Deep Learning Optimisé - Jean Zay

Les parallélismes des gros modèles

INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

DLO-JZ

6ème partie de la formation de l'IDRIS.

Diapositives commentées.

Auteurs : Bertrand Cabot, Nathan Cassereau

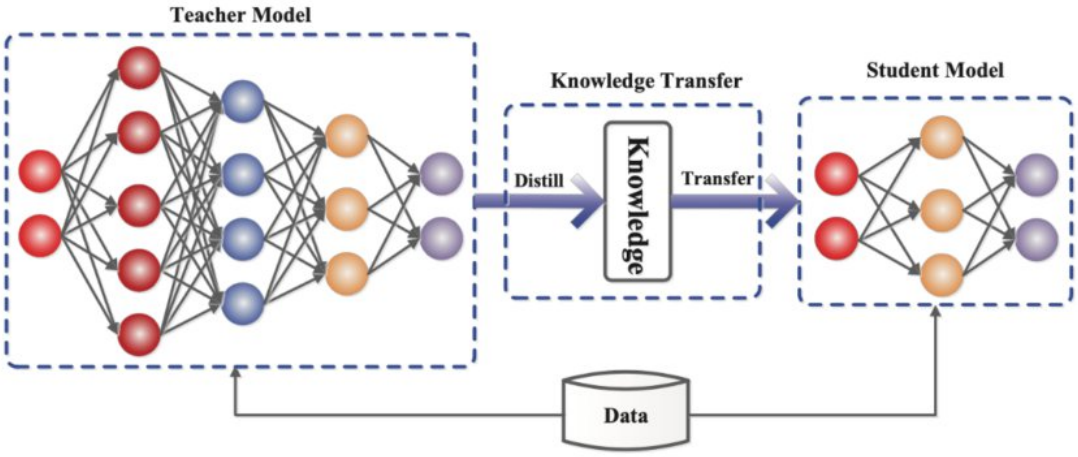
Mai 2022. Mise à jour Octobre 2023.

Chapitres :

- Inférence et fine-tuning
- Les *Vision Transformer*
- Les Parallélismes de modèle pour les très gros modèles
- API parallélismes de modèle

Cette partie a pour objectif de brièvement présenter quelques méthodes liées à l'inférence et au fine-tuning des modèles très larges.

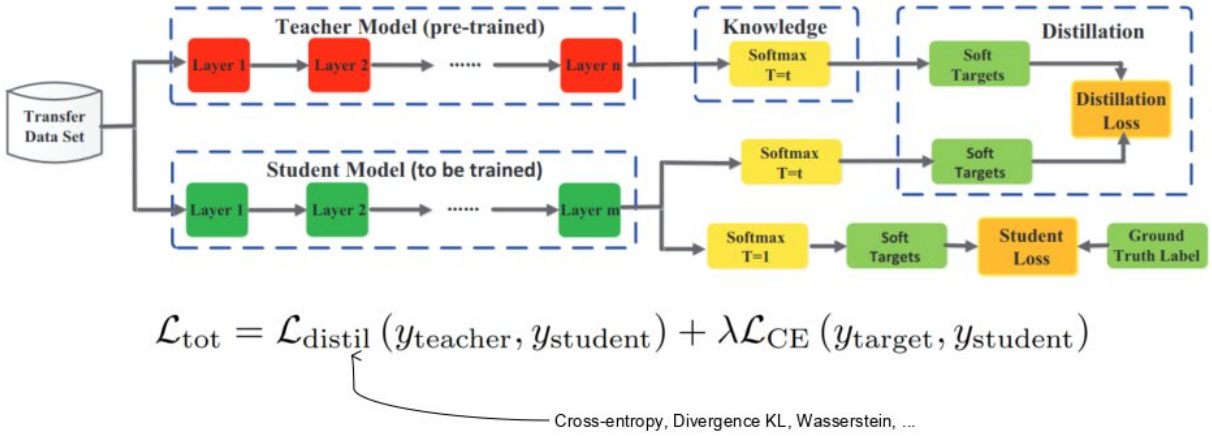
Distillation



[18]

La distillation permet de résoudre une tâche complexe dans un réseau un peu plus petit. On peut exploiter toute la connaissance apprise par le gros réseau et la compacter dans un plus petit réseau. On aurait alors une perte de performance négligeable pour un gain en temps de calcul potentiellement important.

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.

Pour faire la distillation, on entraîne le petit réseau en se servant des prédictions du gros réseau. Il est également possible d'utiliser les vrais labels s'ils sont encore disponibles.

Les fonctions de perte entre le professeur et l'étudiant peuvent être des erreurs quadratiques, entropie croisée, divergence de Kullback-Leibler, etc.

Grâce à cette méthode, DistilBERT a réussi à atteindre 97% des performances de BERT avec seulement 60% de ses poids.

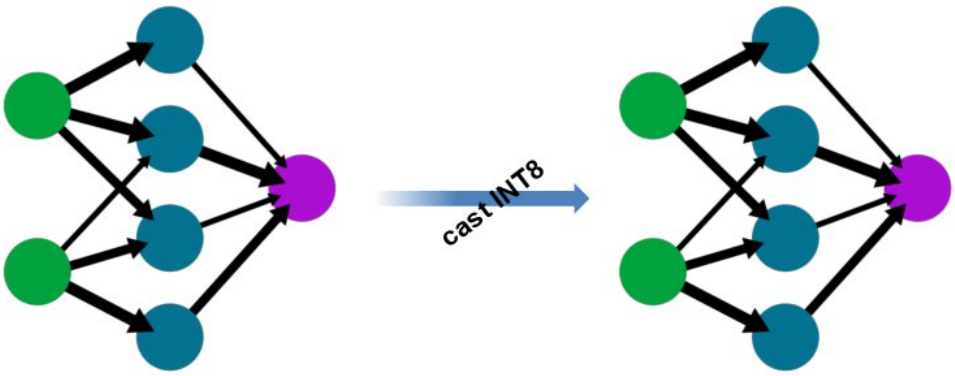
Quantification

	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

La quantisation permet de profiter d'une consommation mémoire réduite ainsi que des performances accrues des GPU avec des entiers 8 bits qu'avec d'autres formats flottants.

Quantification

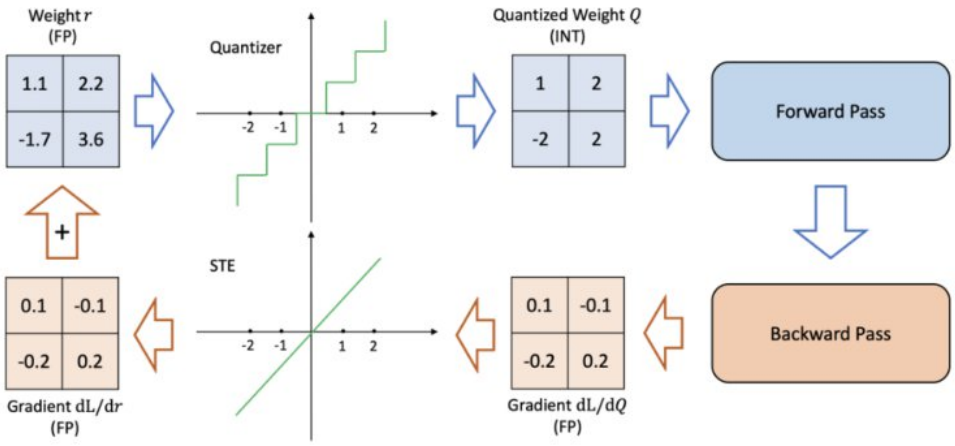


Il est possible de rencontrer une perte en performance

6

La quantisation à la prédiction consiste tout simplement à convertir les poids en des entiers 8 bits. Il faut toutefois surveiller cette opération car elle peut fortement dégrader les performances du réseau entraîné. Pour compenser cet effet, on peut faire un léger fine-tuning post-quantisation pour retrouver les performances perdues.

Quantification

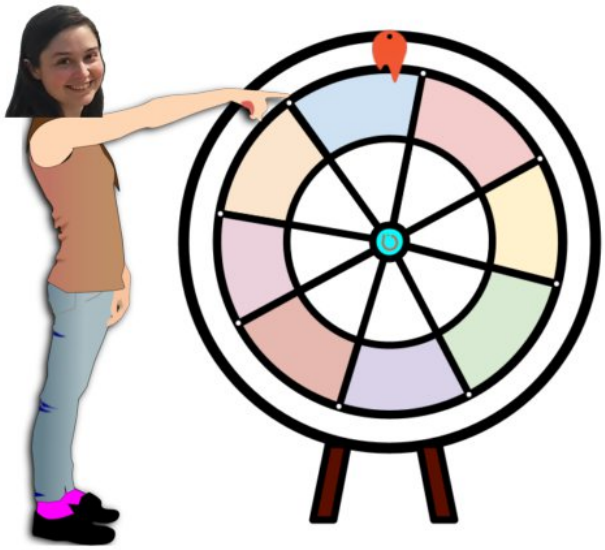


Exemple d'entraînement post-quantisation

[19, 23]

L'entraînement avec des poids entiers n'est pas possible à cause d'un soucis de différentiabilité. On peut contourner le problème en maintenant une copie flottante de nos poids. On peut passer du domaine continu au domaine discret via une fonction en escalier. On peut réaliser l'opération inverse via le *Straight-Through Estimator* [23], qui ignore l'opération d'arrondi, et estime les gradients flottants via une opération identité.

Pruning

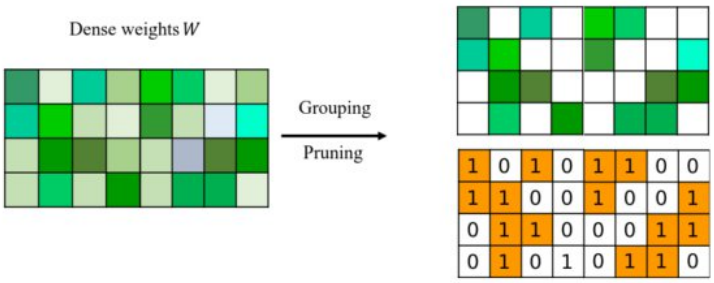


The Lottery Ticket Hypothesis.
A randomly-initialized, dense neural network contains a subnetwork that is initialized such that—when trained in isolation—it can match the test accuracy of the original network after training for at most the same number of iterations.

[24, 25, 26]

L’hypothèse du ticket gagnant est l’idée qu’un réseau de neurones initialisés aléatoirement consiste en réalité en un ensemble de sous-réseaux, dont certains ont des performances équivalentes au réseau originel. Cela signifie que l’on pourrait supprimer tous les autres poids. Les calculs en seraient accélérés sans perte significative de performance. Il semblerait [26] qu’un gros réseau creux serait bien plus performant que de plus petits réseaux denses.

Pruning



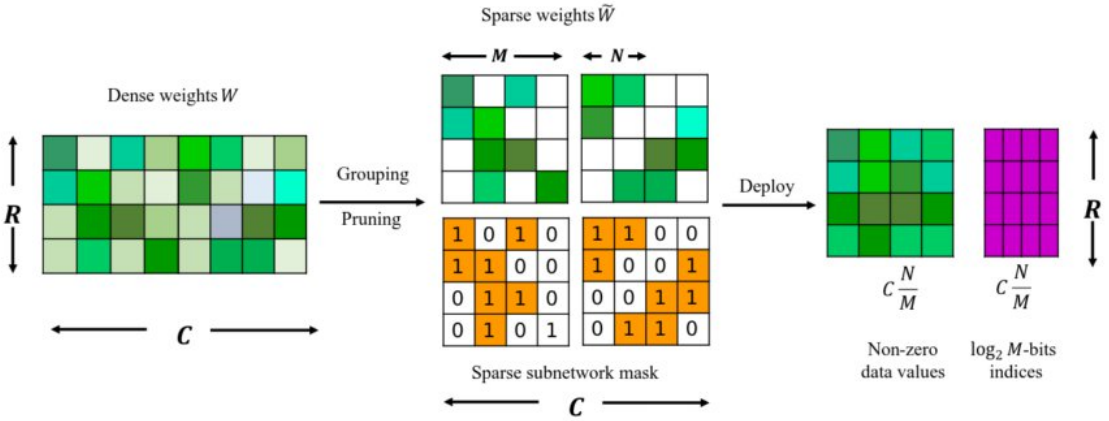
Les poids les plus petits sont mis à 0. Mais combien ?
Quel impact sur le temps de calcul ?

[20]

Il existe plusieurs méthodes pour sélectionner les poids à garder et les poids à abandonner mais des tests [25] semblent montrer que simplement s'appuyer sur l'amplitude de chacun des poids est tout aussi efficace que des méthodes plus sophistiquées.

On peut ensuite valider les poids conservés et retrouver les performances éventuellement perdues via une étape de fine-tuning. Notamment on peut ré-utiliser les *Straight-Through Estimator* (ou plutôt une extension des STE [20]) des méthodes de quantisation pour effectuer l'entraînement du petit réseau en ayant le gros réseau à l'esprit.

Pruning



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

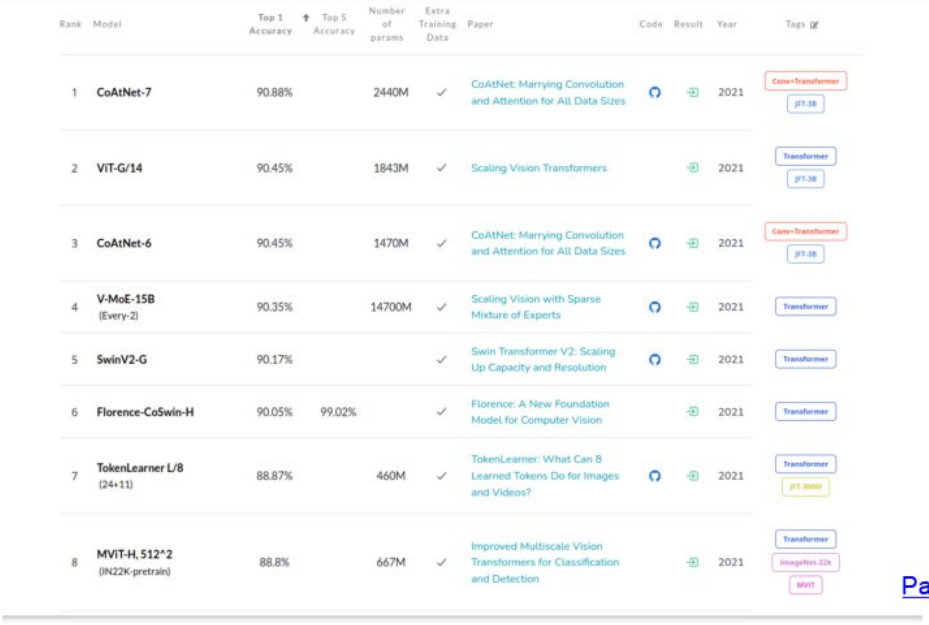
[20, 21]

Mettre certains poids à zéros permet en théorie de diminuer la quantité de calcul et donc le temps d'exécution nécessaire. En pratique, cela ne permet pas de profiter de l'architecture des accélérateurs GPU. C'est pourquoi les GPU les plus récents supportent du calcul avec des matrices creuses. Par exemple les A100 de NVIDIA supportent une dispersion 2:4. Cela signifie que sur tous les quatre poids consécutifs, on peut en abandonner deux.

Si cela est nécessaire pour maximiser notre exploitation du support du calcul creux par les GPU NVIDIA, il est possible de faire des échanges de lignes et de colonnes de certaines couches. Si les échanges sont fait méticuleusement la sortie de la couche est inchangée. Par exemple dans le mécanisme d'attention, on peut faire un échange entre les colonnes de Q tant qu'on fait l'échange équivalent sur les lignes K.T. Même commentaire sur les couches denses si on en considère deux consécutives ensemble.

Cette partie a pour objectif de présenter les *Transformer*, les *Vision Transformer,* puis CoAtNet qui servira de gros modèle type pour le TP sur le *Model Parallelism*.

Vision Transformers >> Resnet-50: 25M

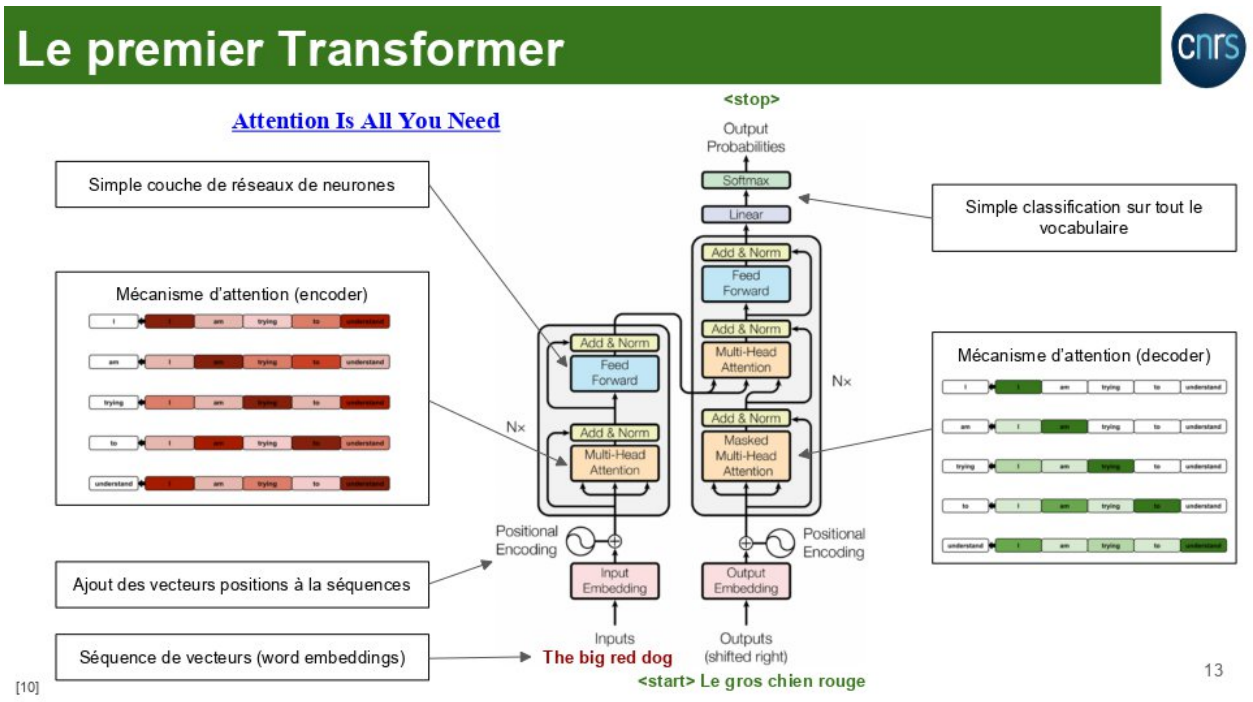


Rank	Model	Top 1 Accuracy	Top 5 Accuracy	Number of params	Extra Training Data	Paper	Code	Result	Year	Tags
1	CoAtNet-7	90.88%		2440M	✓	CoAtNet: Marrying Convolution and Attention for All Data Sizes			2021	Conv+Transformer, JFT-3B
2	ViT-G/14	90.45%		1843M	✓	Scaling Vision Transformers			2021	Transformer, 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 12

Sur *Paperswithcode* nous pouvons voir que l'état de l'art des modèles sur *Imagenet* sont des *Vision Transformer* de plus d'un milliard de paramètres, soit des gros modèles qui rentrent dans les problématiques vues précédemment.

A noter que *CoAtNet* que l'on utilisera pendant le TP a la première place du classement.

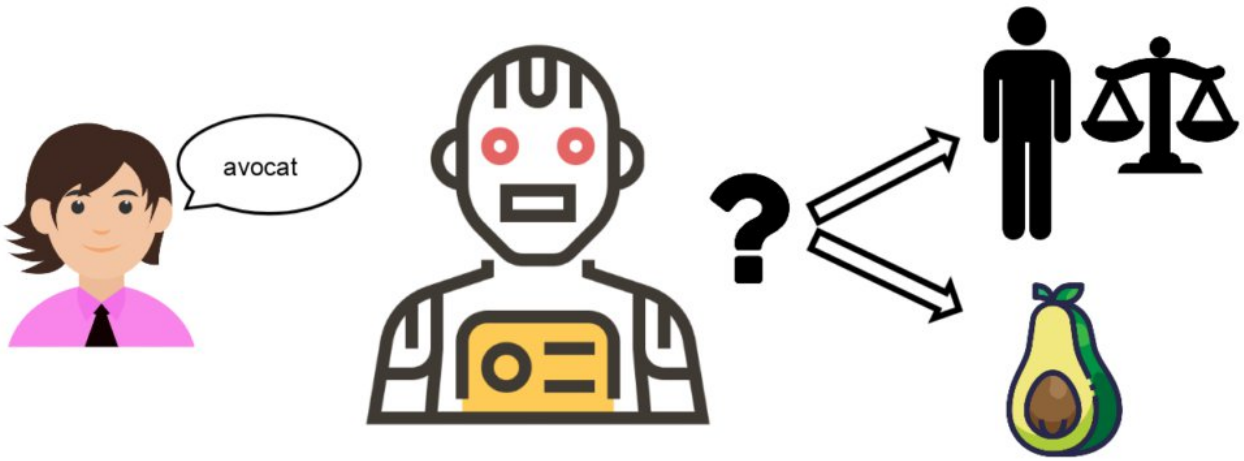


Le premier *Transformer* décrit dans le papier “Attention Is All You Need” correspondait alors à un nouveau type d’architecture permettant la traduction de texte de l’anglais au français. Ce premier *Transformer* s’inspire et améliore les RNN utilisés pour cette même application de traduction. Il est composé d’un *encoder* et d’un *decoder*.

Les RNN les plus sophistiqués exploitaient un mécanisme d’*attention* pour résoudre le problème de la représentation d’une phrase entre l’*encoder* et le *decoder.* Le Transformer exploite exclusivement ce mécanisme d’*attention* sans avoir de couches récurrentes.

Au lieu d’encoder la phrase mot à mot comme le fait les RNN, il prend la séquence entière à encoder en ajoutant une information de position du mot dans la phrase et transforme la séquence en une séquence latente en passant par le mécanisme de *self-attention*.

Mécanisme d’attention (intuition)



14

En français et dans la plupart des langues, certains mots (les homographes) peuvent avoir plusieurs significations. Le vecteur associé à ses mots contient donc des informations relatives à tous ses sens. Par exemple, en français, le mot « avocat » peut désigner un fruit ou une personne.

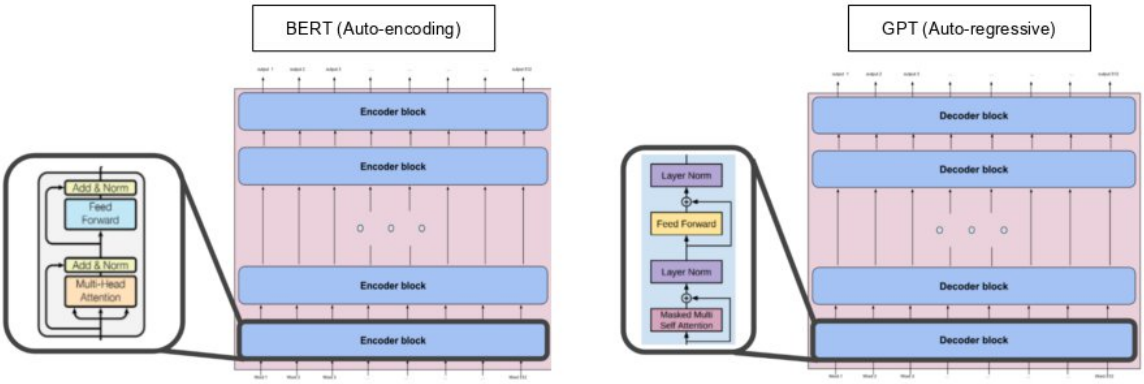
Mécanisme d’attention (intuition)

15

Le transformer peut clarifier le sens grâce au contexte. Si la phrase contient des mots comme « justice » ou « plaidoyer » alors on peut clarifier la signification dans un sens ; dans l’autre si on retrouve des mots comme « arbre » ou « noyau ».

Le vecteur associé au mot « avocat » sera alors enrichi pour lever l’ambiguïté.

BERT vs GPT

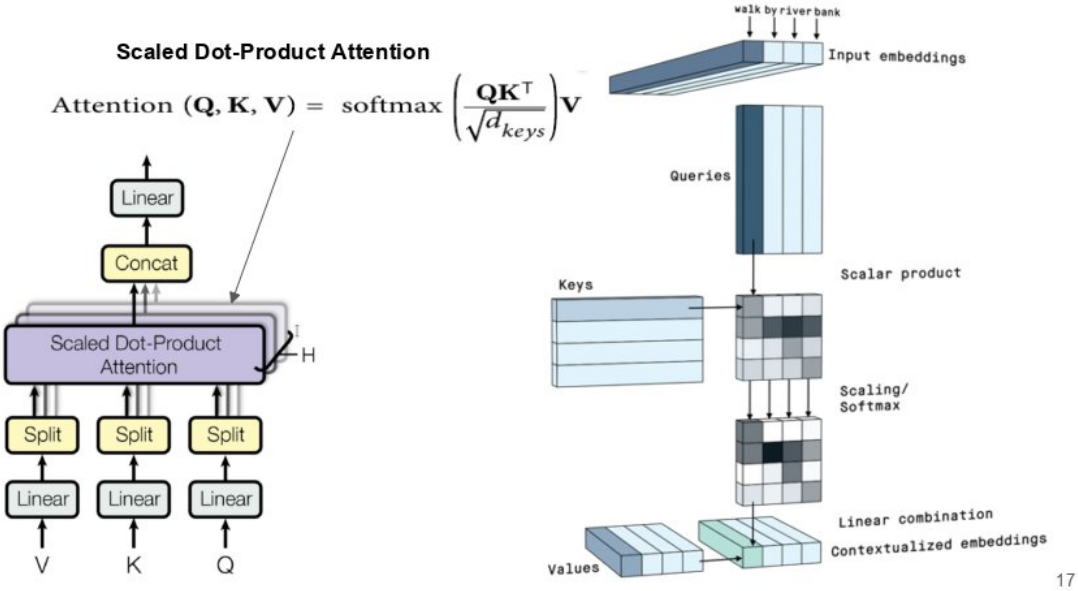


Le premier transformer contient une pile d'encodeurs, et une pile de décodeurs. Le transformer le plus célèbre avec cette architecture est T5. D'autres transformers très célèbres ont adopté une architecture un peu différente.

BERT est le transformer qui a lancé la « transformer-mania ». Il est simplement composé d'une pile d'encodeurs. Il est notamment très adapté pour ce qui concerne la classification (de la phrase ou de chacun des tokens).

Les transformers de type GPT sont composés d'une pile de décodeurs. Cela les rend particulièrement adapté à la génération de texte. Une découverte récente grâce aux LLM (Large Language Models) qui sont tous des décodeurs est que cette génération lui offre également certaines capacités de classification.

Le mécanisme de Self-Attention



[10, 11]

Le mécanisme de *Self-Attention* consiste à appliquer le *Scaled Dot-Product Attention* sur une séquence d'*Input embeddings* (qui a déjà subit une transformation linéaire).

Les tenseurs *Queries*, *Keys* et *Values* sont identiques dans la *Self-Attention* ce qui permet de transformer la séquence en une séquence latente qui prend en compte la dépendance de chaque mot par rapport à chaque autre mot. QK^T correspond à la matrice de dépendance.

Le *Multi-head Attention* divise la séquence d'*Input Embeddings* pour la passer dans plusieurs *heads* de *Scaled Dot-Product*. Cela permet de prendre en compte plusieurs niveaux de liens de dépendance en même temps et plusieurs représentations d'un même vecteur. Cela devient aussi un processus complètement parallélisable et accéléré. Toutes les transformations linéaires permettent d'avoir en plus une liberté totale sur les dimensions des tenseurs.

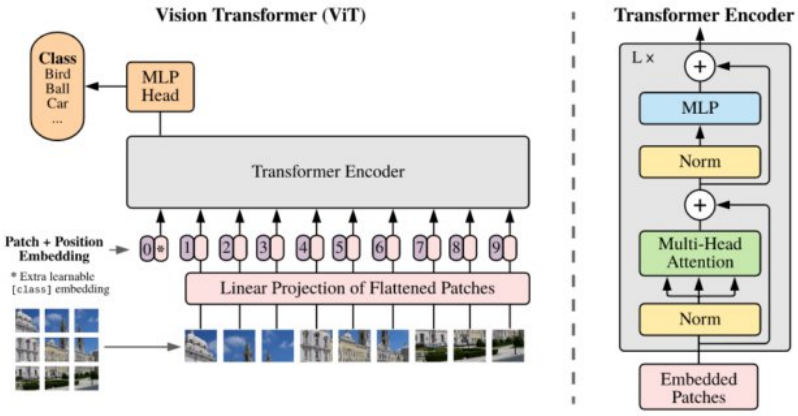
Les Transformers

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

En résumé, les Transformers :

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

Le premier *Vision Transformer* appliqué à de l'imagerie est décrit dans le papier "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale".

Au lieu de traiter une séquence de mot, le *ViT* traite une séquence d'image. Pour cela il *patch* l'image en sous-images représentées en vecteurs de pixels. Ces *patch* après une transformation linéaire représentent la séquence d'I*nput Embeddings* auquel on ajoute une information de position : exactement comme pour le premier *Transformer* en NLP.

Cependant comme ce qui est décrit pour *BERT* en NLP, on ajoute un patch 0, un class token à la séquence qui servira après transformation à la sortie de classification.

CoAtNet

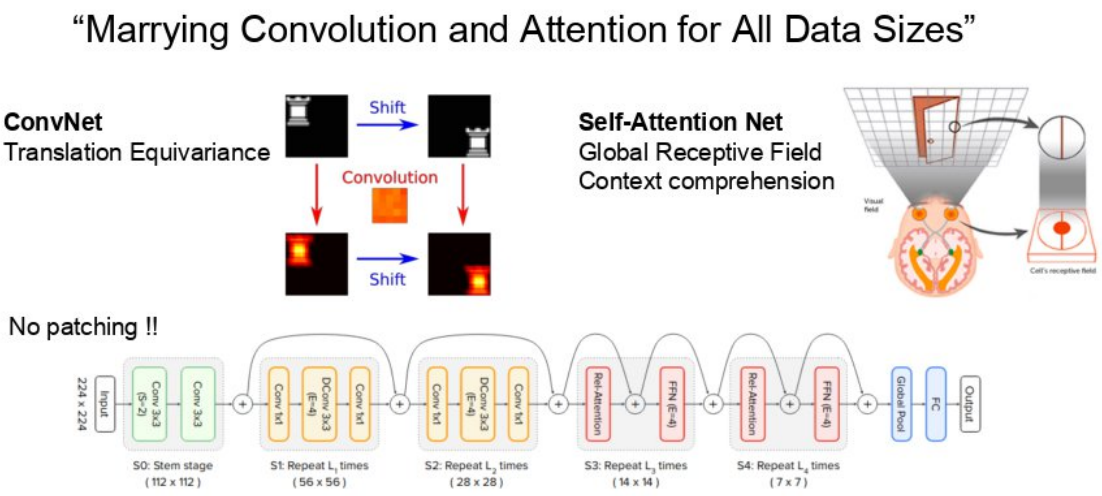


Figure 4: Overview of the proposed CoAtNet.

[13, 14, 15]

Enfin, *CoAtNet* que l'on utilisera pour le TP sur le *Model Parallelism* est décrit dans le papier "CoAtNet: Marrying Convolution and Attention for All Data Sizes".

CoAtNet est l'assemblage d'abord d'un réseau CNN classique puis d'un *Vision Transformer*. Puisque le modèle commence en CNN, il n'y a pas de *patching* d'image à faire en entrée. Nous pouvons donc remplacer facilement un CNN dans un code par un *CoAtNet*. La transition entre CNN et *Vision Transformer* se fait en patchant la sortie de la partie CNN.

Cela a pour avantage de profiter de l'équivariance en translation des CNN et de la compréhension globale du contexte de l'image des *Vision Transformer*, en même temps.

CoAtNet - Résultats

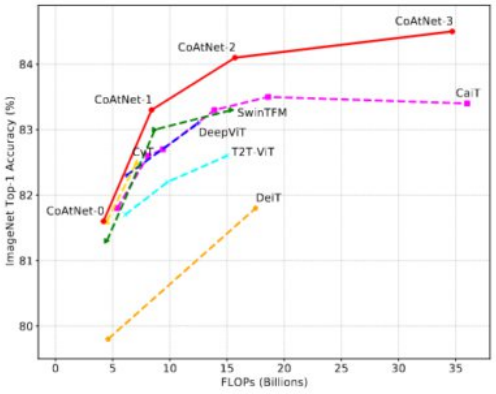


Figure 2: Accuracy-to-FLOPs scaling curve under ImageNet-1K only setting at 224x224.

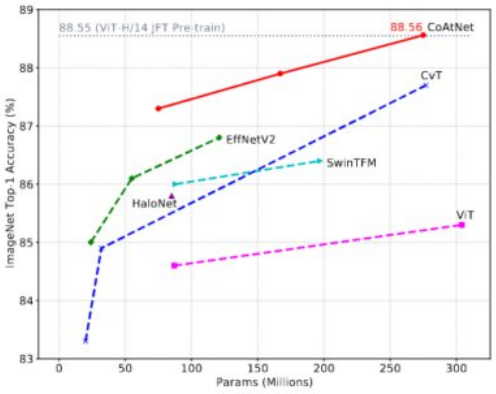
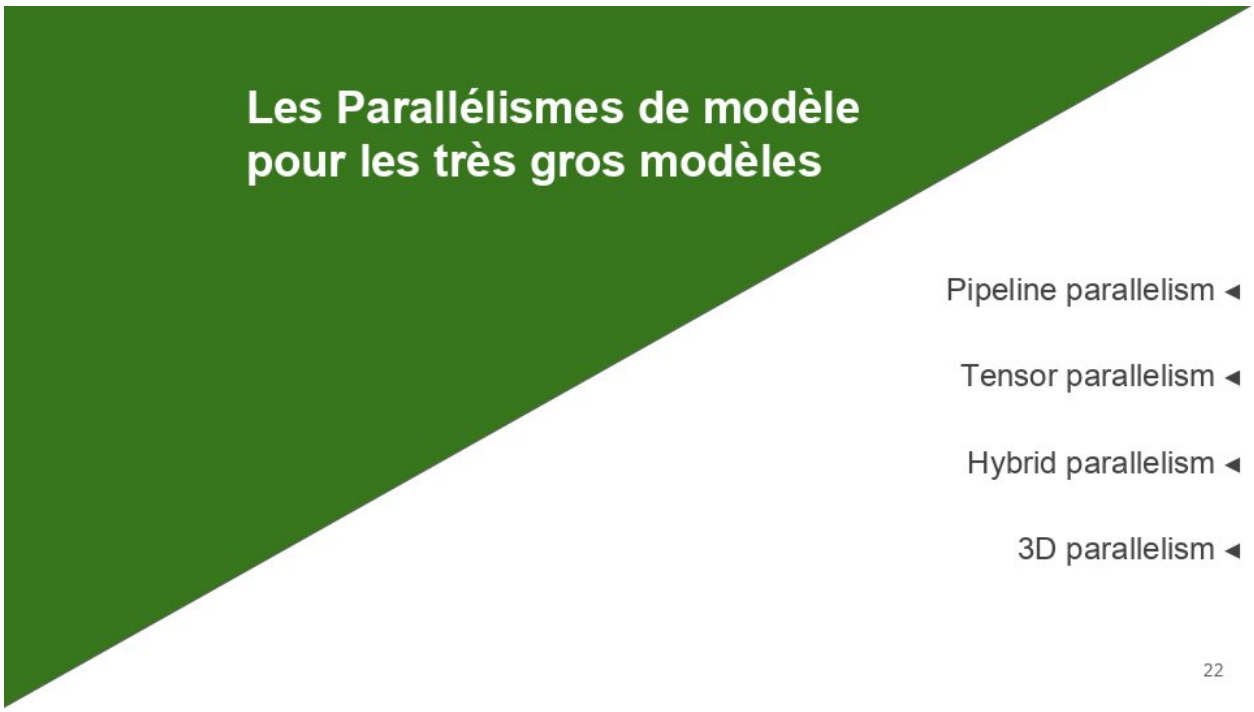


Figure 3: Accuracy-to-Params scaling curve under ImageNet-21K ⇒ ImageNet-1K setting.

[13]

Les résultats des 7 niveaux des modèles *CoAtNet* surpassent tous les *Vision Transformers* actuels et les CNN actuels à nombre de paramètres égal ou à coût de calcul égal.

Il est intéressant de noter que pour augmenter le *score* d'*accuracy* des *Vision Transformer* sur *Imagenet-1k*, il existe des extensions, par exemple *ImageNet-21k* 100 fois plus grosse, pour un apprentissage augmenté.

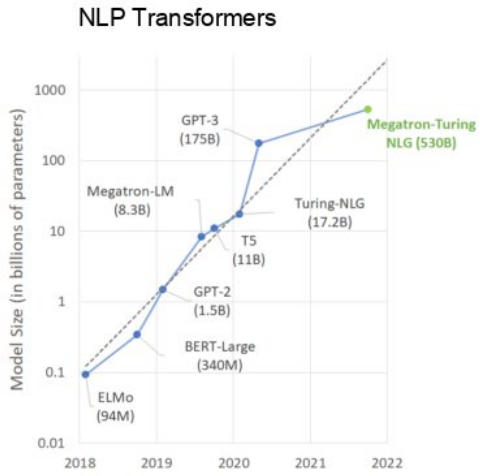


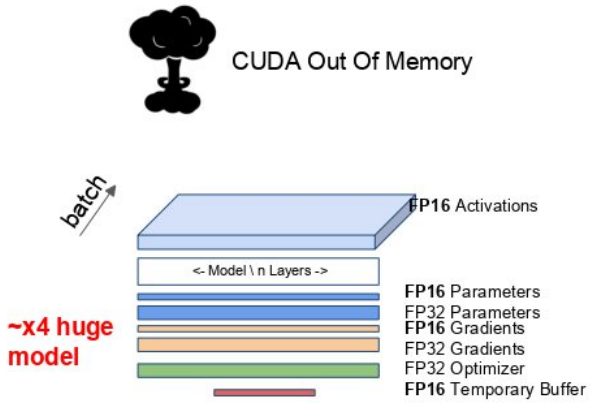
L’objectif de cette partie est de décrire les différentes formes de *Model Parallelism* possibles et l’intérêt de celles-ci.

Du fait de sa complexité, le *Model Parallelism* ne s’utilise seulement que pour les très gros modèles.

Chaque parallélisme doit se penser selon un axe qui lui est spécifique. Nous décrirons alors : le *Pipeline Parallelism*, le *Tensor Parallelism*, l’Hybrid Parallelism, le *3D Parallelism*.

Énormes modèles > 1 Milliard de paramètres





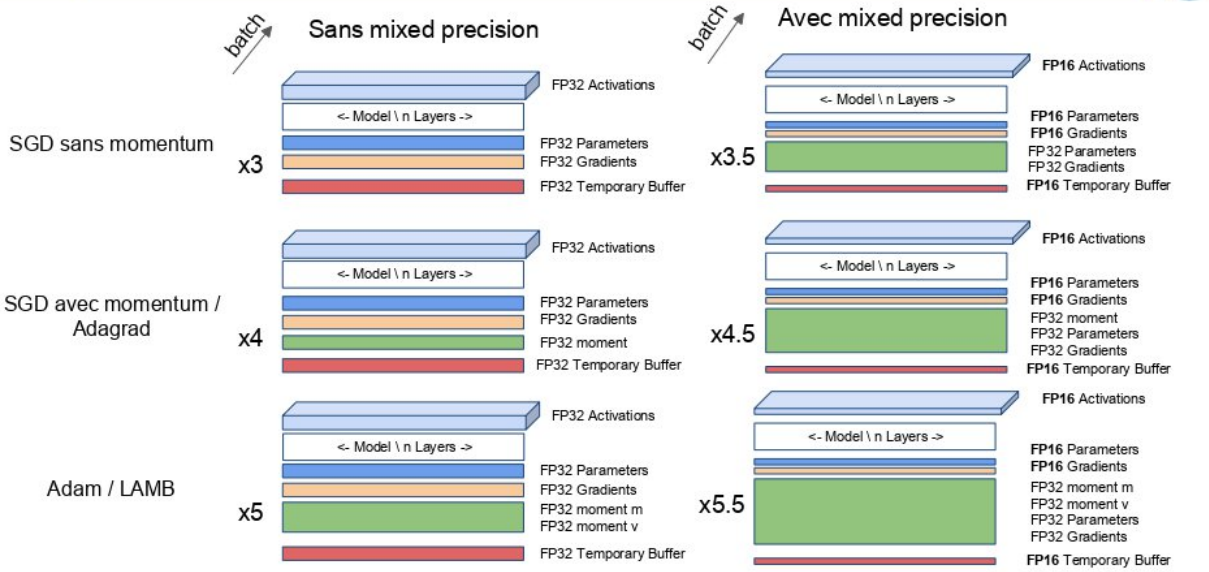
23

[1]

Depuis l'arrivée des *Transformer* en NLP et des *Vision Transformer* en *Computer Vision,* les modèles de réseaux de neurones sont entrés dans une autre dimension. On parlera de gros modèles de plus d'un milliard de paramètres qui apportent de nouvelles problématiques même à l'échelle d'un supercalculateur et qui sont complètement hors de portée d'un ordinateur classique.

D'après ce que l'on a vu précédemment, l'empreinte mémoire directement liée au modèle était négligeable pour des modèles classiques en terme de taille. Pour les gros modèles, cela devient critique.

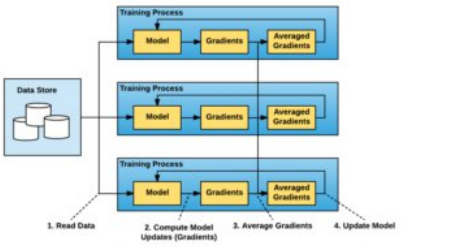
Empreinte mémoire

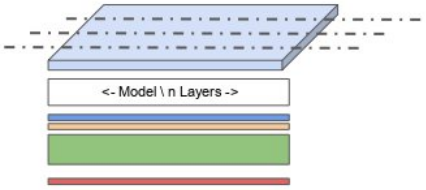


L’empreinte liée au modèle augmente bien sûr en fonction de la taille du modèle, mais aussi lorsque l’on utilise la *Mixed Precision* et selon l’*optimizer* utilisé et le nombre de *momentum* associés.

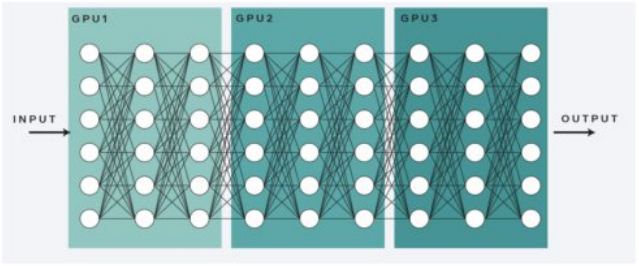
Les différents parallélismes

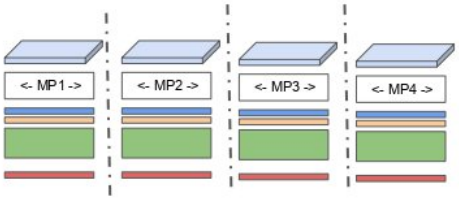
Data Parallelism
- Meilleur Throughput
- Seule l'empreinte mémoire des activations est distribuée
- Multi Processing





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



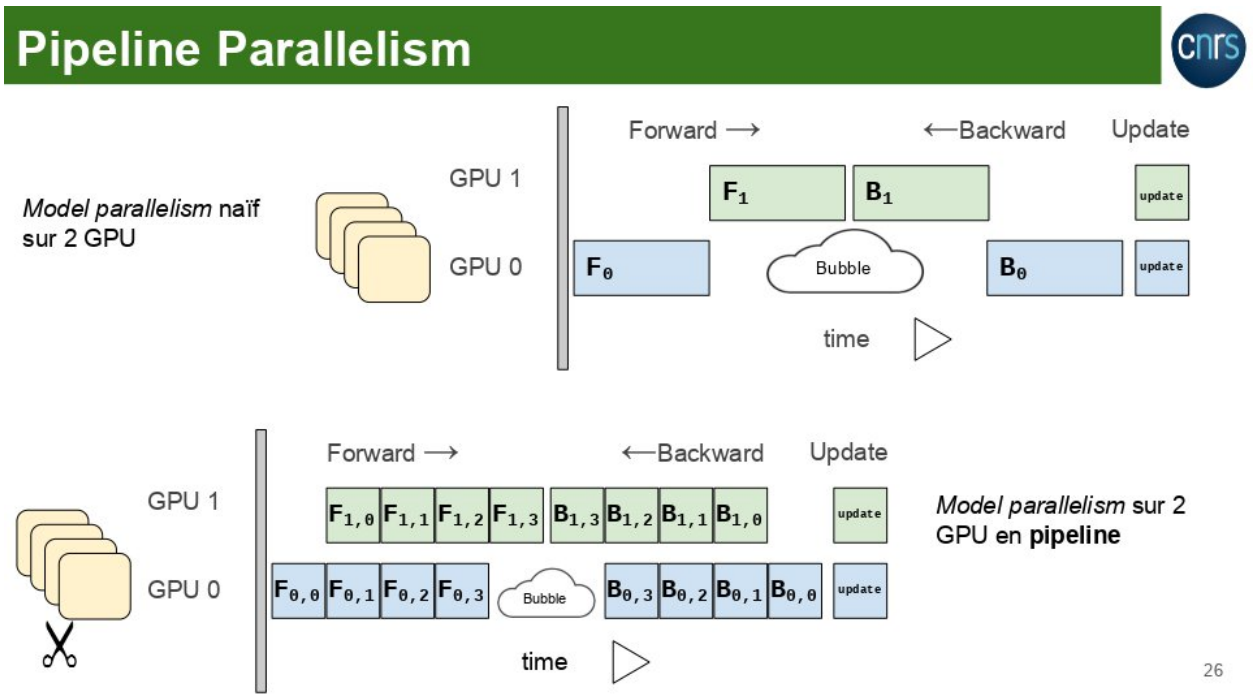


[2, 3]

25

Le *Data Parallelism* que nous avons vu précédemment est la meilleure solution pour le *throughput* et pour sa facilité d'implémentation. Cependant seule l'empreinte mémoire des activations est distribuée. Cela est un problème pour les gros modèles. Le *Data Parallelism* distribue les *batches* à travers les GPU.

Nous ajoutons donc pour les gros modèles le principe de *Model Parallelism* (notamment le *Pipeline Parallelism*). Il s'agit de distribuer le modèle à travers les GPU. Ainsi l'empreinte mémoire complète est distribuée. De plus la taille du *batch* n'est pas augmentée ce qui peut être avantageux lorsque l'on utilise un nombre de GPU très important pour la distribution par rapport à la *DDP*.



La manière naïve de faire du *Model Parallelism* serait de couper le modèle entre 2 couches et d'utiliser les GPU de manière séquentielle pendant la boucle d'apprentissage. Cependant il n'y aurait aucune accélération puisqu'il n'y aurait aucune parallélisation. Seule la distribution de l'empreinte mémoire serait réalisée.

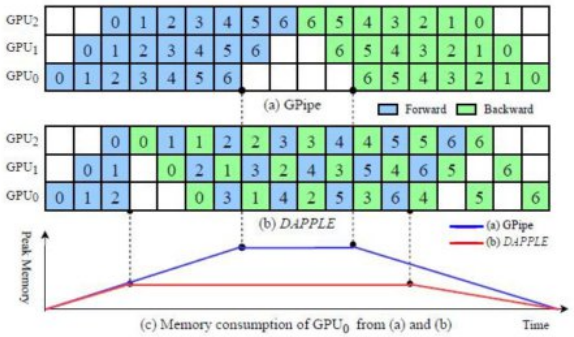
Le *Pipeline Parallelism* en découpant le modèle de la même manière entre 2 couches permet en plus d'accélérer le processus. Pour cela, il subdivise le *batch* en *micro batch*. L'itération d'apprentissage est complète lorsque tous les *micro batches* sont passés en *forward* et *backward*.

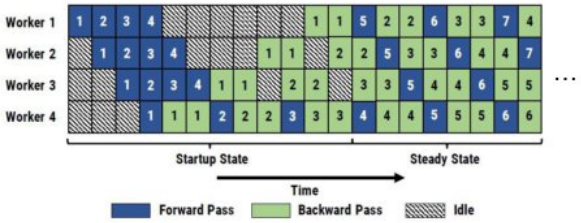
Ainsi, la *bubble* correspondant au temps d'inactivité du GPU est largement réduite. Donc seul, le *Pipeline Parallelism* est utilisé en pratique lorsque l'on coupe le modèle dans le sens des couches.

Pipeline Parallelism

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

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



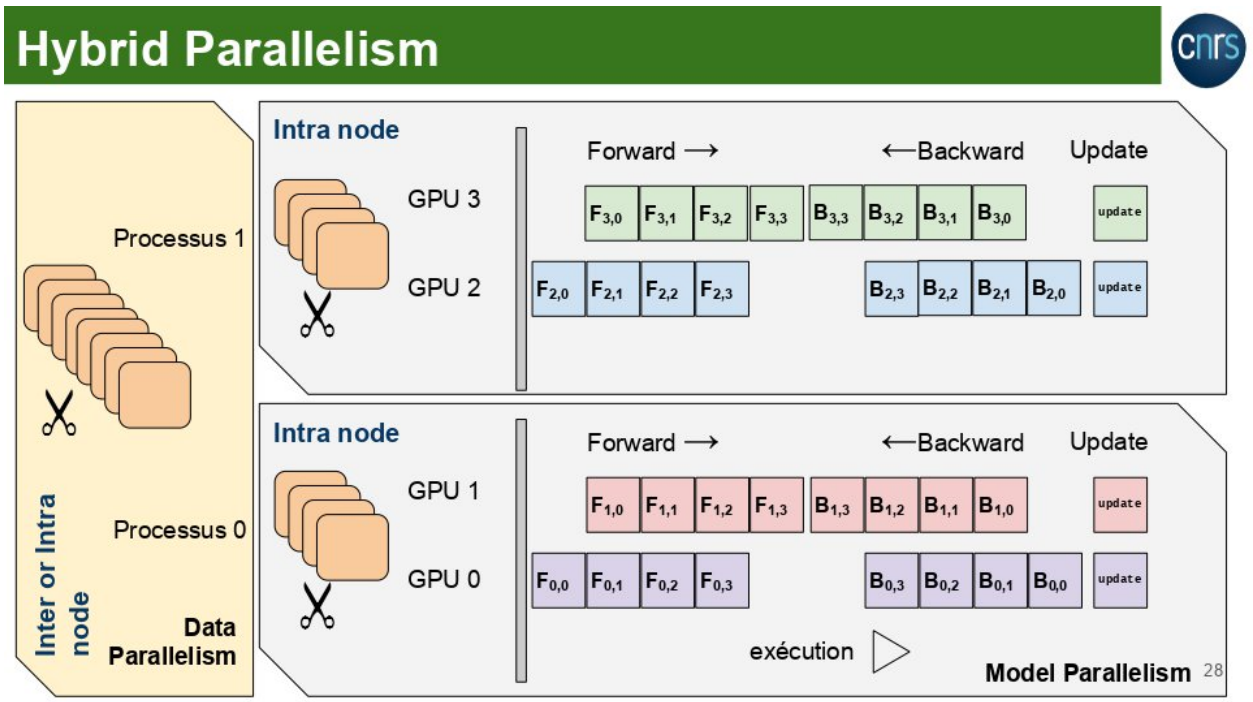


[4, 5]

Les optimisations possibles du *Pipeline Parallelism* s'organisent selon les 2 méthodes suivantes :

Le *Pipeline Parallelism* **synchrone** correspond au *Pipeline Parallelism* décrit dans la diapositive précédente. Une optimisation possible est de changer l'ordre d'exécution des *micro batches* afin de gagner en empreinte mémoire. En effet lorsque le *backward* est exécuté l'empreinte mémoire des sorties d'activation du *micro batch* peut être libérée.

Le *Pipeline Parallelism* **asynchrone** se permet de ne pas attendre la fin d'une itération d'apprentissage pour en commencer une nouvelle. Cela entraîne un calcul de la *Loss* parfois non avec le modèle dans sa version *N-1* mais avec le modèle dans sa version *N-2*. Ainsi, le processus de descente de gradient perd en qualité d'apprentissage, mais le parallélisme est accéléré au maximum puis qu'il n'y a quasiment plus de *bubble*. Alors, l'accélération est comparable au *Data Parallelism*. Cependant le *Pipeline* asynchrone est rarement utilisé.



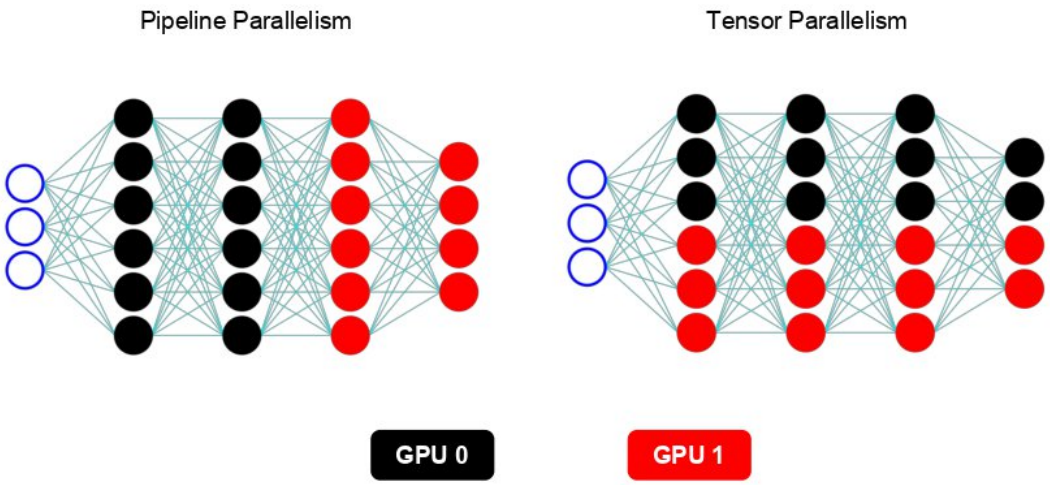
L'*Hybrid Parallelism* correspond à l'utilisation du *Data Parallelism* et du *Pipeline Parallelism* en même temps.

En effet, le *Data Parallelism* est le plus efficace pour accélérer l'apprentissage. Cependant un certain nombre de *Pipeline Parallelism* est nécessaire pour les gros modèles.

Un subtil dosage entre les 2 permettra d'optimiser au mieux le processus.

De plus si on utilise un grand nombre de GPU, le *Data Parallelism* risque de mener à des tailles de *batch* trop importantes. Ainsi, le *Pipeline Parallelism* permet d'équilibrer la taille du batch.

Two paradigms for model parallelism



29

Il existe deux manières de faire du parallélisme de modèles. Il y a la méthode en *Pipeline* présenté précédemment, qui consiste à découper le modèle en répartissant les couches sur différents GPU.

La méthode *Tensor* agit en découpant chacune des couches sur plusieurs GPU.

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

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

30

[22]

En Tensor Parallelism, il est possible de découper chaque couche de deux manières différentes : le long des lignes ou le long des colonnes.

Tensor Parallelism

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

Découpage par colonne

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

GPU 0 GPU 1

Découpage par ligne

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

31

[22]

Cette manière de séparer le tenseur de poids fait apparaître plusieurs sous-calculs qui peuvent être réalisés simultanément sur différents GPU. Puisque la sortie d'une couche est l'entrée de la couche suivante, et que cette dernière a besoin du vecteur complet, cela implique une communication entre les deux GPU pour mettre en commun leurs résultats respectifs.

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

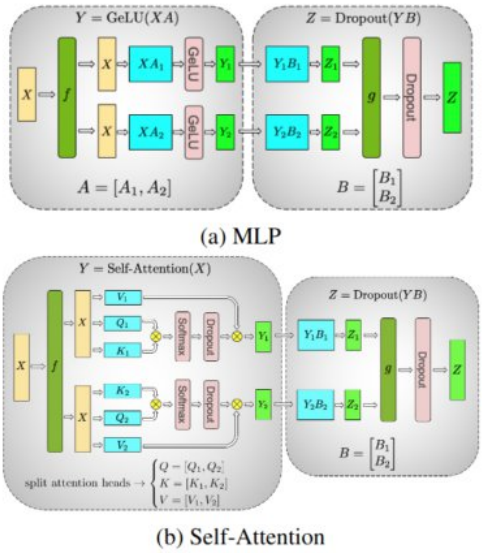
$$\mathbf{W} = \begin{pmatrix} \mathbf{W_1} \\ \mathbf{W_2} \end{pmatrix} \qquad \text{Linear}\left(\begin{pmatrix} \mathbf{X_1} & \mathbf{X_2} \end{pmatrix}\right) = \mathbf{X_1W_1} + \mathbf{X_2W_2}$$

AllReduce

[22]

Lors du découpage par colonne, on doit réaliser une communication AllGather pour concaténer les résultats des différents GPU. Lors du découpage par ligne, on doit faire un AllReduce pour faire la somme des résultats des différents GPU. Il est nécessaire de faire ces communications à chaque couche.

Tensor Parallelism



(a) MLP

(b) Self-Attention

[22]

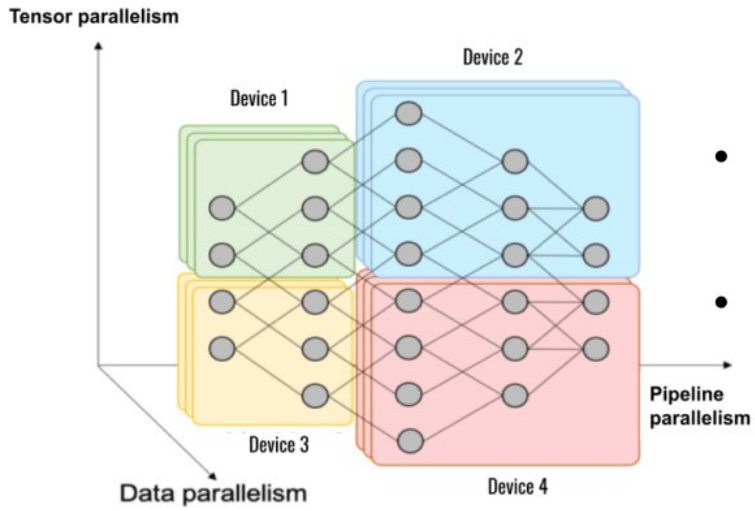
Par défaut, le tensor parallelism exige des synchronisations à **chaque** couche.

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

Puisque lors du découpage par ligne, chaque GPU n'a besoin en entrée que d'une sous-partie du vecteur d'entrée, et que le découpage par colonne provoque un découpage de la sortie, alors mettre une couche dense découpée selon les colonnes, suivie d'une couche dense découpée selon les lignes, alors on peut se permettre de ne faire qu'une opération AllReduce qu'à la sortie de la seconde couche. Cela enlève le besoin de communiquer après la première couche découpée selon les colonnes, on divise alors par deux le volume des communications exigées par le Tensor Parallelism.

Le même raisonnement est applicable avec le mécanisme d'attention, qui est encore plus fondamentalement partitionné.

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)

34

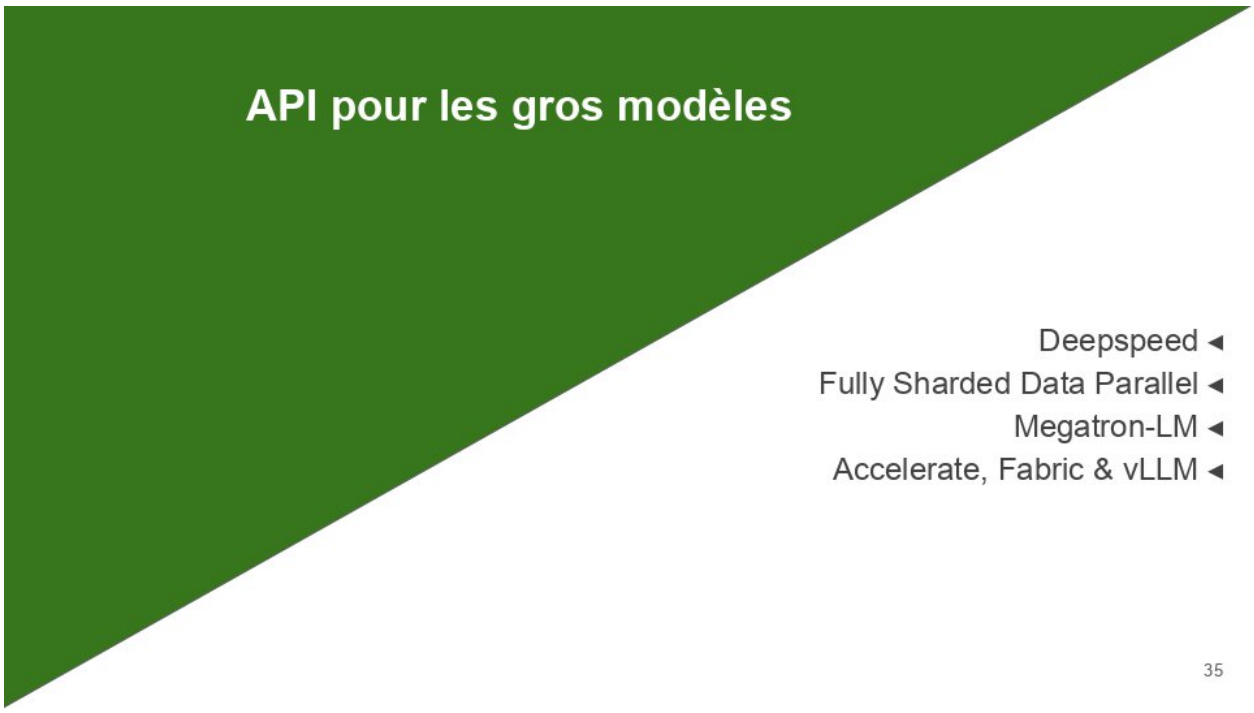
Le *3D Parallelism* distribue selon 3 axes :

- Le *Data Parallelism* selon l'axe des *batches*
- Le *Pipeline Parallelism* selon l'axe des couches
- Et, en plus, le *Tensor Parallelism* selon l'axe des nœuds des couches ou à l'intérieur des calculs des tenseurs propres à chaque couche.

Nous avons vu que le *Data Parallelism* est simple à implémenter car aujourd'hui tous les framework de Deep Learning intègre une solution de *Data Parallelism*, et est la meilleure solution pour accélérer le code. Cependant il tend vers des *batches* de plus en plus larges.

Nous avons vu que le *Pipeline Parallelism* permet tout en accélérant le code de distribuer l'empreinte mémoire liée au modèle. Il demande un certain effort d'implémentation ou est accessible via un ensemble de librairies dédiées au *Model Parallelism*.

Le *Tensor Parallelism* accessible seulement grâce à un effort conséquent d'implémentation ou grâce à des librairies très spécialisées (Megatron-LM) permet une économie de mémoire conséquente mais génère un flux de communication plus important entre les GPU. Il sera utilisé seulement entre les GPU d'un même nœud de calcul.



La complexité de la mise en œuvre du *Model Parallelism* et de l'apprentissage de gros modèles ou de l'accélération à très grande échelle que l'on vient de décrire est accessible grâce à certaines librairies dédiées.

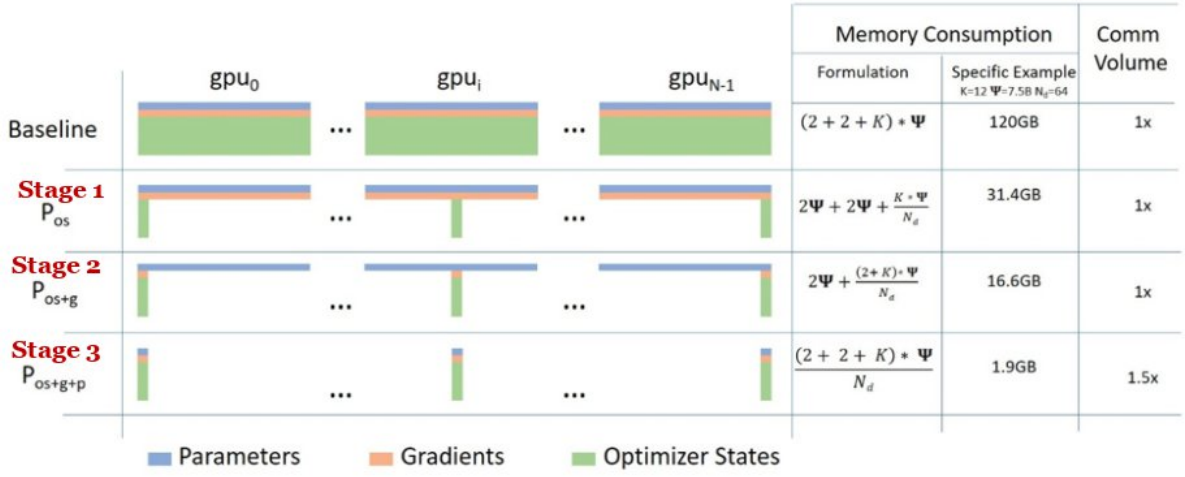
Cette partie décrit les librairies dédiées suivantes :

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

La librairie *Deepspeed* de Microsoft, dédiée à l'accélération des gros modèles et très gros modèles, propose un ensemble de techniques en PyTorch.

Deepspeed est assez facile à intégrer à un code PyTorch et possède une multitude de fonctions d'accélération et de parallélisme.

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

37

[6]

La fonctionnalité la plus intéressante de *Deepspeed* est ZeRO, soit *Zero Redundancy Optimizer*. ZeRO est une optimisation du *Data Parallelism* pour les gros modèles.

Il permet d'économiser l'empreinte mémoire liée au modèle en utilisant le *Data Parallelism*. Le *Data Parallelism* recopie dans chaque GPU entièrement les variables liées au modèle (poids, gradients, l'historique de l'*optimizer*).

ZeRO permet de partager l'empreinte mémoire liée au modèle. Chaque GPU conserve une portion différente de l'empreinte du modèle. Lorsqu'un GPU a besoin d'une partie du modèle qu'elle ne possède pas, le GPU détenteur de cette portion la communique au GPU en question. Ainsi, ZeRO fonctionne exactement comme le *Data Parallelism* en distribuant selon l'axe des *batches* tout en distribuant le stockage des informations liées au modèle.

Il existe 3 "stage" de ZeRO:

- Le stage 1 qui distribue la partie *Optimizer* équivalent au *Data Parallelism* en terme de communication inter GPU.
- Le stage 2 qui distribue les parties *Optimizer* et gradient équivalent au *Data Parallelism* en terme de communication inter GPU.
- Le stage 3 qui distribue les parties *Optimizer*, gradient et poids du modèle plus coûteux de 50% en terme de communication.

Il existe aussi des optimisations de ZeRO : *ZeRO Offload* et *ZeRO Infinity* qui utilisent la mémoire CPU pour distribuer des modèles encore plus gros de plus de 1000 milliards de paramètres correspondant aux modèles du futur.

Fused optimizers

Implémentations présentent dans *APEX*

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

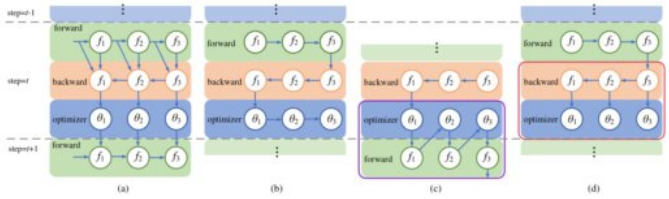
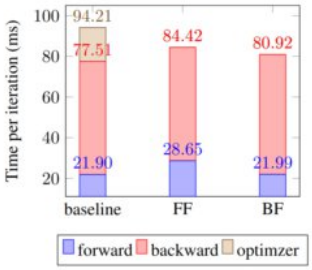


Figure 1: (a) Data dependency graph. (b) Baseline method. (c) `Forward-fusion`. (d) `Backward-fusion`. θ_i represents the trainable parameters in the layer f_i.



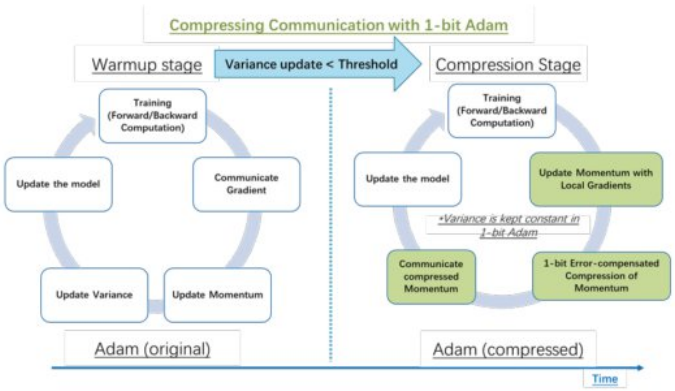
But : accélère l'étape des optimiseurs sur GPU.

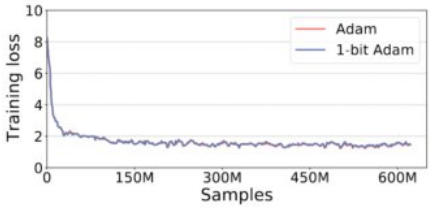
[7]

Deepspeed intègre aussi les *optimizer* dit fusionnés proposés par APEX.

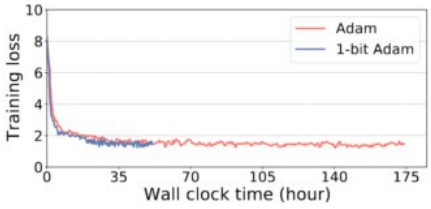
Il fusionne des ***kernels* GPU** pour économiser les opérations d'initialisation et de lecture / écriture de mémoire, ce qui engendre une légère accélération des *optimizer*.

1-bit optimizers





(a) Sample-wise



(b) Time-wise

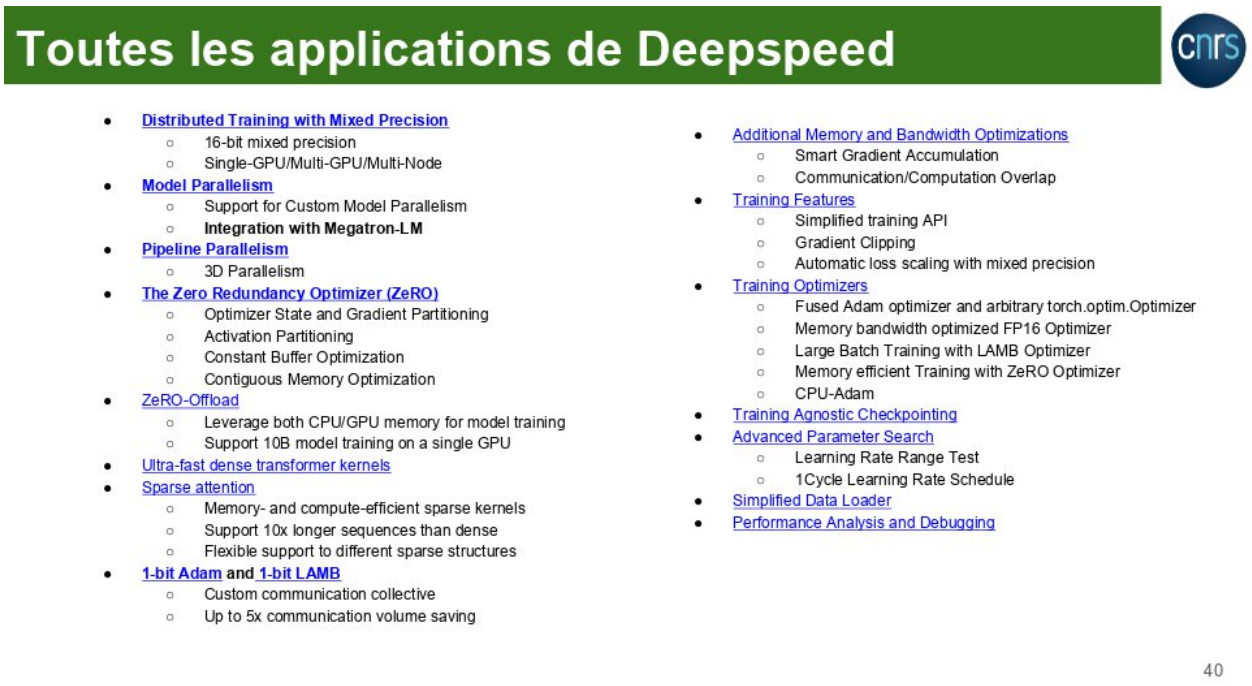
But : diminuent les communications nécessaires et donc accélère l'étape des optimiseurs pour un modèle distribué.

[8, 9]

Deepspeed propose aussi des *optimizer* dit *One-bit* qui diminuent les communications nécessaires et donc accélèrent l'étape des optimiseurs pour un modèle distribué en *Data Parallelism*.

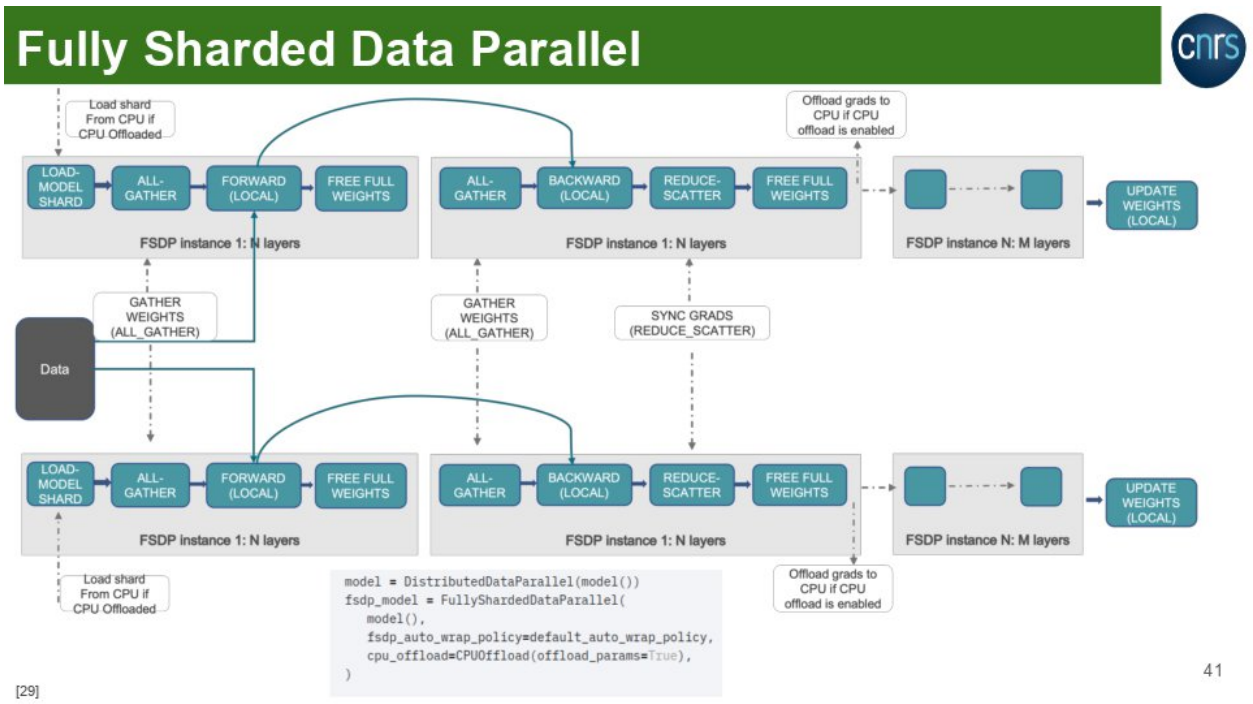
Le principe, par exemple du 1-*bit Adam* est de calculer le *momentum* seulement localement, puis d'échanger seulement une information codée sur 1 bit de divergence entre les *momentum* locaux. La variance, étant non linéaire, ne peut être traitée de la même façon. Sur le constat empirique que la variance rapidement devient quasiment stable, le 1-bit Adam, après un *warm-up stage* correspondant à un Adam classique pendant quelques *epoch*, bascule dans son mode compressé où la variance est fixée à la dernière valeur calculée.

Cela permet une réduction des communications de l'ordre de 97%. Pour un gros modèle déployé sur un nombre important de GPU en *Data Parallelism*, on observe une réduction de temps d'un facteur 3 pour une *accuracy* quasiment identique.



Deespeed propose en plus de ZeRO, des *Fused optimizer* et des *1-bit optimize*r un nombre important d'autres applications pour l'accélération :

- Le *Pipeline Parallelism*
- L'intégration de *Megatron-LM* permettant d'utiliser le *3D Parallelism* couplé avec ZeRO
- Le *0/1 Adam* qui est une optimisation du *1-bit Adam*
- Le paramétrage de la mémoire et des *buffers*
- Le *Sparse Attention*
- etc

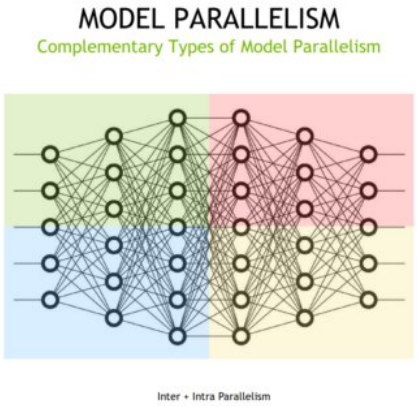


Fully Sharded Data Parallel (FSDP) est une optimisation du parallélisme de données qui est très fortement inspirée de ZeRO Stage 3. Tous les GPU ne retiennent pas en mémoire tous les poids. Contrairement à ZeRO-3 qui répartit les couches sur les différents GPUs (répartition analogue à du *Pipeline Parallelism*), FSDP découpe chaque couche sur les différents GPUs (répartition analogue à du *Tensor Parallelism*). Il est important de noter que la FSDP (ainsi que ZeRO-3) ne font pas du parallélisme de modèle. Les mêmes poids sont utilisés sur tous les GPUs, c'est simplement qu'ils ne sont pas recopiés sur tous les processus, il faut donc réaliser des communications pour chaque GPU puisse faire la *forward* puis la *backward* correctement.

La méthode FSDP a l'avantage d'être incluse nativement dans PyTorch et on n'a plus besoin de faire appel à *FairScale* comme ça a été le cas par le passé. Son utilisation est relativement simple, très semblable à de la *DDP* classique (on peut cependant rajouter quelques optimisations supplémentaires). L'expérience montre que l'on peut entraîner de très gros LLM simplement avec FSDP et sans introduire d'autres méthodes de parallélisation. Selon la configuration et les optimisations utilisées, les performances peuvent même être supérieures à celle de DeepSpeed ZeRO-3.

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 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.

NVIDIA développe *Megatron-LM* qui permet de gérer un *3D Parallelism* clef en main utilisant la *Mixed Precision* pour les plus célèbres architectures de *Transformer* comme GPT, BERT, T5 avec des GPU NVIDIA.

Megatron-LM est un apport majeur pour l'apprentissage des plus gros modèles actuels de Transformer. Les versions plus récentes de *Megatron-LM* ont également introduit le *Sequence Parallelism*, une méthode pour partitionner les activations dans le sous-groupe de GPU pour les couches où le *Tensor Parallelism* n'intervient pas (par exemple les *LayerNorm*).

Fabric & Accelerate

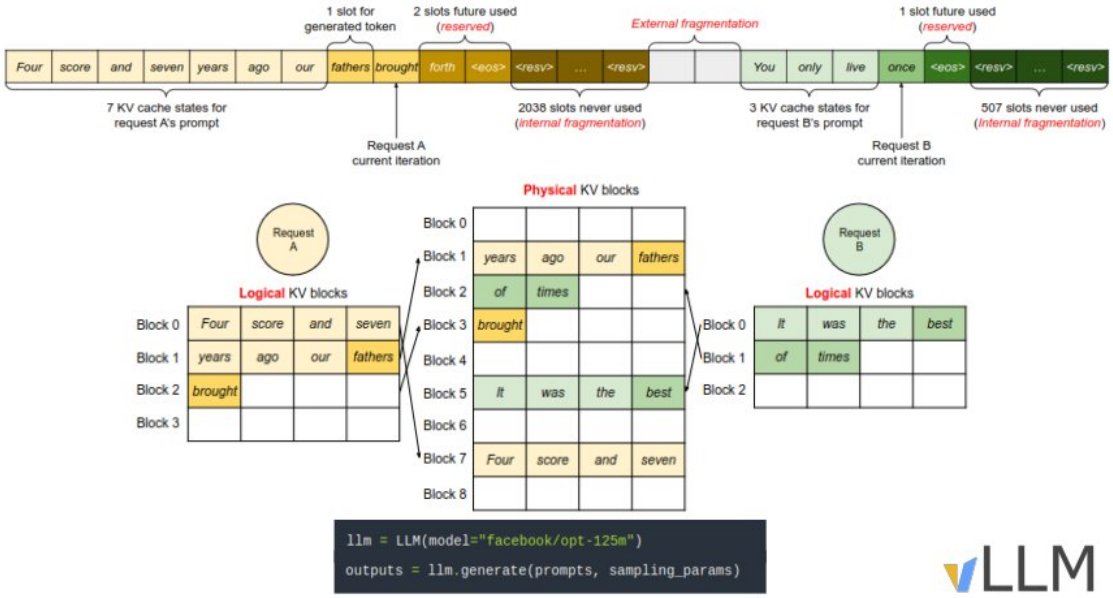
[17, 28]

Accelerate est une bibliothèque développée par HuggingFace qui a pour objectif de simplifier l'utilisation de nombreuses optimisations telles que la parallélisation, la précision mixte, etc. Le développement d'*Accelerate* est très dynamique et implémente les découvertes les plus récentes assez rapidement. *Accelerate* permet d'utiliser *Megatron-LM*, *DeepSpeed*, etc. Lors de tests sur Jean Zay, on s'est rendu compte que l'utilisation d'*Accelerate* sur plusieurs noeuds est possible mais fastidieuse car elle requiert un fichier de configuration par noeud. C'est pourquoi nous avons développé *idr_accelerate*, un wrapper de la commande *accelerate* qui gère ces fichiers de configuration de manière transparente.

Son utilisation est la suivante : srun (ou torchrun) idr_accelerate <options pour accelerate telles que config_file> fichier_train.py <options pour le fichier d'entraînement>.

Lightning Fabric est une version allégée de Pytorch Lightning qui est moins intrusive dans le code et moins obscure. Elle permet également d'utiliser plusieurs optimisations en modifiant peu le code d'entraînement.

vLLM (Inférence des transformers)



[16]

L'inférence sur les LLM est un sujet devenu très important puisque de nos jours, en plus d'être la force derrière certaines API accessibles en ligne, de nombreux chercheurs sont amenés à en faire sur supercalculateurs pour étudier le comportement de certains LLM.

Puisque le mécanisme d'attention ne s'applique pas indépendamment sur chaque token, il est important de garder la *Key* et la *Value* des mots précédants pour calculer les scores d'attention plus rapidement, ce qui est très gourmand en mémoire.

Le fait que les LLM génèrent des séquences de taille variable, que chaque mot doit être généré séquentiellement, et que certaines méthodes (par exemple Beam Search) peuvent impliquer la génération de plusieurs mots à un même prompt pousse à une utilisation déraisonnable et irresponsable de la mémoire du GPU. Une grande partie de la mémoire réservée n'est pas réellement exploitée et cela limite fortement la taille du batch de manière injustifiée.

Pour cela vLLM introduit l'attention paginée. Plutôt que de stocker toute une séquence de manière contigüe en mémoire, quitte à trop en réserver, on rassemble les tokens par petits groupes. Les tokens d'un groupe sont contigus en mémoire mais différents groupes ne sont pas contigus dans la mémoire physique. Puisque les kernels CUDA de deep learning exige généralement une contiguïté mémoire pour réaliser leurs opérations, vLLM reproduit le comportement d'un OS sur CPU via l'introduction d'une mémoire logique et de pages sur le GPU. On peut même réutiliser la mémoire en pointant plusieurs blocs logiques vers un même bloc physique. Ces optimisations permettent de limiter le gaspillage et de démultiplier le batch size possible.

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/