CMSC – 35200
Deep Learning Systems

Ian Foster and Rick Stevens

Deep Learning Systems 2023: Generative Models for Science

Scaling up LLMs

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

Scaling up LLMs

Language Models are Unsupervised Multitask Learners

Alec Radford [*1] **Jeffrey Wu** [*1] **Rewon Child** [1] **David Luan** [1] **Dario Amodei** [**1] **Ilya Sutskever** [**1]

Abstract

Natural language processing tasks, such as question answering, machine translation, reading comprehension, and summarization, are typically approached with supervised learning on task-specific datasets. We demonstrate that language models begin to learn these tasks without any explicit supervision when trained on a new dataset of millions of webpages called WebText. When conditioned on a document plus questions, the answers generated by the language model reach 55 F1 on the CoQA dataset - matching or exceeding the performance of 3 out of 4 baseline systems without using the 127,000+ training examples. The capacity of the language model is essential to the success of zero-shot task transfer and increasing it improves performance in a log-linear fashion across tasks. Our largest model, GPT-2, is a 1.5B parameter Transformer that achieves state of the art results on 7 out of 8 tested language modeling datasets in a zero-shot setting but still underfits WebText. Samples from the model reflect these improvements and contain coherent paragraphs of text. These findings suggest a promising path towards building language processing systems which learn to perform tasks from their naturally occurring demonstrations.

competent generalists. We would like to move towards more general systems which can perform many tasks – eventually without the need to manually create and label a training dataset for each one.

The dominant approach to creating ML systems is to collect a dataset of training examples demonstrating correct behavior for a desired task, train a system to imitate these behaviors, and then test its performance on independent and identically distributed (IID) held-out examples. This has served well to make progress on narrow experts. But the often erratic behavior of captioning models (Lake et al., 2017), reading comprehension systems (Jia & Liang, 2017), and image classifiers (Alcorn et al., 2018) on the diversity and variety of possible inputs highlights some of the shortcomings of this approach.

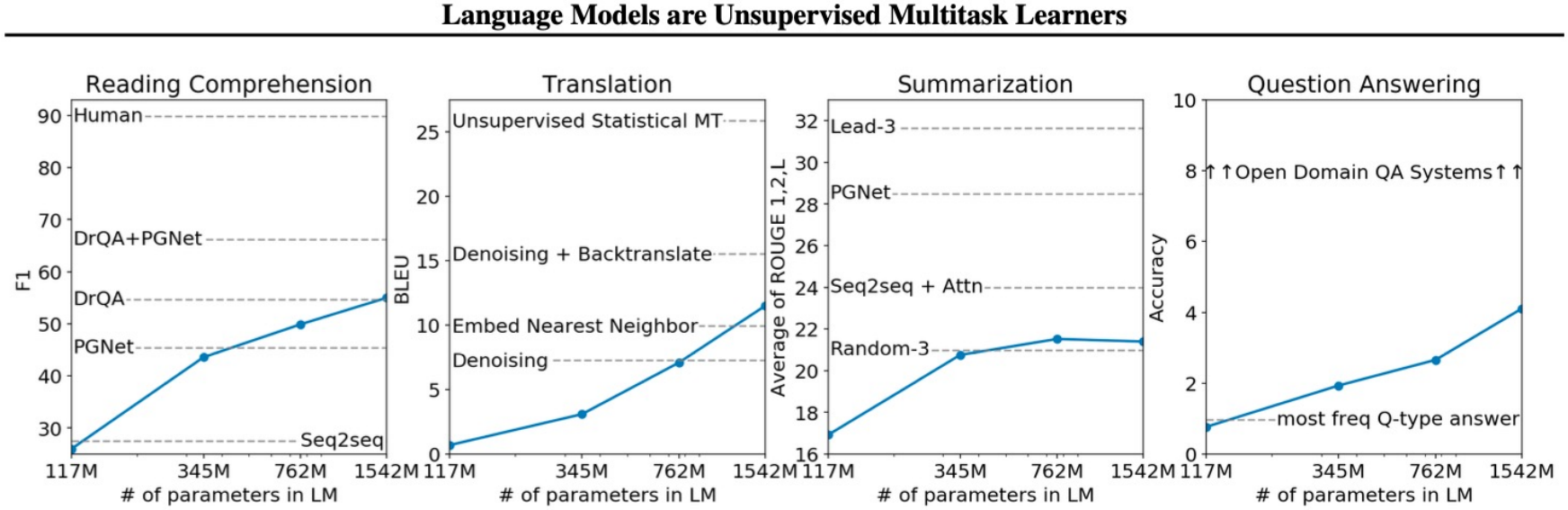
Our suspicion is that the prevalence of single task training on single domain datasets is a major contributor to the lack of generalization observed in current systems. Progress towards robust systems with current architectures is likely to require training and measuring performance on a wide range of domains and tasks. Recently, several benchmarks have been proposed such as GLUE (Wang et al., 2018) and decaNLP (McCann et al., 2018) to begin studying this.

Multitask learning (Caruana, 1997) is a promising framework for improving general performance. However, multitask training in NLP is still nascent. Recent work reports modest performance improvements (Yogatama et al.,

GPT-2 Models

Parameters	Layers	d_{model}
117M	12	768
345M	24	1024
762M	36	1280
1542M	48	1600

GPT-2 Skill as a function of #Parameters



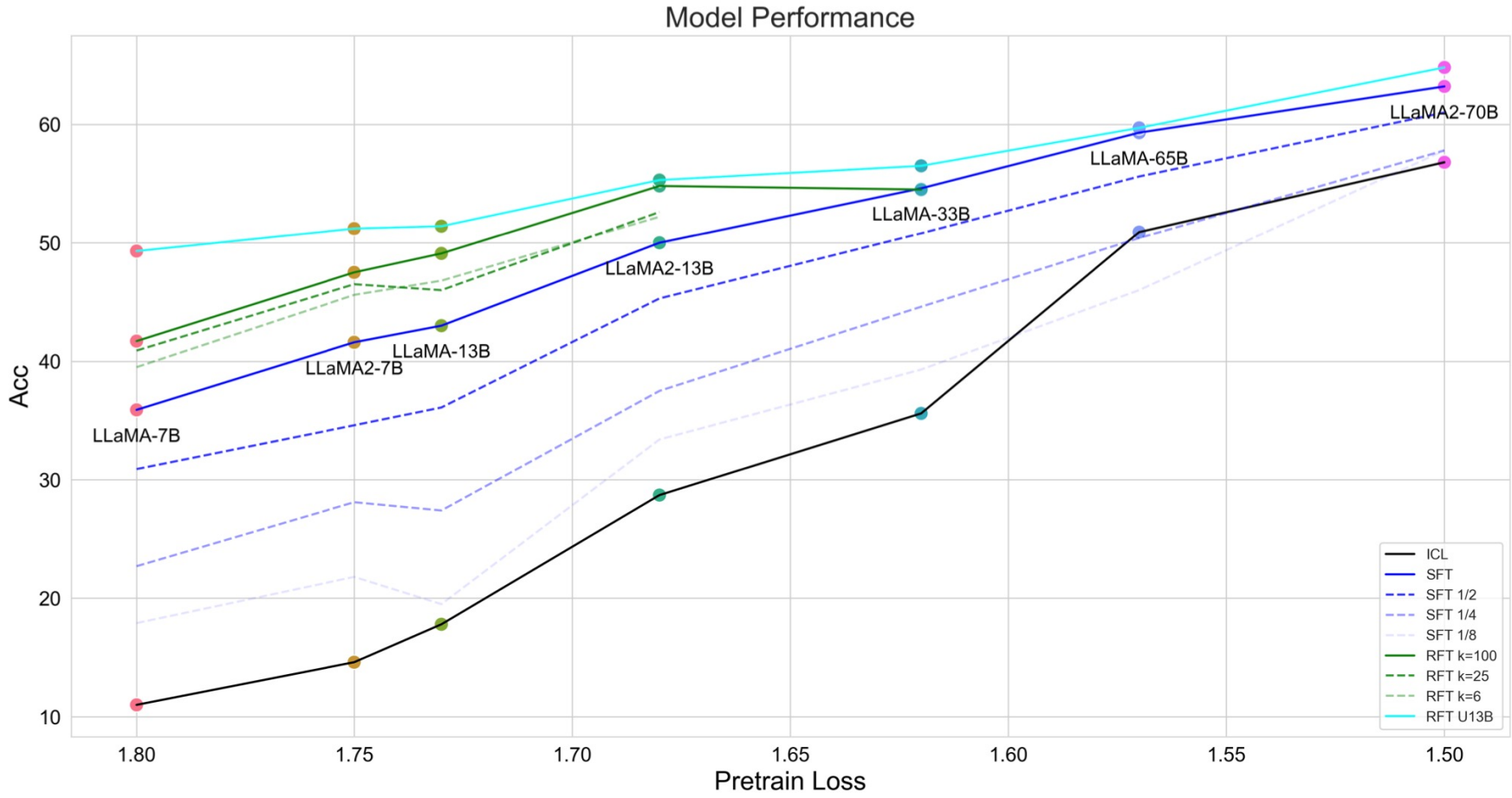


Figure 1: The key findings of scaling relationship on learning math reasoning ability with LLMs.

Scaling Models Up!

- Why do we want larger models?
- What is the largest model we can build?
  - Human brain has roughly 100 Trillion synapses (very shallow and bushy)
- What is the largest model we can build that fits on a single GPU?
- As we scale up what limits to we run into?
  - Memory?
  - Compute?
  - Data?

How do we estimate memory needed?

- Number of parameters?
- Memory for Activations?
- Memory for Batch size?
- Optimizer state?
- Working storage?

- How do memory requirements change from training to inference?
  - KV Cache for faster inference
  - What limits inference speed?

Compute Performance

- For a given sized model how much compute is needed to train it to convergence?
- Do we need to train to convergence?
- Are there trade offs between computing and memory?
- How do we scale the model beyond one CPU/GPU?
- How much numerical precision do we need?
- How fast are computers getting faster?
- Could we train a 100 T parameter model?

How much data is needed to train models?

- Assuming we could have any data we desired for training; can we estimate how much we need to train a model?

3 Predictable Scaling

A large focus of the GPT-4 project was building a deep learning stack that scales predictably. The primary reason is that for very large training runs like GPT-4, it is not feasible to do extensive model-specific tuning. To address this, we developed infrastructure and optimization methods that have very predictable behavior across multiple scales. These improvements allowed us to reliably predict some aspects of the performance of GPT-4 from smaller models trained using $1,000\times - 10,000\times$ less compute.

https://arxiv.org/pdf/2303.08774.pdf

Predicting Loss

3.1 Loss Prediction

The final loss of properly-trained large language models is thought to be well approximated by power laws in the amount of compute used to train the model [41, 42, 2, 14, 15].

To verify the scalability of our optimization infrastructure, we predicted GPT-4's final loss on our internal codebase (not part of the training set) by fitting a scaling law with an irreducible loss term (as in Henighan et al. [15]): $L(C) = aC^b + c$, from models trained using the same methodology but using at most 10,000x less compute than GPT-4. This prediction was made shortly after the run started, without use of any partial results. The fitted scaling law predicted GPT-4's final loss with high accuracy (Figure 1).

https://arxiv.org/pdf/2303.08774.pdf

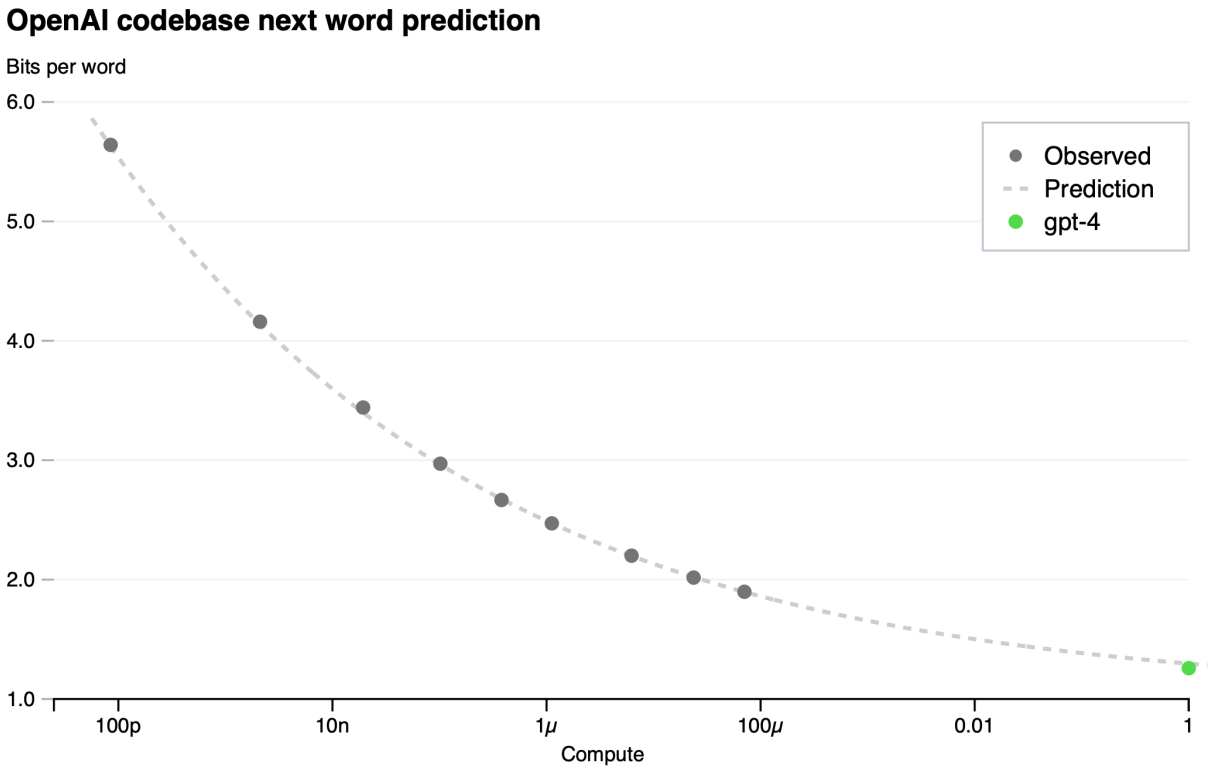


Figure 1. Performance of GPT-4 and smaller models. The metric is final loss on a dataset derived from our internal codebase. This is a convenient, large dataset of code tokens which is not contained in the training set. We chose to look at loss because it tends to be less noisy than other measures across different amounts of training compute. A power law fit to the smaller models (excluding GPT-4) is shown as the dotted line; this fit accurately predicts GPT-4's final loss. The x-axis is training compute normalized so that GPT-4 is 1.

Scaling Laws of Large Language Models It is important to understand and predict the performance gain as the language model scales up. Kaplan et al. (2020) first investigated and derived a predictable relationship on how the number of model parameters and data sizes contribute to the loss over many orders of magnitudes. Hoffmann et al. (2022) refined the scaling laws in (Kaplan et al., 2020) and found the scaling laws for computation-optimal training. Muennighoff et al. (2023) explored and extended the scaling laws under a data-constrained scenario. Besides investigating the scaling performance for pre-training, Gao et al. (2022) discussed the scaling laws for overparameterized reward models for alignment with human preference, and Hernandez et al. (2021) developed scaling laws for transferring performance from pre-trained models to downstream tasks. Henighan et al. (2020); Caballero et al. (2022) investigated scaling laws of math problems. In this paper, we are investigating the scaling relationships of large language models on learning math word problems with pre-training losses, supervised data amount, and augmented data amount.

LLMs can be characterized by four elements

- The number of parameters in the model N
- The size of the training set D in tokens
- The compute budget used to train in in FLOPS (floating point operations per second)
- The network architecture of the model (i.e. Transformer – Decoder)

- The question is therefore: Given a fixed increase in the compute budget, **in what proportion should we increase the number of parameters *N* and the size of the training dataset *D* to achieve the optimal loss *L*?**

Two Scaling Laws

- **OpenAI Scaling Laws (Brown/Kaplan)**
- The trend to train increasingly larger models was motivated by a paper published by OpenAI in 2020 titled ***Scaling Laws for Neural Language Models***.
- This paper suggests that to train larger models, increasing the number of parameters is **3x more important** than increasing the size of the training set.

"Scaling the model is more important than scaling the data"

"Compute efficient scaling doesn't require convergence"

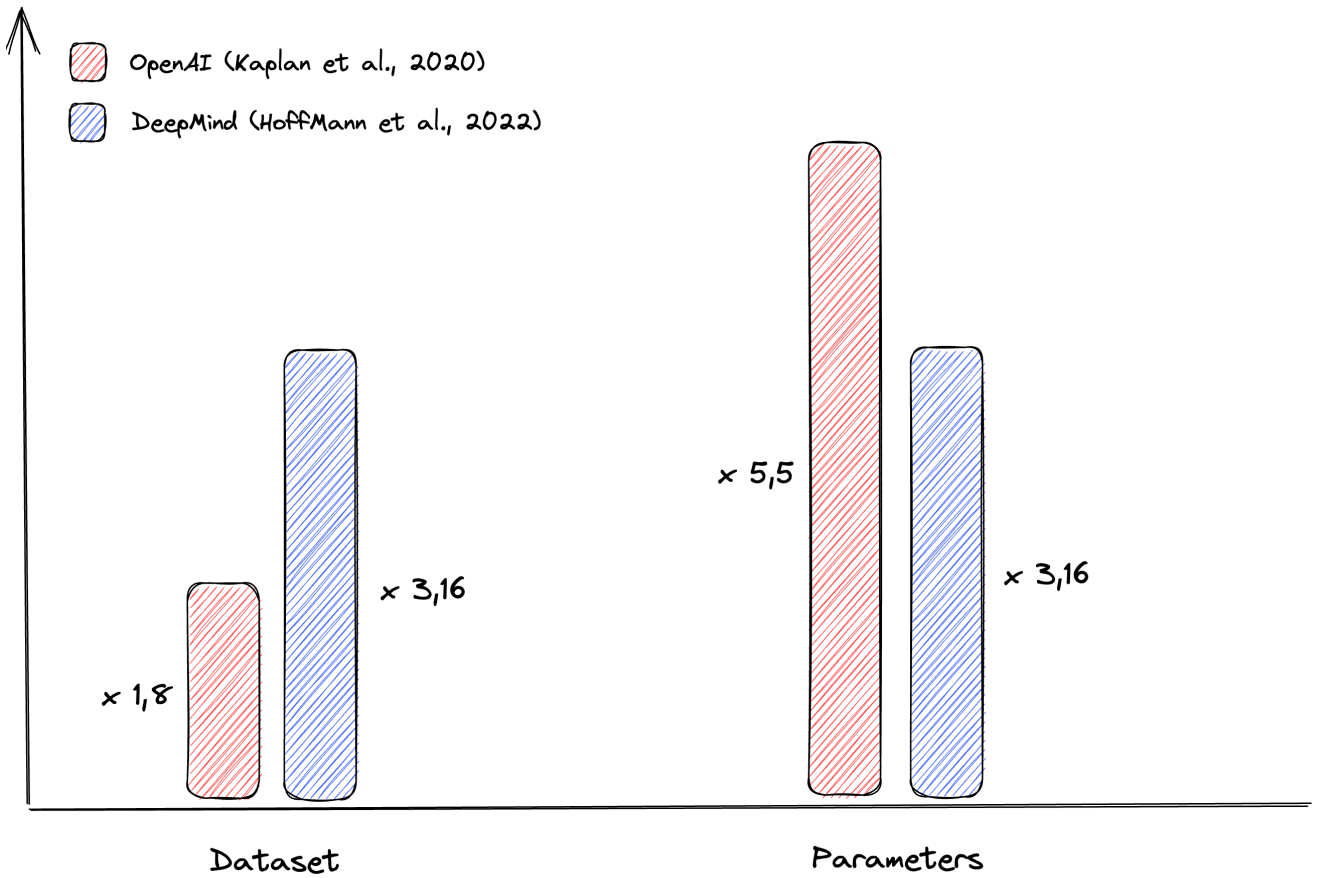
Two Scaling Laws

- **(Updated) DeepMind Scaling Laws (Chinchilla)**
- **However**, another paper, **published by DeepMind in 2022**, shows empirically that increasing the size of the training data is just as important as increasing the number of parameters itself. They **must be increased in equal proportions**.
- Moreover, training an optimal model requires about **20x more tokens than parameters**.

"Scaling the model is equally important to scaling data"

"Closer to convergence is better"

10x increase in compute budget



RLHF dramatically improves models

- Recent research papers (**InstructGPT**, **Chinchilla**) have shown that an efficiently trained GPT model (with Reinforcement Learning from Human Feedback and lots of data) containing 1.3 billion parameters, equals in performance a GPT-3 model containing 175 billion parameters.
- Labelers significantly prefer InstructGPT outputs over outputs from GPT-3. On our test set, outputs from the 1.3B parameter InstructGPT model are preferred to outputs from the 175B GPT-3, despite having over 100x fewer parameters. These models have the same architecture, and differ only by the fact that InstructGPT is fine-tuned on our human data. This result holds true even when we add a few-shot prompt to GPT-3 to make it better at following instructions. Outputs from our 175B InstructGPT are preferred to 175B GPT-3 outputs 85 ± 3% of the time, and preferred 71 ± 4% of the time to few-shot 175B GPT-3. InstructGPT models also generate more appropriate outputs according to our labelers, and more reliably follow explicit constraints in the instruction.

Brainscale model ~100 trillion parameters

- Suppose we want to train a large model with 100 trillion parameters to study its emergent properties.
- According to the Chinchilla paper, this would mean training this model on a database of... **180 petabytes of text**.
- We are simply **running out of data** since the entire Common Crawl dataset is "only" 12 petabytes ...

Efficient Training on a Single GPU .. Nice analysis of the issues .. huggingface

Method	Speed	Memory
Gradient accumulation	No	Yes
Gradient checkpointing	No	Yes
Mixed precision training	Yes	(No)
Batch size	Yes	Yes
Optimizer choice	Yes	Yes
DataLoader	Yes	No
DeepSpeed Zero	No	Yes

Anatomy of Model's Operations

Transformers architecture includes 3 main groups of operations grouped below by compute-intensity.

1. **Tensor Contractions**

   Linear layers and components of Multi-Head Attention all do batched **matrix-matrix multiplications**. These operations are the most compute-intensive part of training a transformer.

2. **Statistical Normalizations**

   Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more **reduction operations**, the result of which is then applied via a map.

3. **Element-wise Operators**

   These are the remaining operators: **biases, dropout, activations, and residual connections**. These are the least compute-intensive operations.

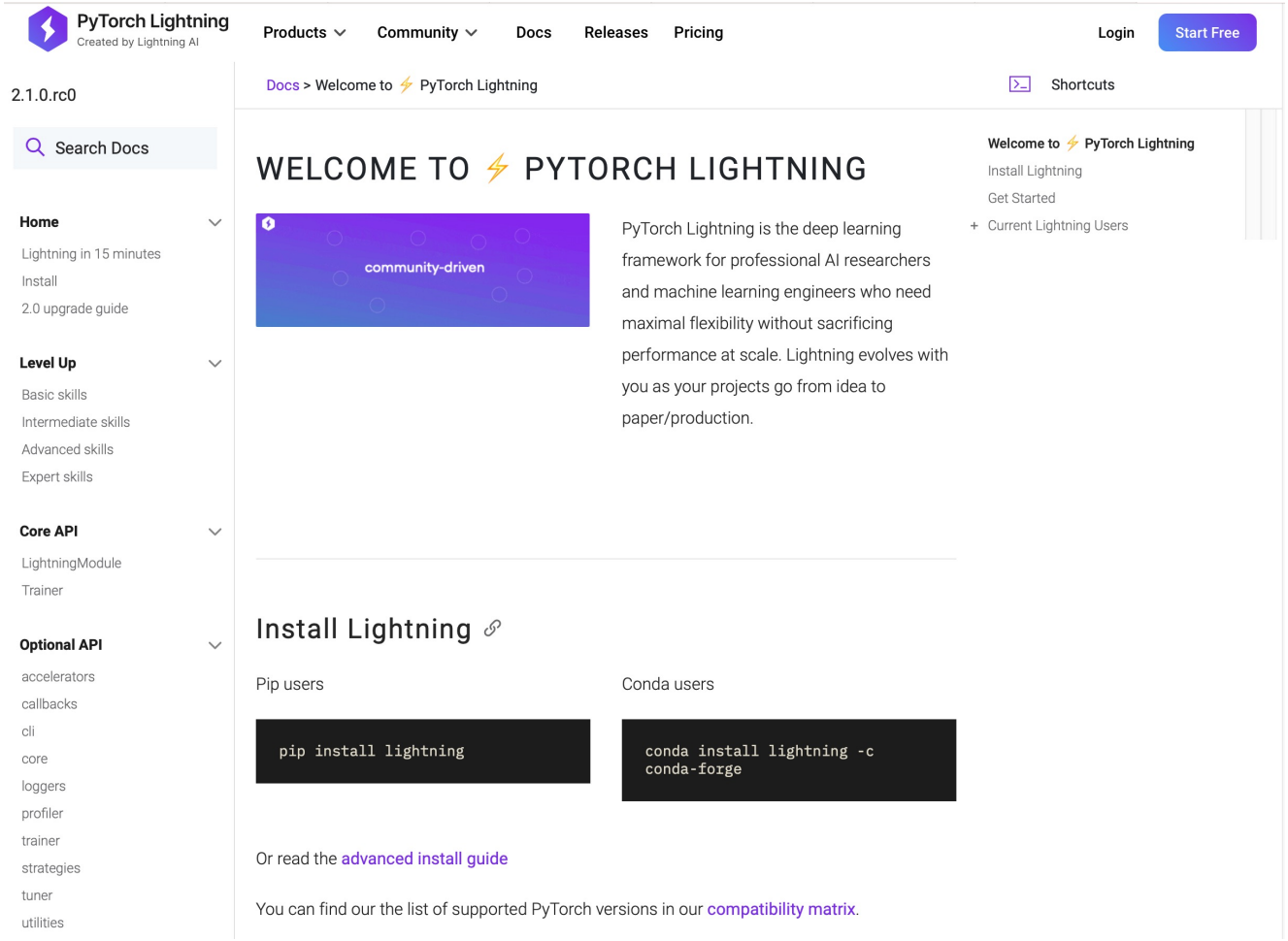
This knowledge can be helpful to know when analyzing performance bottlenecks.

This summary is derived from Data Movement Is All You Need: A Case Study on Optimizing Transformers 2020

Attacking Memory Consumption

- Accounting for all the memory usage
- Devising schemes for reducing memory usage for each component process that uses memory
- Exploring tradeoffs between memory, computing, communication and accuracy

https://lightning.ai/docs/pytorch/latest/



https://github.com/rasbt/pytorch-memory-optim

PyTorch Lightning is the deep learning framework for professional AI researchers and machine learning engineers who need maximal flexibility without sacrificing performance at scale. Lightning evolves with you as your projects go from idea to paper/production.

We are using their tutorial to present the key ideas

You should experiment with the tools yourself, perhaps on NanoGPT or other models.

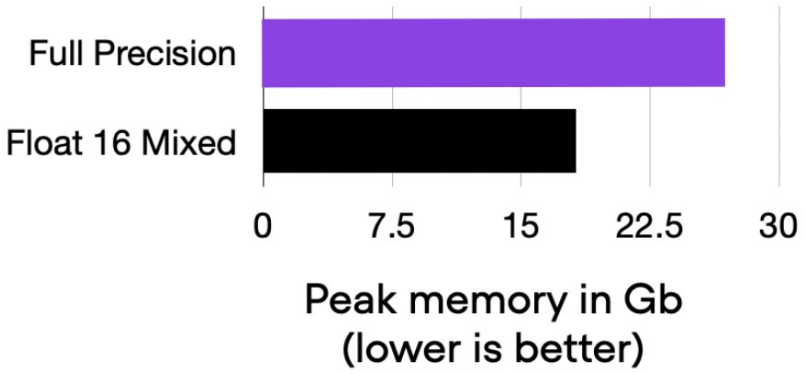
Each of these techniques are covered in various papers

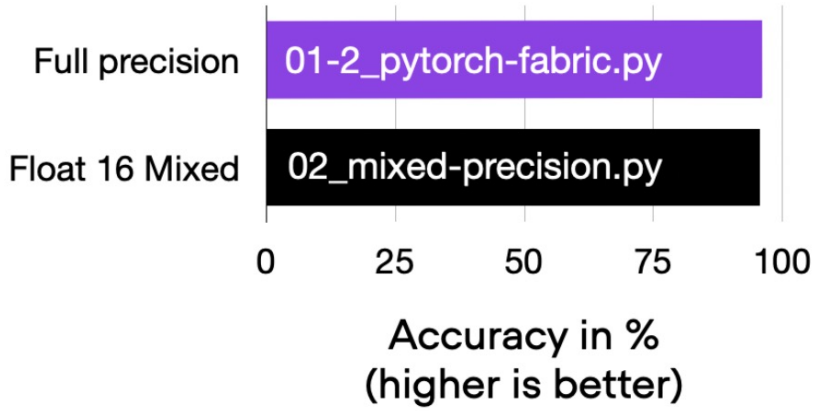
Megatron-LM, DeepSpeed ZeRO, Flash Attention, etc.

Table of Contents

Mixed Precision (32bit and 16bit)

- By default frameworks use float32 (four bytes) per parameter and for operations
- Mixed precision can replace some parameters and operations with float16 or bfloat16
- Some implementation keep two copies f32 and f16





Mixed-Precision Training

What Is Mixed-Precision Training?

Mixed precision training uses both 16-bit and 32-bit precision to ensure no loss in accuracy. The computation of gradients in the 16-bit representation is much faster than in the 32-bit format and saves a significant amount of memory. This strategy is beneficial, especially when we are memory or compute-constrained.

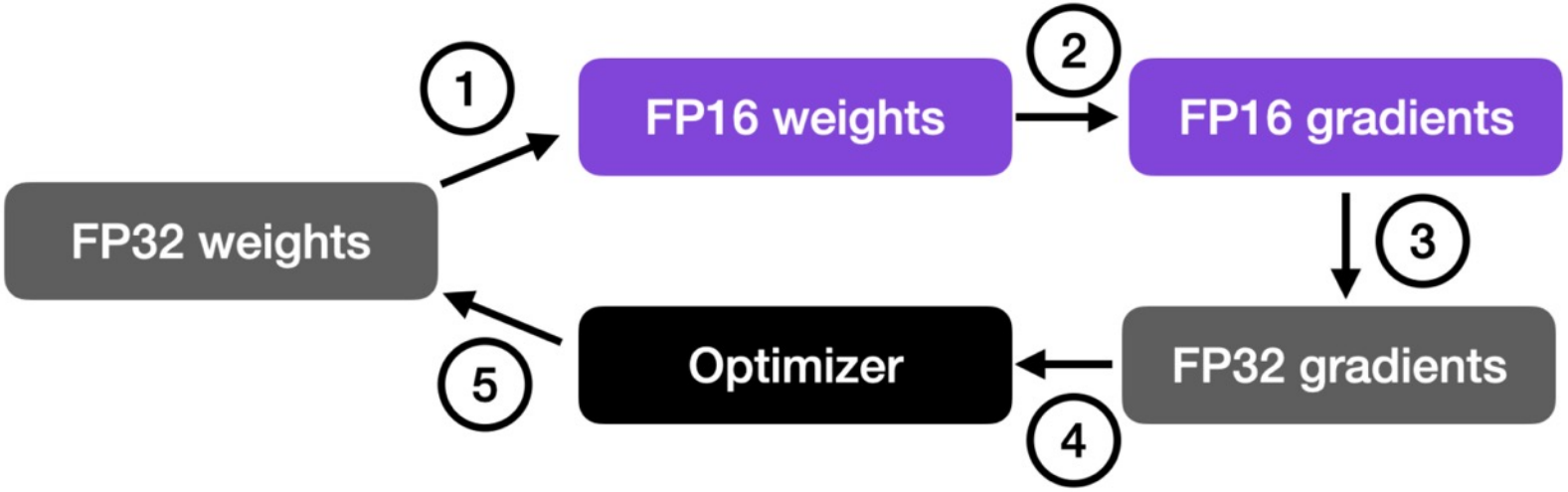
It's called "mixed-"rather than "low-"precision training because we don't transfer *all* parameters and operations to 16-bit floats. Instead, we switch between 32-bit and 16-bit operations during training, hence, the term "mixed" precision.

As illustrated in the figure below, mixed-precision training involves converting weights to lower-precision (FP16) for faster computation, calculating gradients, converting gradients back to higher-precision (FP32) for numerical stability, and updating the original weights with the scaled gradients.

This approach allows for efficient training while maintaining the accuracy and stability of the neural network.

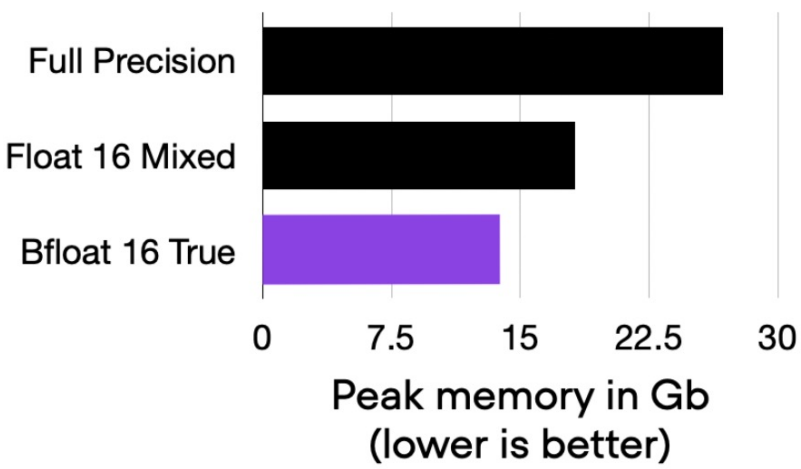
Mixed-Precision flow

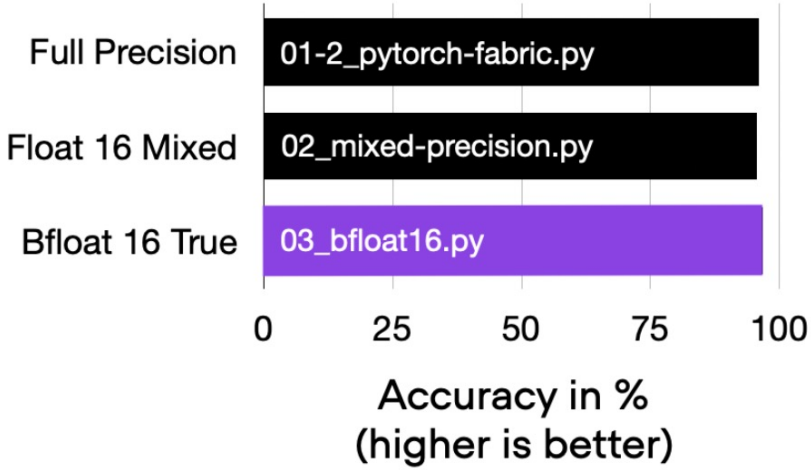
- Conversion to/from FP32 and FP16



Full Low Precision Training (BFP16)

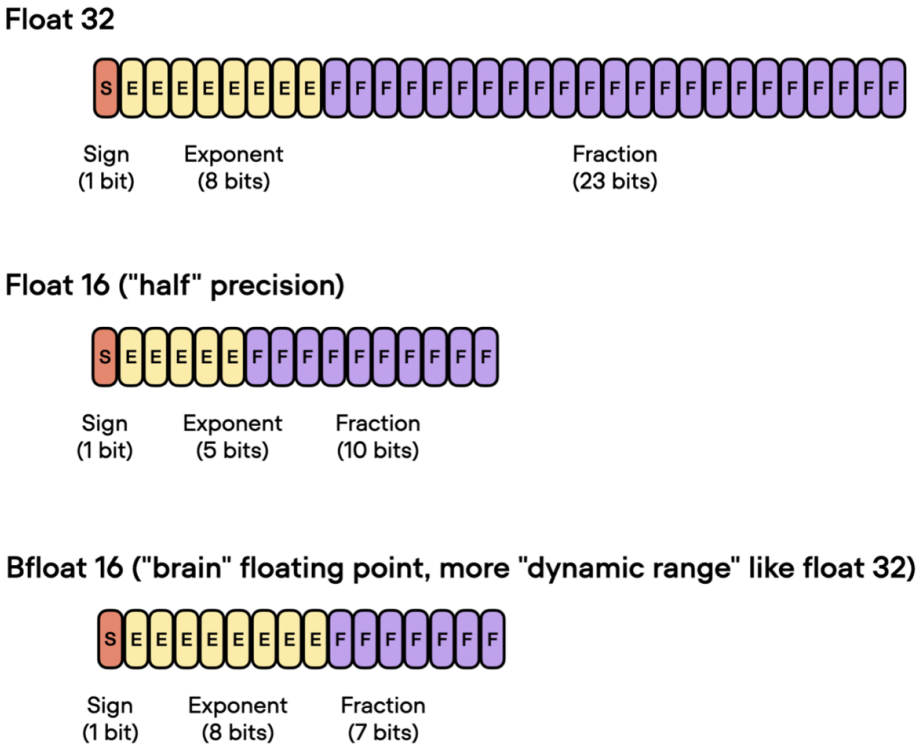
We can also take it a step further and try running with “full” lower 16-bit precision (instead of mixed-precision which converts intermediate results to a 32-bit representation.)





What Is Bfloat16?

The “bf16” in `"bf16-mixed"` stands for Brain Floating Point (bfloat16). Google developed this format for machine learning and deep learning applications, particularly in their Tensor Processing Units (TPUs). Bfloat16 extends the dynamic range compared to the conventional float16 format at the expense of decreased precision



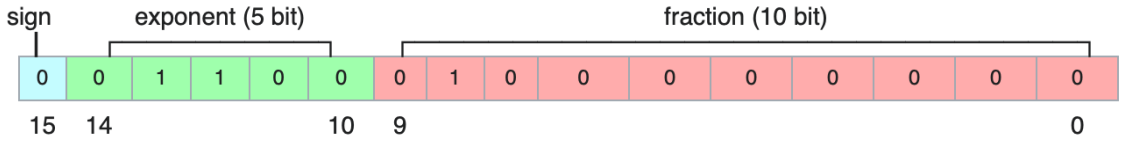
Brain Floating Point was Promoted by Google

Most GPUs now support it

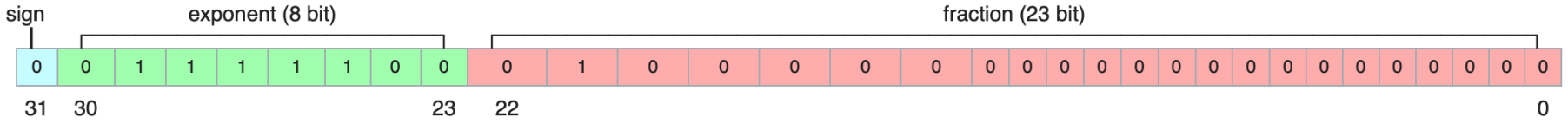
Solves many numerical stability Problems

FP8 and BFP8 are also

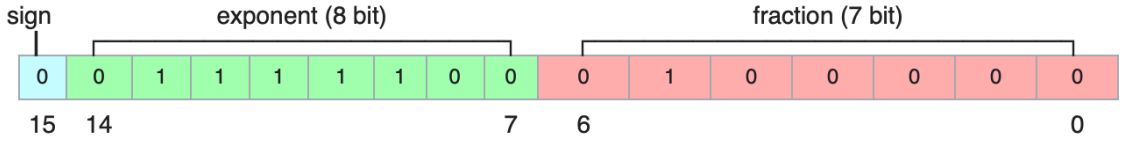
IEEE half-precision 16-bit float



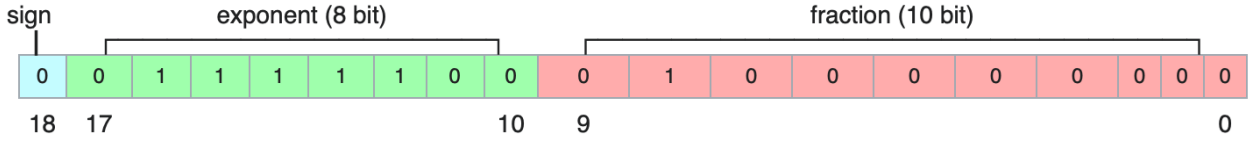
IEEE 754 single-precision 32-bit float



bfloat16



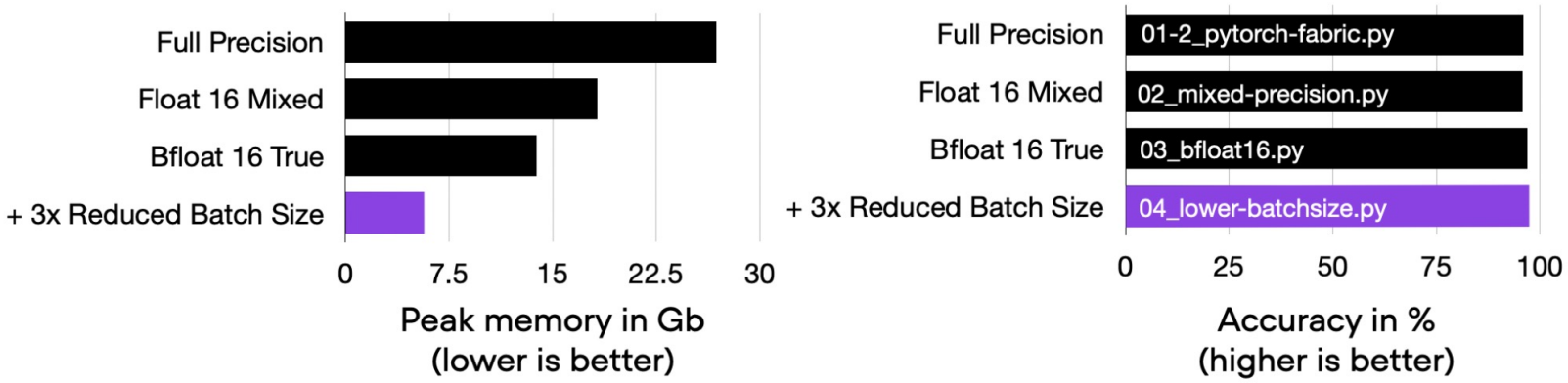
NVidia's TensorFloat



Reducing the Batch Size

Let's tackle one of the big elephants in the room: why don't we simply reduce the batch size? This is usually always an option to reduce memory consumption. However, it can sometimes result in worse predictive performance since it alters the training dynamics. (For more details, see Lecture 9.5 in my Deep Learning Fundamentals course.)

Either way, let's reduce the batch size to see how that affects the results. It turns out we can lower the batch size to 16, which brings memory consumption down to 5.69 GB, without sacrificing performance:

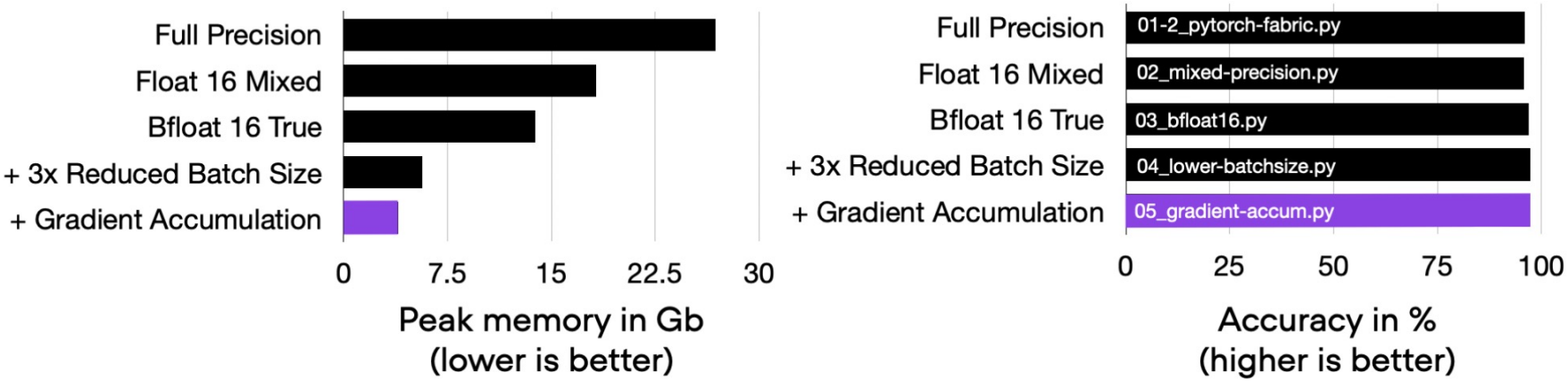


Comparing 04_lower-batchsize.py to the previous codes.

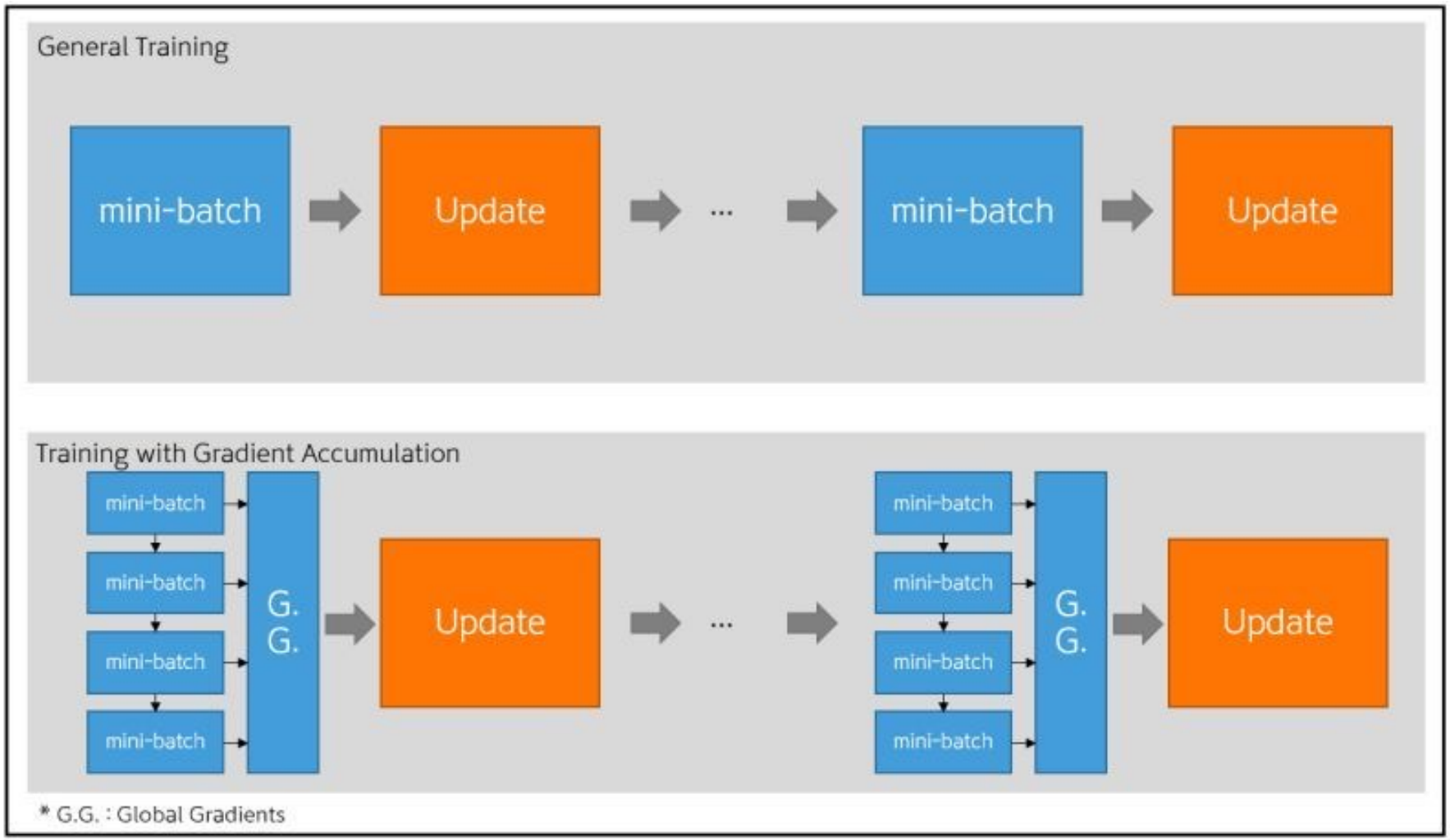
Gradient Accumulation to Create Microbatches

Gradient accumulation is a way to virtually increase the batch size during training, which is very useful when the available GPU memory is insufficient to accommodate the desired batch size. Note that this only affects the runtime, not the modeling performance.

In gradient accumulation, gradients are computed for smaller batches and accumulated (usually summed or averaged) over multiple iterations instead of updating the model weights after every batch. Once the accumulated gradients reach the target "virtual" batch size, the model weights are updated with the accumulated gradients.



Result of 05_gradient-accum.py


General Training
mini-batch
Update
...
mini-batch
Update
Training with Gradient Accumulation
mini-batch
mini-batch
mini-batch
mini-batch
G.
G.
Update
...
mini-batch
mini-batch
mini-batch
mini-batch
G.
G.
Update
* G.G. : Global Gradients

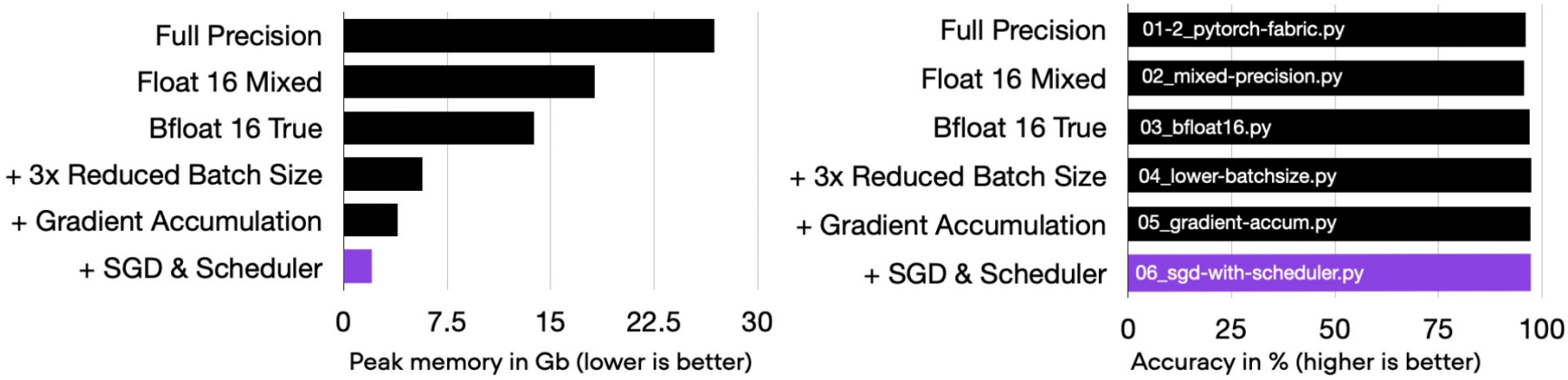
Using a “leaner” Optimizer

Did you know that the popular Adam optimizer comes with additional parameters? For instance, Adam has 2 additional optimizer parameters (a mean and a variance) for each model parameter.

So, by swapping Adam with a stateless optimizer like SGD, we can reduce the number of parameters by 2/3, which can be quite significant when working with vision transformers and LLMs.

The downside of plain SGD is that it usually has worse convergence properties. So, let’s swap Adam with SGD and introduce a cosine decay learning rate scheduler to compensate for this and achieve better convergence.

SGD can save memory
But doesn’t often doesn’t
Converge as fast



Result of 06_sgd-with-scheduler.py

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

$$For\ each\ Parameter\ w^j$$

(j subscript dropped for clarity)

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

η : *Initial Learning rate*

g_t : *Gradient at time t along* ω^j

ν_t : *Exponential Average of gradients along* ω_j

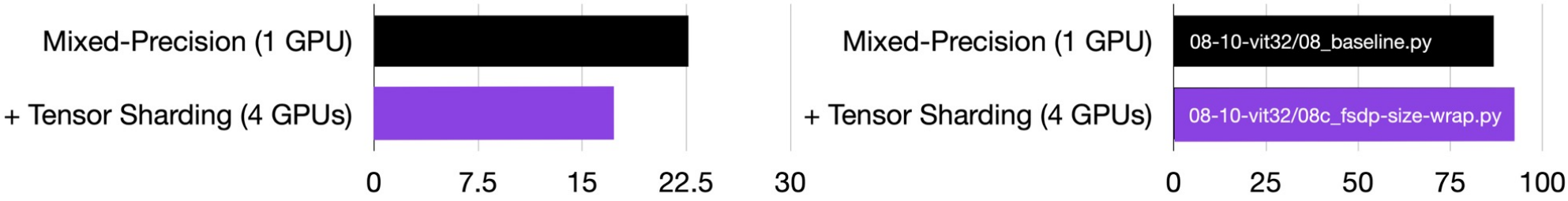
s_t : *Exponential Average of squares of gradients along* ω_j

β_1, β_2 : *Hyperparameters*

Distributed Training and Tensor Sharding

The next modification we are going to try is multi-GPU training. It becomes beneficial if we have multiple GPUs at our disposal since it allows us to train our models even faster.

However, here, we are mainly interested in the memory saving. So, we are going to use a more advanced, distributed multi-GPU strategy called Fully Sharded Data Parallelism (FSDP), which utilizes both data parallelism and tensor parallelism for sharding large weight matrices across multiple devices.

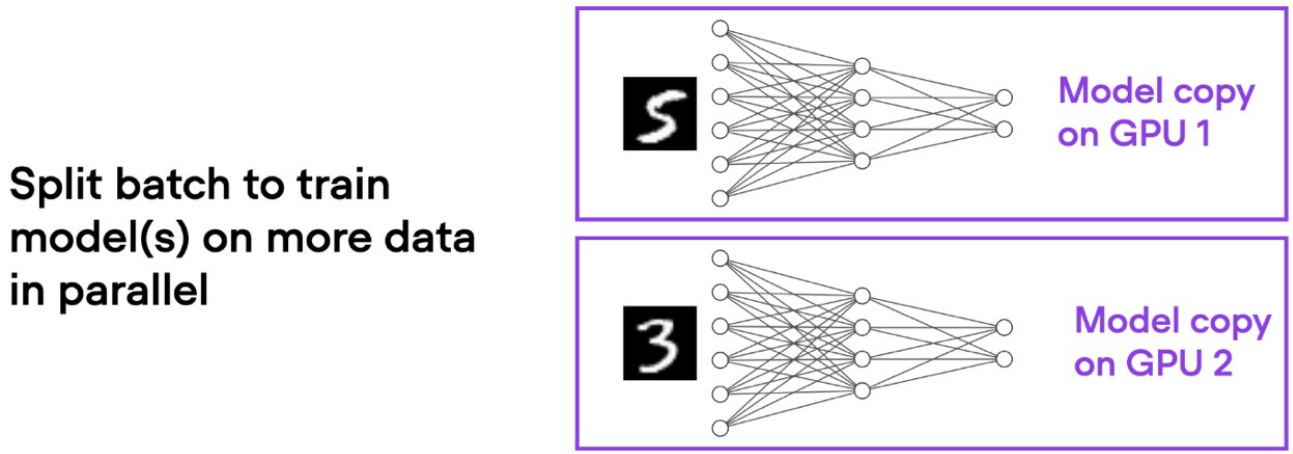


Via code example 08c_fsdp-size-wrap.py

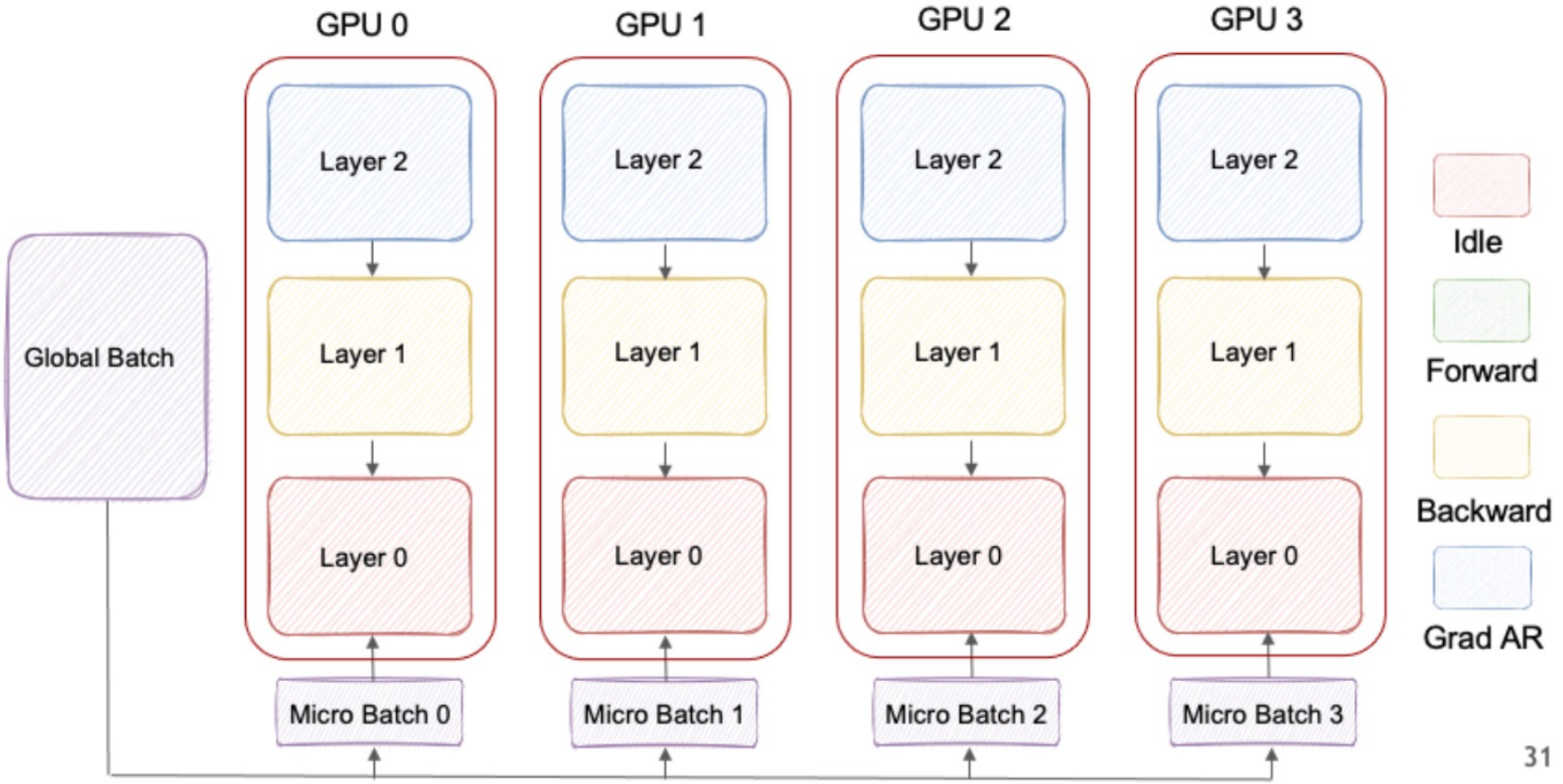
Data Parallelism

Understanding Data Parallelism and Tensor Parallelism

In data parallelism, the mini-batch is divided, and a copy of the model is available on each of the GPUs. This process speeds up model training as multiple GPUs work in parallel.



Distributed Data Parallelism



Data Parallelism

Here's how it works in a nutshell:

1. The same model is replicated across all the GPUs.
2. Each GPU is then fed a different subset of the input data (a different mini-batch).
3. All GPUs independently perform forward and backward passes of the model, computing their own local gradients.
4. Then, the gradients are collected and averaged across all GPUs.
5. The averaged gradients are then used to update the model's parameters.

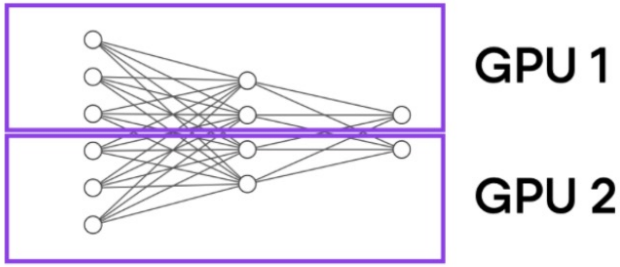
The primary advantage of this approach is speed. Since each GPU is processing a unique mini-batch of data concurrently with the others, the model can be trained on more data in less time. This can significantly reduce the time required to train our model, especially when working with large datasets.

However, data parallelism has some limitations. Most importantly, each GPU must have a complete copy of the model and its parameters. This places a limit on the size of the model we can train, as the model must fit within a single GPU's memory — this is not feasible for modern ViTs or LLMs.

Tensor Parallelism

- Sharding or Partitioning the network “horizontally”

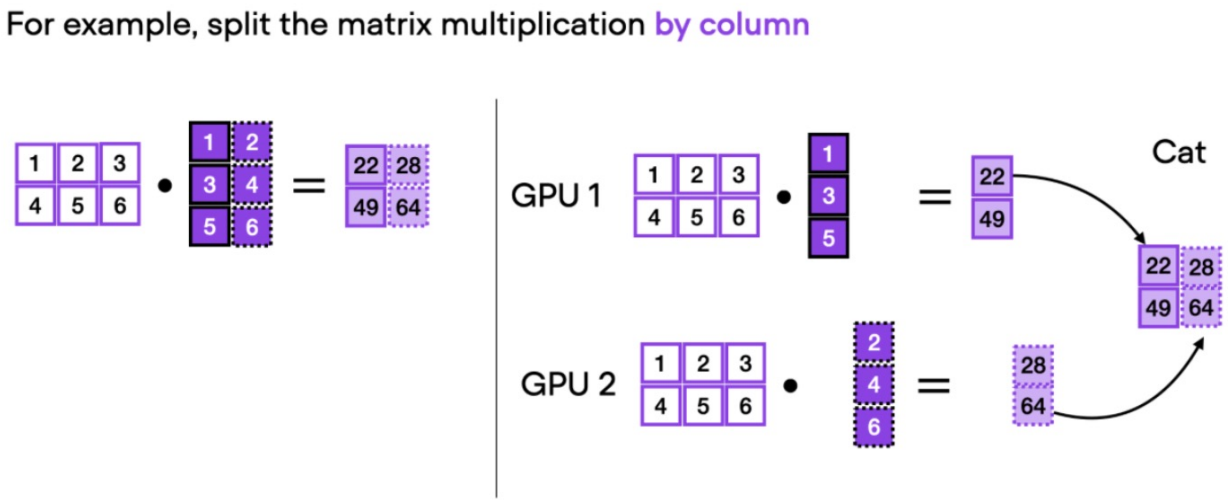
Related to model parallelism, but split horizontally instead of vertically



(Depending on the implementation, you may split weight matrices, optimizer states, gradients)

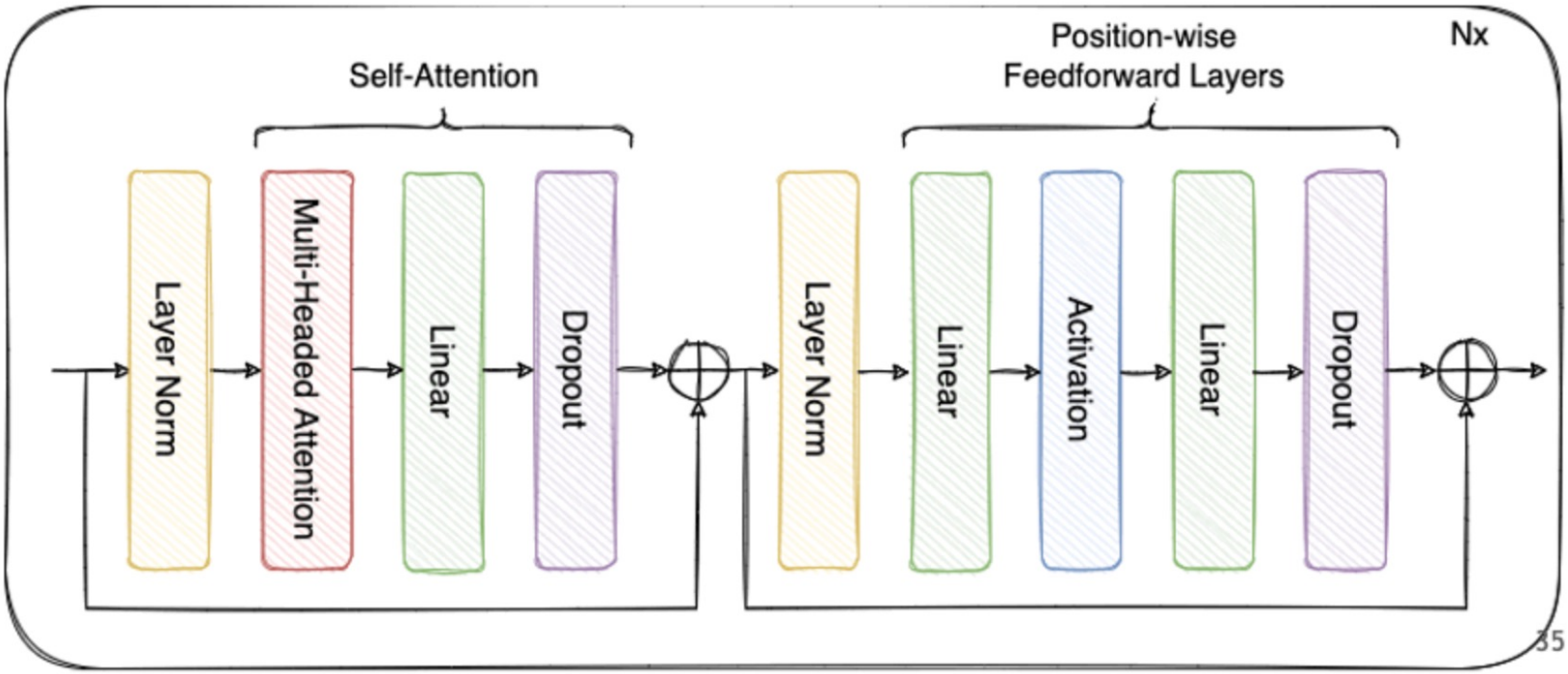
Tensor Parallelism

How does it work? Think of matrix multiplication. There are two ways to distribute it — by row or by column. For simplicity, let's consider distribution by column. For instance, we can break down a large matrix multiplication operation into separate computations, each of which can be carried out on a different GPU, as shown in the figure below. The results are then concatenated to get the original result, effectively distributing the computational load.



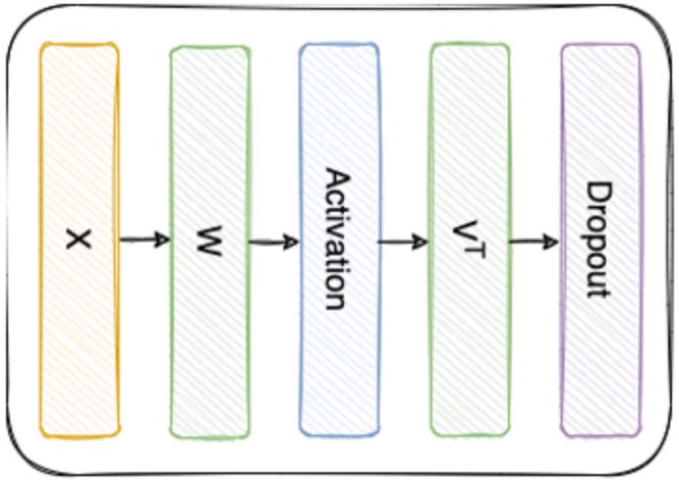
Transformers

"Pre-Layernorm" Transformer Block



Tensor Parallelism (Intra-Layer)

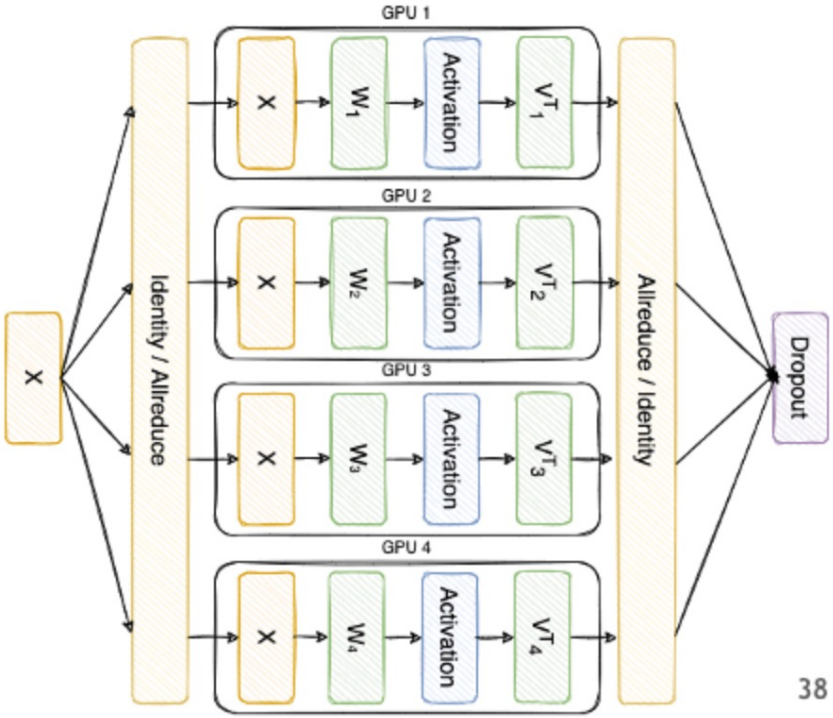
MLP



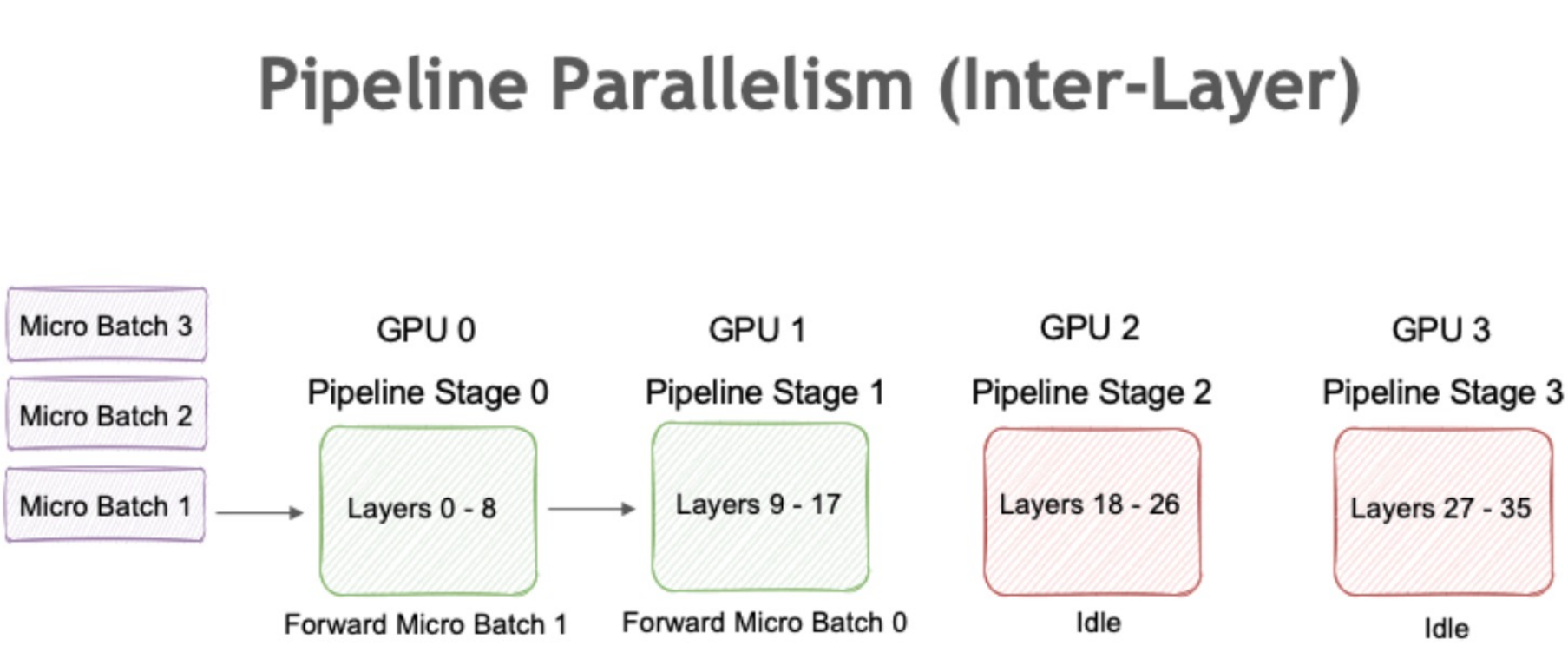
Split weights W and V^T across multiple GPUs

$$\mathbf{W} = [\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{W}_4] \qquad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \\ \mathbf{V}_3 \\ \mathbf{V}_4 \end{bmatrix}$$

Tensor Parallel = 4 MLP

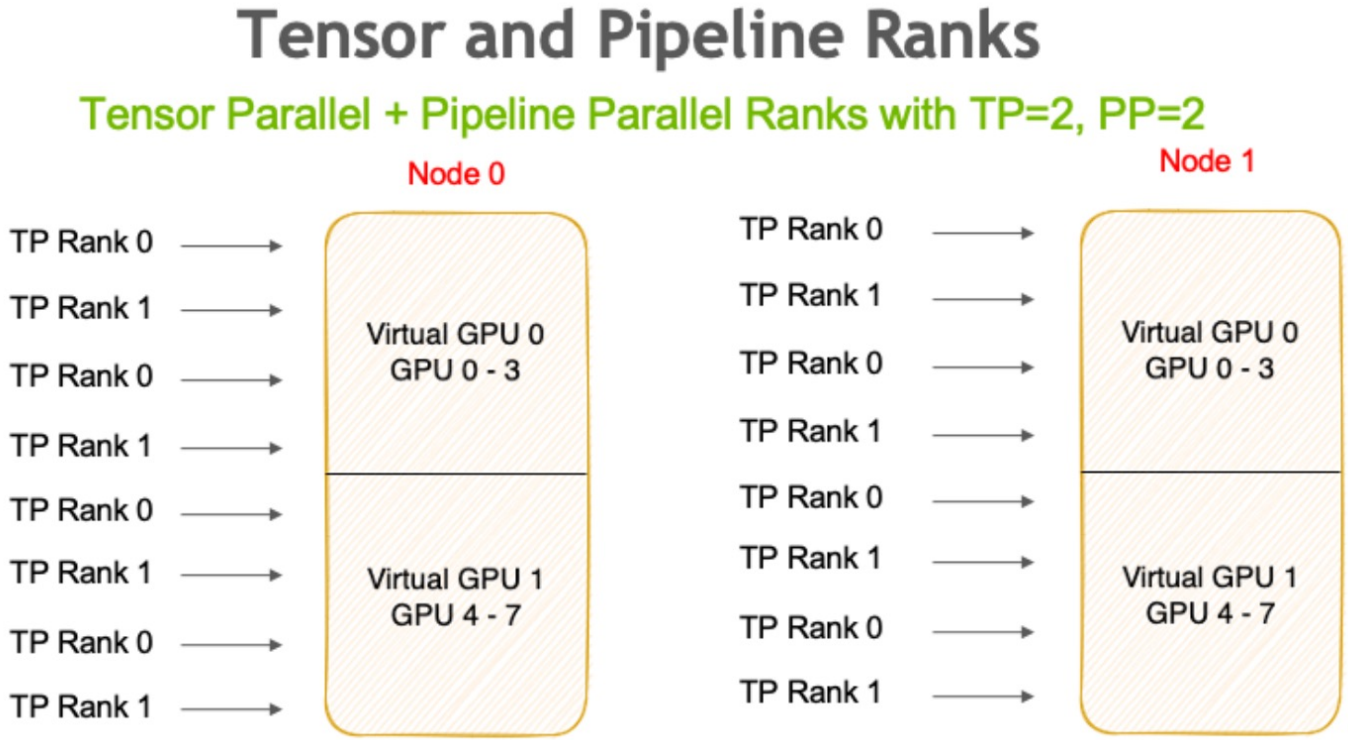


Pipeline Parallelism



Parallelism nomenclature

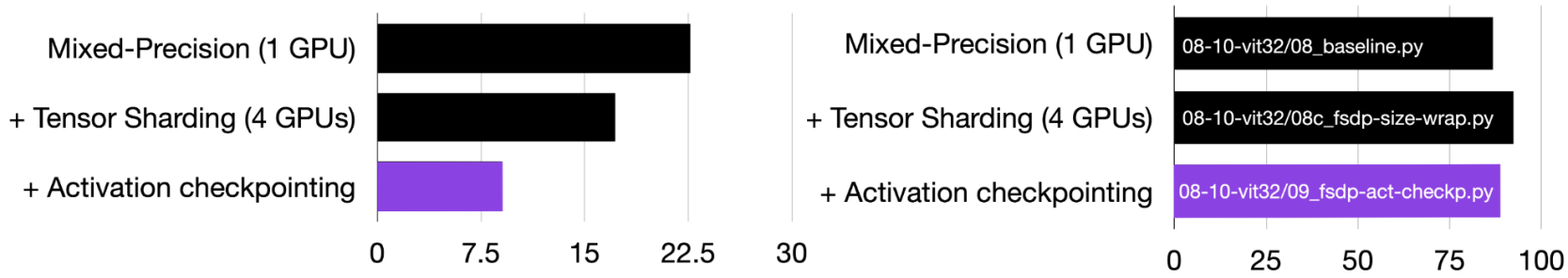
When reading and modifying NeMo Megatron code you will encounter the following terms.



Activation Checkpointing

To further minimize memory usage during neural network computations, we can add gradient checkpointing (also known as activation checkpointing). This method selectively eliminates certain layer activations during the forward pass and later recalculates them in the backward pass. This approach essentially compromises some computational time to conserve memory.
In other words, a layer's inputs and outputs are retained in memory after the forward pass, but any intermediate tensors that were involved in the computation within the module are released. When the backward pass is computed for these checkpointed modules, the previously cleared tensors are recalculated.

> This lowers the memory consumption from 17.23 GB to 9.03 GB. However, this slightly increased the runtime from 18.95 min to 22.58 min.

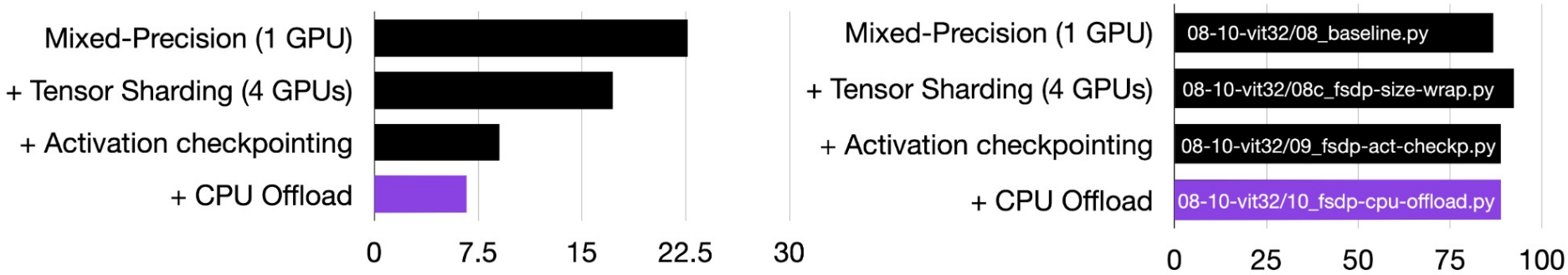


Via code example 09_fsdp-with-act-checkpointing

Parameter Offloading

Moving the optimization parameters from GPU memory to the CPU memory

This reduces the memory consumption from 9.03 GB with activation checkpointing to 6.68 GB with additional CPU offloading. Thus substantially increased the runtime, however, from 22.58 min to 101.53 min.

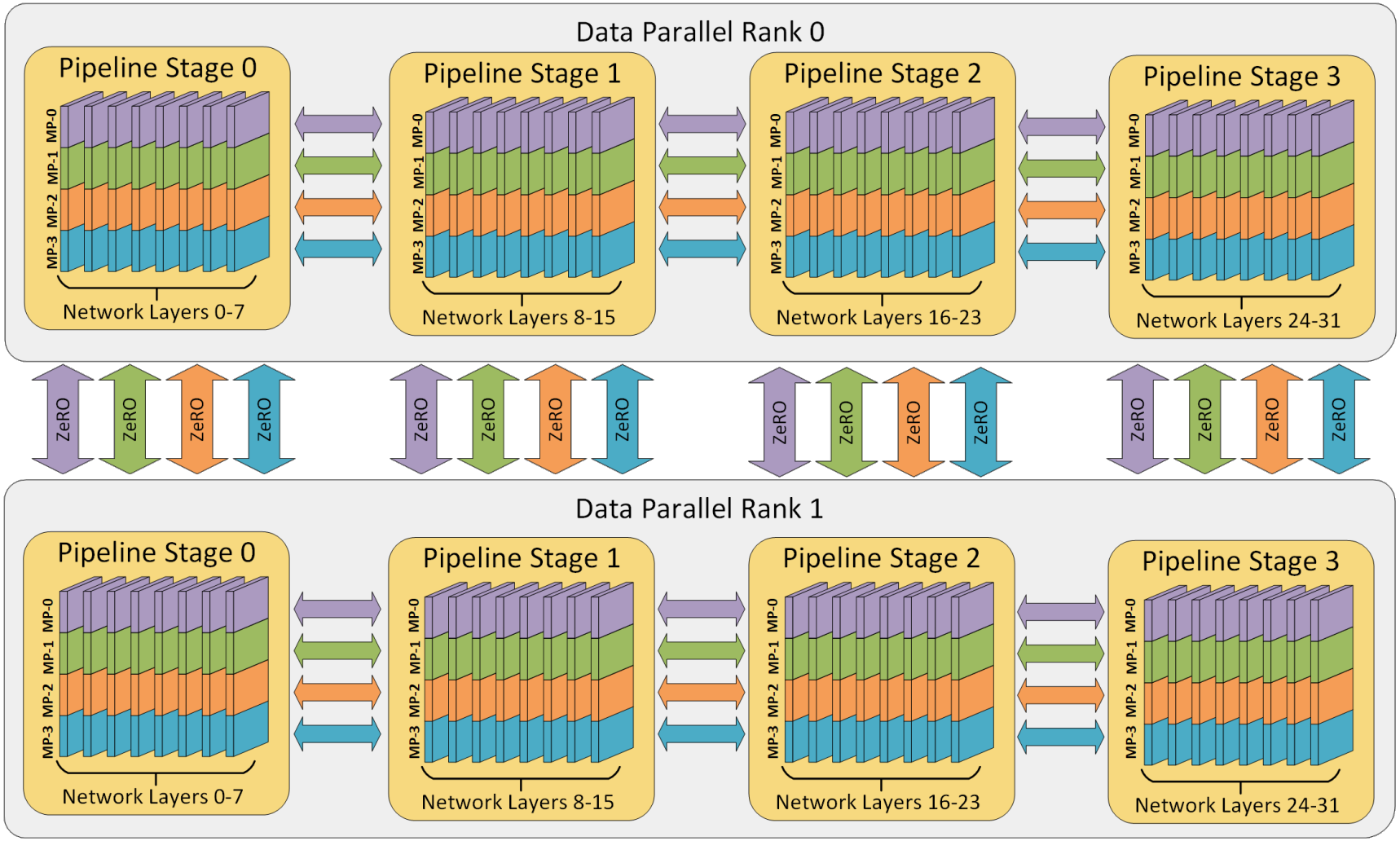


Via code example 10_fsdp-cpu-offload.py

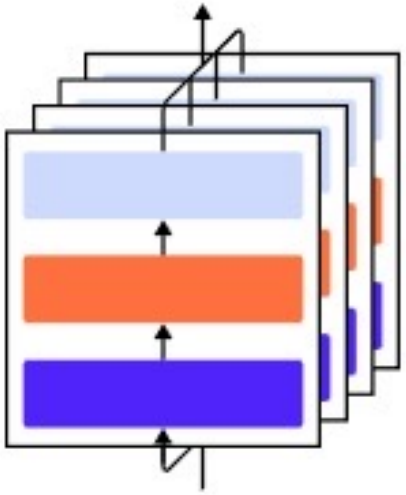
Composing these Approaches

- It is very common today to compose many of these optimizations using tools and libraries build by various groups.

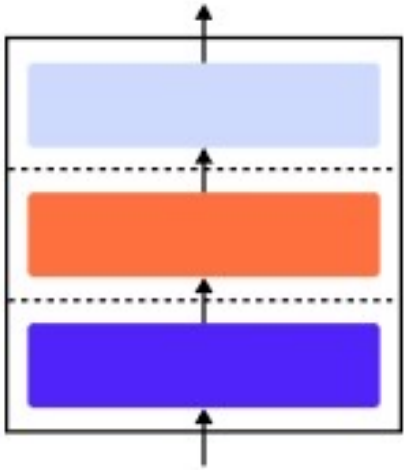
- Megatron-LM is a transformer implementation that supports data and tensor parallelism to scale to 1T parameters
- DeepSpeed project at Microsoft supports offloading (Zero Redunancy Optimization, etc.)
- Flash Attention reduces the size of the attention cache
- Etc.


Data Parallel Rank 0
Pipeline Stage 0
Pipeline Stage 1
Pipeline Stage 2
Pipeline Stage 3
MP-0
MP-1
MP-2
MP-3
Network Layers 0-7
Network Layers 8-15
Network Layers 16-23
Network Layers 24-31
ZeRO
Data Parallel Rank 1

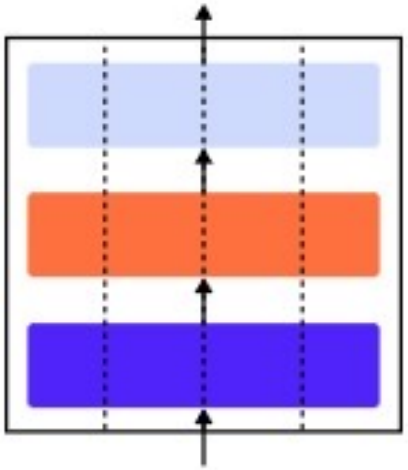
Data Parallelism



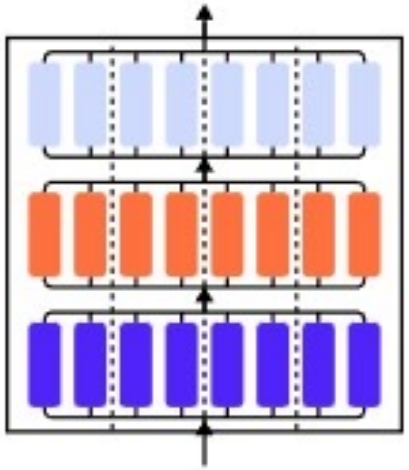
Pipeline Parallelism



Tensor Parallelism



Expert Parallelism



Computation of: z=h(f(x), g(y))