

Attention residual

Arxiv Tech report - MoonshortAI's Kimi Team

Bastien DOIGNIES - GDL. LIRIS

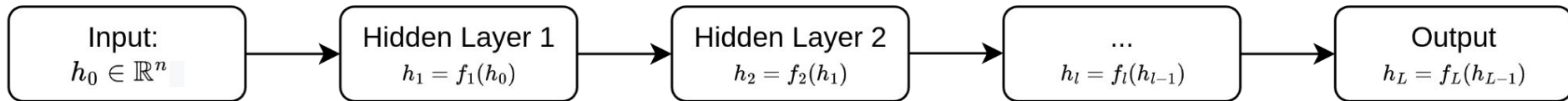
Old problem: vanishing gradient

Exemple (Dense)

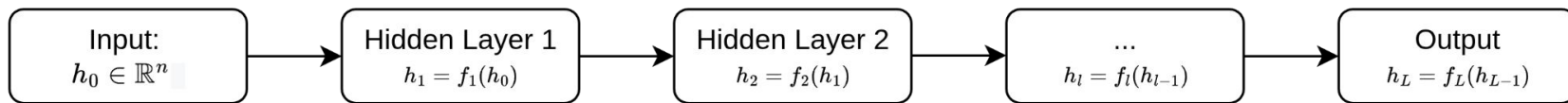
$$f_k(x) = \sigma(W_k x + b_k)$$

Neural networks

A classical neural network is a chain of hidden layers. Early versions use dense or convolutional layers followed by activation functions to enhance the expressive power of the architecture.



Old problem: vanishing gradient



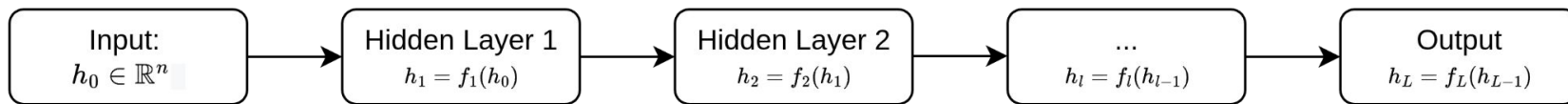
The gradient of the loss (assuming dense layers) w.r.t to first bias:

$$\frac{\partial \mathcal{L}}{\partial b_1} = \frac{\partial \mathcal{L}}{\partial h_L} \frac{\partial h_L}{\partial h_{L-1}} \cdots \frac{\partial h_1}{\partial b_1}$$

But we have (for sigmoids) :

$$\frac{\partial h_l}{\partial h_{l-1}} \propto \sigma'(h_{l-1}) \leq \frac{1}{2}$$

Old problem: vanishing gradient

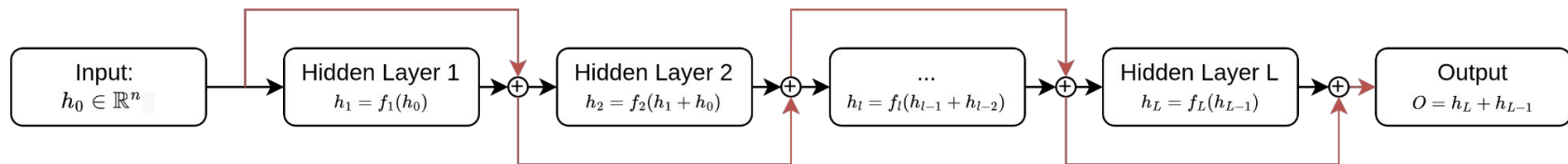


Hence,

$$\frac{\partial \mathcal{L}}{\partial b_1} \propto \prod_{l=1}^L \sigma'(h_l) \leq \frac{1}{2^L}$$

Or: the deeper the network, the smaller the magnitude of the gradient. Hence, the first layers receive very little gradient updates.

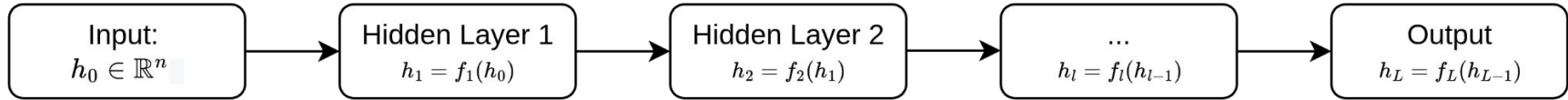
Solution: ResNet [He 2015]



The idea is to add the output of a layer to the output of the previous layer (i.e. its input). This allows the gradient to flow more naturally backward through the network (via the identity term in the gradient equation):

$$\frac{\partial \mathcal{L}}{\partial h_l} = \frac{\partial \mathcal{L}}{\partial h_L} \prod_{i=l}^{L-1} \left(I + \frac{\partial f_i}{\partial h_i} \right)$$

Another old problem: input magnitude change



TL;DR: the output magnitude of a layer changes during training, and subsequent layers must adapt to this change (“in addition to moving in their own correction direction”).

Solution(s): Normalization [Ioffe 2015]

The issue is solved by introducing a “normalization layer,” which ensures that the input remains within a “fixed” range. Essentially, it subtracts the mean and divides by the variance.

Method	Normalizing axis	Typical models	Ref
BatchNorm	Batch	CNNs	[Ioffe 2015]
LayerNorm RMSNorm	Features	Transformers (Claude, GPT, Mistral)	[Lei Ba 2016] [Zhang 2019]
InstanceNorm	Spatial per channel	Style transfers (StyleGAN)	[Ulyanov 2016]
GroupNorm	By channels	Detection/Segmentation/Diffusion	[Wu 2018]

Now: what happens with both?

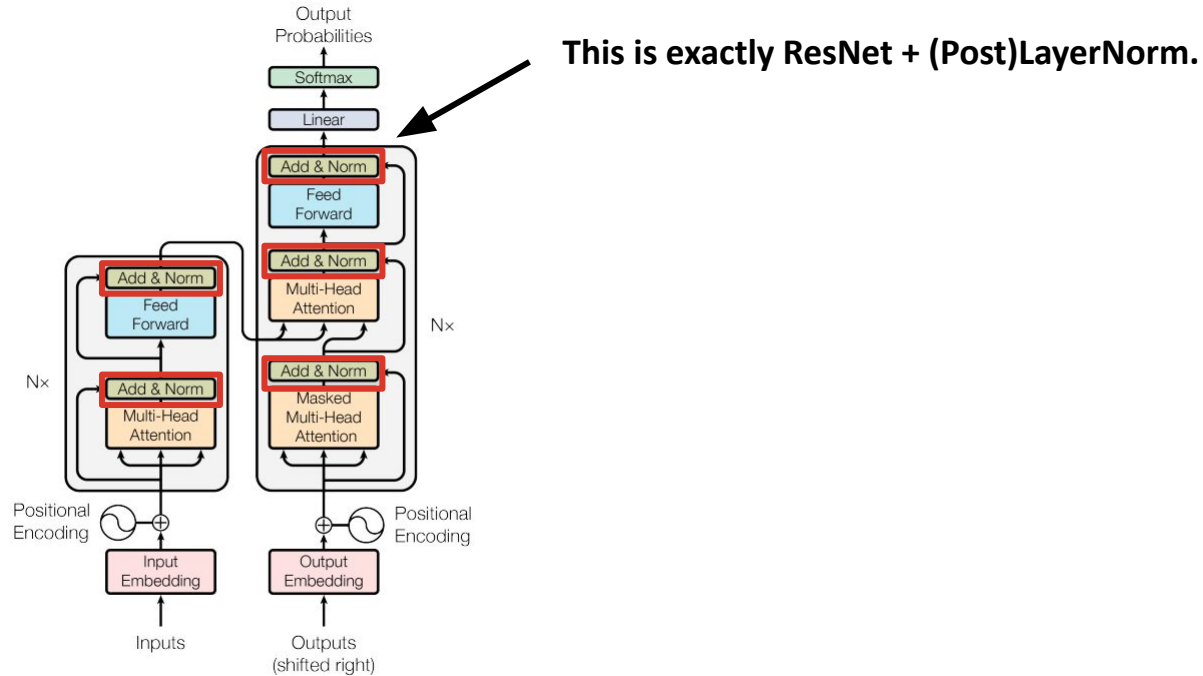
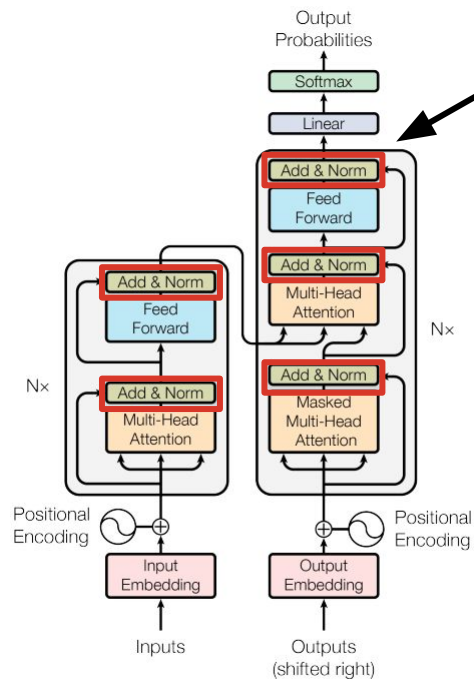


Figure 1: The Transformer - model architecture.

[Vaswani 2017]

Now: what happens with both?



This is exactly ResNet + (Post)LayerNorm.

The gradient through the ResNet is given by:

$$\frac{\partial \mathcal{L}}{\partial h_l} = \frac{\partial \mathcal{L}}{\partial h_L} \prod_{i=l}^{L-1} \left(I + \frac{\partial f_i}{\partial h_i} \right)$$

Constrained by normalization

Figure 1: The Transformer - model architecture.

[Vaswani 2017]

Now: what happens with both?

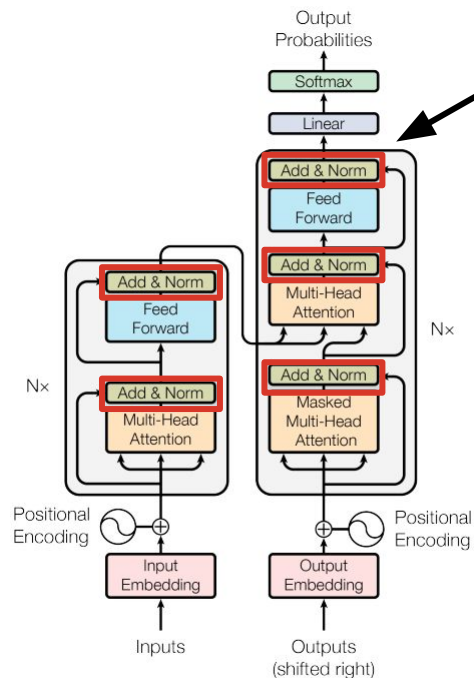


Figure 1: The Transformer - model architecture.

[Vaswani 2017]

This is exactly ResNet + (Post)LayerNorm.

The gradient through the ResNet is given by:

$$\frac{\partial \mathcal{L}}{\partial h_l} = \frac{\partial \mathcal{L}}{\partial h_L} \prod_{i=l}^{L-1} \left(I + \frac{\partial f_i}{\partial h_i} \right)$$

The product of $(1+x)$ tends to grow...

Constrained by normalization

The normalization reverses the old problem: now early layer tend to receive more gradient information...

Partial Solution: Highway networks [Srivastava 2015]

Note: the paper is one month older than ResNet.

Instead of simply adding the previous layer's output, why not perform a learned interpolation between it and the new layer's output?

$$h_l = g_l h_{l-1} + (1 - g_l) f_l(h_{l-1})$$

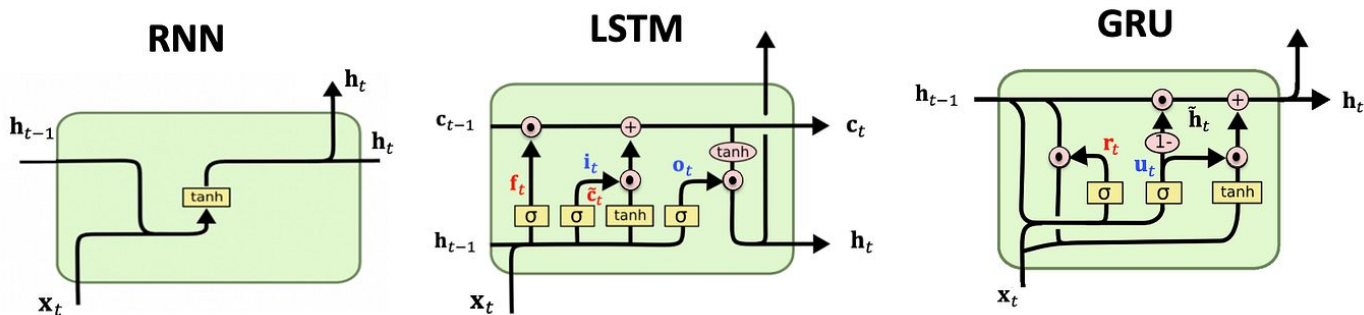
This does not totally solve the issue, but mitigates it.

Another problem with ResNet (not solved):

One fundamental flaw of ResNets is that they only look at the previous vector. In some sense, they compress the information into a single vector, which is highly lossy.

Another problem with ResNet (not solved):

One fundamental flaw of ResNets is that they only look at the previous vector. In some sense, they compress the information into a single vector, which is highly lossy. This is reminiscent of RNN/LSTM networks, which suffered from memory loss as sequences became longer and longer.



Solution (as always): Attention

Let the network attend to previous layers and compute the appropriate weights for them:

$$\mathbf{h}_l = \alpha_{0 \rightarrow l} \cdot \mathbf{h}_1 + \sum_{i=1}^{l-1} \alpha_{i \rightarrow l} \cdot f_i(\mathbf{h}_i)$$

Solution (as always): Attention

The attention weights can be written as $\alpha_{i \rightarrow l} = \phi(\mathbf{q}_l, \mathbf{k}_i)$ for a kernel function $\phi: \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}_{\geq 0}$, where $\mathbf{q}_l$ and $\mathbf{k}_i$ are query and key vectors [23, 70]. Different choices of ϕ recover different residual variants (§6.2); we adopt $\phi(\mathbf{q}, \mathbf{k}) = \exp(\mathbf{q}^\top \text{RMSNorm}(\mathbf{k}))$ [66] with normalization, yielding softmax attention over depth:

$$\alpha_{i \rightarrow l} = \frac{\phi(\mathbf{q}_l, \mathbf{k}_i)}{\sum_{j=0}^{l-1} \phi(\mathbf{q}_l, \mathbf{k}_j)} \quad (2)$$

For each layer l , we define:

$$\mathbf{q}_l = \mathbf{w}_l, \quad \mathbf{k}_i = \mathbf{v}_i = \begin{cases} \mathbf{h}_1 & i = 0 \\ f_i(\mathbf{h}_i) & 1 \leq i \leq l-1 \end{cases} \quad (3)$$

where the query $\mathbf{q}_l = \mathbf{w}_l$ is a layer-specific learnable vector in $\mathbb{R}^d$. The RMSNorm inside ϕ prevents layers with large-magnitude outputs from dominating the attention weights. The input to layer l is then:

$$\mathbf{h}_l = \sum_{i=0}^{l-1} \alpha_{i \rightarrow l} \cdot \mathbf{v}_i \quad (4)$$

We call this form *full attention residuals*. For each token, Full AttnRes requires $O(L^2 d)$ arithmetic and $O(Ld)$ memory to store layer outputs. Since depth is far smaller than sequence length, the arithmetic cost is modest.

Block Attention residual

3.2 Block Attention Residuals

We propose *Block Attention Residuals*, which partitions the L layers into N blocks: within each block, the layer outputs are reduced to a single representation via summation, and across blocks, we apply full attention over only N block-level representations and the token embedding. This reduces both memory and communication overhead from $O(Ld)$ to $O(Nd)$.

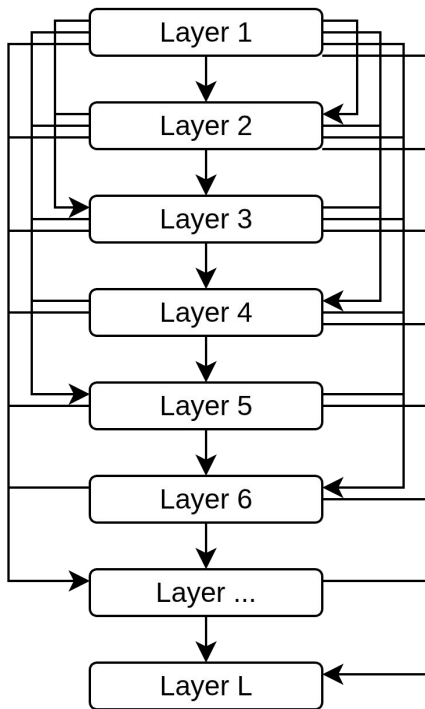
Intra-Block Accumulation. Specifically, we divide the L layers into N blocks of $S = L/N$ layers each, assuming L is divisible by N ; otherwise, the last block contains the remaining $L \bmod N$ layers. Let $\mathcal{B}_n$ denote the set of layer indices in block n ($n = 1, \dots, N$). To form a block, we sum all of its layer outputs:

$$\mathbf{b}_n = \sum_{j \in \mathcal{B}_n} f_j(\mathbf{h}_j) \quad (5)$$

Inter-Block Attention. In Full AttnRes, the input to layer l is computed by attending over all outputs up to $f_{l-1}(\mathbf{h}_{l-1})$. The block-wise variant replaces these individual outputs with block representations, defining $\mathbf{b}_0 = \mathbf{h}_1$ so that the token embedding is always included as a source. For the i -th layer in block n , the value matrix is:

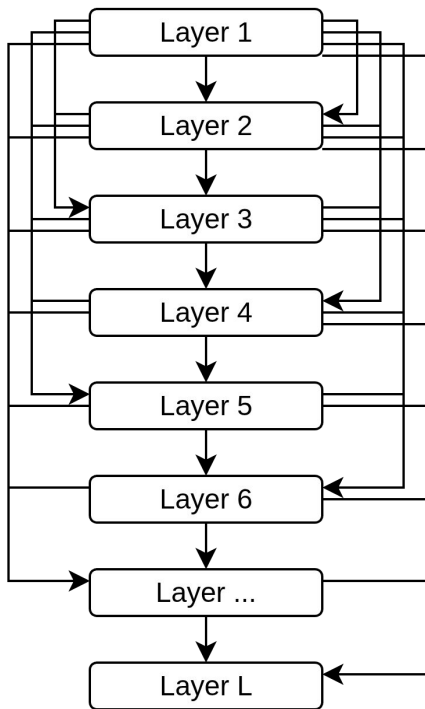
$$\mathbf{V} = \begin{cases} [\mathbf{b}_0, \mathbf{b}_1, \dots, \mathbf{b}_{n-1}]^\top & \text{if } i = 1 \text{ (first layer of block } n) \\ [\mathbf{b}_0, \mathbf{b}_1, \dots, \mathbf{b}_{n-1}, \mathbf{b}_n^{i-1}]^\top & \text{if } i \geq 2 \text{ (subsequent layers)} \end{cases} \quad (6)$$

Block Attention residual

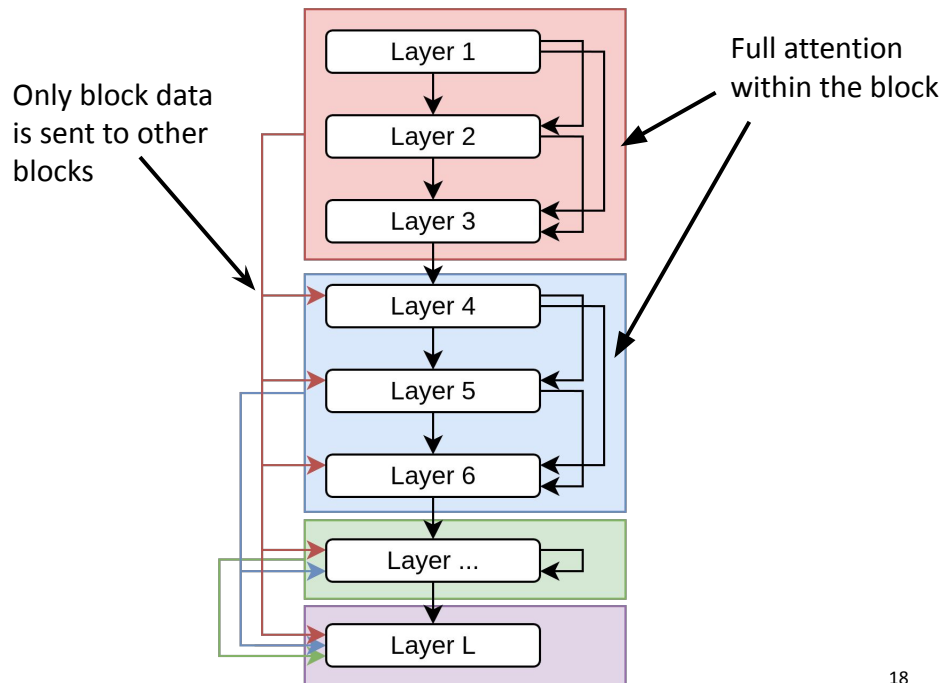


Full attention residual connections

Block Attention residual



Full attention residual connections

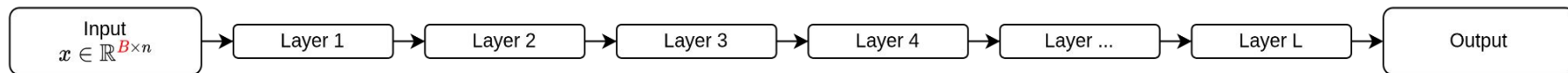


Block attention residuals

Very big model problems (ie the one we can't afford to train in public research)

How to train a model on multiple GPUs?

- Data-Parallelization?



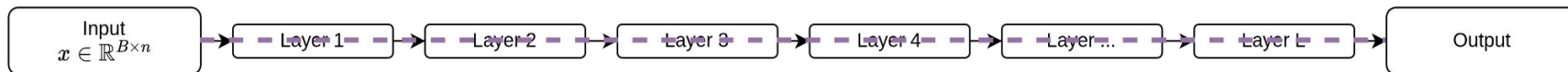
- Requires the model to fit within each GPU's memory (which is non-trivial for billion-parameter networks).
- Requires aggregating gradients and results.
- Requires updating weights (i.e. applying gradients).

Overall: wastes a lot of compute, either by duplicating computation or by spending excessive time moving data around.

Very big model problems (ie the one we can't afford to train in public research)

How to train a model on multiple GPUs?

- ~~Data-Parallelization~~
- Tensor-Parallelization?



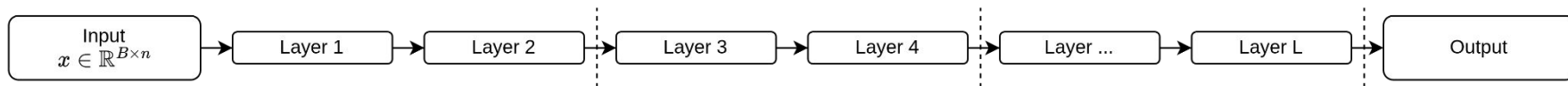
- Requires transferring and aggregating results.
- Involves a large amount of communication between GPUs.

Overall: it could be a good approach for a small number of very large layers that do not fit in memory, especially on a small number of devices. However, when training on 1024 A100 GPUs for a month (the reported setup for GPT-3 according to NVIDIA), communication overhead would become predominant.

Very big model problems (ie the one we can't afford to train in public research)

How to train a model on multiple GPUs?

- ~~Data Parallelization~~
- ~~Tensor Parallelization~~
- Pipeline parallelization?



- Each GPU sequentially processes a small part of the pipeline (e.g. layers 1 & 2).
- Wastes time because of pipeline bubbles (when GPUs are inactive).
- Requires at least one layer to fit on a single GPU (which people do run into in practice...).
Load imbalance: some GPUs may become bottlenecks depending on the layers they have to run, preventing efficient sharding...

Overall: also not ideal when wasting compute resources can literally cost millions per hour, especially when you've decided that trillions of parameters are what you want...

Solution: Data-Tensor-Pipeline parallelization [Narayanan 2021]

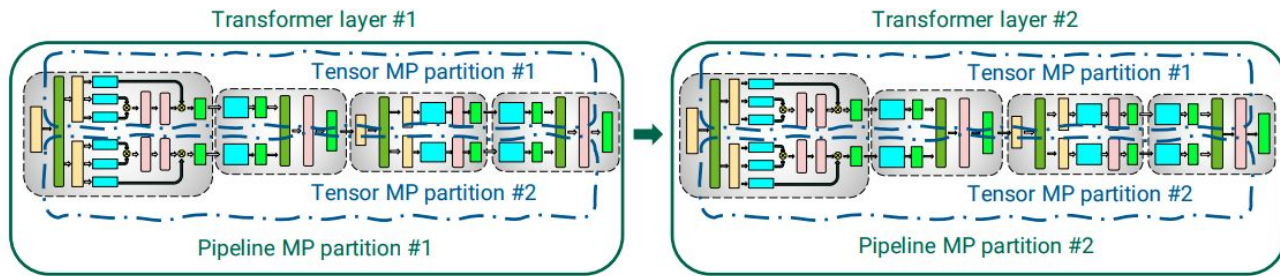
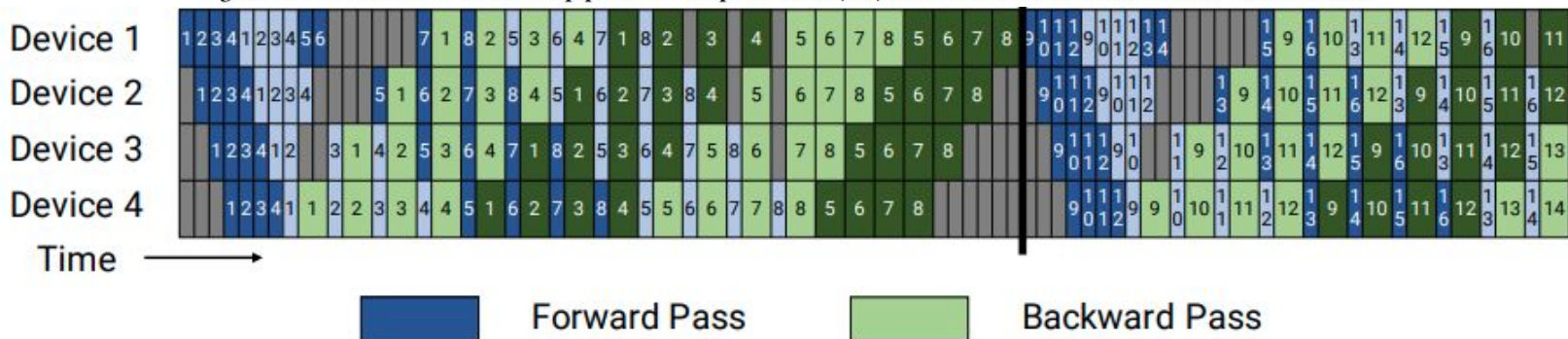


Figure 2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.



As a side note: 8 microbatches for a 16-way pipeline with 8-way tensor parallelism = 1024 GPUs :)! Btw, this means that each layer is computed across 64 GPUs... GPT-4 is rumored to have used between 10k and 25k GPUs in parallel.

Coming back to our problem: we need to send the block residual attention to corresponding GPUs

TLDR: As each GPU compute one part of the pipeline we need not to send the full attention history but only what was not present in the GPU. Essentially the memory and transfer cost is very low especially when compared to layer outputs.

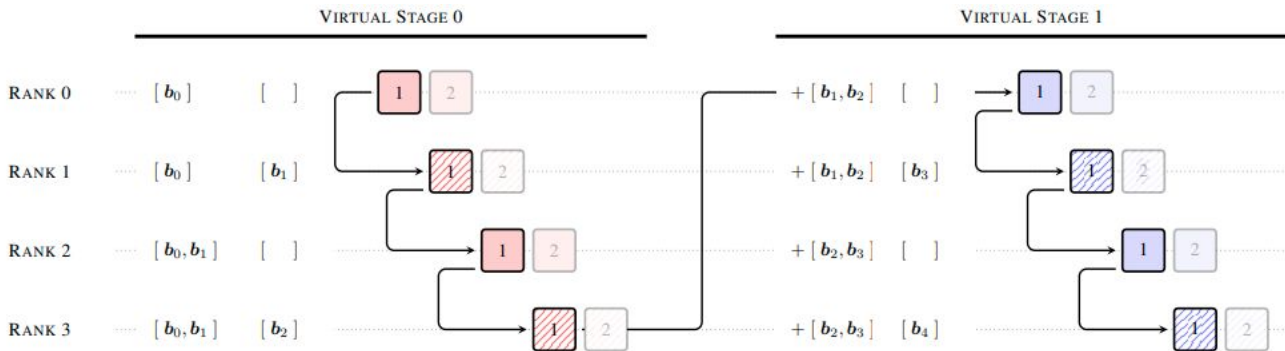
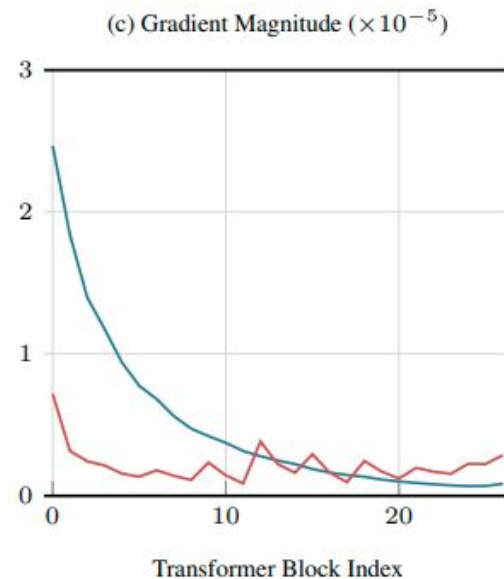
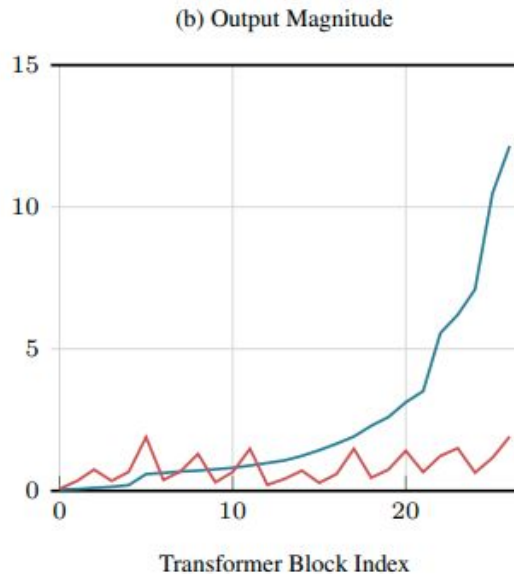
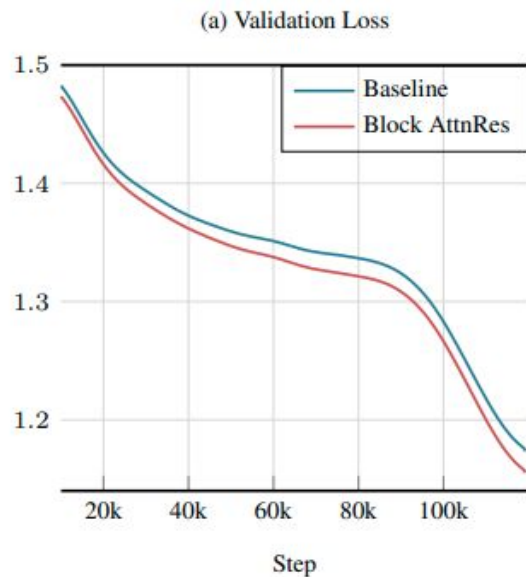
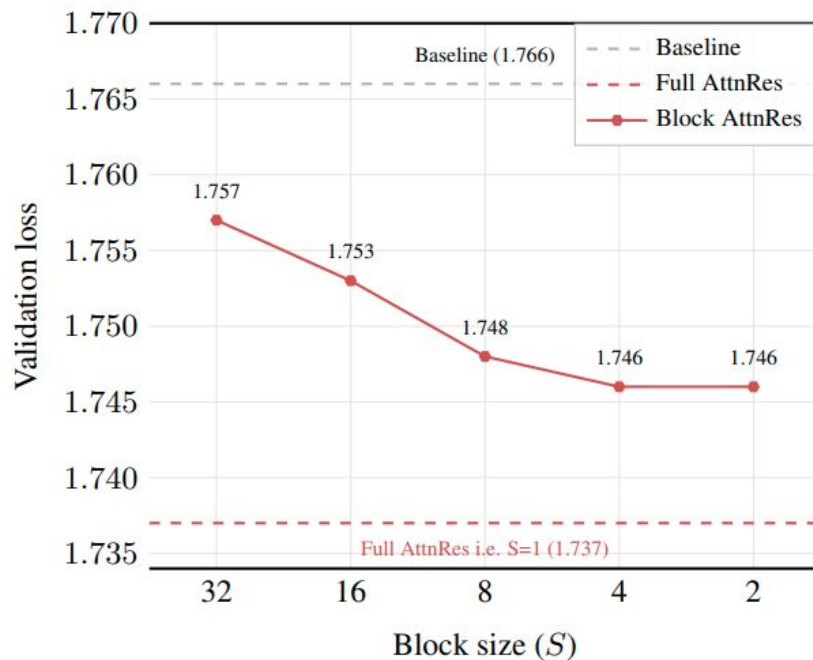
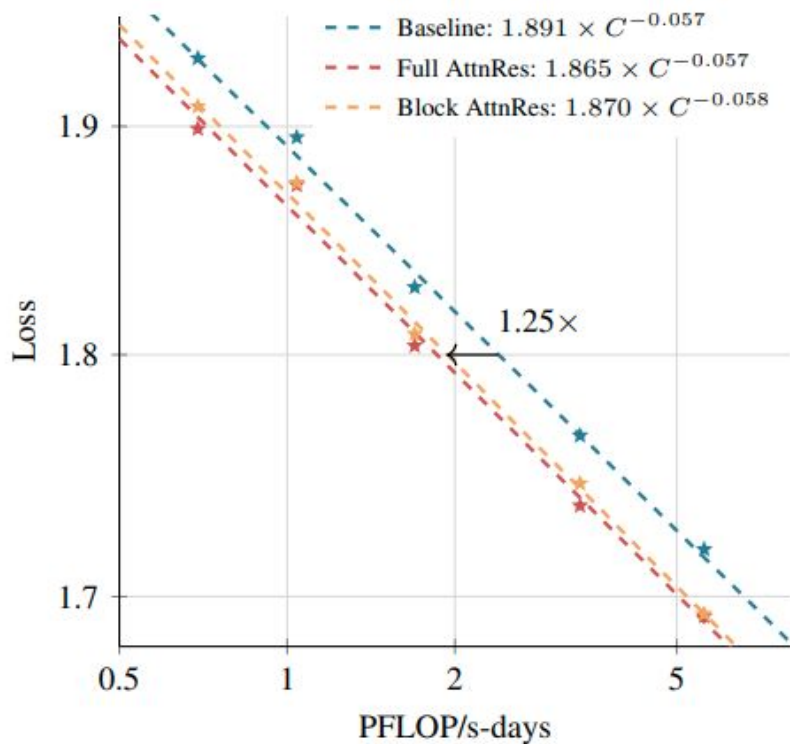


Figure 3: Cache-based pipeline communication example with 4 physical ranks and 2 virtual stages per rank, where hatched boxes denote end of AttnRes blocks. Numbers indicate micro-batch indices. Each rank caches previously received blocks; stage transitions only transmit incremental blocks ($+ [b_1, b_2]$) instead of the full history.

Some results :)



Some results :)



FIN