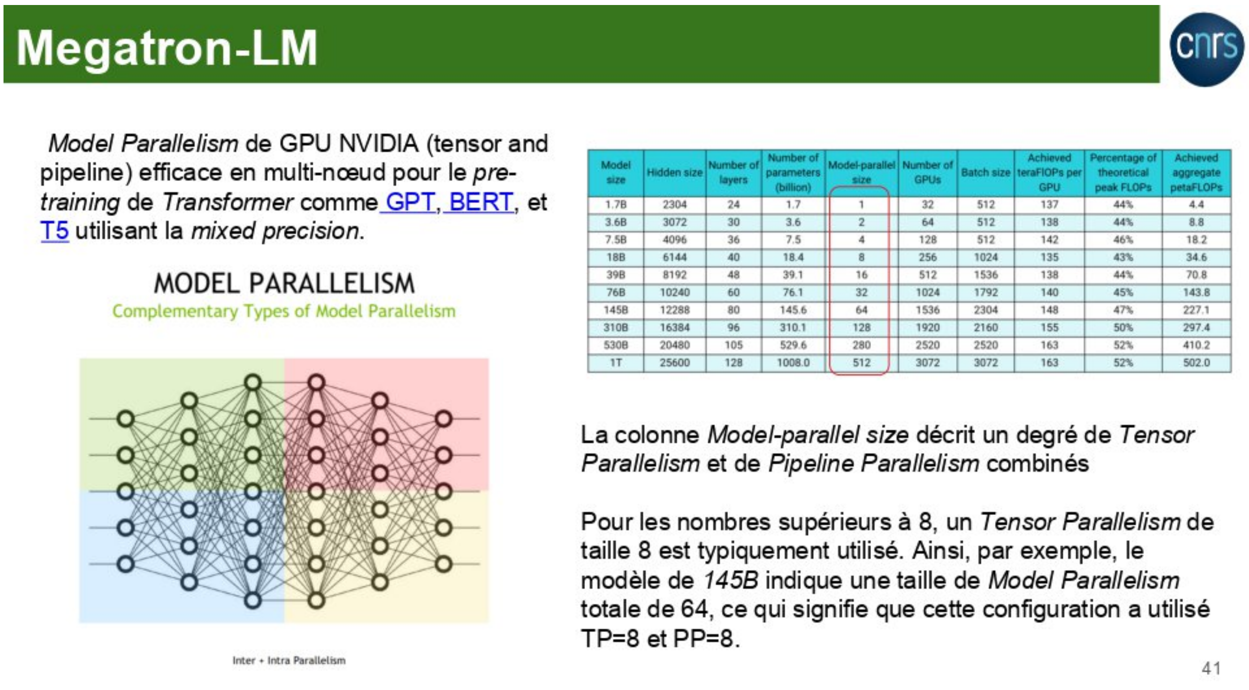
```
model = DistributedDataParallel(model())
fsdp_model = FullyShardedDataParallel(
   model(),
   fsdp_auto_wrap_policy=default_auto_wrap_policy,
   cpu_offload=CPUOffload(offload_params=True),
)
```

[29]

41

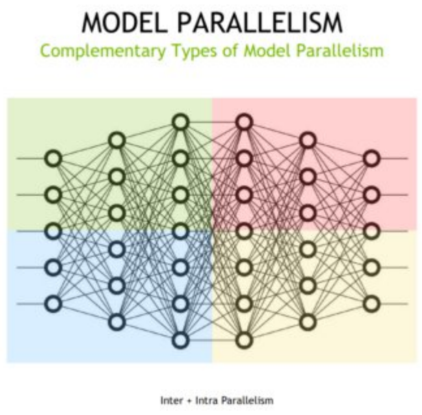
Fully Sharded Data Parallel (FSDP) is a Data Parallelism optimization which is heavily inspired by ZeRO-Stage 3. GPUs do not store all weights at all times. Unlike ZeRO-3 which scatters layers across GPUs (a partitioning similar to Pipeline Parallelism), FSDP partitions each layer across GPUs (a partitioning similar to Tensor Parallelism). It is essential to note that FSDP, as well as ZeRO-3, does not perform model parallelism. All weights are used on all GPUs, it is just that they are not stored and duplicated on all processes. Therefore, we need to execute additional communication steps so that each GPU is able to do the forward and the backward steps properly.

The FSDP implementation has the advantage of being included natively in PyTorch and no longer requires an external library like it used to (FairScale). Its usage is quite simple, and very similar to standard DDP (although we have the possibility to add some specific optimizations as well). Experiments have demonstrated that FSDP can be used to train very large language models, without requiring other parallelism methods. Depending on the configuration and chosen optimizations, FSDP may even reach higher performance than ZeRO-3 from DeepSpeed.



Megatron-LM

Model Parallelism de GPU NVIDIA (tensor and pipeline) efficace en multi-nœud pour le *pre-training* de *Transformer* comme GPT, BERT, et T5 utilisant la *mixed precision*.


MODEL PARALLELISM
Complementary Types of Model Parallelism
Inter + Intra Parallelism

Model size	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Batch size	Achieved teraFlOPs per GPU	Percentage of theoretical peak FLOPs	Achieved aggregate petaFLOPs
1.7B	2304	24	1.7	1	32	512	137	44%	4.4
3.6B	3072	30	3.6	2	64	512	138	44%	8.8
7.5B	4096	36	7.5	4	128	512	142	46%	18.2
18B	6144	40	18.4	8	256	1024	135	43%	34.6
39B	8192	48	39.1	16	512	1536	138	44%	70.8
76B	10240	60	76.1	32	1024	1792	140	45%	143.8
145B	12288	80	145.6	64	1536	2304	148	47%	227.1
310B	16384	96	310.1	128	1920	2160	155	50%	297.4
530B	20480	105	529.6	280	2520	2520	163	52%	410.2
1T	25600	128	1008.0	512	3072	3072	163	52%	502.0

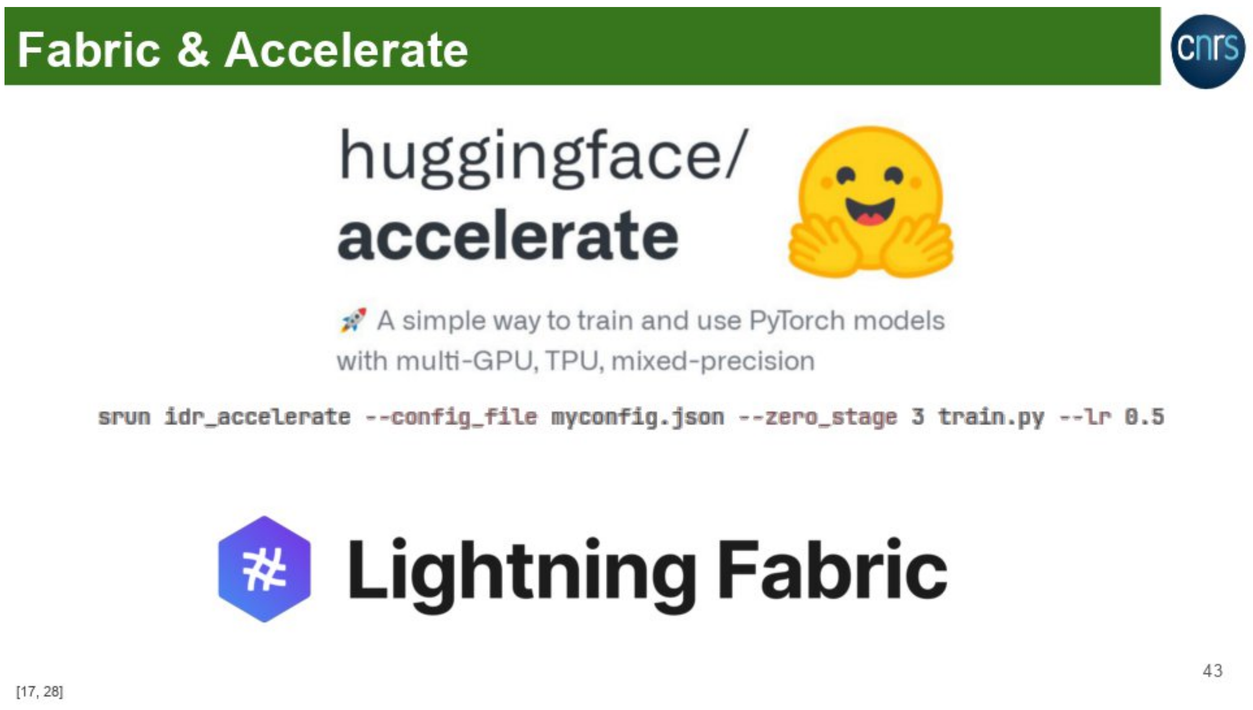
La colonne *Model-parallel size* décrit un degré de *Tensor Parallelism* et de *Pipeline Parallelism* combinés

Pour les nombres supérieurs à 8, un *Tensor Parallelism* de taille 8 est typiquement utilisé. Ainsi, par exemple, le modèle de *145B* indique une taille de *Model Parallelism* totale de 64, ce qui signifie que cette configuration a utilisé TP=8 et PP=8.

41

NVIDIA is developing Megatron-LM which enables the turnkey management of 3D Parallelism using Mixed Precision for the most well-known Transformer architectures with NVIDIA GPUs such as GPT, BERT, and T5.

Megatron-LM represents a major contribution to the training of what are currently the largest Transformer models. The most recent versions of Megatron-LM have also introduced sequence parallelism, a method for partitioning the activations into the GPU subgroup for the layers where tensor parallelism does not intervene (for example, LayerNorms).



Fabric & Accelerate

huggingface/
accelerate

🚀 A simple way to train and use PyTorch models with multi-GPU, TPU, mixed-precision

```
srun idr_accelerate --config_file myconfig.json --zero_stage 3 train.py --lr 0.5
```

Lightning Fabric

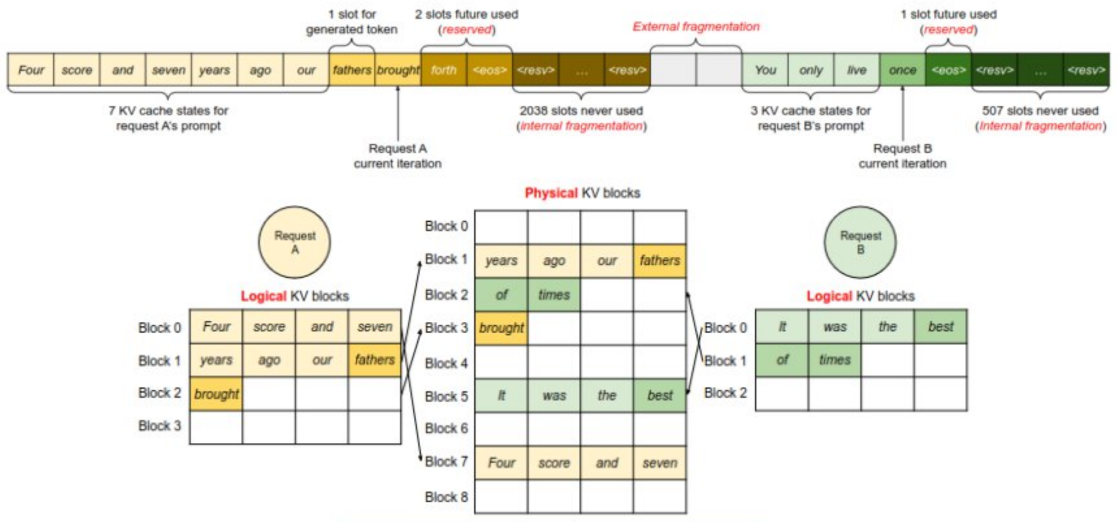
[17, 28]

Accelerate is a library developed by HuggingFace with a similar objective. *Accelerate* is actively being developed and implements new methods very quickly. It allows the usage of Megatron-LM, DeepSpeed, etc. IDRIS' technological watch group realized that multi-node training with Accelerate is possible but also very tedious since it requires one specific configuration file per node. That is why we developped *idr_accelerate*, a wrapper of the *accelerate* command which manages those configuration files behind the scenes. It is used the following way:

`srun (or torchrun) idr_accelerate <accelerate options such as config file> train_file.py <train file arguments>`

Lightning Fabric is a light version of Pytorch Lightning which is not as invasive in the code and also not as obscure. It also enables multiple optimizations without many code updates.

vLLM (Inférence des transformers)



```
llm = LLM(model="facebook/opt-125m")
outputs = llm.generate(prompts, sampling_params)
```

[16]

vLLM

44

LLM’s inference has become a very signification concern since lately, on top of powering many online APIs, many scientists use it on supercomputers to study their behaviours and their predictions.

Since attention mechanism do not apply independently on each token, it is more efficient to store the *Key* and the *Value* of previous tokens to compute attention scores more quickly. However this is extremely memory-consuming.

The fact that LLM generate variable-sized sequences, that each token needs to be generated sequentially, and that some methods (for instance Beam Search) requires generating multiple tokens for a single prompt (therefore branching the generation process) leads to extreme and unreasonable memory usage. More of it is not even exploited (it is just memory fragmentation), but it limits the batch size quite severely.

For this reason, vLLM creates Paged Attention. Instead of storing an entire sequence contiguously in memory (which leads to unused reserved memory), we split tokens in small groups. Within a group, tokens are stored contiguously in physical memory, but groups are not. Since most CUDA kernels require tensors to be stored contiguously, vLLM implements the behaviour of an OS on GPU with the usage of a logical memory and pages. We can even re-use memory by pointing multiple logical blocks to the same physical block. Those optimizations limit memory waste and enable larger batches.

TP : CoAtNet-7

- Limite du *Data Parallelism* avec CoAtNet
- Implémenter ZeRO
- Implementer le Pipeline Parallelism
- Recherche du meilleur compromis

Références des images utilisées et articles

1. HuggingFace 2021, https://huggingface.co/blog/large-language-models
2. Nicolae, Bogdan, et al. "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models." *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 2020.
3. FairScale authors. (2021). FairScale: A general purpose modular PyTorch library for high performance and large scale training. https://fairscale.readthedocs.io/en/latest/deep_dive/pipeline_parallelism.html
4. Fan, Shiqing, et al. "DAPPLE: A pipelined data parallel approach for training large models." *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 2021.
5. Narayanan, Deepak, et al. "PipeDream: Generalized pipeline parallelism for DNN training." *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 2019.
6. Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 2020.
7. Jiang, Zixuan, et al. "Optimizer Fusion: Efficient Training with Better Locality and Parallelism." *arXiv preprint arXiv:2104.00237* (2021).
8. Deepspeed 2020, https://www.deepspeed.ai/2020/09/08/onebit-adam-blog-post.html
9. Tang, Hanlin, et al. "1-bit adam: Communication efficient large-scale training with adam's convergence speed." *International Conference on Machine Learning.* PMLR, 2021.
10. Vaswani, Ashish, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).
11. Peltarion, https://peltarion.com/blog/data-science/self-attention-video
12. Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv preprint arXiv:2010.11929* (2020).
13. Dai, Zihang, et al. "Coatnet: Marrying convolution and attention for all data sizes." *Advances in Neural Information Processing Systems* 34 (2021): 3965-3977.
14. Medium, https://medium.com/@oskyhn_77789/current-convolutional-neural-networks-are-not-translation-equivariant-2f04bb9062e3
15. AI Summer, https://theaisummer.com/receptive-field/
16. https://vllm.ai/
17. https://huggingface.co/docs/accelerate/index
18. Gou, Jianping, et al. "Knowledge distillation: A survey." *International Journal of Computer Vision* 129 (2021): 1789-1819.
19. Gholami, Amir, et al. "A survey of quantization methods for efficient neural network inference." *arXiv preprint arXiv:2103.13630* (2021).
20. Zhou, Aojun, et al. "Learning N: M fine-grained structured sparse neural networks from scratch." *arXiv preprint arXiv:2102.04010* (2021).
21. https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/
22. Shoeybi, Mohammad, et al. "Megatron-lm: Training multi-billion parameter language models using model parallelism." *arXiv preprint arXiv:1909.08053* (2019).
23. Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation." *arXiv preprint arXiv:1308.3432* (2013).
24. Frankle, Jonathan, and Michael Carbin. "The lottery ticket hypothesis: Finding sparse, trainable neural networks." *arXiv preprint arXiv:1803.03635* (2018).
25. Gale, Trevor, Erich Elsen, and Sara Hooker. "The state of sparsity in deep neural networks." *arXiv preprint arXiv:1902.09574* (2019).
26. Zhu, Michael, and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression." *arXiv preprint arXiv:1710.01878* (2017).
27. Sanh, Victor, et al. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." *arXiv preprint arXiv:1910.01108* (2019).
28. https://lightning.ai/docs/fabric/stable/
29. https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/