Large-scale model training

Machine Learning Modena – May 10 2023

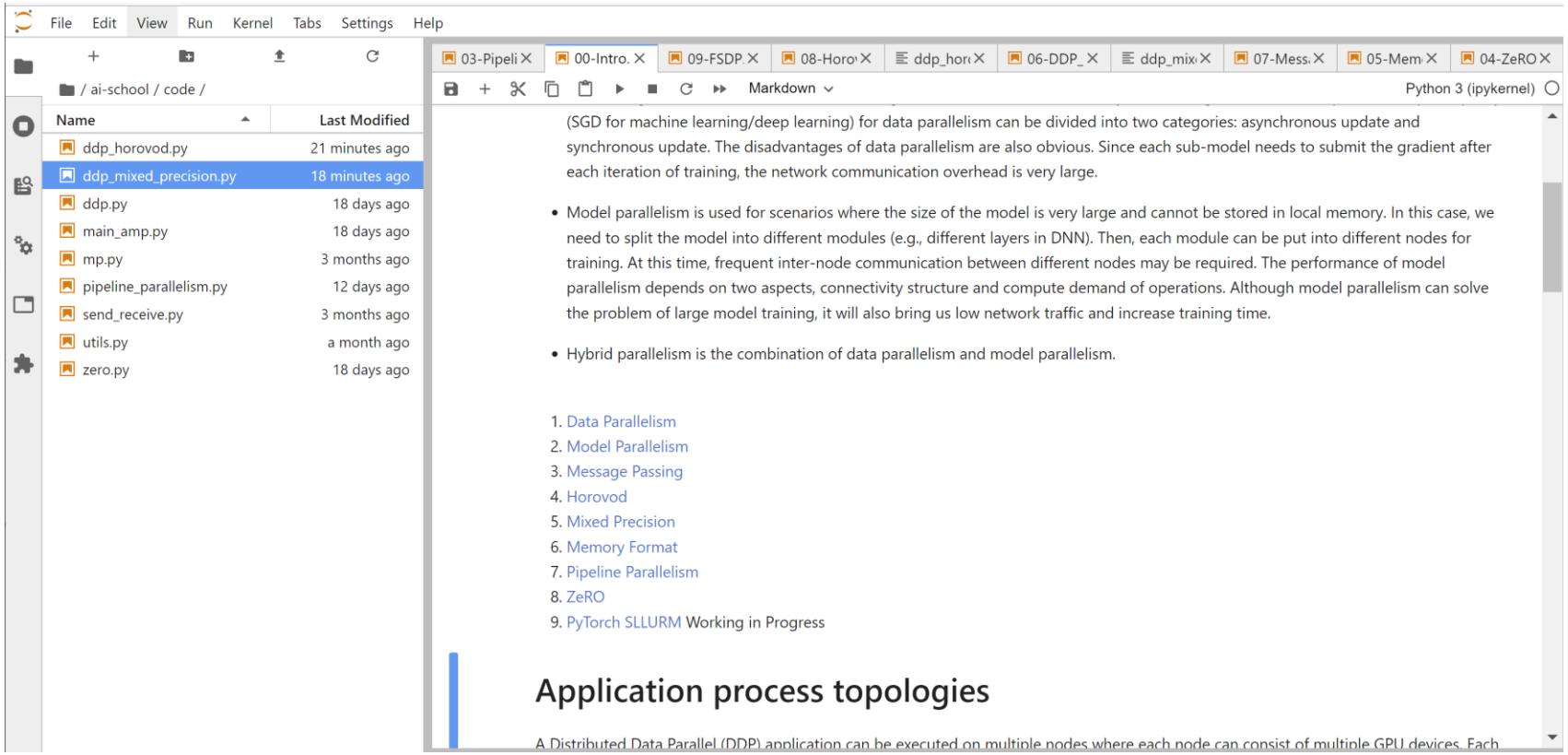
About Me

Giuseppe Fiameni – gfiameni@nvidia.com

- AI & HPC @ NVIDIA
  - *Support Higher Education and Research through collaboration projects*
- Lead engineer of the NVIDIA AI Technology Center, Italy
  - *Agreement among CINI, CINECA and NVIDIA to accelerate scientific discovery*
- 10+ years experience delivering HPC applications and large data processing solutions at CINECA (www.hpc.cineca.it)

NOTEBOOKS & CODE

https://github.com/gfiameni/hpdl/
https://gitlab.hpc.cineca.it/sorland/hpc-dl-sapienza-2023



Why this talk?

- Deep Learning is transitioning from being a ***computer science*** *towards a* ***computational science***.
- Advanced computing and large-scale infrastructure are fundamental to conduct science i.e., train gigantic models, process massive data, achieve better performance, reduce time to solutions.

CINECA Leonardo system

https://leonardo-supercomputer.cineca.eu/

- Nodes: 3456 booster nodes
- Processors: Intel Xeon 8358 32 cores, 2.6 GHz
- Accelerators: **4 x NVIDIA custom Ampere GPU 64GB HBM2**
- RAM: (8x64) GB DDR4 3200 MHz
- Network: 2 x NVIDIA HDR cards 2x100Gb/s
- 106 PB (raw) Large capacity storage, 620 GB/s

4th fastest supercomputer worldwide

Andrej Karpathy 
@karpathy

The most dramatic optimization to nanoGPT so far (~25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64). This calculates added useless dimensions but goes down a different kernel path with much higher occupancy. Careful with your Powers of 2.

Andrej Karpathy  @karpathy · Feb 4

Replying to @abhi_venigalla and @MosaicML

I love how sometimes changing one integer/flag can have the same impact as a 1 month optimization project. You just know there is some OMP_NEVER_HEARD_OF=3 that gets addition 3% MFU. Or my personal favorite - that undocumented bios flag that only 4 people on Earth know exists :D

AGENDA

Why Large datasets and Large Neural Networks?

Challenges of Supervised training

Scaling law

Training large neural networks

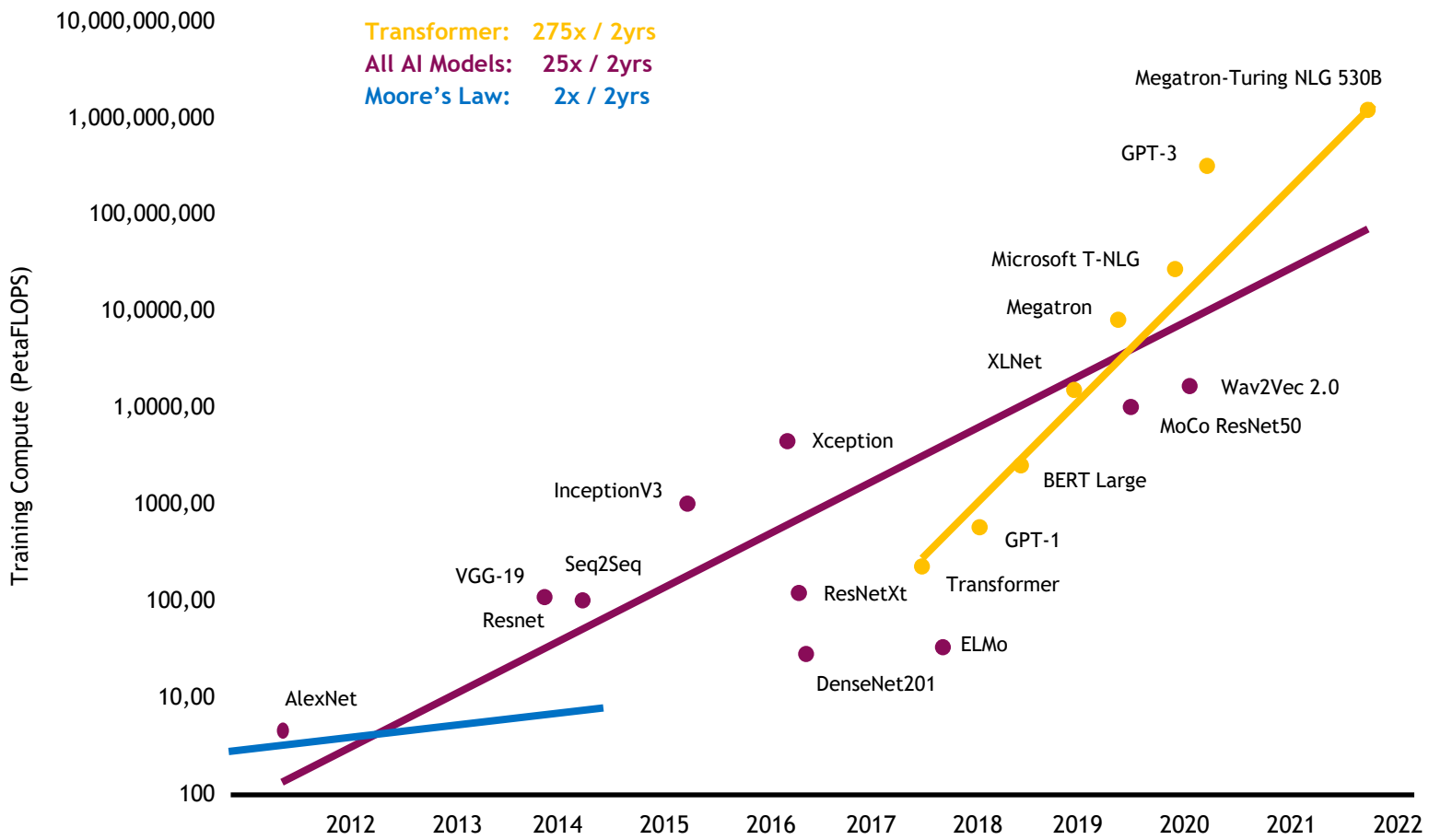
Single GPU optimization

Multi GPU optimization

The case of LLM

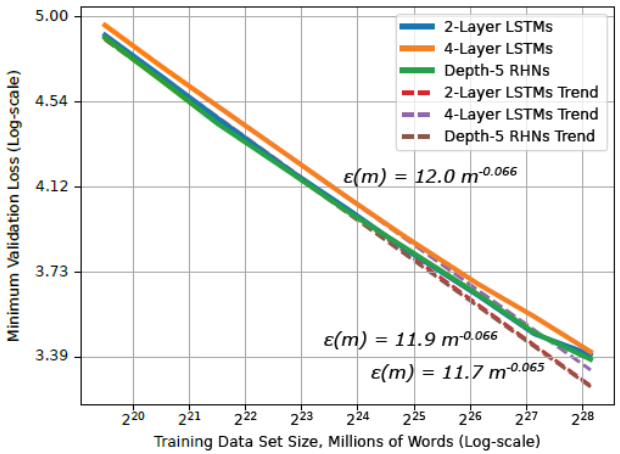
Dramatic increase in Model Sizes

The Trend Continues

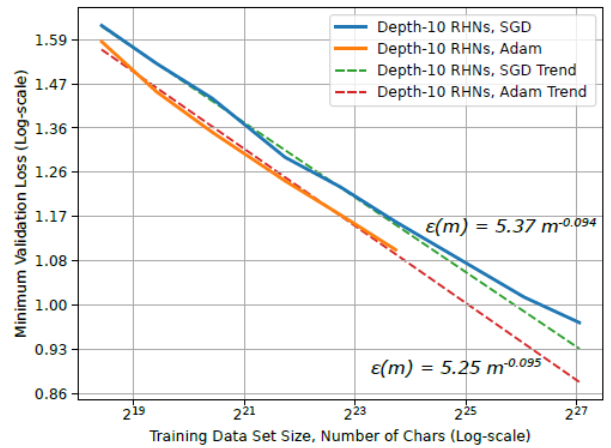


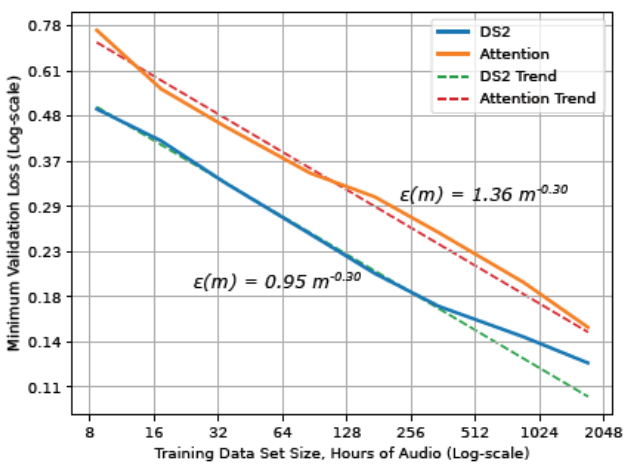
RELIABLE WAY FOR IMPROVING MODEL PERFORMANCE

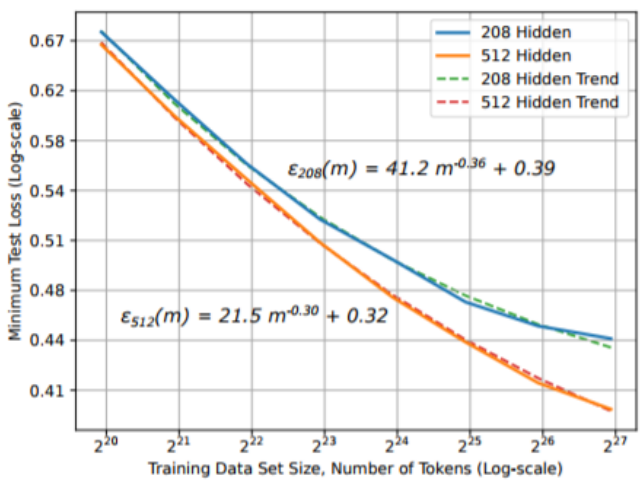
Logarithmic relationship between the dataset size and accuracy

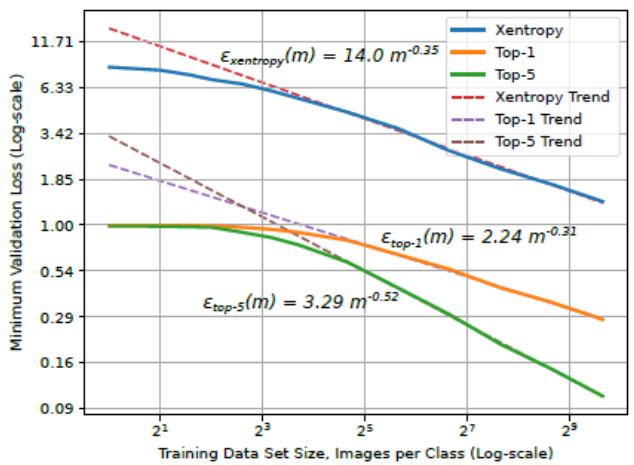


- Translation
- Language Models
- Character Language Models
- Image Classification
- Attention Speech Models





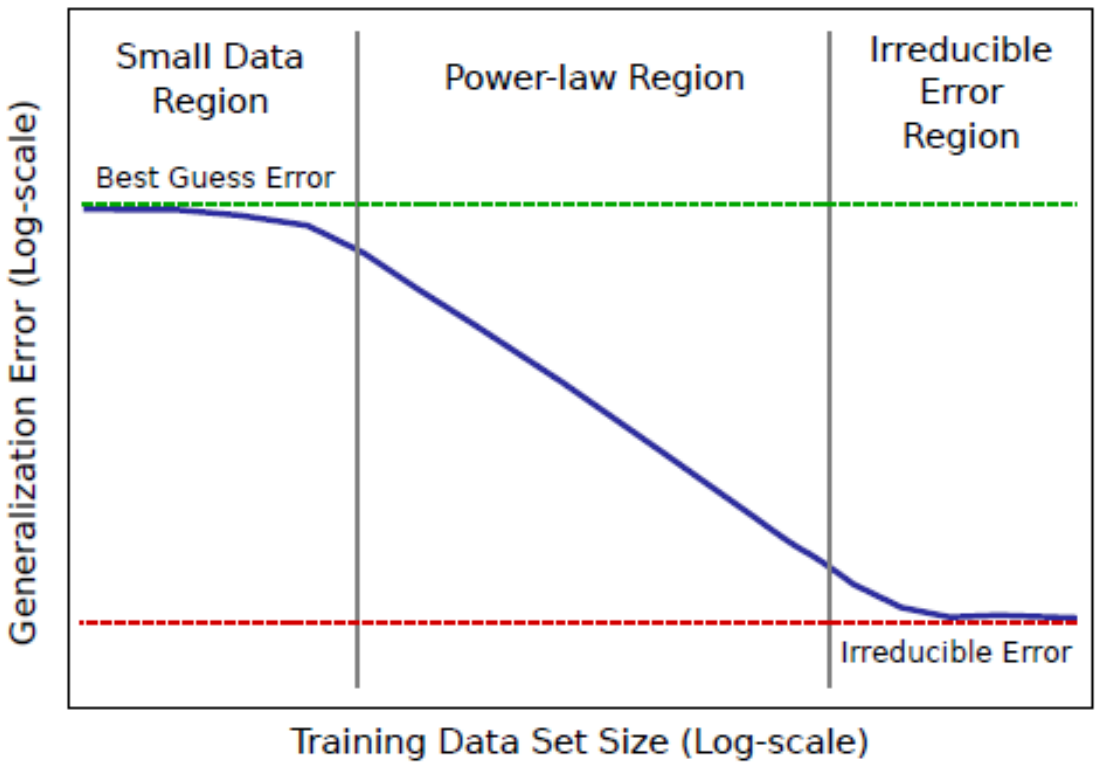




Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv preprint arXiv:1712.00409.

NEW PROMISE

Logarithmic relationship between the dataset size and accuracy

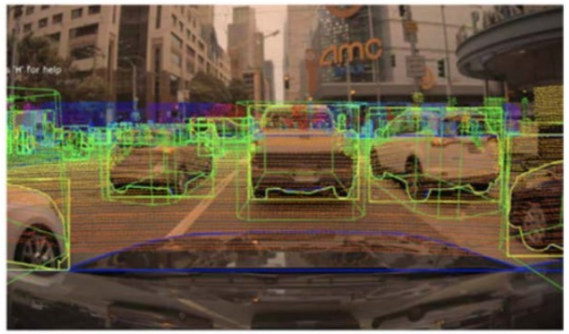


Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv preprint arXiv:1712.00409.

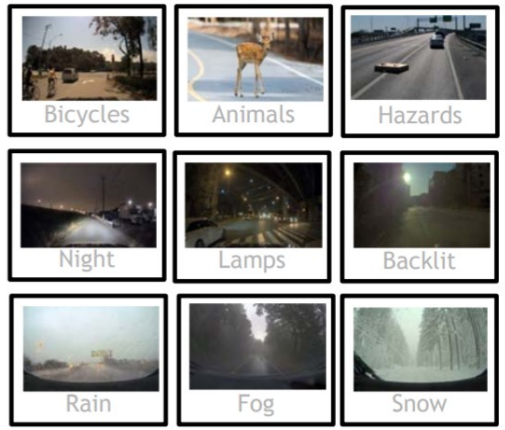
THE COMPLEXITY

Direct impact on data collection and labelling effort

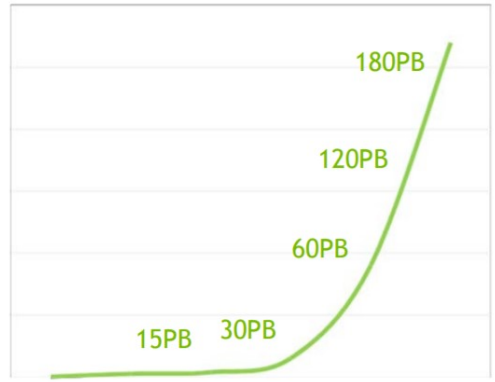
More Functionality × More Conditions = MASSIVE Data



New features (i.e. lane keeping) require new data…



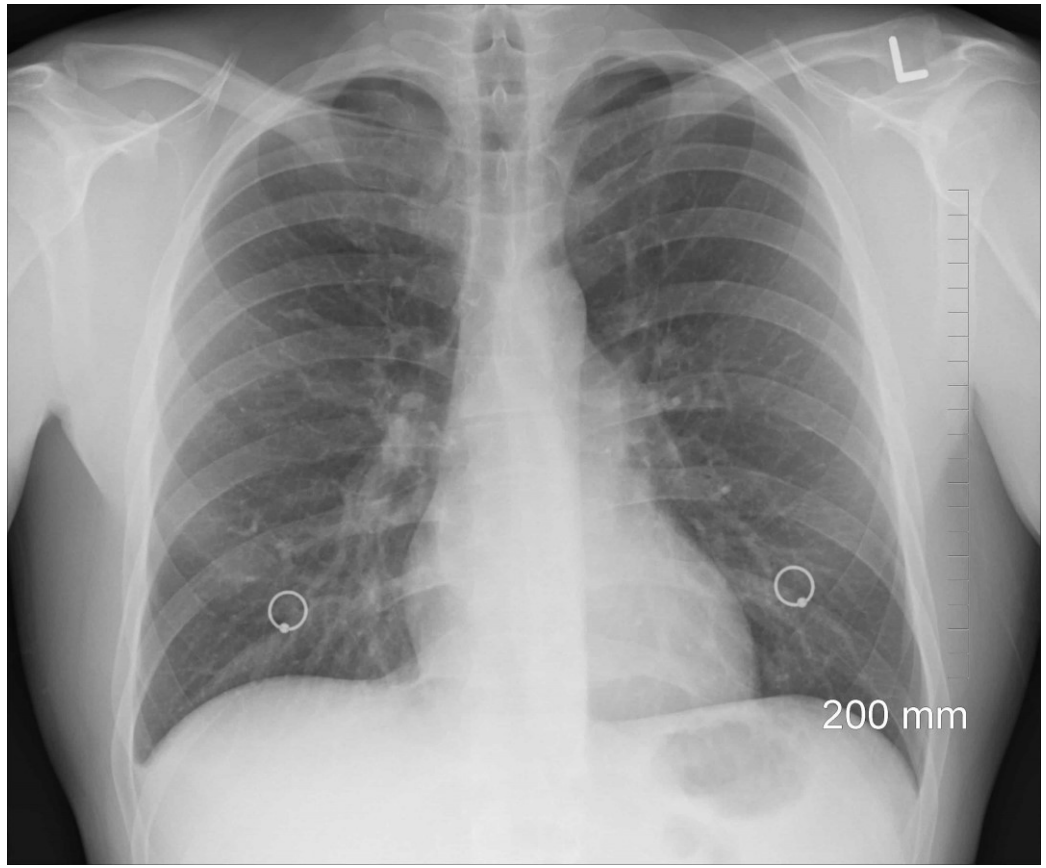
…and require more real examples to meet safety targets…



…resulting in exponential data and compute needs

THE COMPLEXITY

What about problems that require much higher level of expertise



THE COMPLEXITY

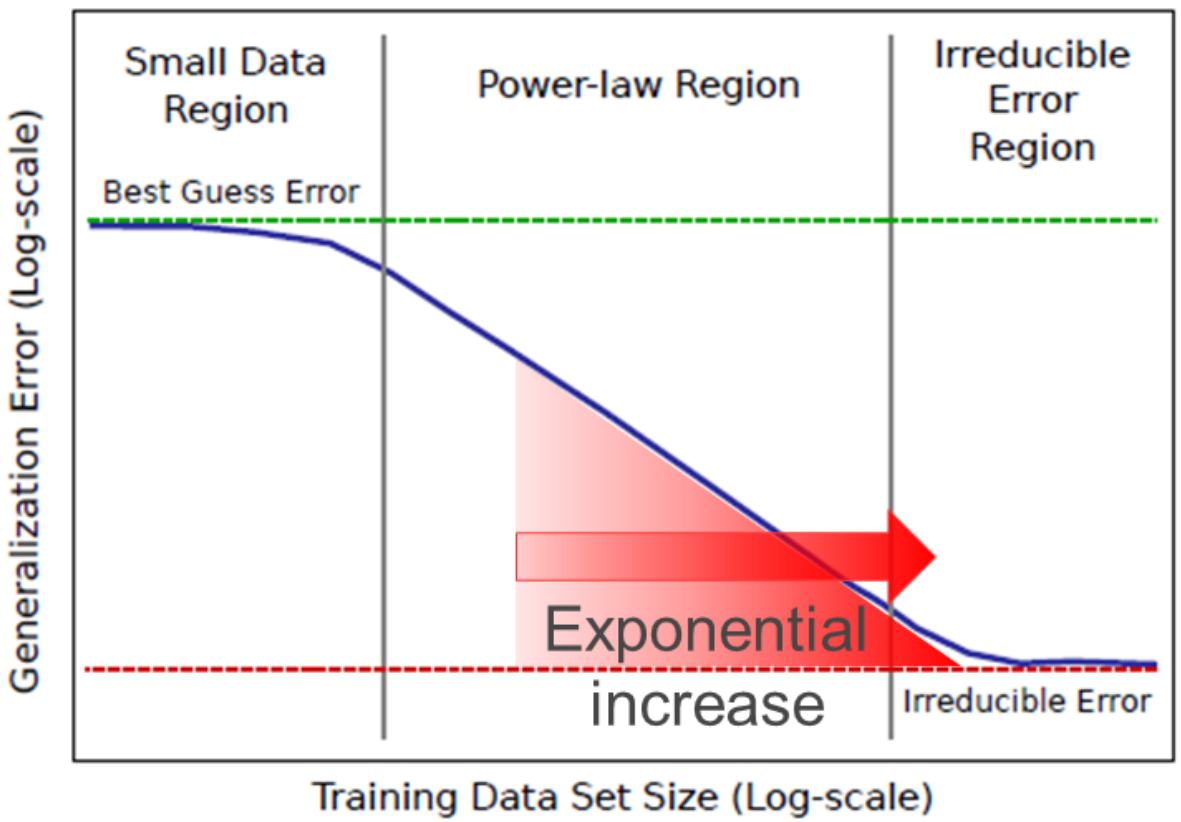
What about problems that require much higher level of expertise

Nawet prostsze problemy niż diagnoza raka wymagają często ekspertyzy.

Even problems that are simpler than cancer diagnosis often require expert judgment.

THE COST OF LABELLING

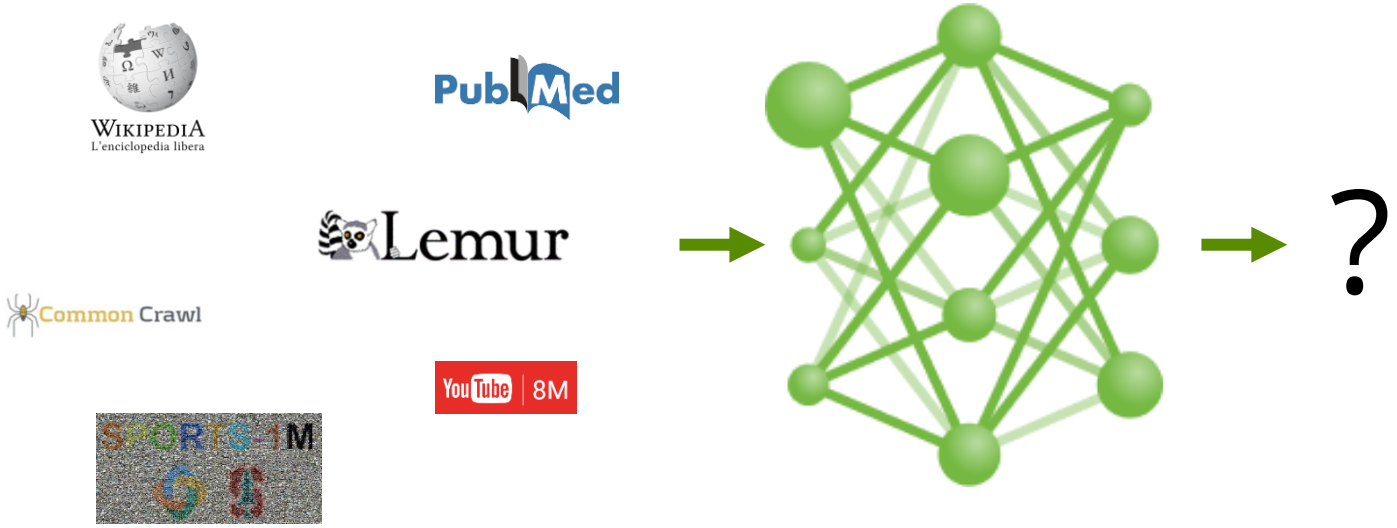
Limits the utility of deep learning models



Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv preprint arXiv:1712.00409.

SELF AND UN-SUPERVISED LEARNING

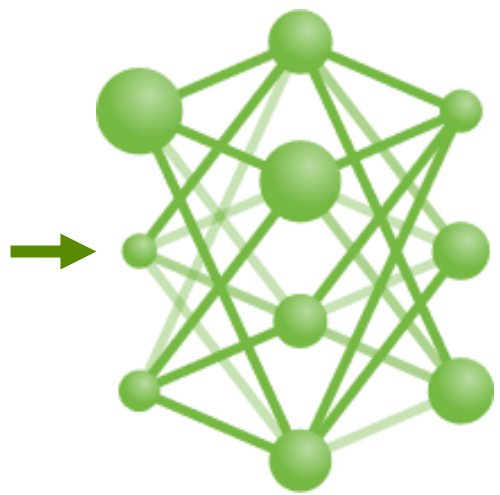
What to do in the absence of labels



SELF-SUPERVISED AND UNSUPERVISED LEARNING

Natural Language Processing – Masked Language Models

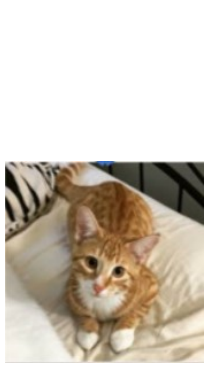
Lorem ipsum [MASK] sit amet, consectetur adipiscing elit.

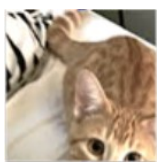


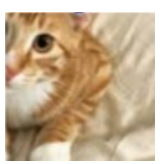
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

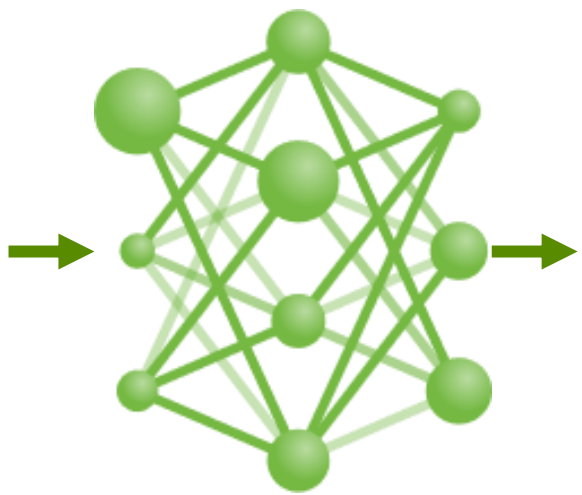
SELF-SUPERVISED AND UNSUPERVISED LEARNING

Computer Vision – Contrastive Learning





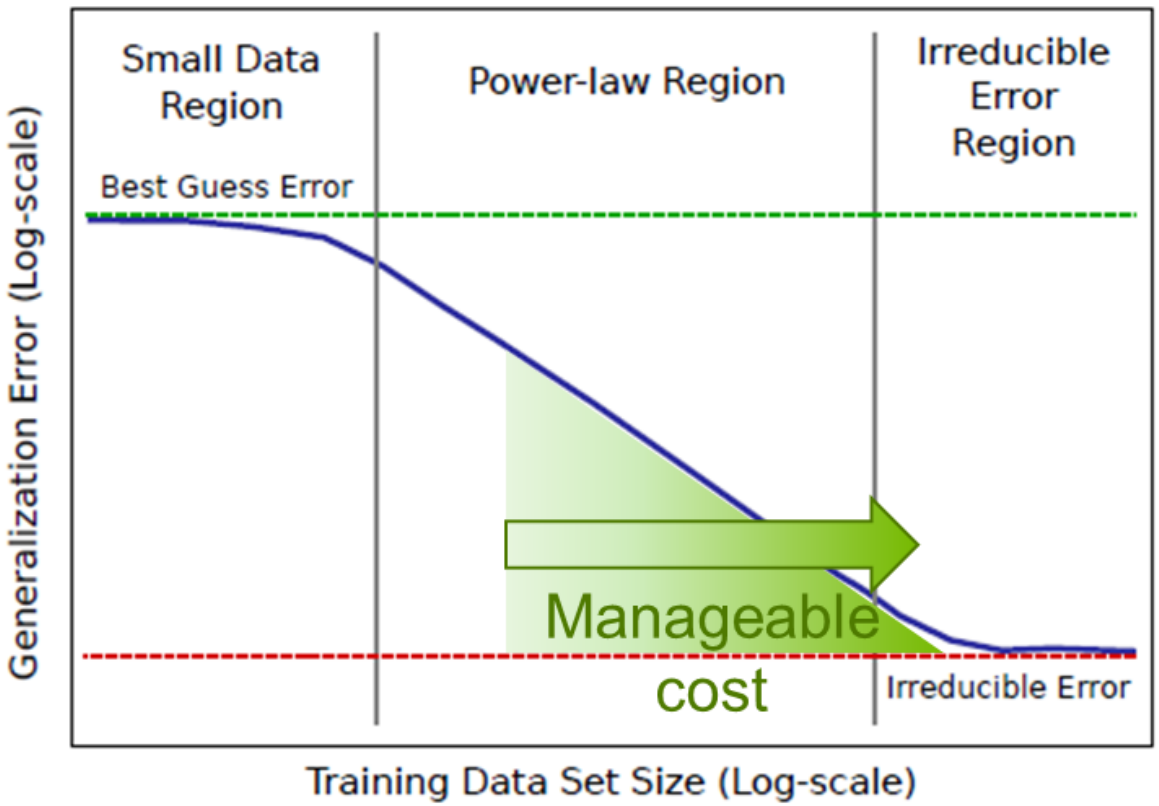




Do the pictures have the same origin: Yes/No

THE COST OF LABELLING

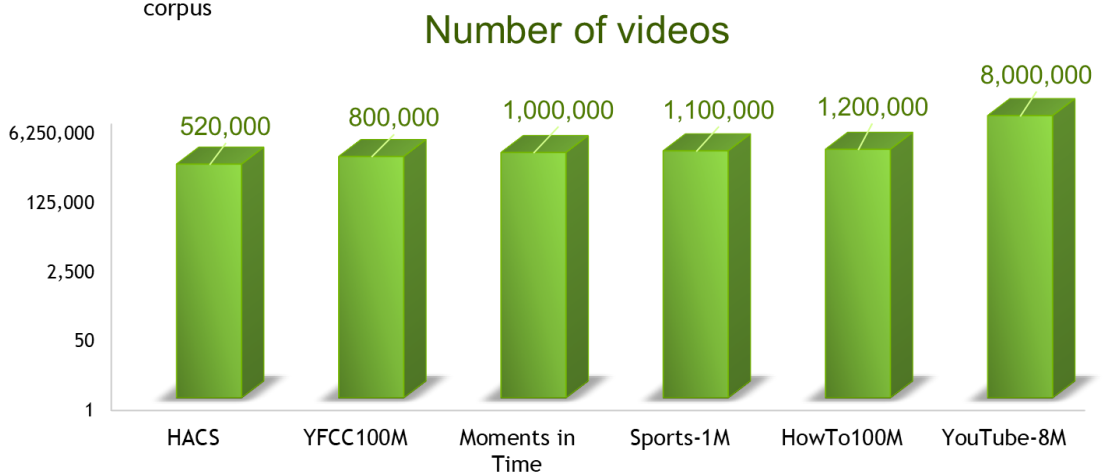
Semi supervised models



Hestness, J., Narang, S., Ardalani, N., Diamos, G., Jun, H., Kianinejad, H., ... & Zhou, Y. (2017). Deep Learning Scaling is Predictable, Empirically. arXiv preprint arXiv:1712.00409.

SELF SUPERVISED AND UNSUPERVISED LEARNING

Abundance of unlabelled data in Text



SCALING LAWS

Scaling Laws apply to NLP

As you increase the dataset size, you must also increase the model size

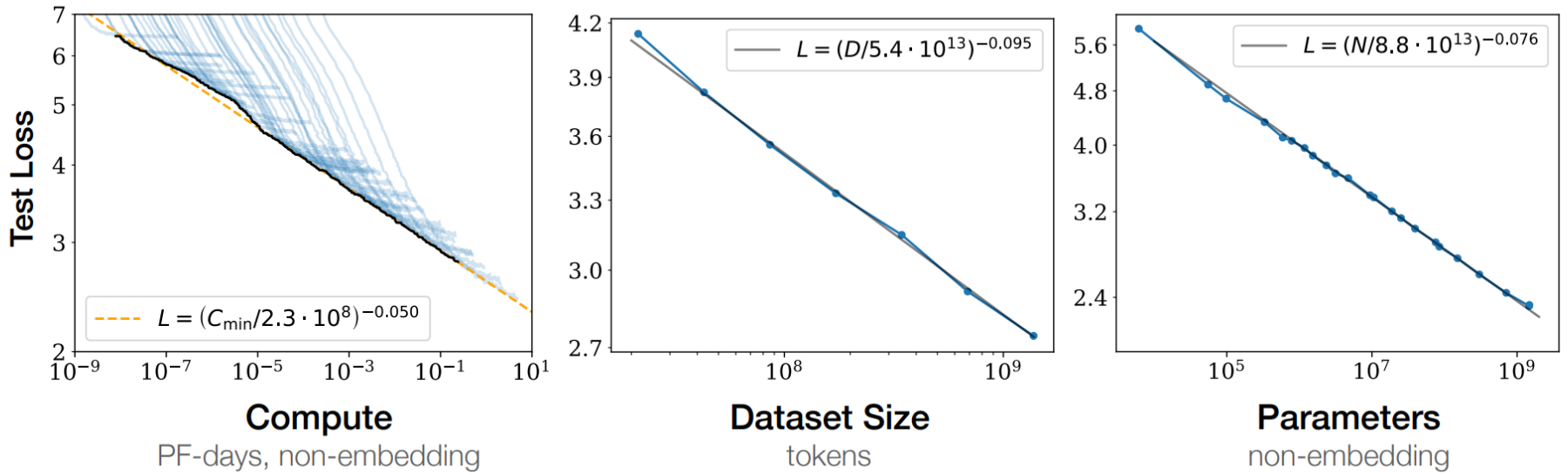


Figure 1 Language modeling performance improves smoothly as we increase the model size, datasetset size, and amount of compute[2] used for training. For optimal performance all three factors must be scaled up in tandem. Empirical performance has a power-law relationship with each individual factor when not bottlenecked by the other two.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

Scaling Laws apply to computer vision too

Increase in performance is proportional to the model size and dataset size

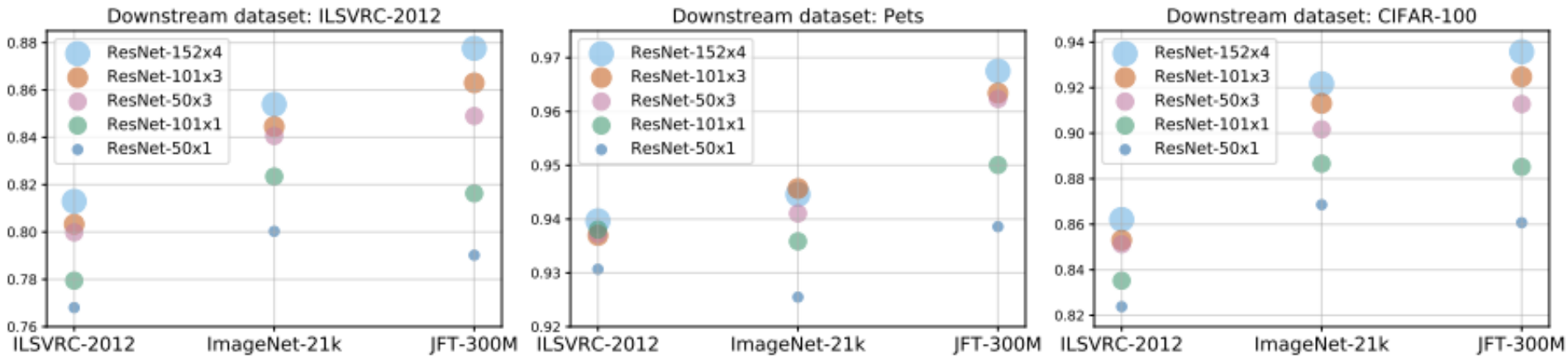


Fig. 5: Effect of upstream data (shown on the x-axis) and model size on downstream performance. Note that exclusively using more data or larger models may hurt performance; instead, both need to be increased in tandem.

IT IS MORE THAN JUST ACCURACY

Importance of Dataset size

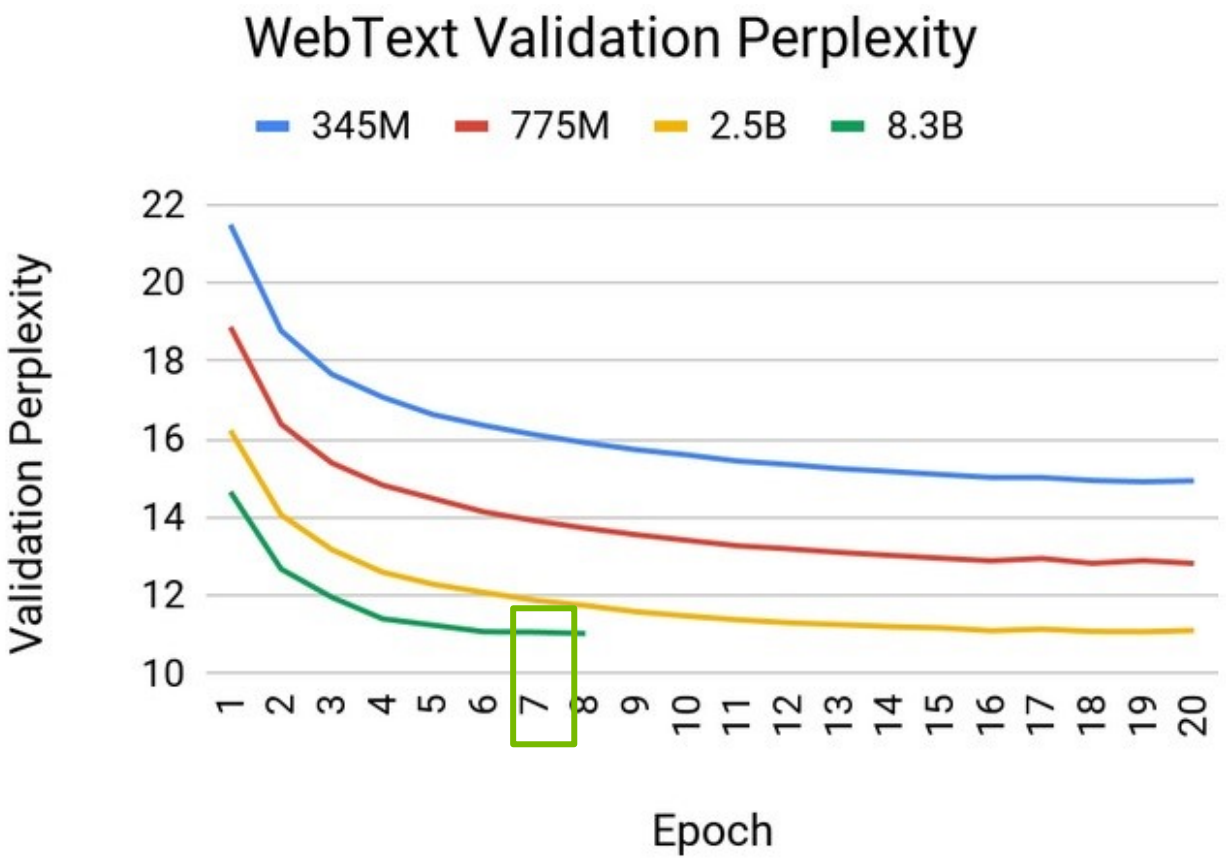
Dataset size more important than neural network design

"… more importantly, we find that the precise architectural hyperparameters are unimportant compared to the overall scale of the language model."

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... & Amodei, D. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.

Even more importantly

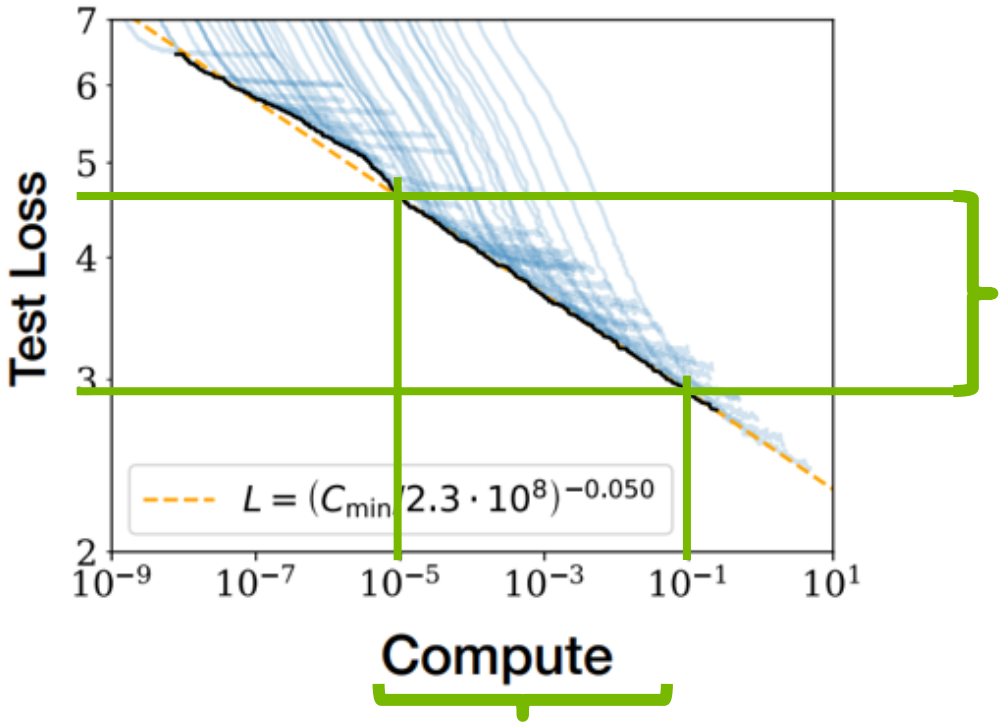
Large Neural Networks use data more efficiently



Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. arXiv preprint arXiv:1909.08053
Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*..

Are Large language models worth it?

The cost of incremental improvement



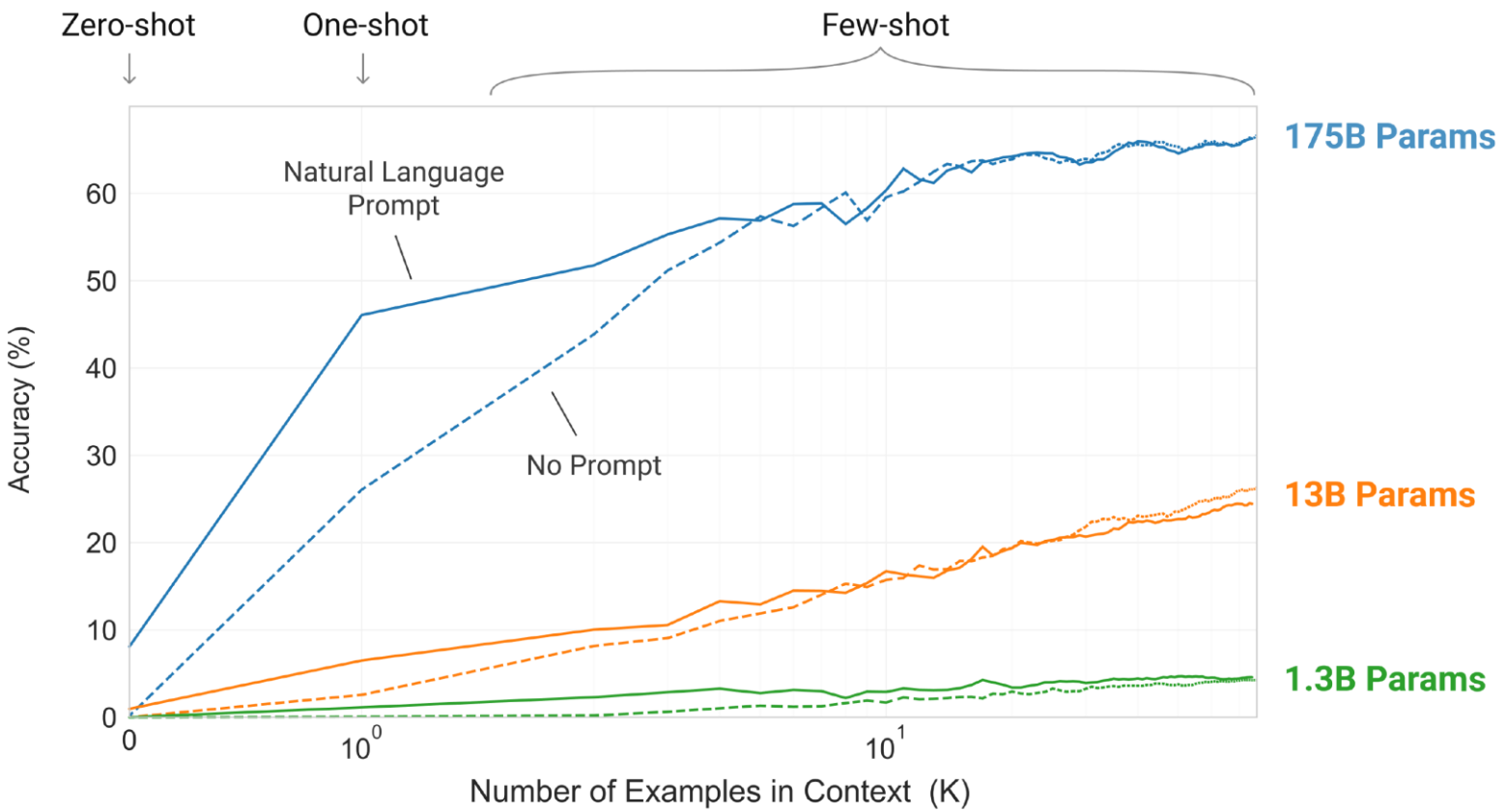
Are we building those models only for the small incremental improvement in their performance?

Is it worth all the engineering and computational investment?

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, Sam McCandlish. Scaling Laws for Autoregressive Generative Modeling.2020

Few shot learning

Learning from far fewer examples



Zero-shot

The model predicts the answer given only a natural language description of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
cheese =>                           ← prompt
```

One-shot

In addition to the task description, the model sees a single example of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
sea otter => loutre de mer          ← example
cheese =>                           ← prompt
```

Few-shot

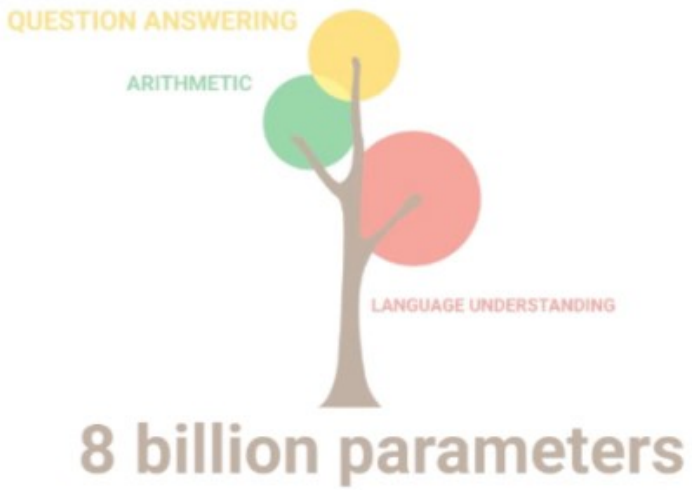
In addition to the task description, the model sees a few examples of the task. No gradient updates are performed.

```
Translate English to French:        ← task description
sea otter => loutre de mer          ← examples
peppermint => menthe poivrée        ←
plush girafe => girafe peluche      ←
cheese =>                           ← prompt
```

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Agarwal, S. (2020). **Language models are few-shot learners**. arXiv preprint arXiv:2005.14165..

Model sizes vs tasks

LLM



https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html

Chatbot

From small quantitative gains to large qualitative ones

PERSPECTIVE

WHAT DO I MEAN BY BIG

GPT-3 size comparison: 538x Bigger than BERT-Large

Not a linear scale

3.14*10^8 PFLOPS

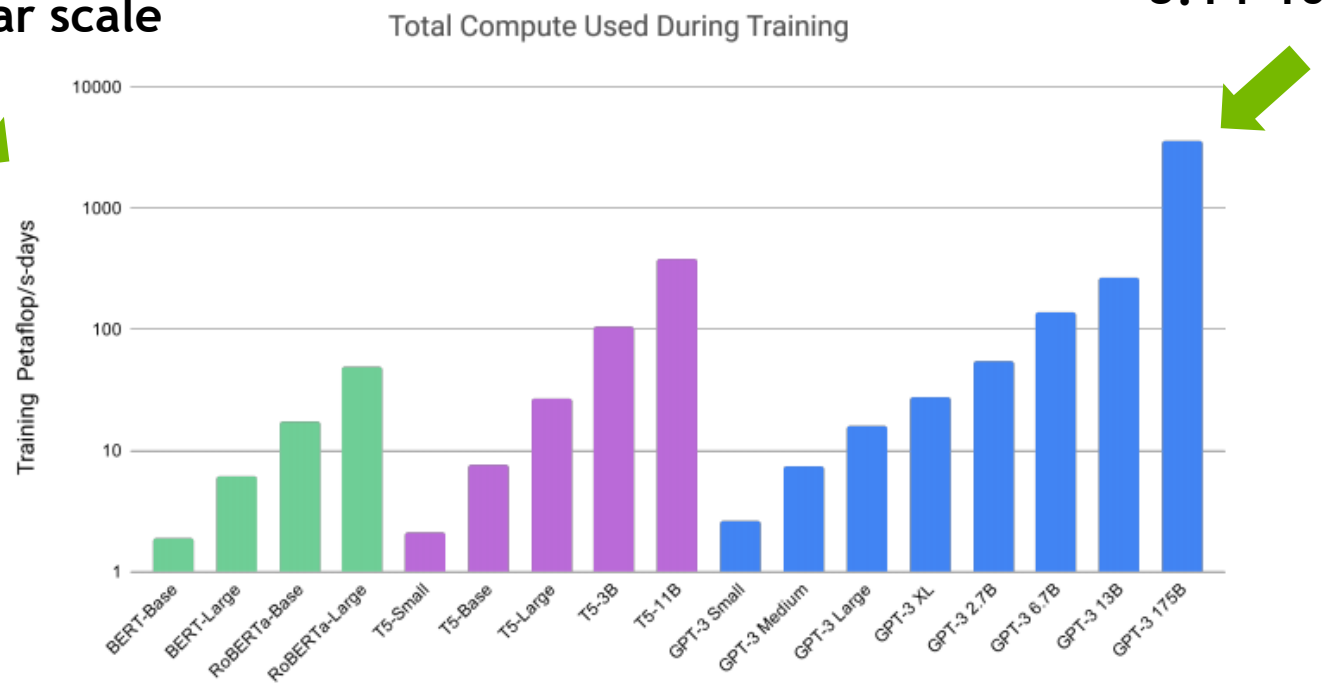


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

WHAT DO I MEAN BY BIG

GPT-3 size comparison: 538x Bigger than BERT-Large

Not a linear scale

~31 years on a single A100

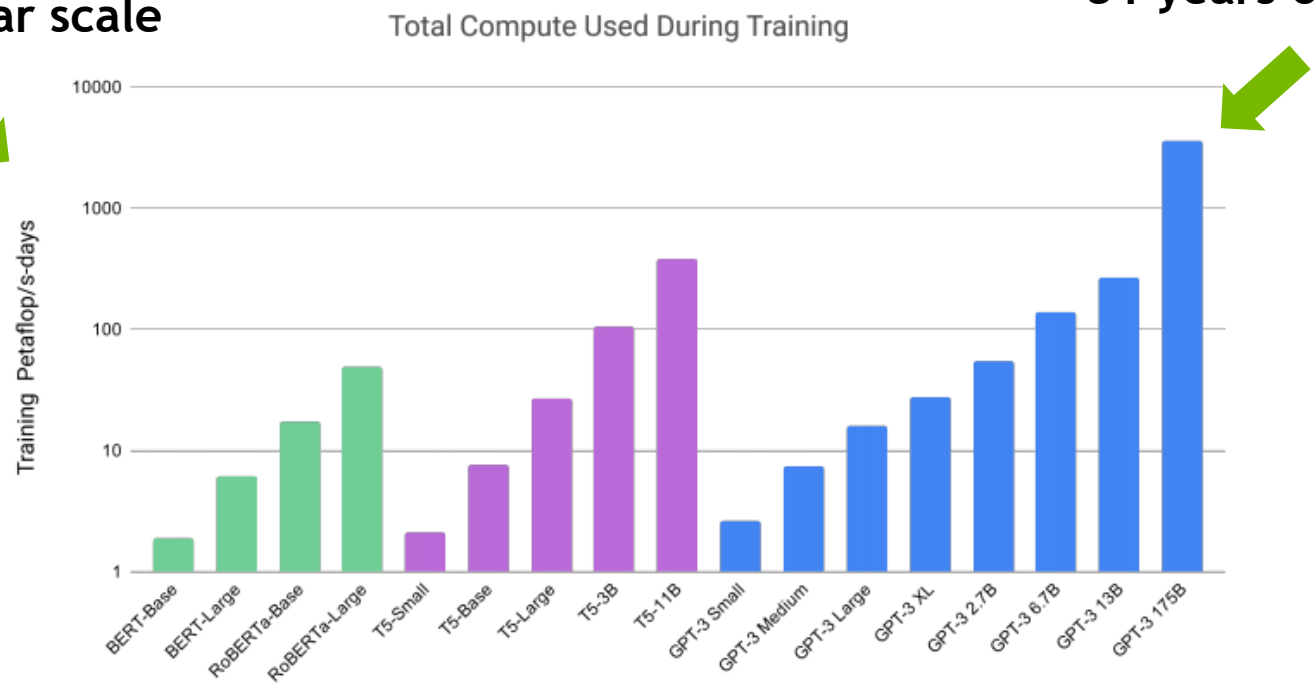


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

ESTIMATE COMPUTE NEEDED

Calculate how many hours/days compute resource need

paper : https://arxiv.org/pdf/2005.14165.pdf

```
[2]: import numpy as np
T=300*1e+9 #oftokens in the dataset
#P=175*1e+9 # number of model parameters
n= 480 # Berzelius 480 # number of GPUs in the compute cluster

def calculate_days_needed(T , P , n ,x):
    if x is None:
        return '1-2 weeks'
    else:
        #x=140*1e+12 # TeraFlop/s per GPU
        tot=8*T*P
        div=n*x
        compute_sec=tot/div
        #convert compute seconds to days
        to_days=round(compute_sec/(3600*24),1)
        return to_days

GPT3_models_labels=[ 'gpt3_2.7B', 'gpt3_6.7B','gpt3_13B', 'gpt3_175B']
GPT3_model_params=[ 2.7*1e+9, 6.7*1e+9 , 13*1e+9, 175*1e+9,1*1e+12 ]
GPT3_model_params_str=['1.3 Billion' ,'2.7 Billion', '13 Billion', '175 Billion']
#according to the table above
GPT3_X=[127*1e+12, 130*1e+12,135*1e+12,140*1e+12 ]
print("all below are measured with dataset size **300 billion** measured in tokens \n")
for gpt3_name, gpt3_params, gpt3_param_str, x in zip(GPT3_models_labels,GPT3_model_params,G
    days_needed=calculate_days_needed(T,gpt3_params,n,x)
    print(" ----------------------------------------------------------------------------
    print(" language model :{} with {} number of parameters , it will need {} days to compu
```

```
all below are measured with dataset size **300 billion** measured in tokens

 ----------------------------------------------------------------------------
 language model :gpt3_2.7B with 1.3 Billion number of parameters , it will need 1.2 days to compute

 ----------------------------------------------------------------------------
 language model :gpt3_6.7B with 2.7 Billion number of parameters , it will need 3.0 days to compute

 ----------------------------------------------------------------------------
 language model :gpt3_13B with 13 Billion number of parameters , it will need 5.6 days to compute

 ----------------------------------------------------------------------------
 language model :gpt3_175B with 175 Billion number of parameters , it will need 72.3 days to compute
```

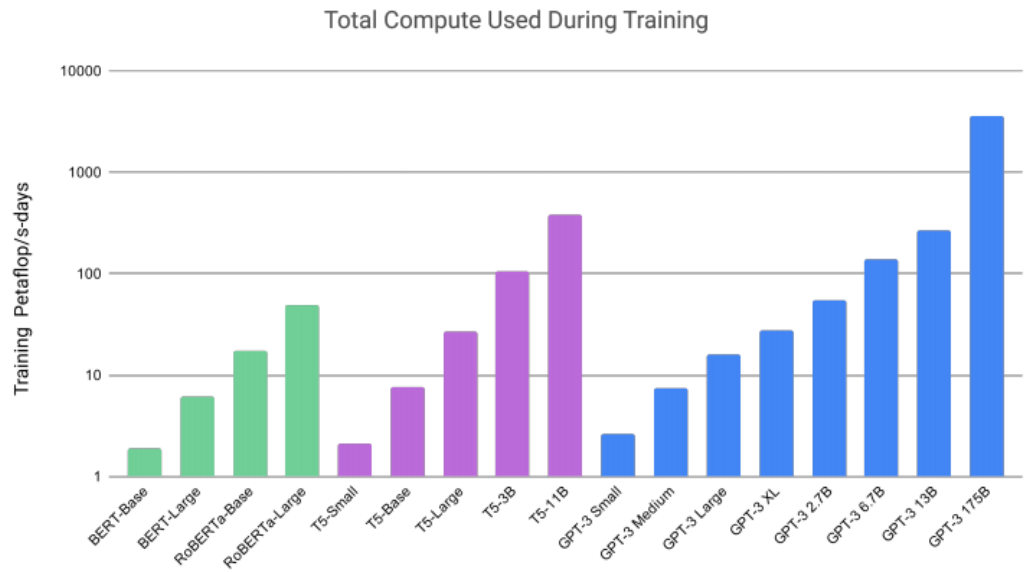


Figure 2.2: Total compute used during training. Based on the analysis in Scaling Laws For Neural Language Models [KMH+20] we train much larger models on many fewer tokens than is typical. As a consequence, although GPT-3 3B is almost 10x larger than RoBERTa-Large (355M params), both models took roughly 50 petaflop/s-days of compute during pre-training. Methodology for these calculations can be found in Appendix D.

Source :https://arxiv.org/pdf/2005.14165.pdf

GPU Memory occupation

1. model weights
2. optimizer states
3. gradients
4. forward activations saved for gradient computation
5. temporary buffers
6. functionality-specific memory

```
Tue Jan 17 15:04:19 2017
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 367.57                 Driver Version: 367.57                    |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 1080    Off  | 0000:01:00.0      On |                  N/A |
| 72%   78C    P2    90W / 200W |   7830MiB /  8105MiB |     98%      Default |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX 1080    Off  | 0000:02:00.0     Off |                  N/A |
|  2%   48C    P8    13W / 200W |      1MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX 1080    Off  | 0000:05:00.0     Off |                  N/A |
| 53%   67C    P2    56W / 200W |   7830MiB /  8113MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0      1261    G   /usr/lib/xorg/Xorg                           140MiB |
|    0      4065    C   python                                      7655MiB |
|    0     10013    G   compiz                                        31MiB |
|    2      4233    C   python                                      7827MiB |
+-----------------------------------------------------------------------------+
```

GPU Memory occupation

In details

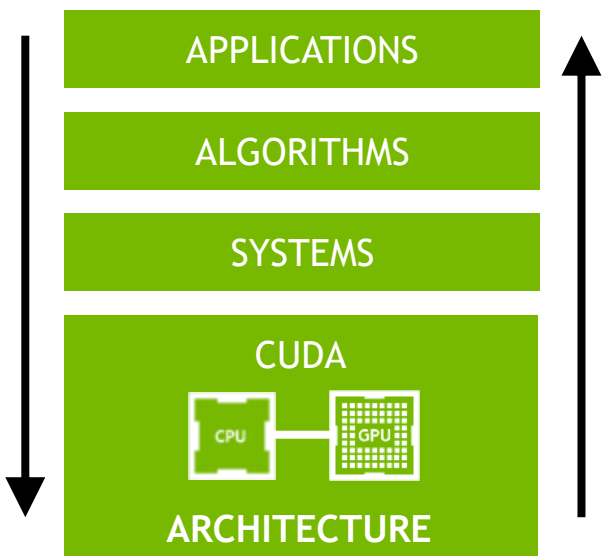
- **Model Weights**
  - 4 bytes * number of parameters for fp32 training
  - 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16 in memory)
- **Optimizer States**
  - 8 bytes * number of parameters for normal AdamW (maintains 2 states)
  - 2 bytes * number of parameters for 8-bit AdamW optimizers like bitsandbytes
  - 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state)
- **Gradients**
  - 4 bytes * number of parameters for either fp32 or mixed precision training (gradients are always kept in fp32)
- **Forward Activations**
  - size depends on many factors, the key ones being sequence length, hidden size and batch size

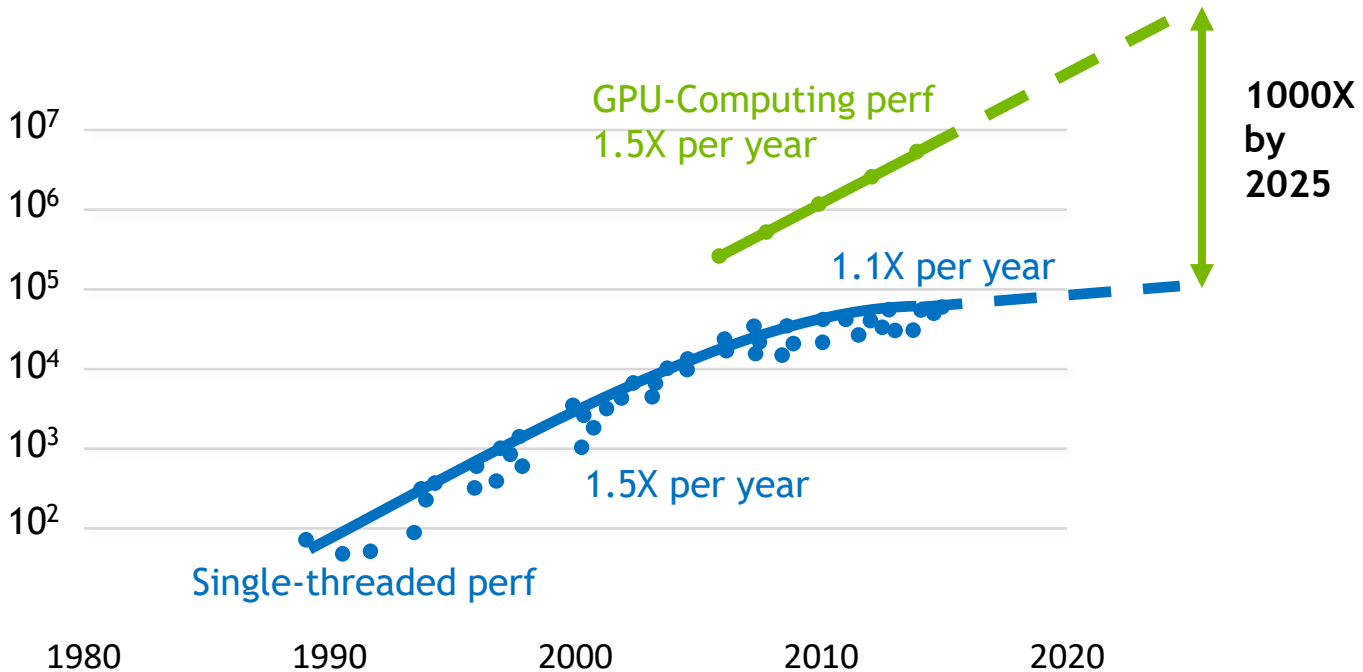
3B-parameter model

T5-3b

- **AdamW** uses 8 bytes for each parameter, here the optimizer will need (8*3) **24GB** of GPU memory.
- **Adafactor** uses slightly more than 4 bytes, so (4*3) **12GB** and then some extra.
- **8bit BNB** quantized optimizer will use only (2*3) **6GB** if all optimizer states are quantized.
- A standard **Adam** uses 16 bytes for each parameter, so (8*3) **48GB** of GPU memory.

RISE OF GPU COMPUTING





Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

NVIDIA

NVIDIA H100

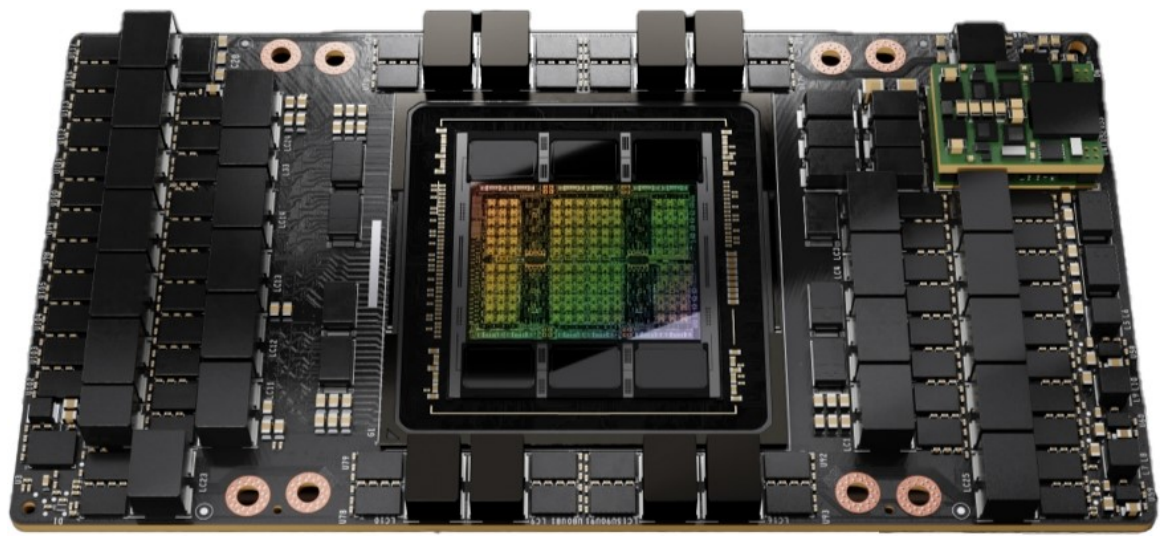
Unprecedented Performance, Scalability, and Security for Every Data Center

HIGHEST AI AND HPC PERFORMANCE
4PF FP8 (6X)| 2PF FP16 (3X)| 1PF TF32 (3X)| 60TF FP64 (3X)
3TB/s (1.5X), 80GB HBM3 memory

TRANFORMER MODEL OPTIMIZATIONS
6X faster on largest transformer models

HIGHEST UTILIZATION EFFICIENCY AND SECURITY
7 Fully isolated & secured instances, guaranteed QoS
2nd Gen MIG | Confidential Computing

FASTEST, SCALABLE INTERCONNECT
900 GB/s GPU-2-GPU connectivity (1.5X)
up to 256 GPUs with NVLink Switch | 128GB/s PCI Gen5



FP8, FP16, TF32 performance include sparsity. X-factor compared to A100

HOW CAN WE HANDLE THIS COMPLEXITY?

Objectives

- Fit very large models into limited hardware
  - e.g. t5-11b is 45GB in just model params
- Significantly speed up training
  - finish training that would take a year in hours

Optimization techniques

primer

- Leverage AMP (Automatic Mixed Precision)
- Activate Gradient Checkpoint
- Empty CUDA Cache:
  - torch.cuda.empty_cache()
- Manually launch Garbage Collection:
  - Import gc
  - gc.collect()
- Enable 8-bit Adam
- Remove bias for convolutional layers followed by batch normalization
- Implement Gradient Accumulation

- Implement Parallelism
  - Data
  - Model
  - Pipeline
  - Tensor
- Offload tensors to CPU
- Shard parameters (Zero Redundancy Optimizer)
- Overlap computing and communication
- Improve data loader
  - Constant Buffer Optimization
  - Contiguous Memory Optimization
- Use Fused Kernel
- Low-Rank adaptation (specific for LLM)

UTILIZING A SINGLE GPU EFFICIENTLY – PRELIMINARY COMMENTS

Arithmetic Intensity

- The operation is said to be **compute-bound** or **data-bound** depending on which one finishes last, with the former scenario being preferable here.

- The threshold for a compute-bound operation is described through the concept of **arithmetic intensity** (ratio between the amount of computation and data transfer).

- A **NVIDIA A100** has a peak computational power of 312 teraflops for half-precision and a memory bandwidth of 2039 GB/s, for an arithmetic intensity threshold of **143 flops/B**.

- A binary addition with an arithmetic intensity of 1/6 lies deeply in the memory-bound region, while the multiplication of two 1024x1024 matrices has an arithmetic intensity of 341 and is compute-bound.

Transformers architecture

- **Tensor Contractions**
  - Linear layers and components of Multi-Head Attention all do batched matrix-matrix multiplications.
- **Statistical Normalizations**
  - Softmax and layer normalization are less compute-intensive than tensor contractions, and involve one or more reduction operations.
- **Element-wise Operators**
  - Biases, dropout, activations, and residual connections.

Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

DATA MOVEMENT IS ALL YOU NEED: A CASE STUDY ON OPTIMIZING TRANSFORMERS https://arxiv.org/pdf/2007.00072.pdf

IMPORTANT CUDNN FLAGS

https://pytorch.org/docs/stable/backends.html#torch-backends-cudnn

Important aspects to consider:

- In NGC containers, the usage of TensorFloat-32 is enabled by default in order to accelerate FP32 calculations using tensor cores on Ampere or newer GPUs.
- Certain classes of CUDA functions are a potential source of non-determinism, such as atomicAdd, where the order of parallel additions to the same value is undetermined and, for floating-point variables, a source of variance in the results.
- cuDNN can automatically determine which combination of primitives is most optimal. Only use this flag when input sizes of a model are no changing!

```
# get the cuDNN version
torch.backends.cudnn.version()

# check availability
torch.backends.cudnn.is_available()

# enabling cuDNN (default = True)
torch.backends.cudnn.enabled = True

# enabling TF32 (default = True for DL)
torch.backends.cudnn.allow_tf32 = True

# enable determinism (default = False)
torch.backends.cudnn.deterministic = False

# enable auto-tuning (default = False)
torch.backends.cudnn.benchmark = True
```

A Note on Time Measurements

https://pytorch.org/docs/stable/backends.html#torch-backends-cudnn

Important aspects to consider:

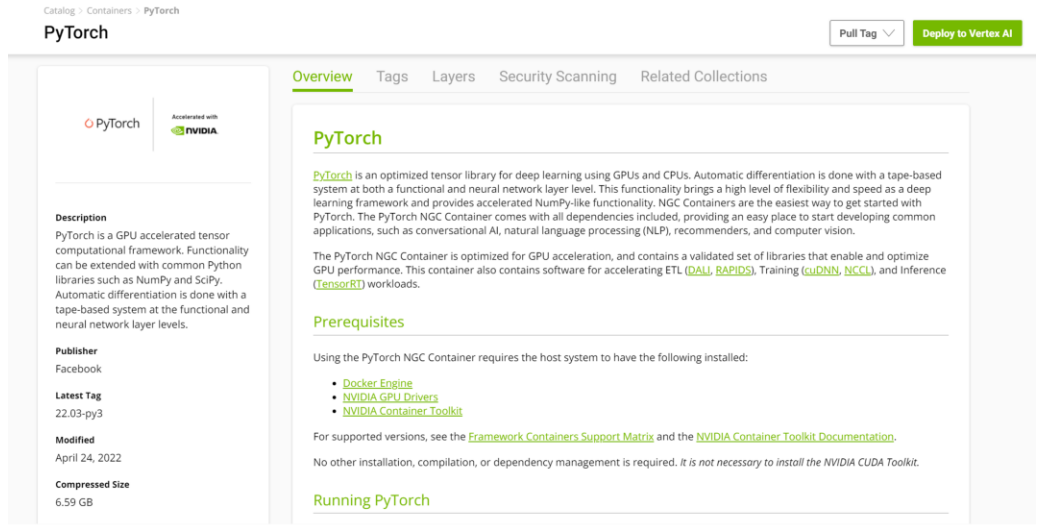
- Be careful with measuring time on the host
- CUDA events are synchronization markers that can be used to monitor the device's progress, to accurately measure timing, and to synchronize CUDA streams.
- Make sure you are measuring large enough workloads.
- Always perform multiple repetitions and average the results.
- Never measure the 1st API call and perform GPU warmup.

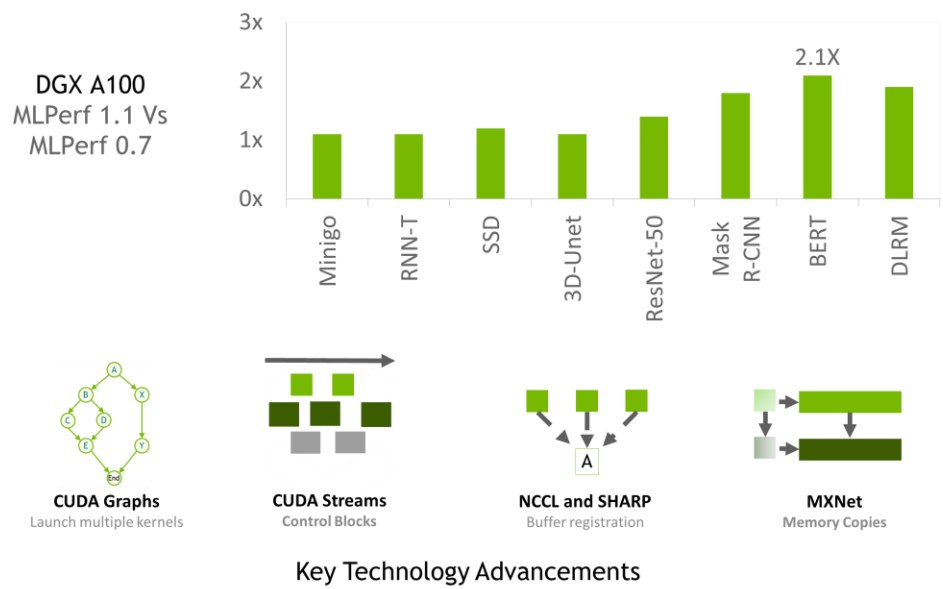
```
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

start.record()
# code to be measured
…
end.record()

torch.cuda.synchronize()

elapsed_time_in_ms = start.elapsed_time(end)
```

WHY USING NGC CONTAINERS?

https://catalog.ngc.nvidia.com/containers

- Always the latest cuDNN version
- Access to all technologies we use for MLPerf (NCCL, SHARP, etc.)
- Achieve reproducibility





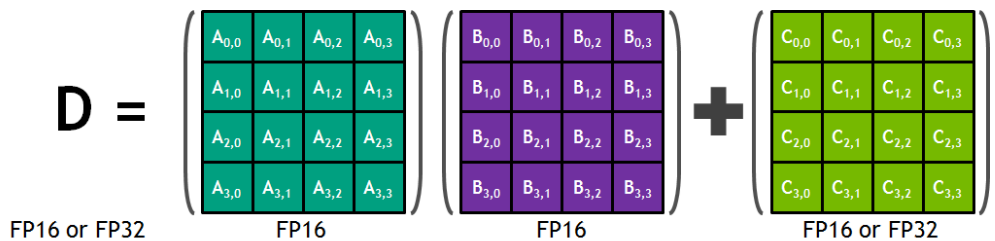
UTILIZING A SINGLE GPU EFFICIENTLY – MAXIMIZING OCCUPANCY & UTILIZATION

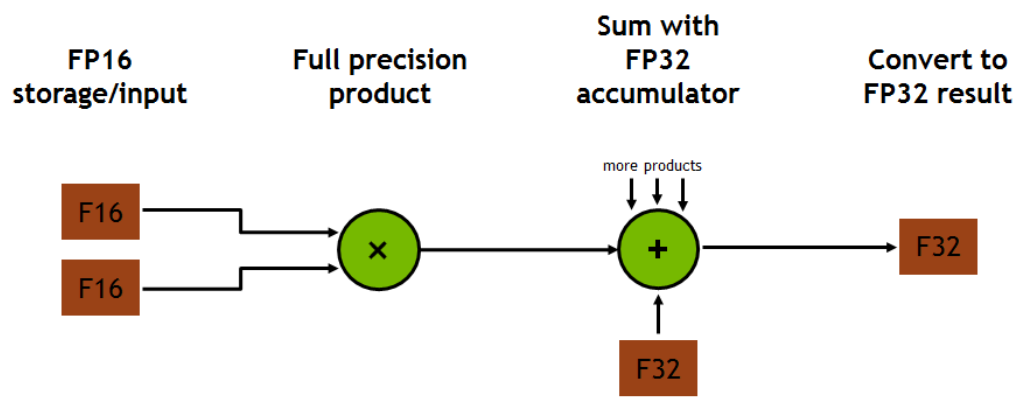
SIZING RECOMMENDATIONS

THE FUNDAMENTAL OPERATION OF DEEP LEARNING - FUSED MULTIPLY ADD

"Matrix tiles = toasts"

- Fused matrix multiply and accumulate (FMA) operations are the core operations of deep learning training and inference.
- 1st generation Tensor Cores (V100) perform 64 floating point FMA mixed-precision operations per clock (FP16 input multiply with full-precision product and FP32 accumulate), i.e., 4 4x4 matrix tiles.
- Higher generation Tensor Cores support additional precisions.

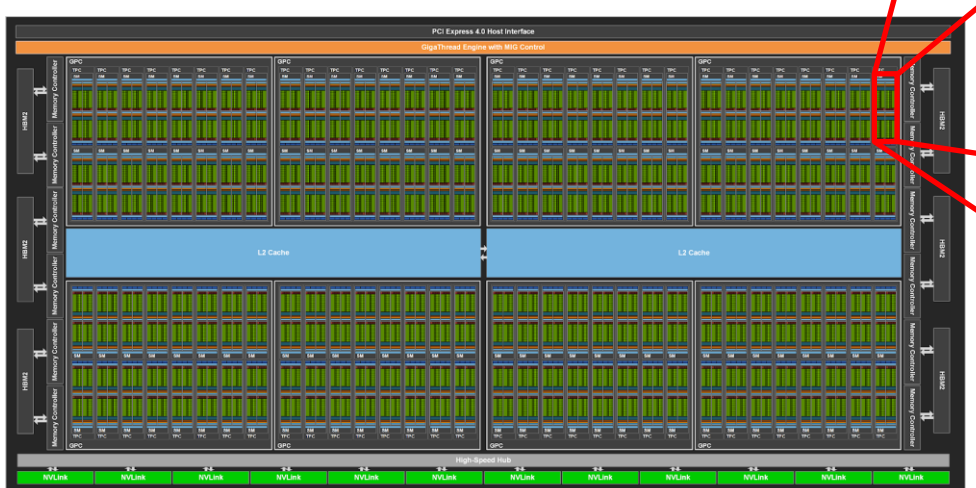




TENSOR CORES

"How many pans do it have?"

- Introduced in the Volta architecture to accelerate matrix multiply and accumulate operations.
- Specific unit of the Streaming Multiprocessors (SMs)
- For an example, an A100 has 108 SMs



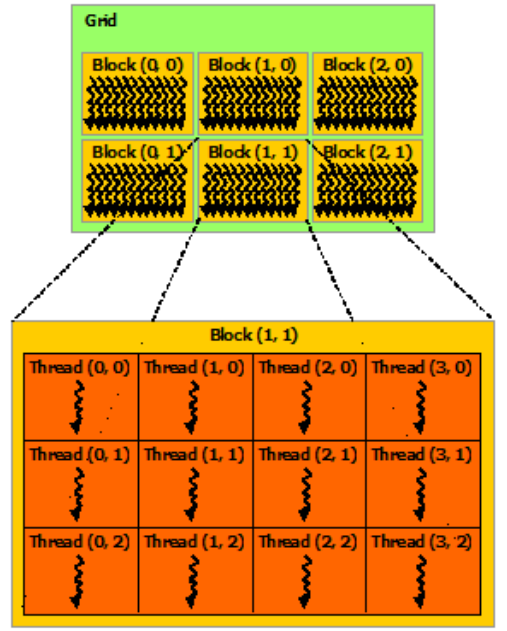
Actually, there are researchers trying to exploit tensor cores for reductions and other ops:
https://arxiv.org/abs/1811.09736

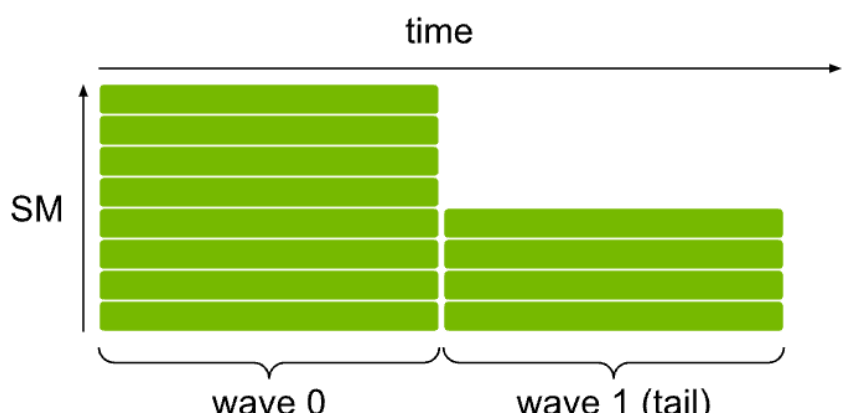
NVIDIA

SIMPLIFIED EXECUTION MODEL

"How to operate with the pans?"

- N *Kernels* are executed in parallel by N different *CUDA threads*.
- Threads are arranged a one-dimensional, two-dimensional, or three-dimensional block of threads, called a *thread block*. A set of thread blocks are launched to execute a function.
- It is usually better the overcommit w.r.t. the number of threads to facilitate instruction latency ("prepare toast while other getting fried").
- When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution.
- It is important to avoid *warp divergence* ("frying toasts with different cooking times in the same pan") whenever possible!
- A set of thread blocks running concurrently is called a *wave*. The more waves, the better to minimize tail effects.

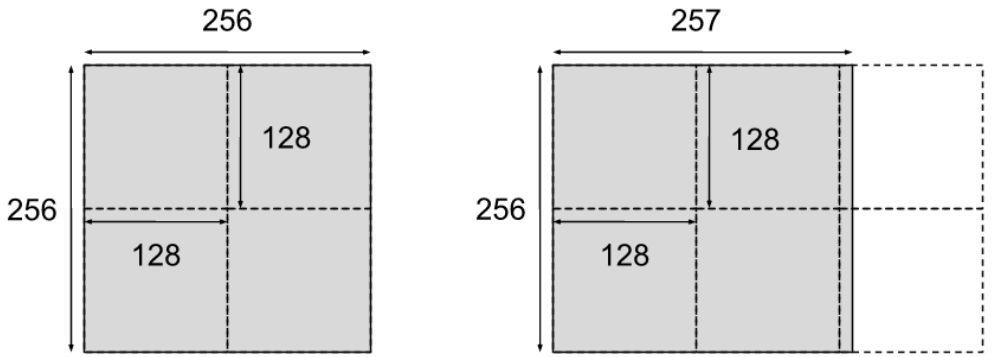


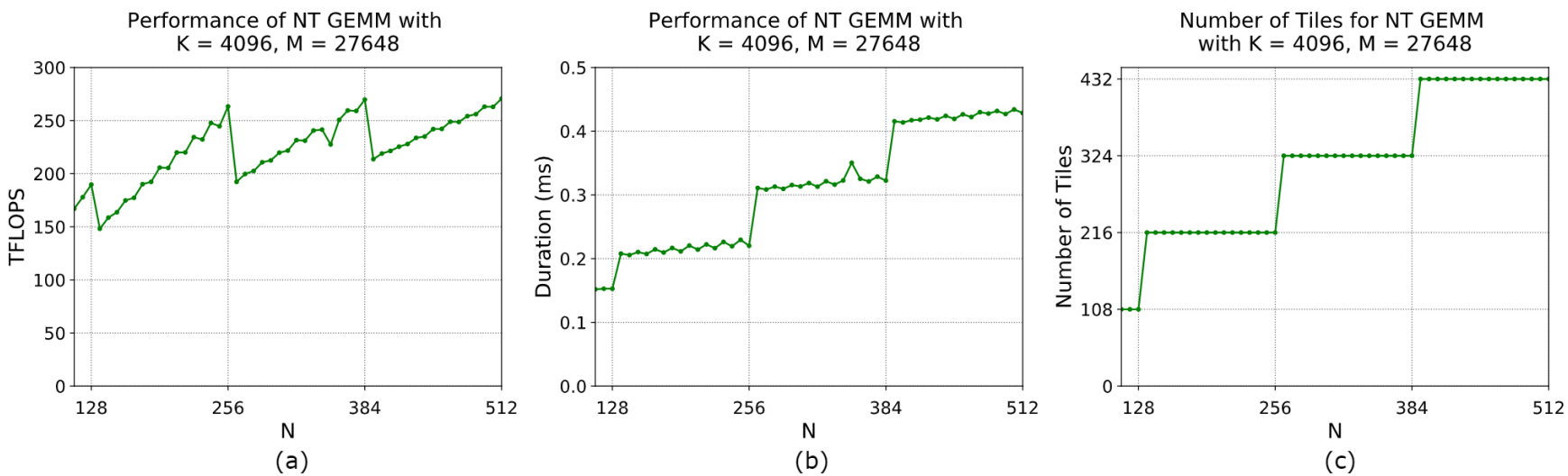


A SIMPLE EXAMPLE

GEMMs

- Tile quantization effect on (a) achieved FLOPS throughput and (b) elapsed time, alongside (c) the number of tiles created.
- Measured with a function that forces the use of 256x128 tiles over the MxN output matrix. In practice, cuBLAS would select narrower tiles (for example, 64-wide) to reduce the quantization effect.
- Experiment performed on NVIDIA A100-SXM4-80GB, CUDA 11.2. cuBLAS 11.4.

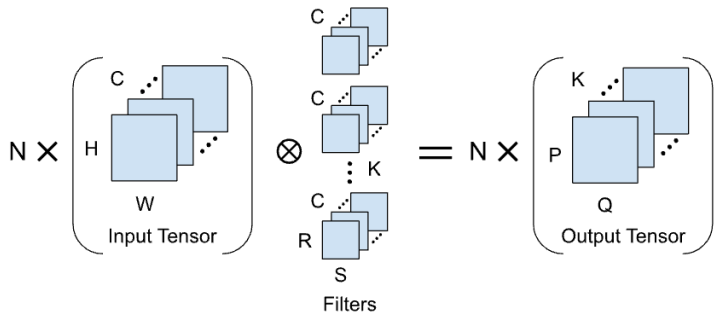




CHECKLIST FOR CONVOLUTIONAL LAYERS

https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html#checklist

- Choose the number of input and output channels to be divisible by 8 (for FP16) or 4 (for TF32). Also consider padding the input channels.
- Choose parameters (batch size, number of input and output channels) to be divisible by at least 64 and ideally 256 to enable efficient tiling and reduce overhead.
- Larger values for size-related parameters (batch size, input and output height and width, and the number of input and output channels) can improve parallelization and hence increase efficiency.
- Make sure auto-tuning is enabled, if applicable.
- Choose tensor layouts in memory to avoid transposing input and output data. We recommend using the NHWC format where possible.



For specific guidance on 2D and particularly 3D convolutions, please also refer to https://docs.nvidia.com/deeplearning/cudnn/best-practices/index.html

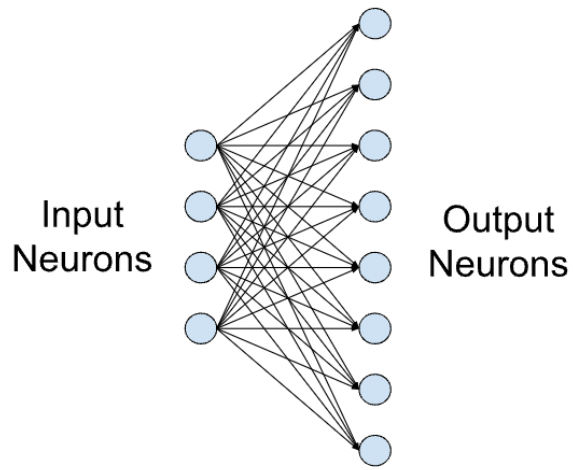
CHECKLIST FOR FULLY CONNECTED LAYERS

https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html#checklist

- Choose the batch size and the number of inputs and outputs to be divisible by 4 (TF32) / 8 (FP16) / 16 (INT8) to run efficiently on Tensor Cores. For best efficiency on A100, choose these parameters to be divisible by 32 (TF32) / 64 (FP16) / 128 (INT8).

- Especially when one or more parameters are small, choosing the batch size and the number of inputs and outputs to be divisible by at least 64 and ideally 256 can streamline tiling and reduce overhead.
  → Larger values for batch size and the number of inputs and outputs improve parallelization and efficiency.

- As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth (NVIDIA A100-SXM4-80GB; this threshold is similar for other A100 and V100 GPUs).



UTILIZING A SINGLE GPU EFFICIENTLY – MAXIMIZING OCCUPANCY & UTILIZATION

TENSOR CORE UTILIZATION

TENSOR CORE CAPABILITIES

FMA operations per clock per SM

	CUDA Cores				Tensor Cores				
NVIDIA Arch.	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4
Volta	32	64	128	256			512		
Turing	2	64	128	256			512	1024	8192
Ampere (A100)	32	64	256	256	64	512	1024	4096	16384
Ampere, sparse						1024	2048	4096	8192

Example - calculate TF32 throughput for A100:

108 (SMs) x 512 (multiply-add ops) x 2 (floating point ops) x 1.41 GHz (clock rate) = 156 TeraFLOPS

MIXED PRECISION TRAINING - THE IDEA

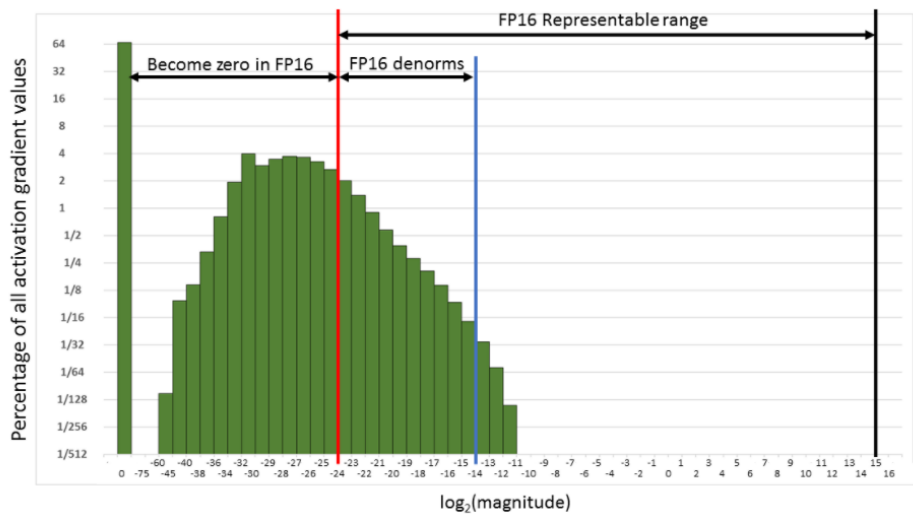
Example: FP32 training of Multibox SSD network

- Histogram shows activation gradient magnitudes throughout FP32 training; both axes are logarithmic.

- Observations:
  - Dynamic range of FP16 would be sufficient to cover the entire histogram. ☺
  - Without “shifting” the histogram, half of the activations would be casted to 0, however. ☹

- Idea: “shifting” = multiplication with a scale factor!

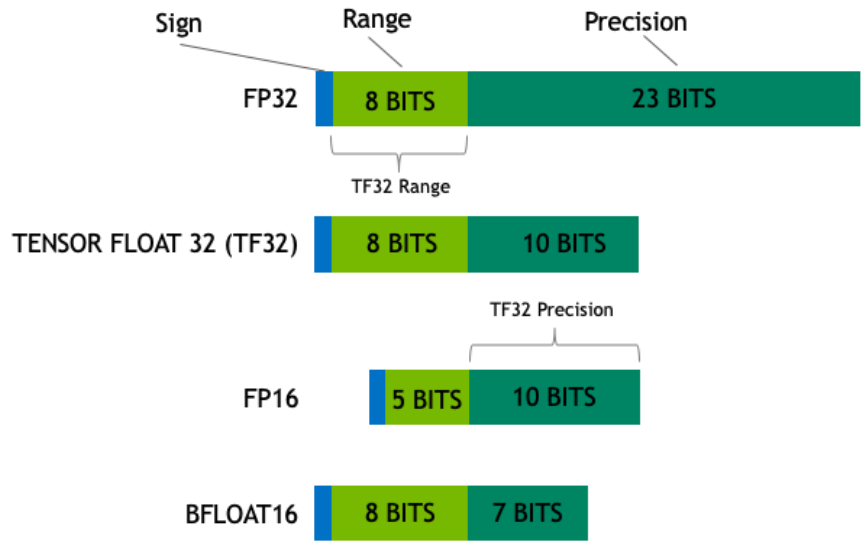
- Concern: Do I need to run a full training in order to find the scaling factor?
  → No, automatic mixed precision comes to the rescue! ☺



A NOTE ON DATA TYPES

Or why TF32 makes sense

- Mixed precision training is mostly about the dynamic range and less about the precision:
  - exponent → dynamic range
  - significand field → precision
- TF32 is a great compromise between FP32 (same range) and FP16 (same precision)
- TF32 is automatically enabled in NGC containers
- No code change is necessary!



HOW TO USE IT?

in pytorch

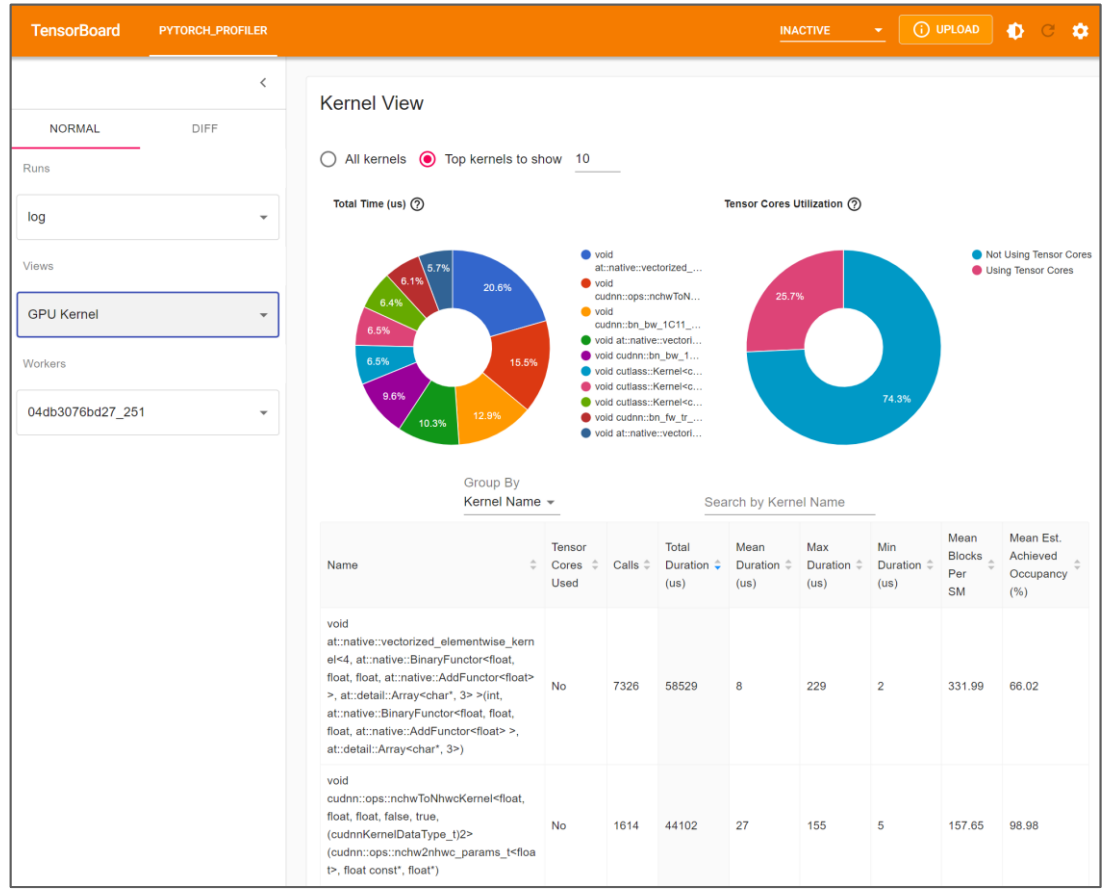
- Backward passes under autocast are not recommended.
- Backward ops run in the same dtype autocast chose for corresponding forward ops.
- scaler.step() first unscales the gradients of the optimizer's assigned params.
- If these gradients contain infs or NaNs, optimizer.step() is skipped.

```
# initialize gradient scaler
scaler = GradScaler()

# training loop
for epoch in epochs:
      for input, target in data:

            # zero gradient buffers
            optimizer.zero_grad()

            # forward pass with autocasting
            with autocast():
                  output = model(input)
                  loss = loss_fn(output, target)

            # call backward() on scaled loss
            scaler.scale(loss).backward()

            # update if no issues
            scaler.step(optimizer)

            # updates the scale for next iteration.
            scaler.update()
```

AM I USING TENSOR CORES?

https://pytorch.org/docs/stable/profiler.html

```
from torch import profiler

prof_schedule = profiler.schedule(wait=2,
                                  warmup=2,
                                  active=5,
                                  repeat=0)

callback =  profiler.tensorboard_trace_handler('./log')

prof = profiler.profile(schedule=prof_schedule,
on_trace_ready=callback,
record_shapes=False,
with_stack=False)

prof.start()

for it in range(num_iterations):
    # code to be profiled
    ...
    prof.step()

prof.stop()
```

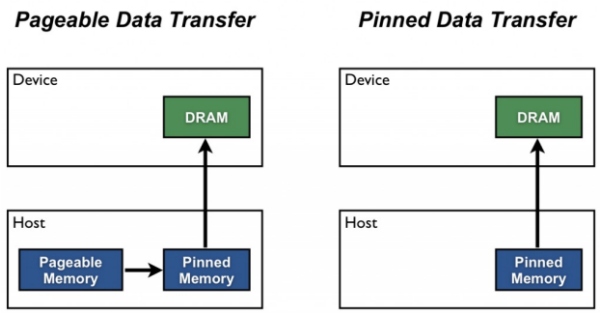


UTILIZING A SINGLE GPU EFFICIENTLY – MINIMIZE LATENCY

OPTIMIZE DATA LOADING

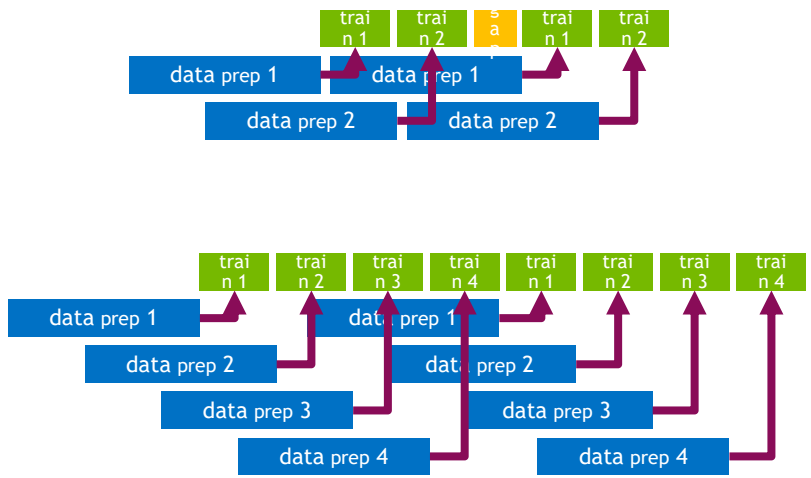
"keep the pan filled"

- If the dataset is small enough, consider moving it entirely onto the GPU.
- Use pinned memory: Host to GPU (H2D) copies are much faster when they originate from pinned (page-locked) memory. This also works for individual tensors!



- Tune the number of workers for loading the data in keep the GPU busy.
- Tune prefetching, i.e., how many samples are prefetched by the dataloader.
- For more information see https://pytorch.org/docs/stable/data.html

```
# in pytorch
loader = DataLoader(dataset, … ,
                    pin_memory=True,
                        num_workers=4,
                        prefetch_factor=2)

# pinning individual tensors
Tensor.pin_memory()
```



Pre-fetching

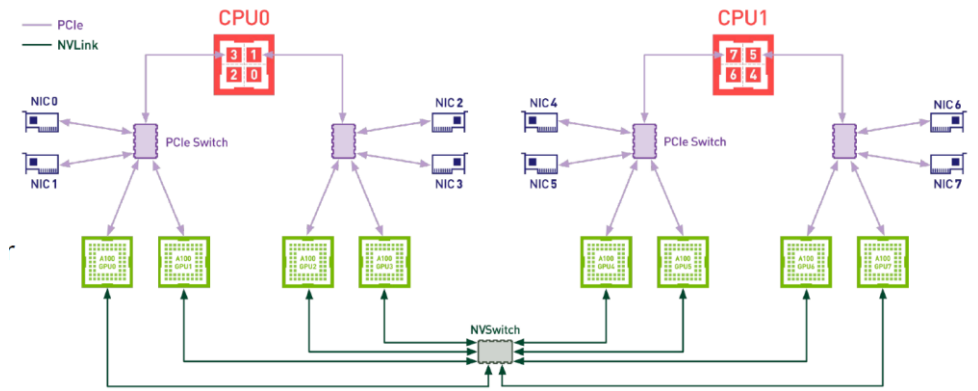
PyTorch

https://www.jpatrickpark.com/post/prefetcher/

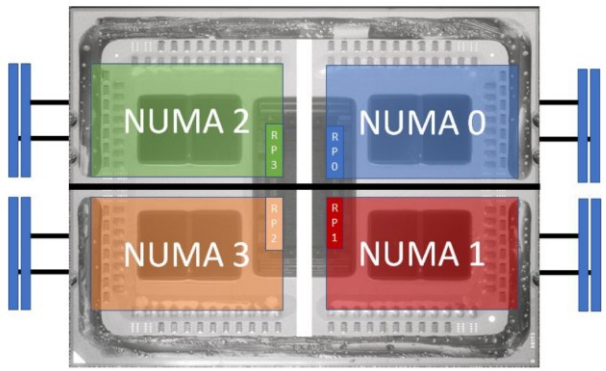
PROCESS/THREAD AFFINITY

Introduction to NUMA on DGX A100

- NUMA = Non-Uniform Memory Access
  - On DGX A100: 4 per socket, 8 per machine (2 sockets)
  - Each NUMA node has 16 physical cores and 2 memory channels with 50GB/s DDR4 BW




```
NUMA node0 CPU(s):    0-15,128-143
NUMA node1 CPU(s):    16-31,144-159
NUMA node2 CPU(s):    32-47,160-175
NUMA node3 CPU(s):    48-63,176-191
NUMA node4 CPU(s):    64-79,192-207
NUMA node5 CPU(s):    80-95,208-223
NUMA node6 CPU(s):    96-111,224-239
NUMA node7 CPU(s):    112-127,240-255
```



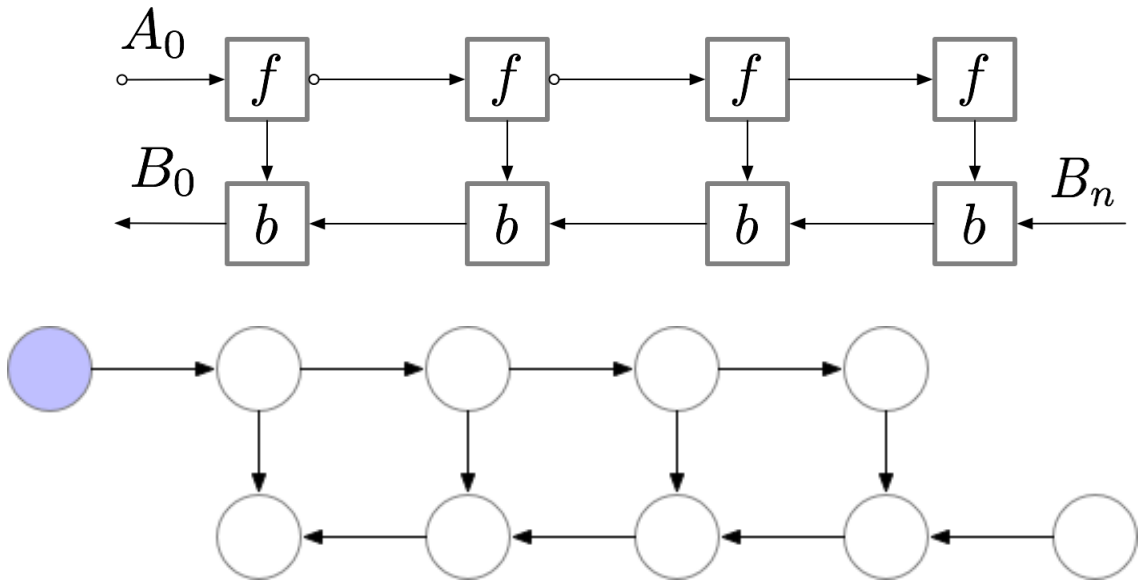

```
$ nvidia-smi topo -m

        GPU0    GPU1    GPU2    GPU3    GPU4    GPU5    GPU6    GPU7    ...  CPU Affinity      [NUMA Node]
GPU0     X      NV12    NV12    NV12    NV12    NV12    NV12    NV12    ...  48-63,176-191     3
GPU1    NV12     X      NV12    NV12    NV12    NV12    NV12    NV12    ...  48-63,176-191     3
GPU2    NV12    NV12     X      NV12    NV12    NV12    NV12    NV12    ...  16-31,144-159     1
GPU3    NV12    NV12    NV12     X      NV12    NV12    NV12    NV12    ...  16-31,144-159     1
GPU4    NV12    NV12    NV12    NV12     X      NV12    NV12    NV12    ...  112-127,240-255   7
GPU5    NV12    NV12    NV12    NV12    NV12     X      NV12    NV12    ...  112-127,240-255   7
GPU6    NV12    NV12    NV12    NV12    NV12    NV12     X      NV12    ...  80-95,208-223     5
GPU7    NV12    NV12    NV12    NV12    NV12    NV12    NV12     X      ...  80-95,208-223     5

Legend:
  X    = Self
  ...
  NV#  = Connection traversing a bonded set of # NVLinks
```

ACTIVATION CHECKPOINT AND GRADIENT ACCUMULATION

Activation Re-computation or gradient checkpointing

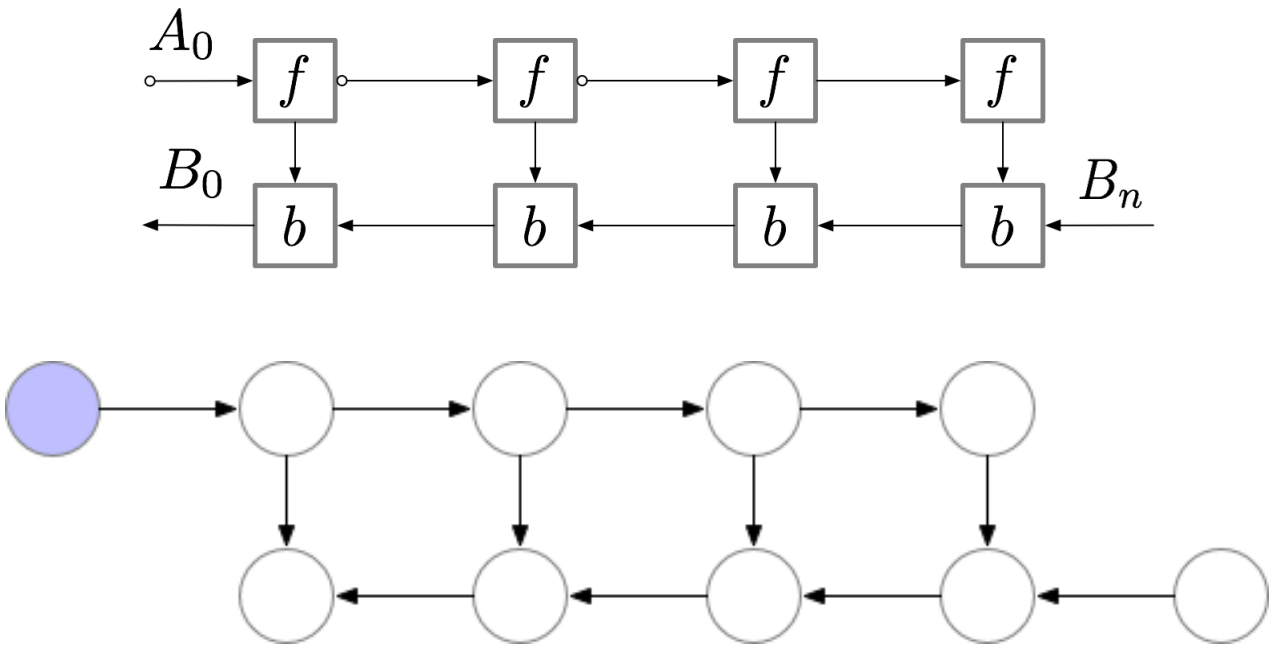
https://pytorch.org/docs/stable/checkpoint.html

- The memory intensive part of training deep neural networks is computing the gradient of the loss by backpropagation.
- By checkpointing nodes in the computation graph defined by your model, and recomputing the parts of the graph in between those nodes during backpropagation, it is possible to calculate gradients at reduced memory cost.
- **When training deep feed-forward neural networks consisting of *n* layers, you can reduce the memory consumption to *O(sqrt(n))*, at the cost of performing one additional forward pass.**



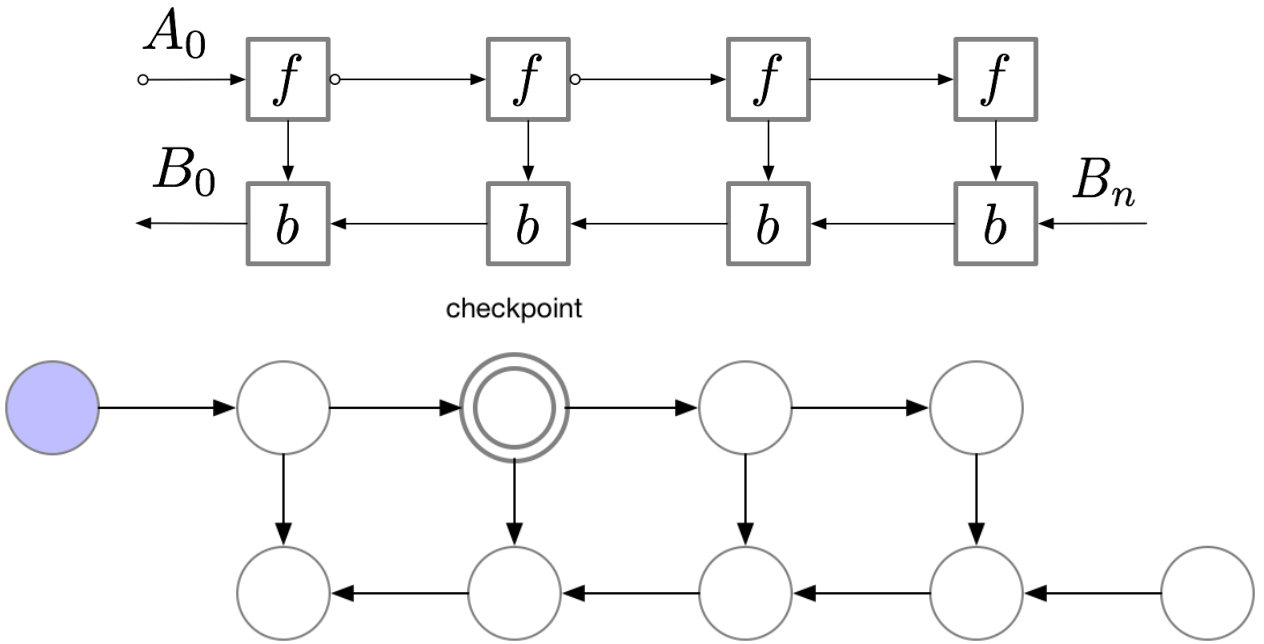
https://github.com/cybertronai/gradient-checkpointing

Activation Re-computation or gradient checkpointing



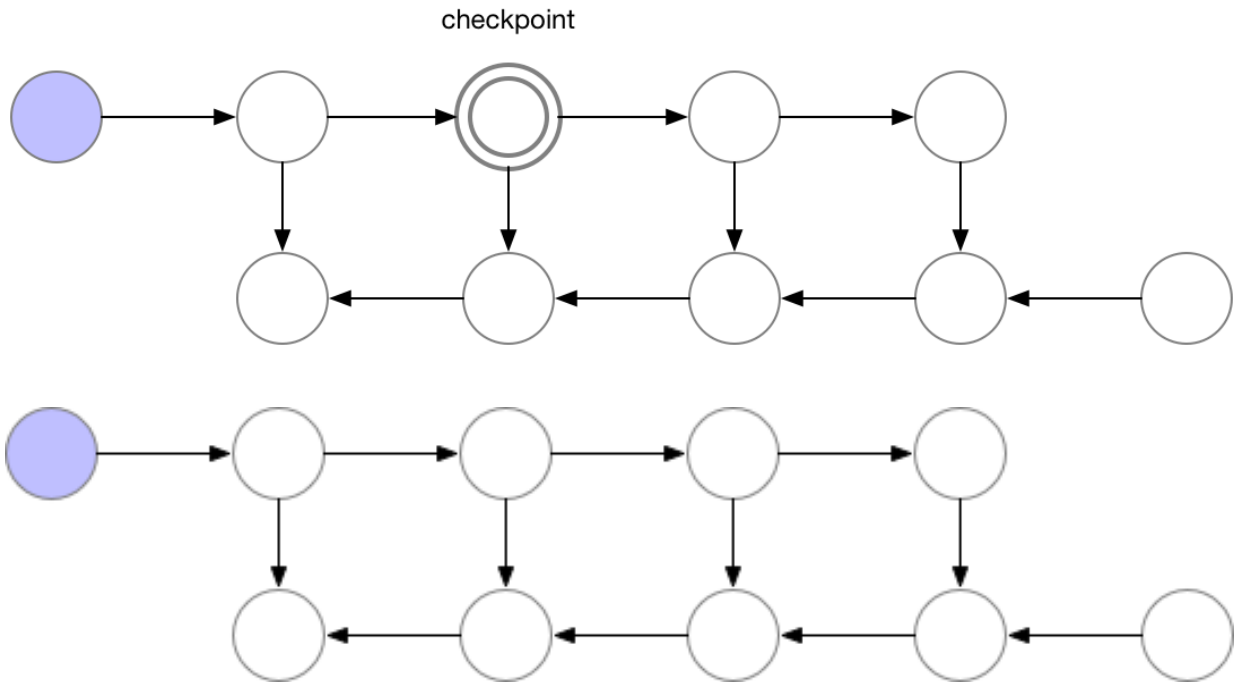
We might for instance simply recompute every node from the forward pass each time we need it.

Activation Re-computation or gradient checkpointing



The strategy we use here is to mark a subset of the neural net activations as checkpoint nodes.

Activation Re-computation or gradient checkpointing



Activation recompute challenges

Microbatch Level Activation Recomputation

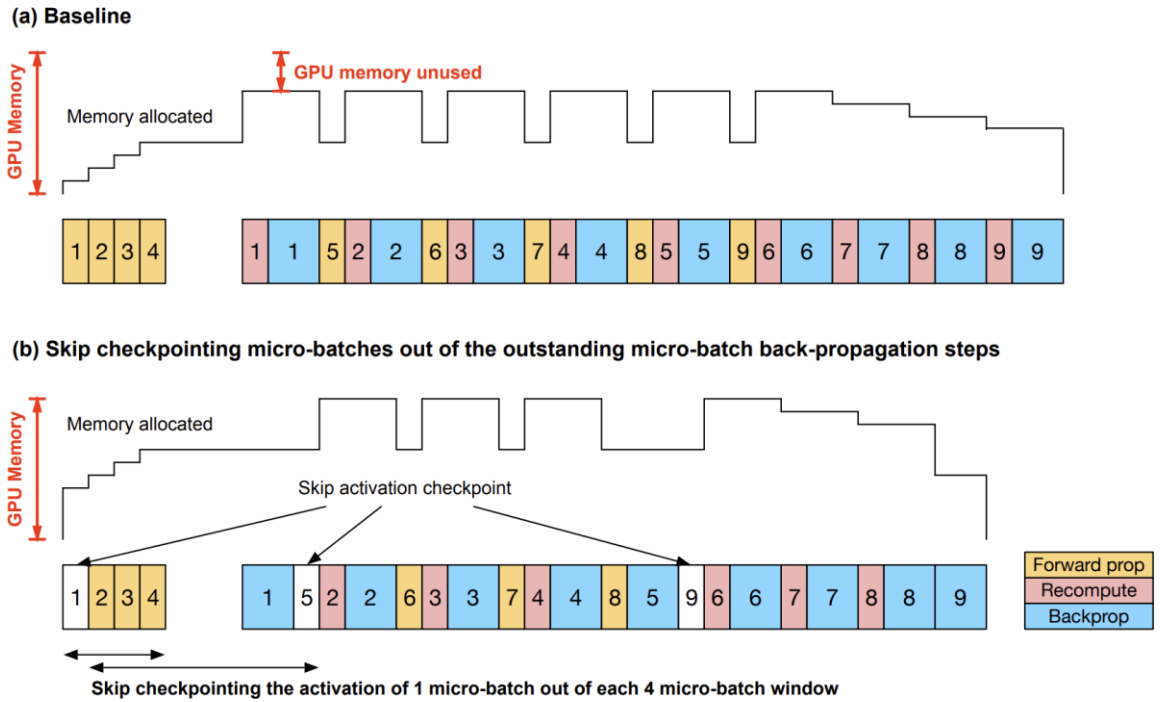


Figure 10: Computation and memory usage patterns of the baseline activation recomputation and microbatch level activation recomputation. Yellow boxes are a forward pass with activations checkpointed (i.e. only some activations are saved), red boxes are activation recomputation, blue boxes are backpropagation, and white boxes are a foward pass with all activations saved.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

Activation recompute challenges

Selective Activation Recompute with Megatron-LM

Selective recomputation:

- Saves the activations that take less space and are expensive to recompute
- Recompute activations that take a lot of space but are relatively cheap to recompute.

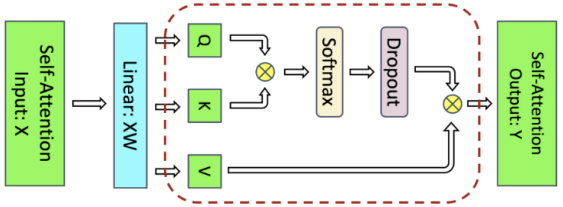


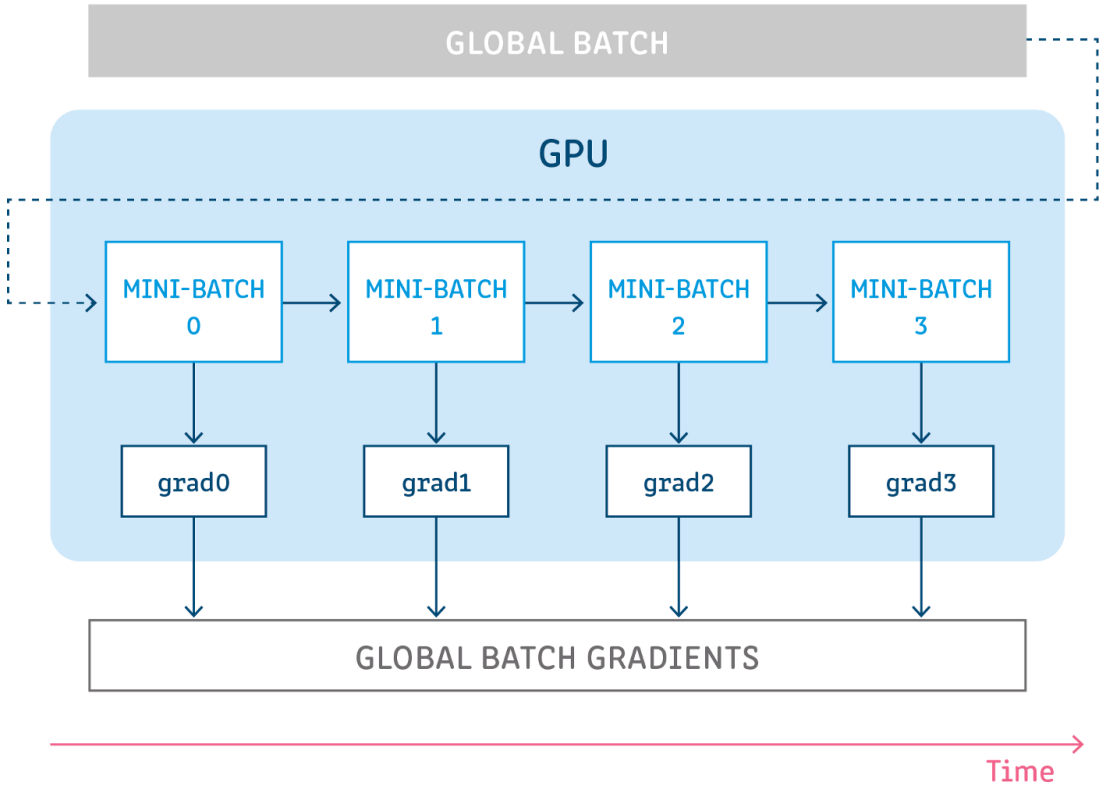
Figure 3: Self-attention block. The red dashed line shows the regions to which selective activation recomputation is applied (see Section 5 for more details on selective activation recomputation).

Figure 1: Parameters, optimizer state, and activations memory. The dashed red line represents the memory capacity of an NVIDIA A100 GPU. Present work reduces the activation memory required to fit the model. Details of the model configurations are provided in Table 3.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

Gradient accumulation

- Gradient accumulation is a mechanism to split the batch of samples — used for training a neural network — into several mini-batches of samples that will be run sequentially.



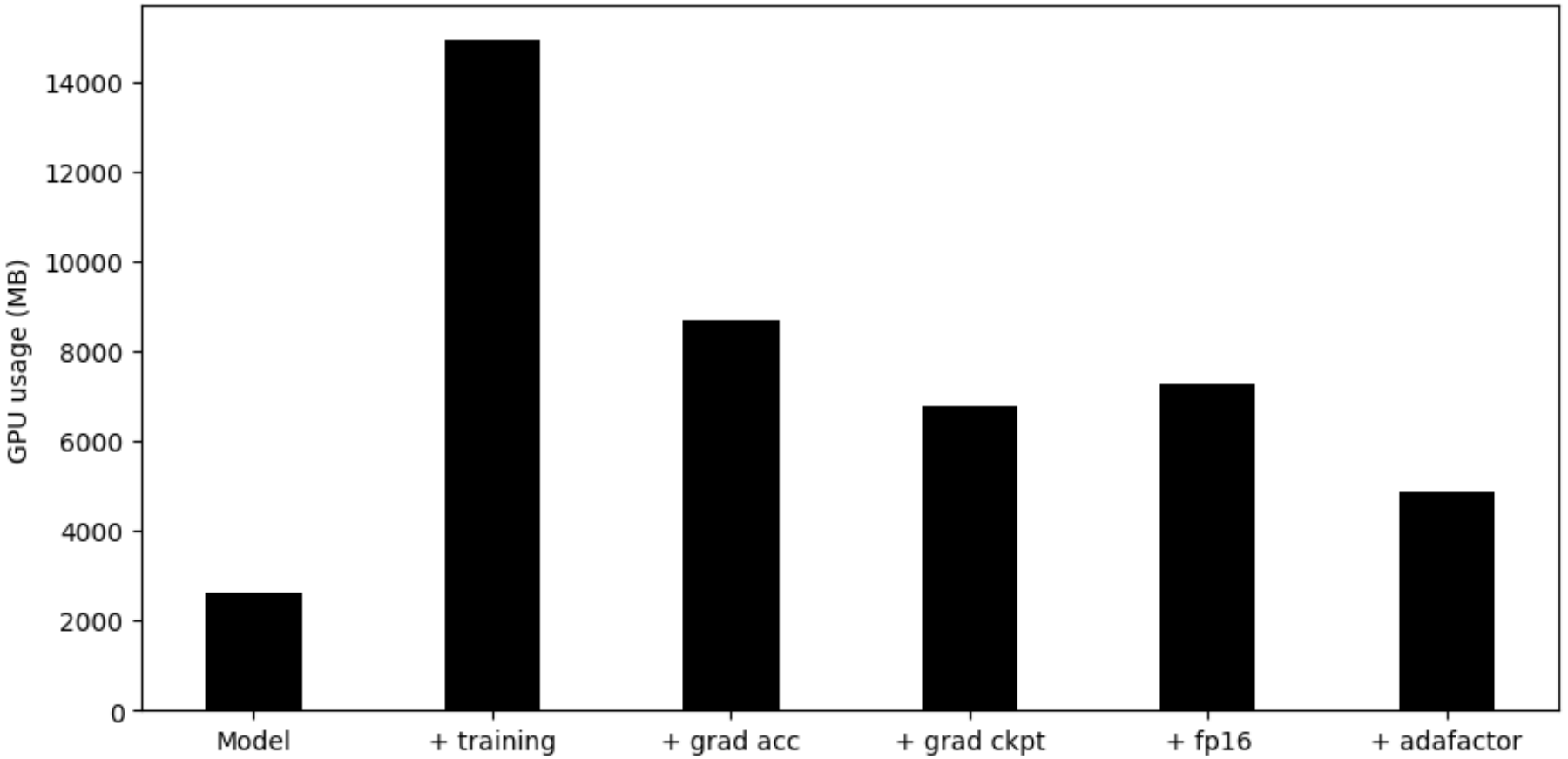
https://towardsdatascience.com/what-is-gradient-accumulation-in-deep-learning-ec034122cfa

Gradient accumulation

```
optimizer = ...


for epoch in range(...):
   for i, sample in enumerate(dataloader):
      inputs, labels = sample
      optimizer.zero_grad()
      # Forward Pass
      outputs = model(inputs)
      # Compute Loss and Perform Back-
      propagation
      loss = loss_fn(outputs, labels)
      loss.backward()
      # Update Optimizer
      optimizer.step()
```

```
optimizer = ...
ACCU_STEPS = ...
for epoch in range(...):
   for i, sample in enumerate(dataloader):
      inputs, labels = sample
      optimizer.zero_grad()
      # Forward Pass
      outputs = model(inputs)
      # Compute Loss and Perform Back-
      propagation
      loss = loss_fn(outputs, labels)
      # Normalize the loss
      loss = loss / ACCU_STEPS
      loss.backward()

      if ((idx + 1) % ACCU_STEPS == 0) or (idx
      + 1 == len(dataloader)):
         # Update Optimizer
         optimizer.step()
```

GPU Memory usage



OPTIMIZE THE OPTIMIZER

8-bit Adam

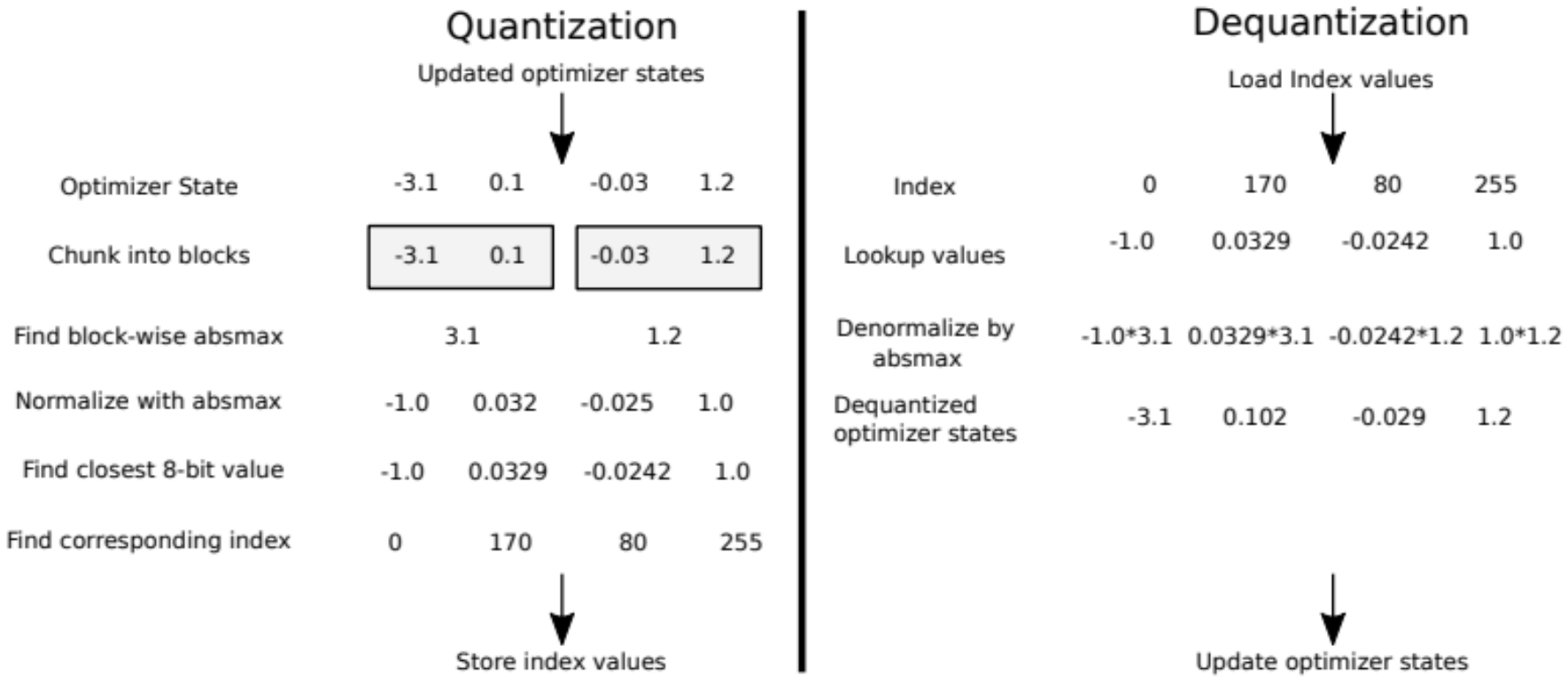
https://github.com/TimDettmers/bitsandbytes

- Do stable optimization with 8 bits using a quantization trick.
  - dynamic quantization, a form of non-linear optimization that is precise for both large and small magnitude values
  - a stable embedding layer to reduce gradient variance

```
import bitsandbytes as bnb

# adam = torch.optim.Adam(model.parameters(),
lr=0.001, betas=(0.9, 0.995)) # comment out old
optimizer

adam = bnb.optim.Adam8bit(model.parameters(),
lr=0.001, betas=(0.9, 0.995)) # add bnb optimizer
adam = bnb.optim.Adam(model.parameters(), lr=0.001,
betas=(0.9, 0.995), optim_bits=8) # equivalent


torch.nn.Embedding(...) ->
bnb.nn.StableEmbedding(...) # recommended for NLP
models
```

Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L., “8-bit Optimizers via Block-wise Quantization”, 2021. doi:10.48550/arXiv.2110.02861.

Schematic of 8-bit optimizers



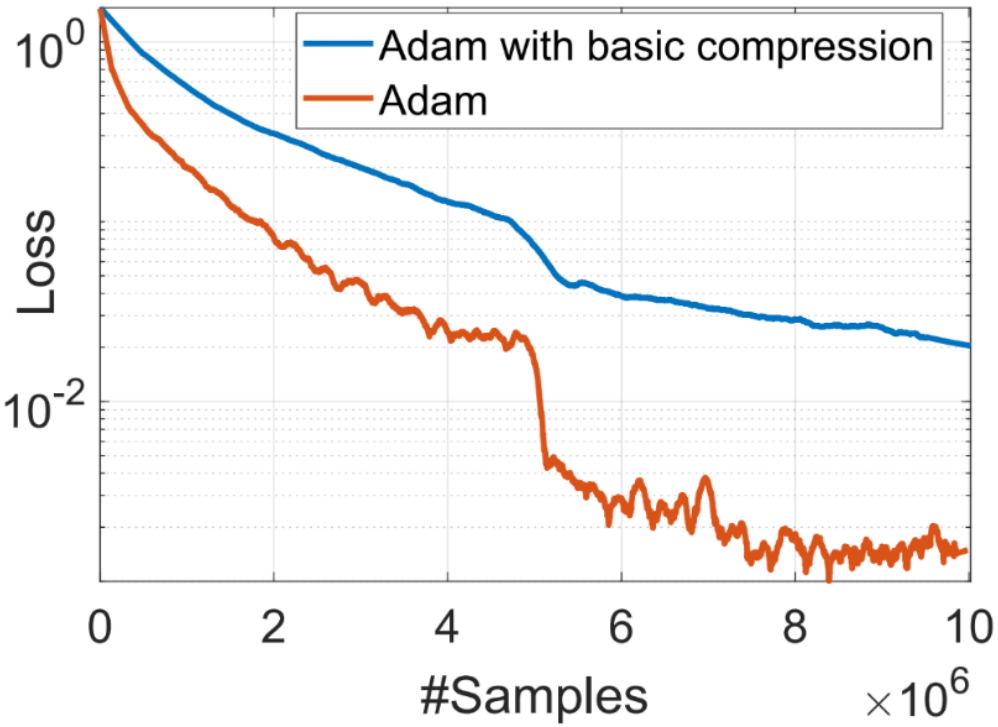
Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L., “8-bit Optimizers via Block-wise Quantization”, 2021. doi:10.48550/arXiv.2110.02861.

Error-compensation to Adam

Basic approach

- The idea of error compensation can be summarized as:
  - 1) doing compression,
  - 2) memorizing the compression error, and then
  - 3) adding the compression error back in during the next iteration.

- This strategy has been proven to work for optimization algorithms that are linearly dependent on the gradient, such as SGD and Momentum SGD.

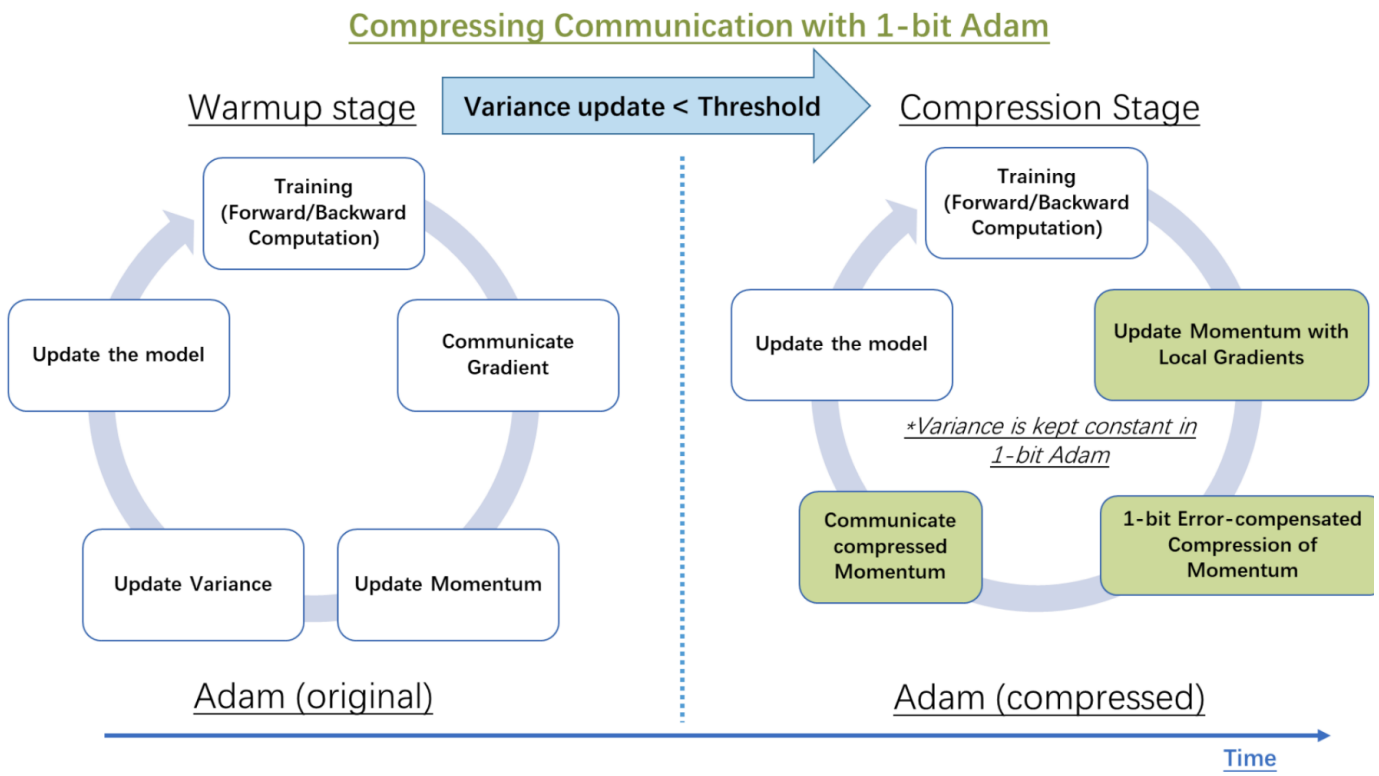


1-bit Adam

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t)^2$$

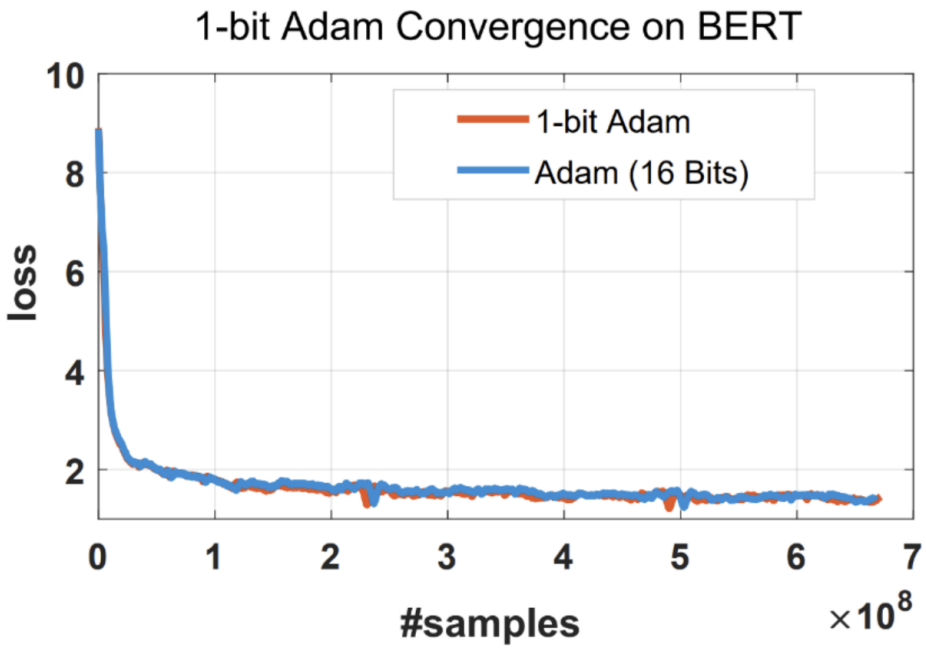
$$x_{t+1} = x_t - \gamma \frac{m_{t+1}}{\sqrt{v_{t+1}} + \eta}$$



1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, Yuxiong He

1-bit Adam

Convergence and Throughput



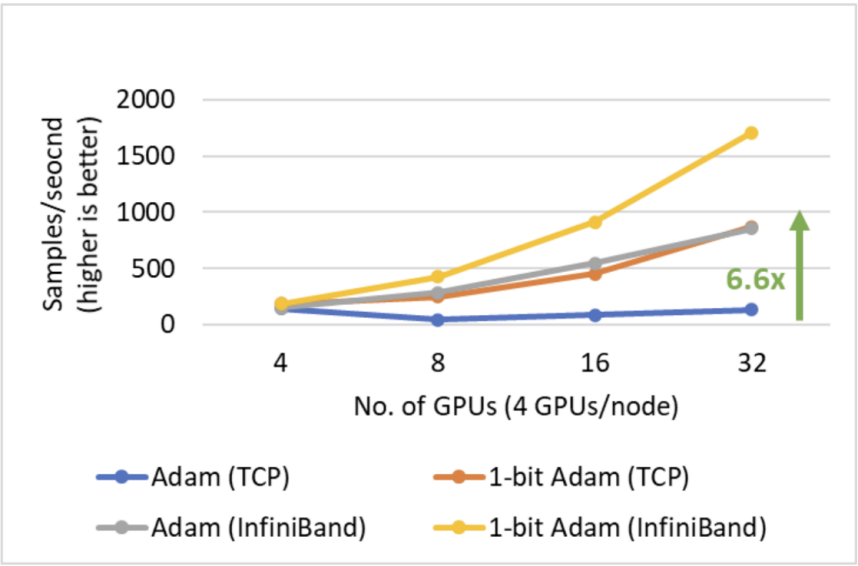


Figure 3: Scalability of 1-bit Adam for BERT-Large Pretraining on V100 GPUs with batch size of 16/GPU.

1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, Yuxiong He

MULTI GPU

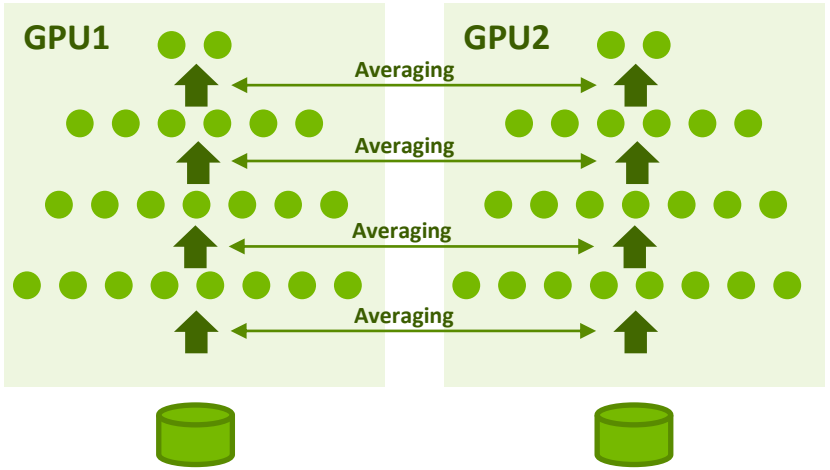
Objective

- Fit very large models into limited hardware
  - e.g. t5-11b is 45GB in just model params
- Significantly speed up training
  - finish training that would take a year in hours

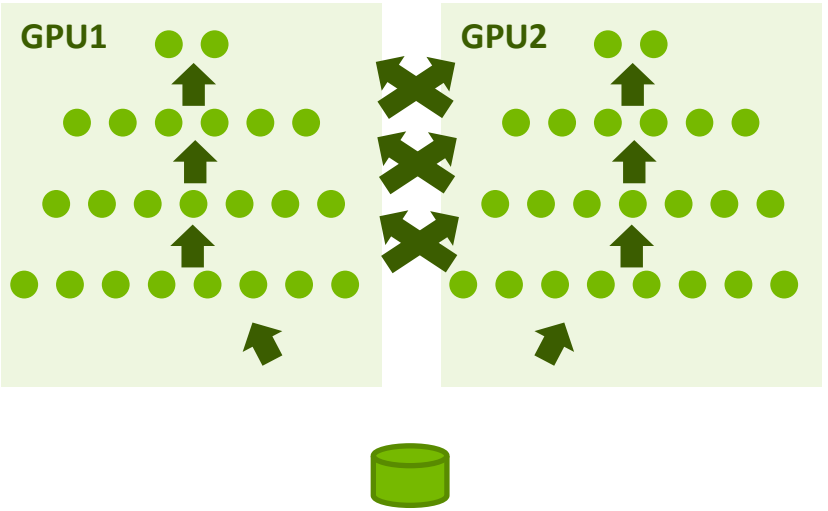
Data vs Model Parallelism

Comparison

- **Data Parallelism**
  - Allows to speed up training
  - All workers train on different data
  - All workers have the same copy of the model
  - Neural network gradients (weight changes) are exchanged

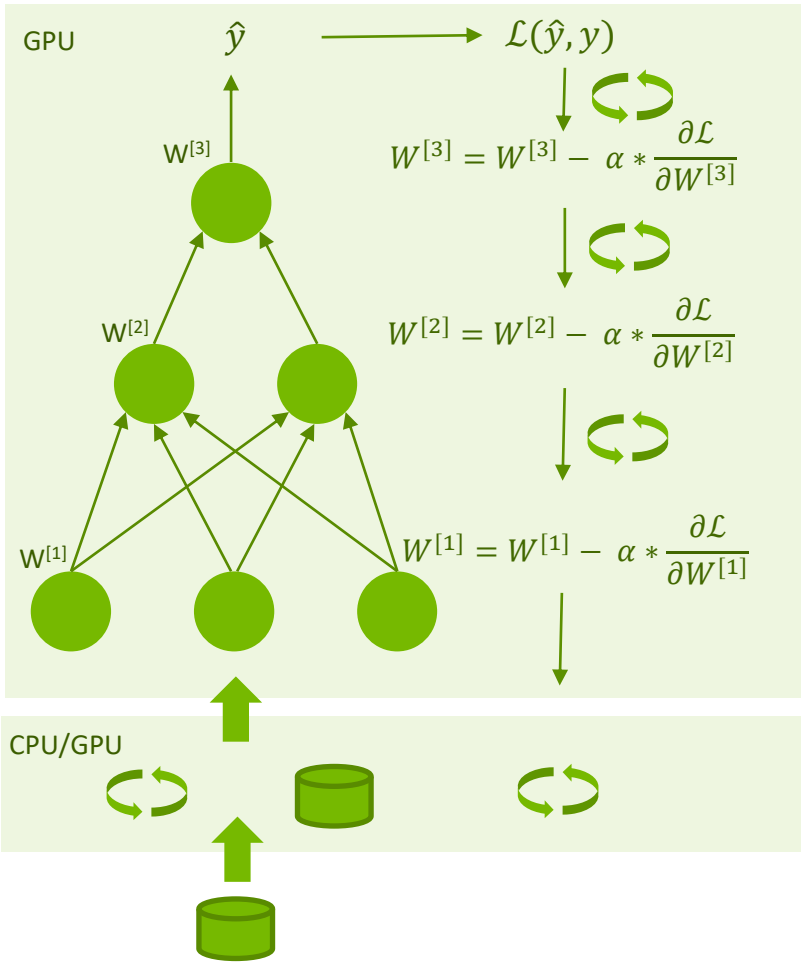


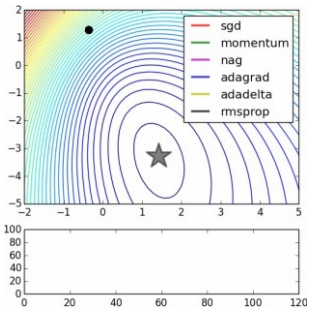
- **Model Parallelism**
  - Allows for a bigger model
  - All workers train on the same data
  - Parts of the model are distributed across GPUs
  - Neural network activations are exchanged



TRAINING A NEURAL NETWORK

Single GPU

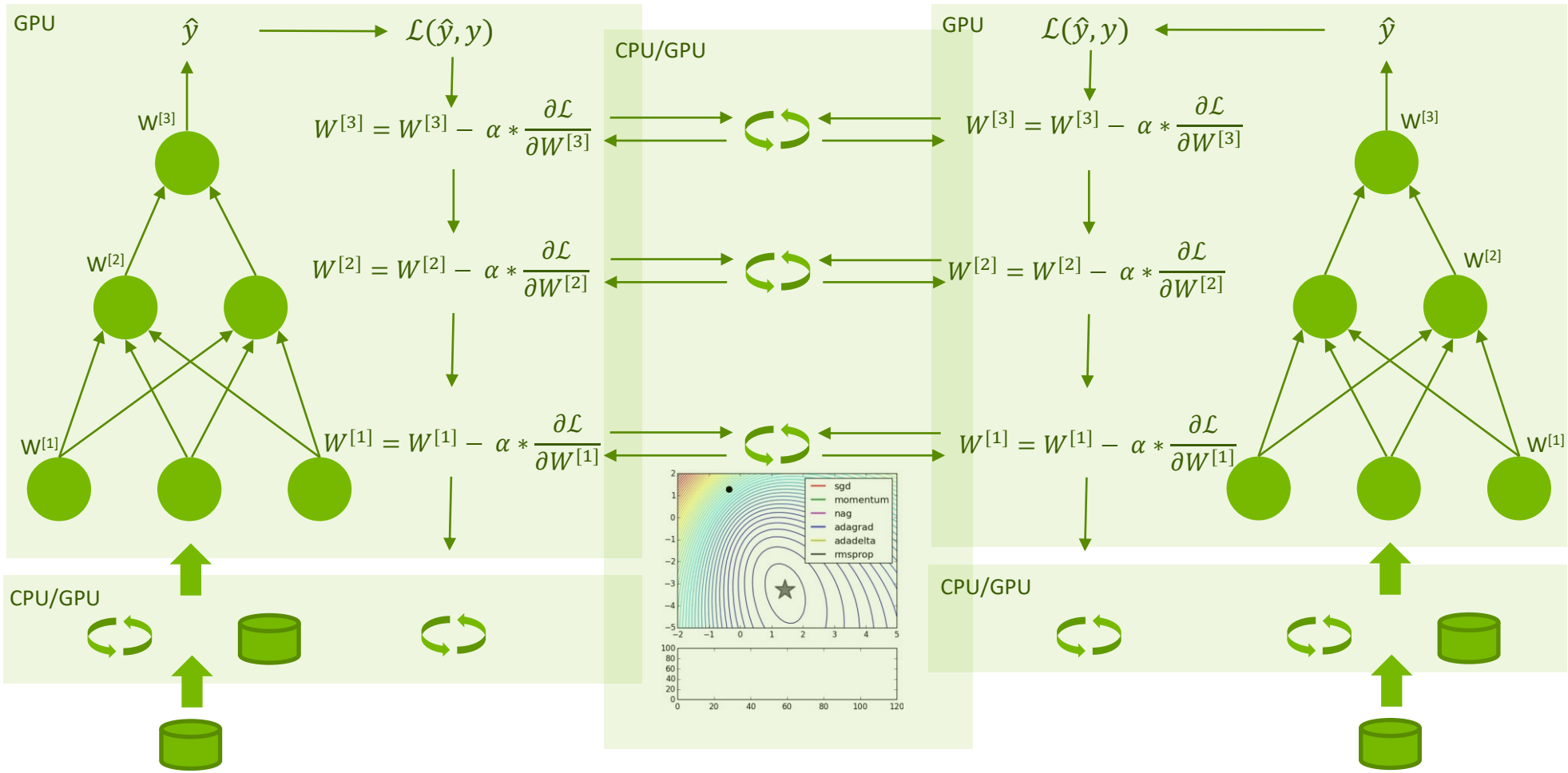




1. Read the data
2. Transport the data
3. Pre-process the data
4. Queue the data
5. Transport the data
6. Calculate activations for layer one*
7. Calculate activations for layer two
8. Calculate the output
9. Calculate the loss
10. Backpropagate through layer three
11. Backpropagate through layer two
12. Backpropagate through layer one
13. Execute optimisation step
14. Update the weights
15. Return control

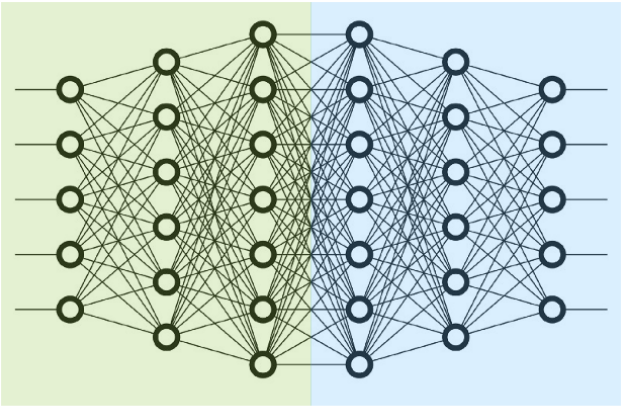
TRAINING A NEURAL NETWORK

Multiple GPUs

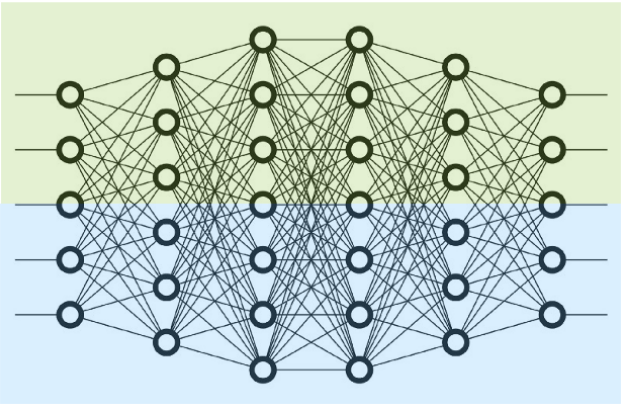


Model Parallelism

- **Pipeline (Inter-Layer) Parallelism**
  - Split sets of layers across multiple devices
  - Layer 0,1,2 and layer 3,4,5 are on difference devices



- **Tensor (Intra-Layer) Parallelism**
  - Split individual layers across multiple devices
  - Both devices compute difference parts of Layer 0,1,2,3,4,5



TENSOR PARALLELISM AND TRANSFORMERS

Gelu | Elu | Relu

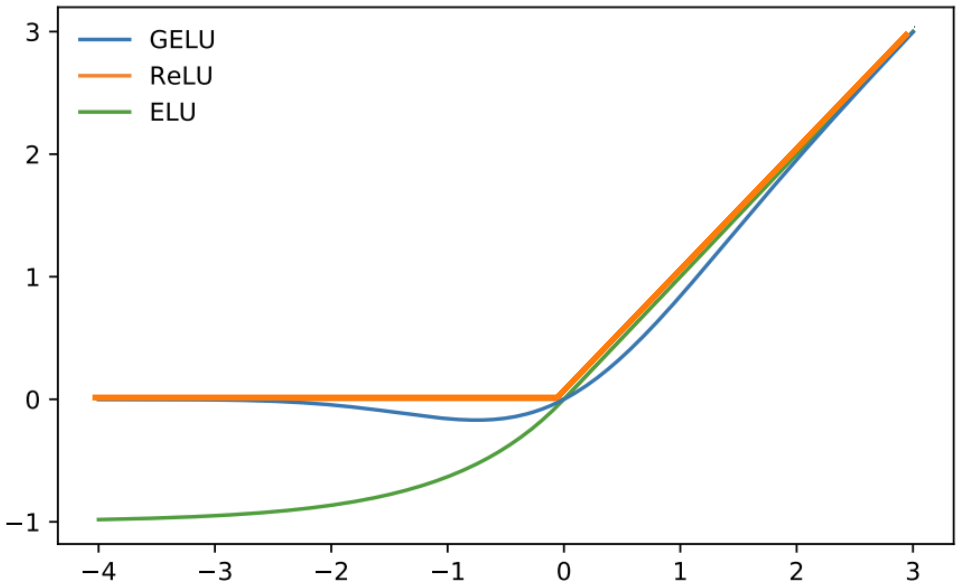
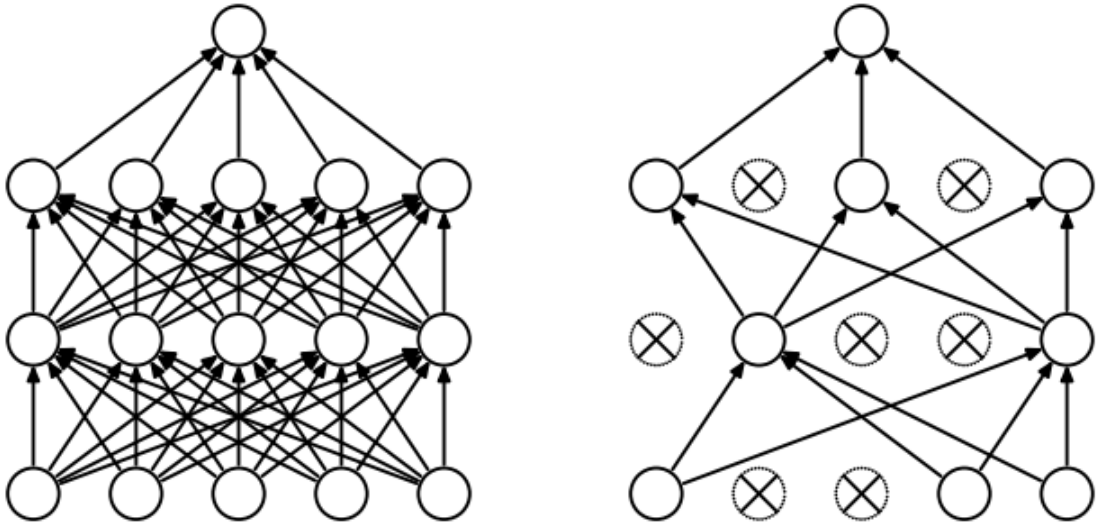


Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

https://paperswithcode.com/method/gelu

Dropout



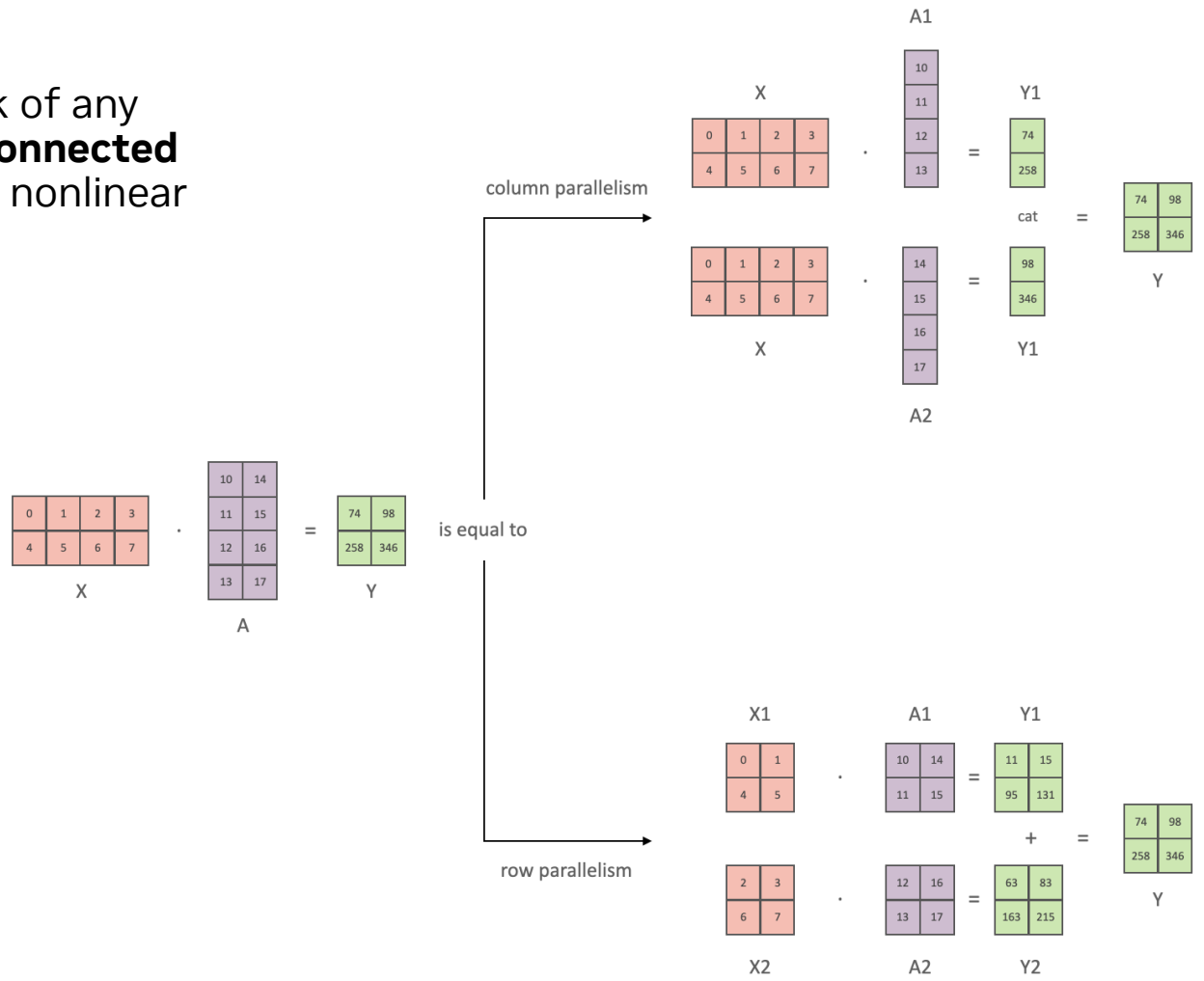
(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

https://proceedings.neurips.cc/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf
https://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf?utm_content=buffer79b43&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer

Tensor Parallelism and Transformers

- The main building block of any transformer is a **fully connected nn.Linear** followed by a nonlinear activation GeLU.



Megatron

https://github.com/NVIDIA/Megatron-LM

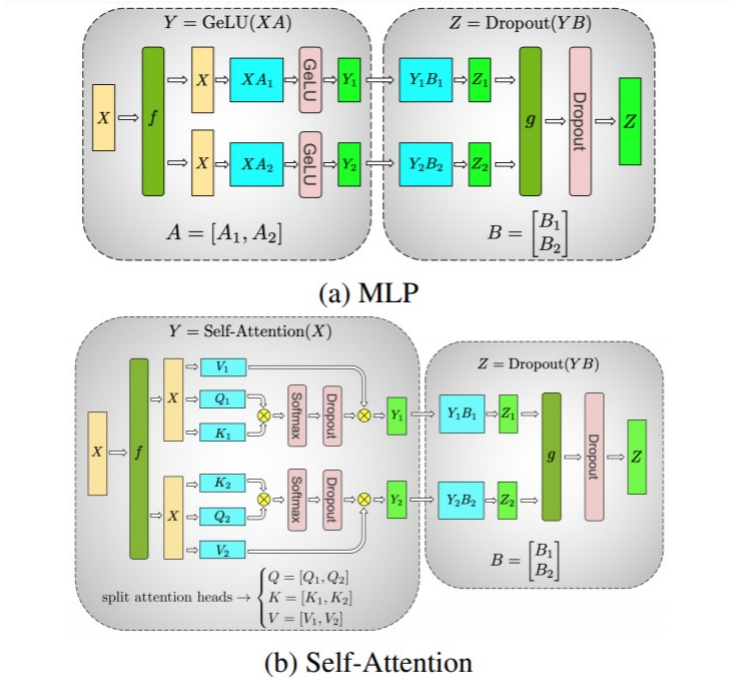


Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

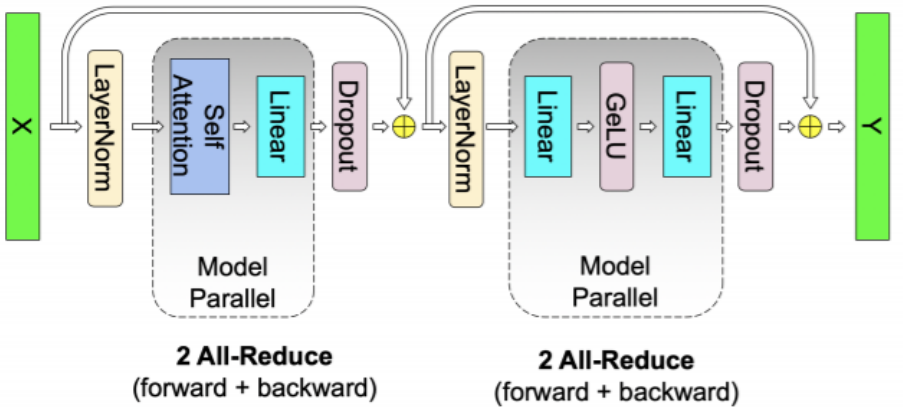
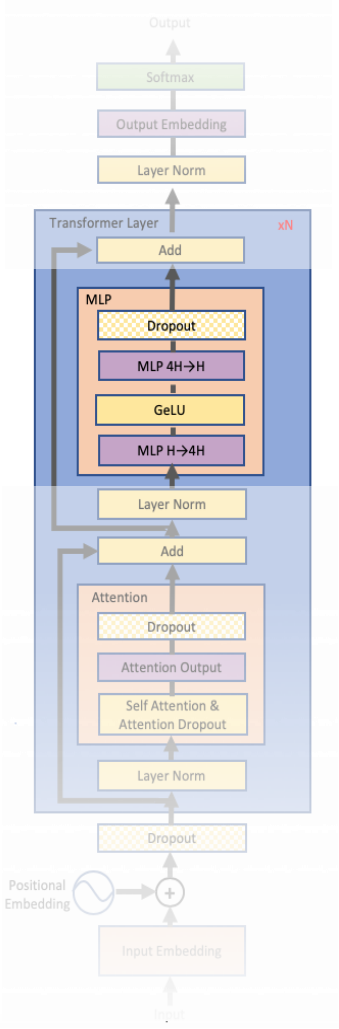


Figure 4. Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., & Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. arXiv preprint arXiv:1909.08053.

Let’s break it down – first up ***MLP***



Partitioning MULTI-LAYER Perceptron (MLP)

MLP:

$$Y = \mathrm{GeLU}(XA)$$
$$Z = \mathrm{Dropout}(YB)$$

Approach 1: split X column-wise and A row-wise:

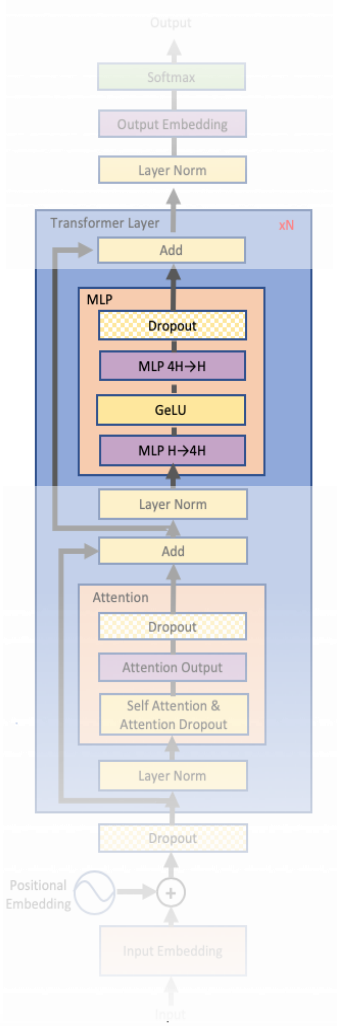
$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \longrightarrow Y = \mathrm{GeLU}(X_1A_1 + X_2A_2)$$

Before GeLU we will need a communication point

Approach 2: split A column-wise:

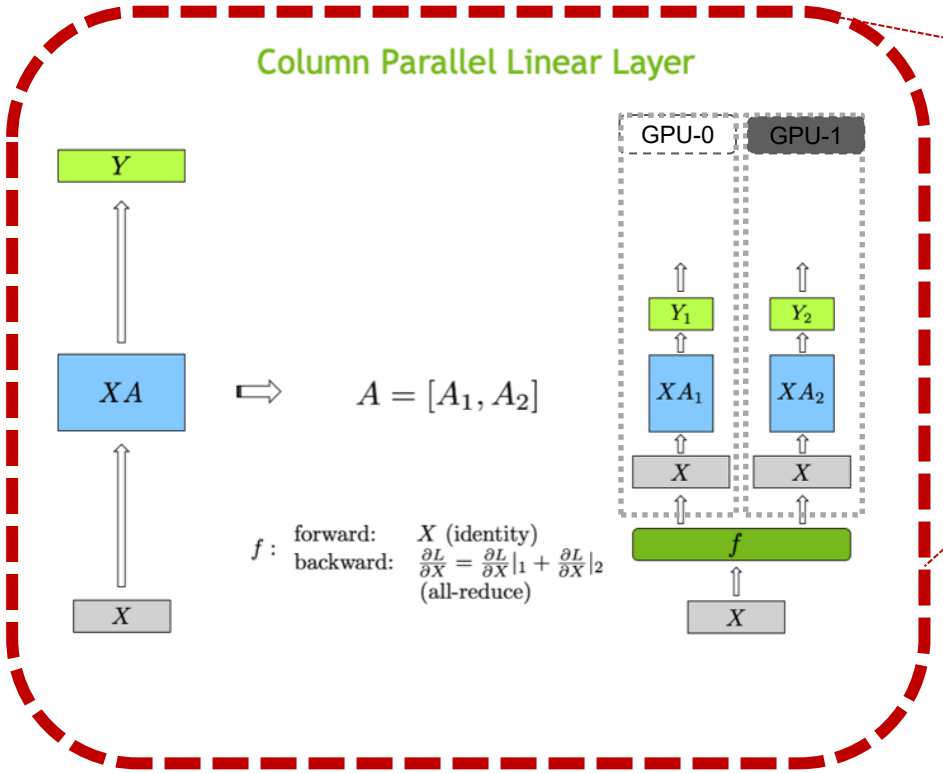
$$A = [A_1, A_2] \longrightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

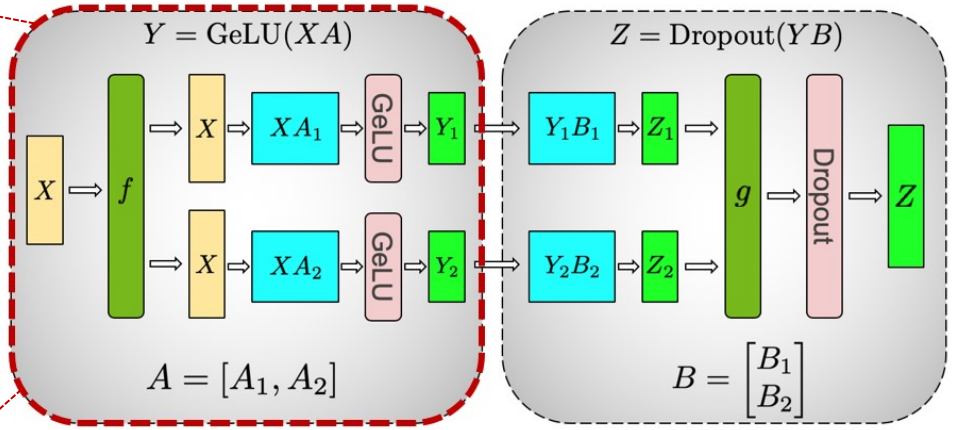
No communication is required



Tensor Parallelism and Transformers

MLP

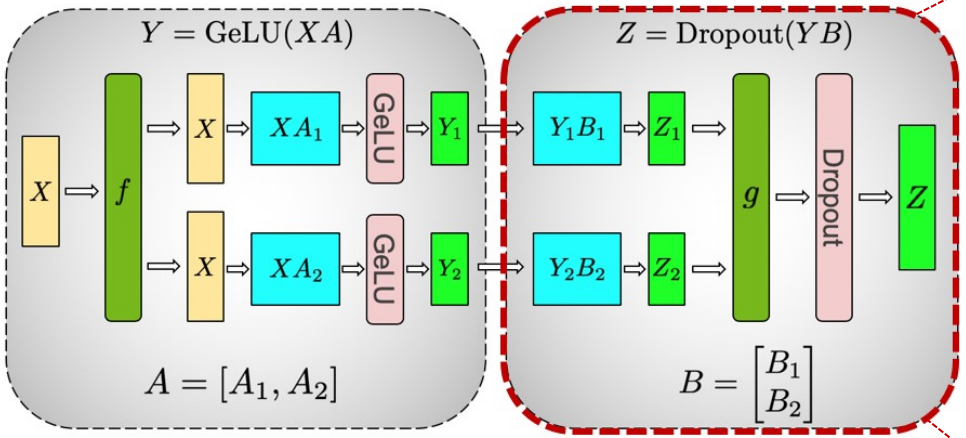




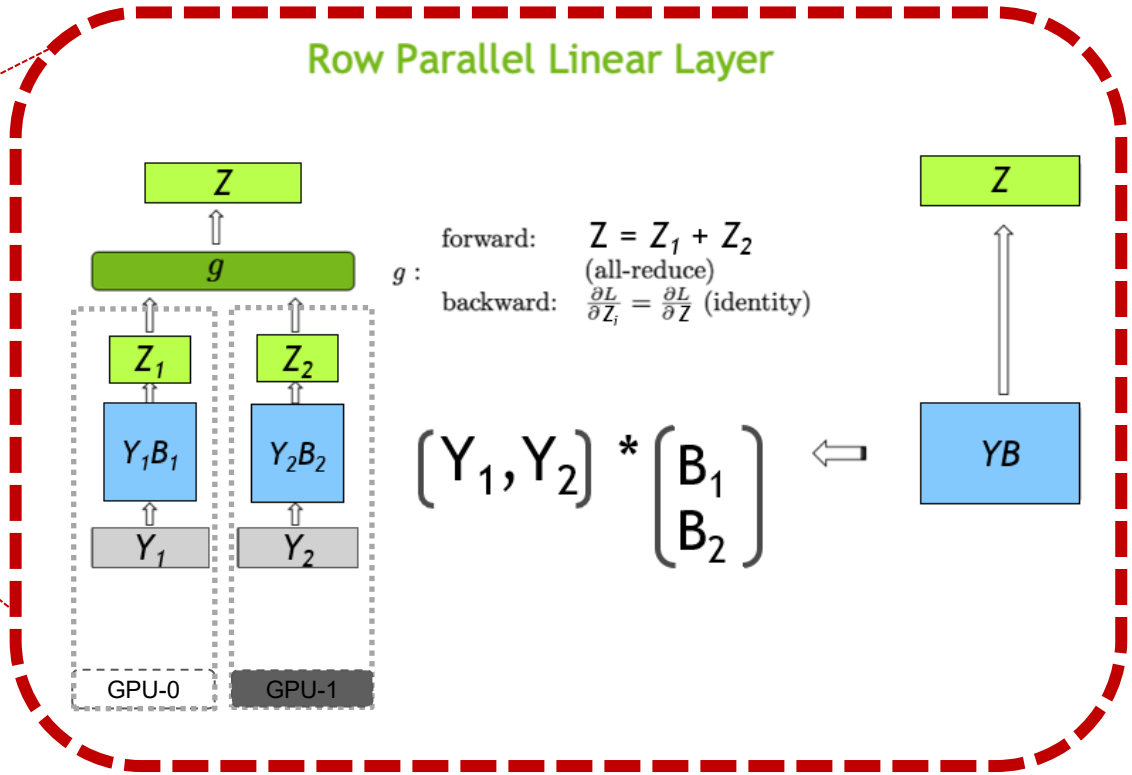
f and *g* are **conjugate**, *f* is **identity** operator in the forward pass and **all-reduce** in the backward pass while *g* is **all-reduce** in forward and **identity** in backward.

$$[Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

Row Parallel – Forward | Backward

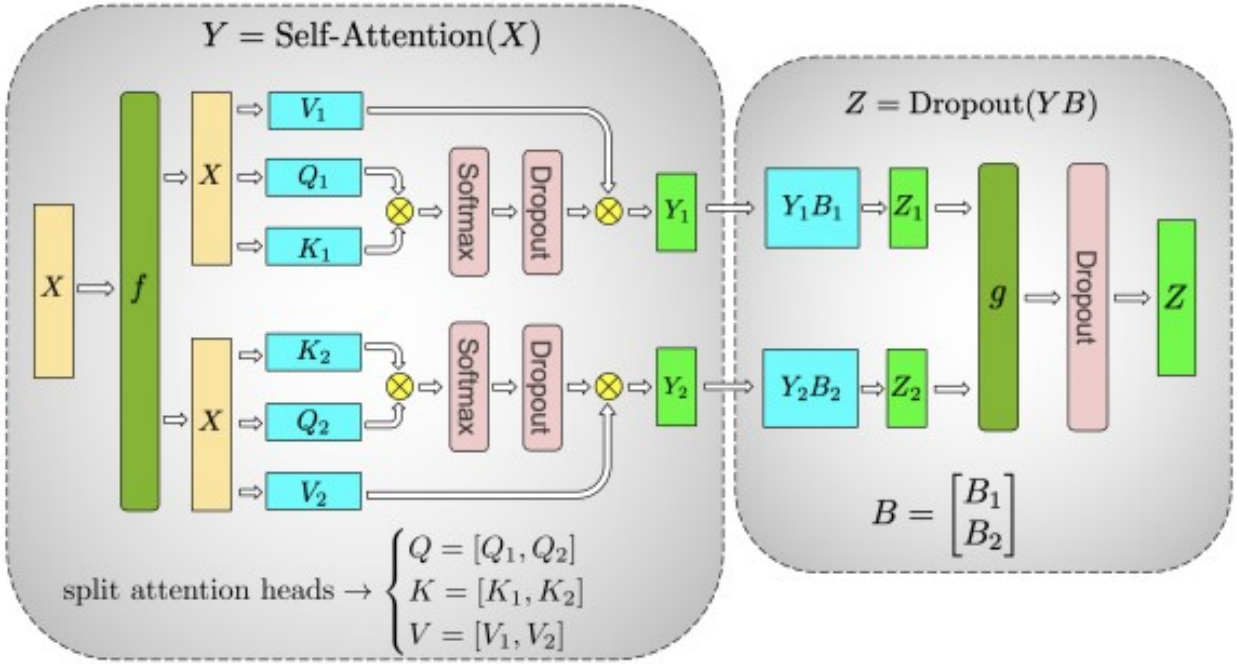


f and *g* are **conjugate**, *f* is **identity** operator in the forward pass and **all-reduce** in the backward pass while *g* is **all-reduce** in forward and **identity** in backward.



Tensor Parallelism and Transformers

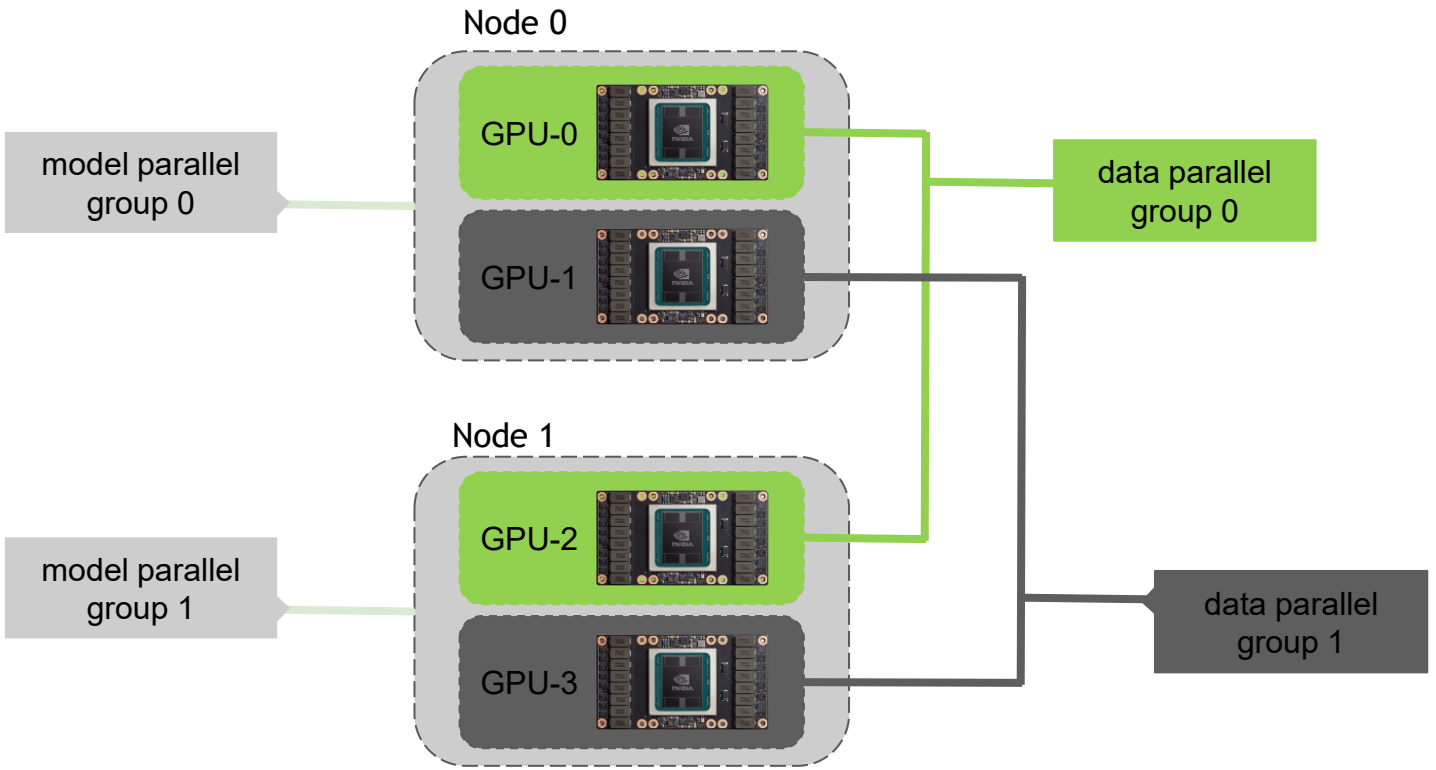
- Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads.



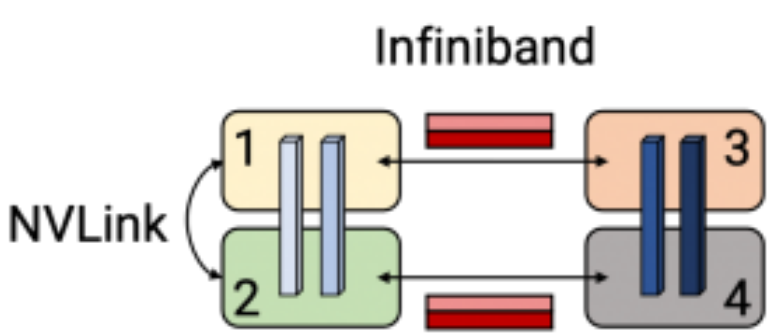
(b) Self-Attention

Hybrid Model

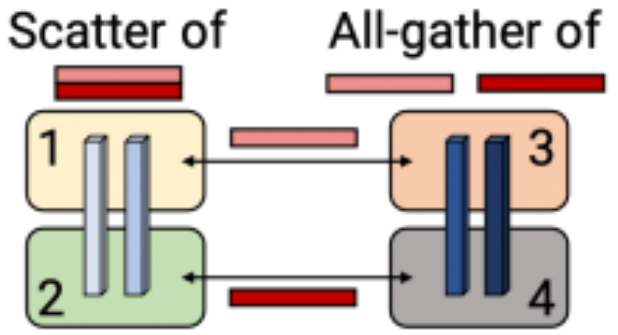
multiple groups of communicators



Scattering/gathering



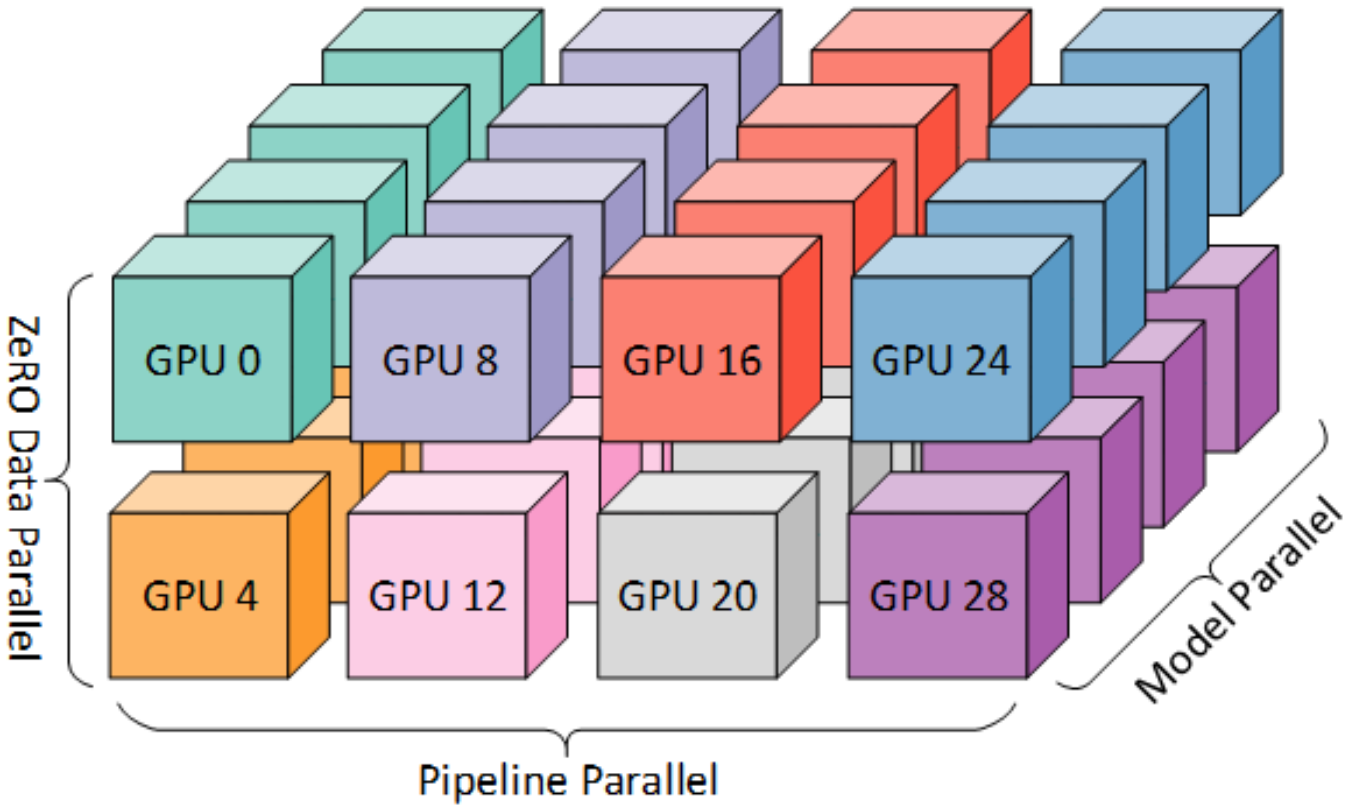
(a) W/o scatter/gather optimization.



(b) W/ scatter/gather optimization.

Scatter/gather communication optimization to reduce the total number of bytes sent between GPUs on different multi-GPU servers.

3D Parallelism



https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/

PYTORCH DISTRIBUTED DATA PARALLEL-

HTTPS://PYTORCH.ORG/TUTORIALS/INTERMEDIATE/DDP_TUTORIAL.HTML

pyTORCH

- **PyTorch** provides several options for distributed training for applications that gradually scale from simple to complex:
  1. Use single-device training if the data and model can fit in one GPU, and training speed is not a concern.
  2. Use single-machine multi-GPU **DataParallel** to make use of multiple GPUs on a single machine to speed up training with minimal code changes.
  3. Use single-machine multi-GPU **DistributedDataParallel**, if you would like to further speed up training and are willing to write a little more code to set it up.
  4. Use multi-machine **DistributedDataParallel** and the launching script, if the application needs to scale across machine boundaries.
  5. Use **torch.distributed.elastic** to launch distributed training if errors (e.g., out-of-memory) are expected or if resources can join and leave dynamically during training.

Data parallel vs distributed data parallel

- **torch.nn.DataParallel**
  - The DataParallel package enables single-machine multi-GPU parallelism with the lowest coding hurdle. It only requires a one-line change to the application code. Although DataParallel is very easy to use, it usually does not offer the best performance because it replicates the model in every forward pass.
- **torch.nn.parallel.DistributedDataParallel**
  - Compared to DataParallel, DistributedDataParallel requires one more step to set up, i.e., calling init_process_group. **DDP uses multi-process parallelism**. Moreover, the model is broadcast at DDP construction time instead of in every forward pass, which also helps to speed up training.
- **torch.distributed.elastic**
  - With the growth of the application complexity and scale, failure recovery becomes a requirement. **torch.distributed.elastic** adds fault tolerance and the ability to make use of a dynamic pool of machines (elasticity).

Data parallelism

- Data Parallelism is when we split the mini-batch of samples into multiple smaller mini-batches and run the computation for each of the smaller mini-batches in parallel.
- Data Parallelism is implemented using ***torch.nn.DataParallel***.
- One can wrap a Module in DataParallel and it will be parallelized over multiple GPUs in the batch dimension.

```
import torch
import torch.nn as nn


class DataParallelModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.block1 = nn.Linear(10, 20)

        # wrap block2 in DataParallel
        self.block2 = nn.Linear(20, 20)
        self.block2 = nn.DataParallel(self.block2)

        self.block3 = nn.Linear(20, 20)

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        return x
```

DISTRIBUTED DATA PARALLEL

- **DistributedDataParallel** (DDP) implements data parallelism at the module level which can run across multiple machines.
- Applications using DDP should spawn multiple processes and create a single DDP instance per process.
- DDP uses collective communications in the ***torch.distributed*** package to synchronize gradients and buffers.

```python
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP


def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()
```

DATA PARALLEL VS DDP

- **DataParallel** is single-process, multi-thread, and only works on a single machine, while DistributedDataParallel is multi-process and works for both single- and multi- machine training.

- **DataParallel** is usually slower than **DistributedDataParallel** even on a single machine due to contention across threads, per-iteration replicated model, and additional overhead introduced by scattering inputs and gathering outputs.

- **DistributedDataParallel** works with model parallel; DataParallel does not at this time. When DDP is combined with model parallel, each DDP process would use model parallel, and all processes collectively would use data parallel.

https://pytorch.org/tutorials/beginner/former_torchies/parallelism_tutorial.html

TORCHRUN

- Single-node multi-worker

```
>>> torchrun
    --standalone
    --nnodes=1
    --nproc_per_node=$NUM_TRAINERS
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

- Fault tolerant (fixed sized number of workers, no elasticity)

```
>>> torchrun
    --nnodes=$NUM_NODES
    --nproc_per_node=$NUM_TRAINERS
    --rdzv_id=$JOB_ID
    --rdzv_backend=c10d
    --rdzv_endpoint=$HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

TORCHRUN

- Elastic (min=1, max=4)

```
>>> torchrun
    --nnodes=1:4
    --nproc_per_node=$NUM_TRAINERS
    --rdzv_id=$JOB_ID
    --rdzv_backend=c10d
    --rdzv_endpoint=$HOST_NODE_ADDR
    YOUR_TRAINING_SCRIPT.py (--arg1 ... train script args...)
```

ZERO (ZERO REDUNDANCY OPTIMIZER)

DeepSpeed

https://github.com/microsoft/DeepSpeed

- DeepSpeed is deep learning optimization software suite that enables scale and speed for Deep Learning Training and Inference.
  - Train/Inference dense or sparse models with billions or trillions of parameters
  - Achieve excellent system throughput and efficiently scale to thousands of GPUs
  - Train/Inference on resource constrained GPU systems
  - Achieve low latency and high throughput for inference
  - Achieve extreme compression for an unparalleled inference latency and model size reduction

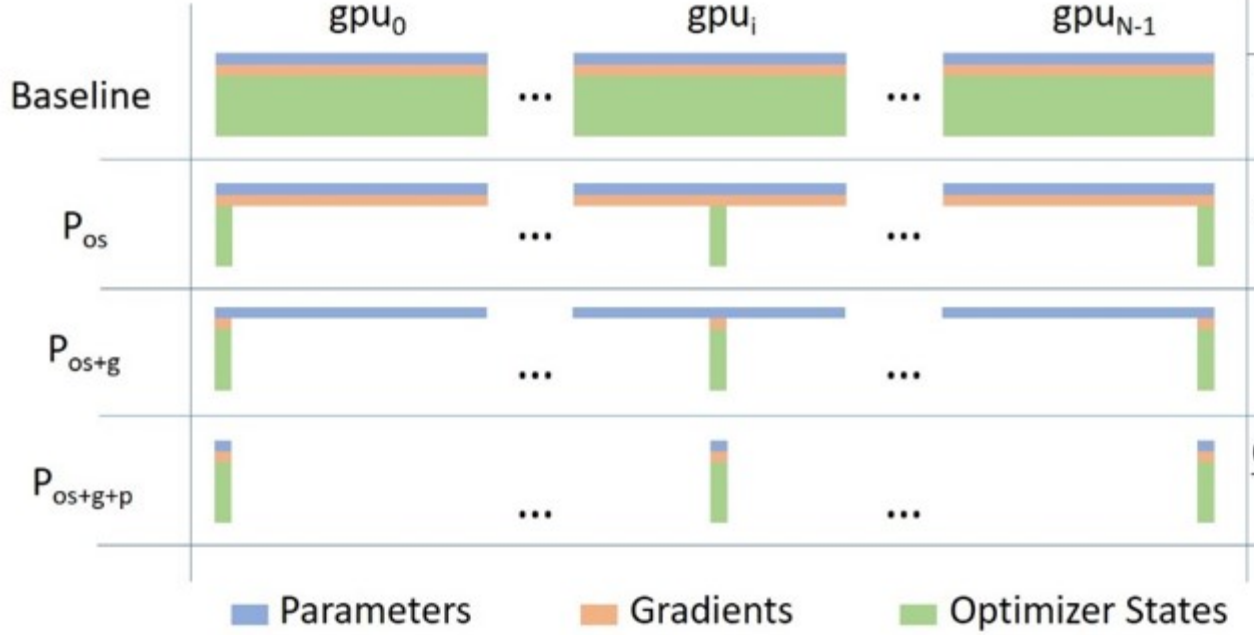
Memory Efficient Optimizer

Zero Redundancy Optimizer

- ***ZeRO* (*Zero Redundancy Optimizer*)** optimizes the memory used for training large models based on the observation about two major memory consumption of large model training:
  - The majority is occupied by *model states*, including optimizer states (e.g. Adam momentums and variances), gradients and parameters.
  - The remaining is consumed by activations, temporary buffers and unusable fragmented memory.

Data Parallelism

ZeRO-powered data parallelism (ZeRO-DP)



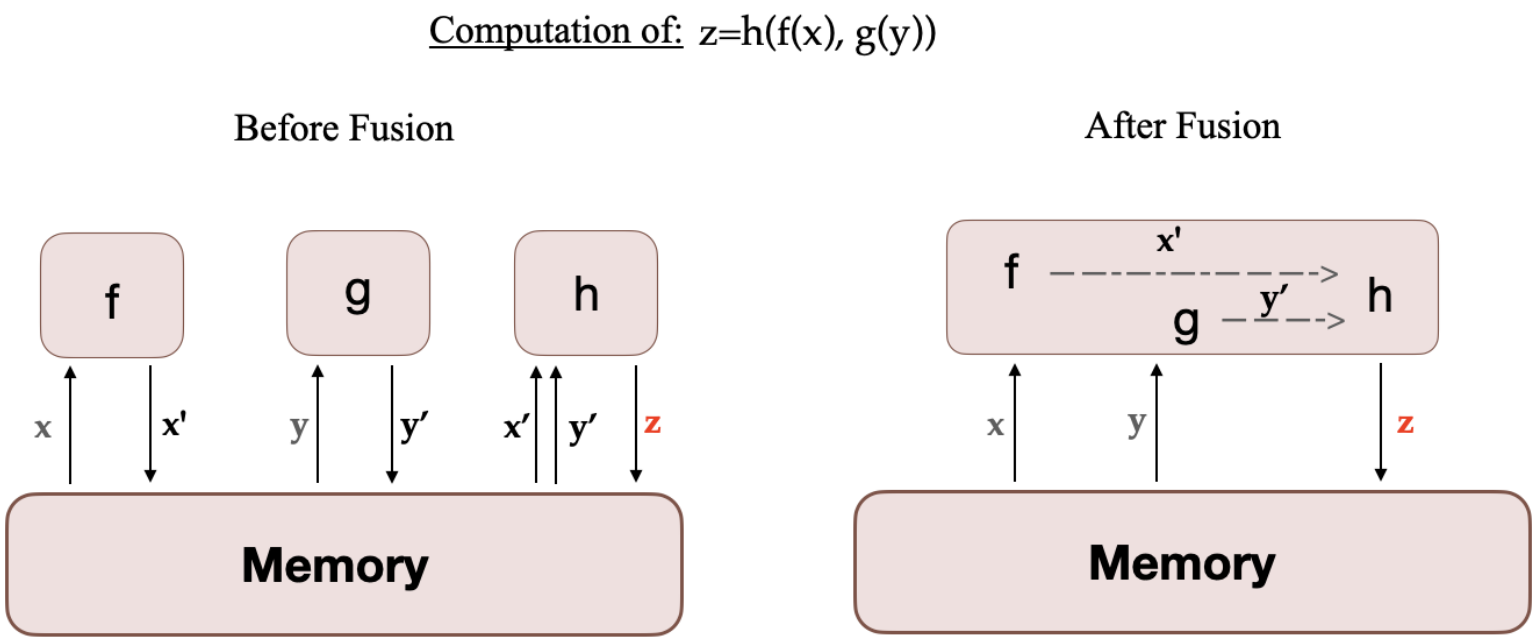
	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

KERNEL FUSION

Fused CUDA Kernels

https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/



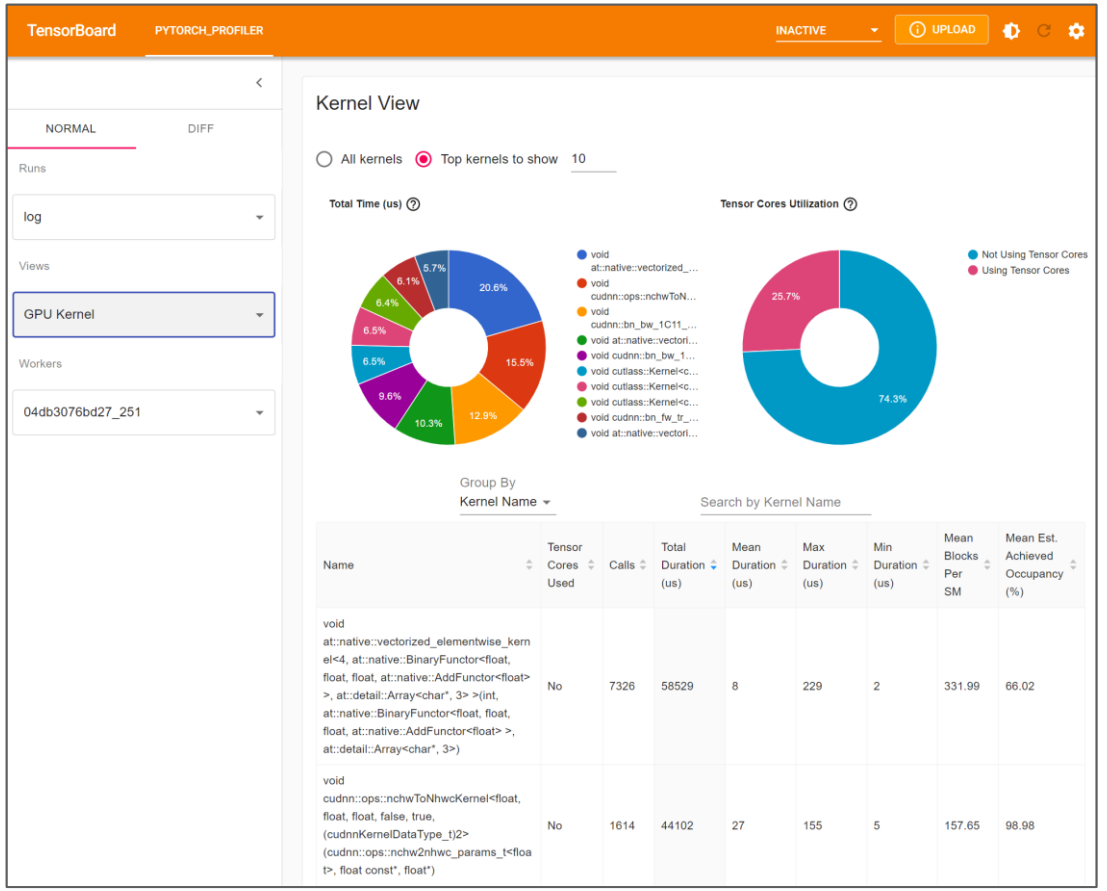
https://huggingface.co/blog/megatron-training

PROFILING YOUR CODE

Am I USING Tensor CORES?

https://pytorch.org/docs/stable/profiler.html

```
from torch import profiler

prof_schedule = profiler.schedule(wait=2,
                                  warmup=2,
                                  active=5,
                                  repeat=0)

callback =
profiler.tensorboard_trace_handler('./log')

prof = profiler.profile(schedule=prof_schedule,
on_trace_ready=callback,
record_shapes=False,
with_stack=False)

prof.start()

for it in range(num_iterations):
    # code to be profiled
    ...
    prof.step()

prof.stop()
```



PYTORCH PROFILER

https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html

- Build-in in PyTorch
- Allows to improve performance of your models visually
- Features
  - Tensor Core Usage and Eligibility Detection
  - Kernel view
  - Stack view
  - Performance recommendations
  - …

```
# ssh into your machine
ssh -L 9999:localhost:9999 user@<server_ip>

# run docker container
$ docker run --gpus all -d -p 9999:9999 -v
/path/to/my/project/:/workspace
nvcr.io/nvidia/pytorch:21.09-py3

# install tensor board plugin
$ pip install torch_tb_profiler

# run application (example from repo)
$ python tiling_and_tensor_cores_pytorch.py

# start tensorboard
$ tensorboard --logdir=./log --port=9999
```

WHAT’S ABOUT THE BATCH SIZE?

Why Batch size matters

- The most efficient performance when batch sizes and input/output neuron counts are divisible by a certain number, which typically starts at **8**, but can be much higher as well.
- That number varies a lot depending on the specific hardware being used and the dtype of the model:
  - *Tensor Core Requirements define the multiplier based on the dtype and the hardware. For example, for fp16 a multiple of **8** is recommended, but on A100 it's **64**!*
- When parameters are too small, there is also Dimension Quantization Effects to consider, this is where tiling happens, and the right multiplier can have a significant speedup.
- Furthermore, the bigger the batch size the less often the optimizer is run, the faster the training is (considering the same dataset length).

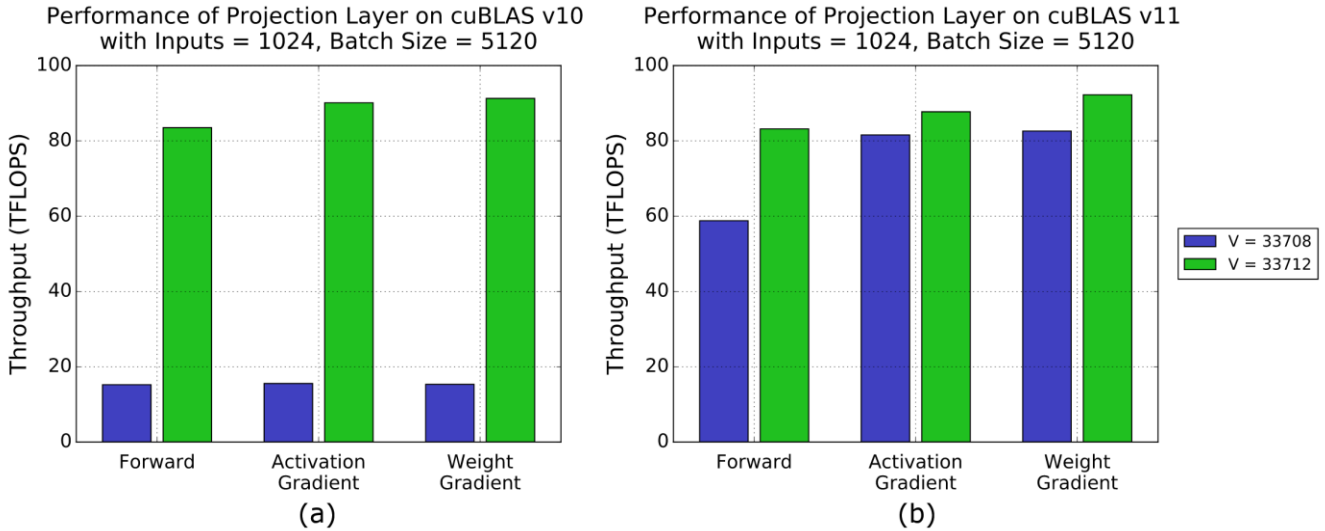
Batch size checklist

fully-connected layers

1. Choose the batch size and the number of inputs and outputs to be divisible by 4 (TF32) / 8 (FP16) / 16 (INT8) to run efficiently on Tensor Cores.
2. For best efficiency on A100, choose these parameters to be divisible by 32 (TF32) / 64 (FP16) / 128 (INT8).
3. Especially when one or more parameters are small, choosing the batch size and the number of inputs and outputs to be divisible by at least 64 and ideally 256 can streamline tiling and reduce overhead.
4. Larger values for batch size and the number of inputs and outputs improve parallelization and efficiency.
5. **As a rough guideline, choose batch sizes and neuron counts greater than 128 to avoid being limited by memory bandwidth.**

The case of transformer

- Transformers are a popular neural network architecture used for sequence-to-sequence mapping tasks, for example for natural language translation. They use an encoder-decoder architecture making heavy use of attention, both to “self-attend” over input sequences, as well as to give the decoder access to the encoder’s context
- From a performance standpoint, Transformers fundamentally process all the tokens in an input sequence in parallel. That makes Transformers very amenable to highly parallel architectures such as GPUs, and leads to large GEMMs that, with a few simple guidelines, can take great advantage of Tensor Core acceleration.



See what happens when the vocabulary size is chosen without regard to alignment. FP16 data is used, so dimensions must be multiples of 8 for best alignment.

Batch size vs learning rate

Theory suggests that when multiplying the batch size by k, one should multiply the learning rate ϵ by $\sqrt{k}$ to keep the variance in the gradient expectation constant.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

Warmup strategy

- A lot of networks will diverge especially at early learning phase.
- Warmup strategies address this challenge.

Gradual warmup. We present an alternative warmup that *gradually* ramps up the learning rate from a small to a large value. This ramp avoids a sudden increase from a small learning rate to a large one, allowing healthy convergence at the start of training. In practice, with a large minibatch of size kn, we start from a learning rate of η and increment it by a constant amount at each iteration such that it reaches $\hat{\eta} = k\eta$ after 5 epochs. After the warmup phase, we go back to the original learning rate schedule.

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., ... & He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv preprint arXiv:1706.02677.

JIT, TORCHSCRIPT, XLA

JIT and TorchScript

- **JIT** is a dynamic tracing compiler that generates optimized code on the fly during runtime.
- It works by tracing the execution of a PyTorch model during training and generates optimized code that can be reused during inference.
- The advantage of JIT is that it can generate optimized code for a specific input size, making it very efficient for that input size.
- JIT does not provide any static guarantees about the correctness of the generated code.

- **TorchScript** is a static graph compiler that generates optimized code ahead of time.
- It works by converting a PyTorch model into a graph representation that can be optimized and compiled for execution on different devices.
- The advantage of TorchScript is that it provides static guarantees about the correctness of the generated code, making it more reliable for production use.
- However, TorchScript requires a bit more effort upfront to convert the PyTorch model into a graph representation.

JIT and TorchScript

```
import torch
import torch.nn as nn

# Define a simple PyTorch model
class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        return self.linear(x)

# Create an instance of the model
model = SimpleModel()
```

```
# Define some example input
input = torch.randn(1, 2)

# Use JIT to generate optimized code on the
# fly
jit_model = torch.jit.trace(model, input)

# Use TorchScript to generate optimized
# code ahead of time
script_model = torch.jit.script(model)

# Evaluate the models on the input
jit_output = jit_model(input)
script_output = script_model(input)

# Compare the outputs
print(jit_output)
print(script_output)
```

XLA

Accelerate Linear Algebra

- XLA stands for Accelerated Linear Algebra and it is a domain-specific compiler for linear algebra operations developed by Google.
- It is designed to optimize the performance of machine learning models by compiling and executing them on a variety of devices, including CPUs, GPUs, and TPUs (Tensor Processing Units).
- The main idea behind XLA is to generate highly optimized device-specific code for linear algebra operations. To achieve this, XLA performs a number of optimizations such as loop unrolling, kernel fusion, and memory layout transformations. XLA also supports automatic differentiation, which is an essential feature for training deep learning models.
- XLA can be used with PyTorch through the PyTorch/XLA package, which provides a PyTorch interface to XLA.

XLA

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as
transforms

# Set up the device to use XLA
device = xm.xla_device()

# Define the CNN architecture
class Net(nn.Module):
[...]

# Create an instance of the model and
# move it to the XLA device
net = Net()
net.to(device)
```

```
# Create an instance of the optimizer and
# move it to the XLA device
optimizer = optim.SGD(net.parameters(),
lr=lr)
optimizer = xla.optimizer(optimizer,
device=device)

# Train the model
for epoch in range(num_epochs):
[...]
        xm.optimizer_step(optimizer)


print('Finished training')
```

JIT, TORCHSCRIPT, XLA

- Performance depends on the specific use case and the hardware being used.
- In general
  - **JIT** provides the most flexibility and can provide significant speedups for inference on a CPU or GPU.
  - **TorchScript** is optimized for deployment and can provide significant speedups and size reductions for models running on a variety of devices.
  - **XLA** is optimized for linear algebra operations and can provide significant speedups for training and inference on a variety of devices, particularly TPUs.

OPEN SOURCE LLM

Google "We Have No Moat, And Neither Does OpenAI"

Leaked Internal Google Document Claims Open Source AI Will Outcompete Google and OpenAI

DYLAN PATEL AND AFZAL AHMAD

MAY 4, 2023 · PAID

The text below is a very recent leaked document, which was shared by an anonymous individual on a public Discord server who has granted permission for its republication. It originates from a researcher within Google. We have verified its authenticity. The only modifications are formatting and removing links to internal web pages. The document is only the opinion of a Google employee, not the entire firm. We do not agree with what is written below, nor do other researchers we asked, but we will publish our opinions on this in a separate piece for subscribers. We simply are a vessel to share this document which raises some very interesting points.

Open Source LLM

The timeline (part 1)

- **Feb 24, 2023 - LLaMA is Launched**
  - Meta launches LLaMA, open sourcing the code, but not the weights.
- **March 3, 2023 - The Inevitable Happens**
  - Within a week, LLaMA is leaked to the public. Existing licenses prevent it from being used for commercial purposes, but suddenly anyone is able to experiment.
- **March 12, 2023 - Language models on a Toaster**
  - A little over a week later, Artem Andreenko gets the model working on a Raspberry Pi. At this point the model runs too slowly to be practical because the weights must be paged in and out of memory.
- **March 13, 2023 - Fine Tuning on a Laptop**
  - The next day, Stanford releases Alpaca, which adds instruction tuning to LLaMA.
  - Suddenly, anyone could fine-tune the model to do anything, kicking off a race to the bottom on low-budget fine-tuning projects. What's more, *the low rank updates can be distributed easily and separately* from the original weights, making them independent of the original license from Meta.
- **March 18, 2023 - Now It's Fast**
  - Georgi Gerganov uses 4 bit quantization to run LLaMA on a MacBook.
- **March 19, 2023 - A 13B model achieves "parity" with Bard**
  - The next day, a cross-university collaboration releases Vicuna, and uses GPT-4-powered eval to provide qualitative comparisons of model outputs. The evaluation method is suspect though.
  - They sampled examples of "impressive" ChatGPT dialogue posted on sites like ShareGPT.

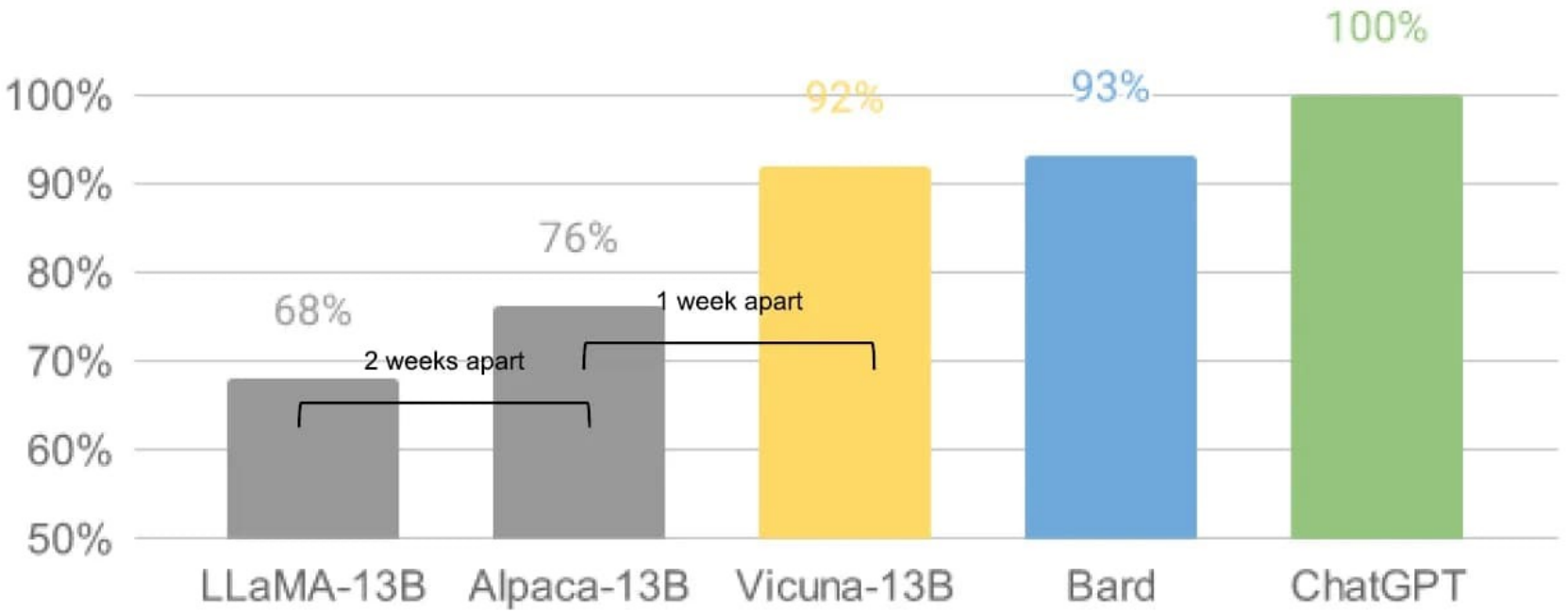
Open Source LLM

The timeline (part 2)

- **March 25, 2023 - Choose Your Own Model**
  - Nomic creates GPT4All, which is both a model and, more importantly, an ecosystem. For the first time, we see models (including Vicuna) being gathered together in one place. Training Cost: $100.
- **March 28, 2023 - Open Source GPT-3**
  - Cerebras trains the GPT-3 architecture using the optimal compute schedule implied by Chinchilla, and the optimal scaling implied by μ-parameterization. These models are trained from scratch, meaning the community is no longer dependent on LLaMA.
- **March 28, 2023 - Multimodal Training in One Hour**
  - Using a novel *Parameter Efficient Fine Tuning (PEFT) technique*, LLaMA-Adapter introduces instruction tuning and multimodality in one hour of training.
- **April 3, 2023 - Real Humans Can't Tell the Difference Between a 13B Open Model and ChatGPT**
  - Berkeley launches Koala, a dialogue model trained entirely using freely available data.
  - They take the crucial step of measuring real human preferences between their model and ChatGPT. While ChatGPT still holds a slight edge, more than 50% of the time users either prefer Koala or have no preference.
- **April 15, 2023 - Open Source RLHF at ChatGPT Levels**
  - Open Assistant launches a model and, more importantly, a dataset for Alignment via RLHF. Their model is close (48.3% vs. 51.7%) to ChatGPT in terms of human preference. In addition to LLaMA, they show that this dataset can be applied to Pythia-12B, giving people the option to use a fully open stack to run the model.

Vicuna

Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality



*GPT-4 grades LLM outputs. Source: https://vicuna.lmsys.org/

https://github.com/lm-sys/FastChat

RETRAINING MODELS FROM SCRATCH IS THE HARD PATH

Retraining models from scratch is the hard path

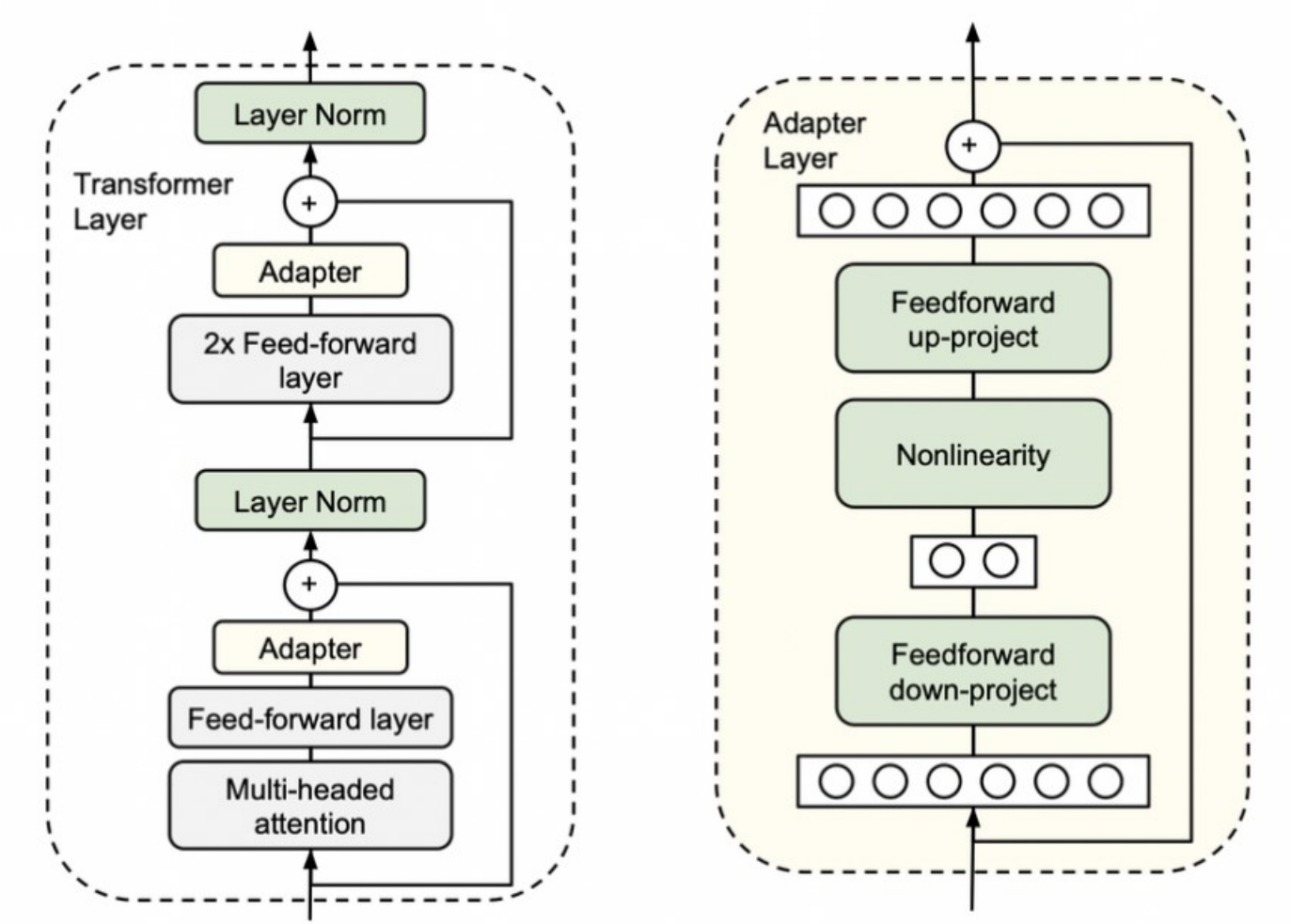
- LoRA (Low-rank adaptation) achieves cost-effective model fine-tuning by utilizing low-rank factorizations to represent model updates, leading to significant reduction in the size of update matrices by several thousand times.
- The effectiveness of LoRA lies in its stackability, which enables the application of improvements such as instruction tuning that can be built upon by other contributors for various tasks such as dialogue, reasoning, and tool use.
- As a result, the model can be easily updated to keep pace with new and better datasets and tasks without the need for expensive full runs.

PEFT

https://github.com/huggingface/peft

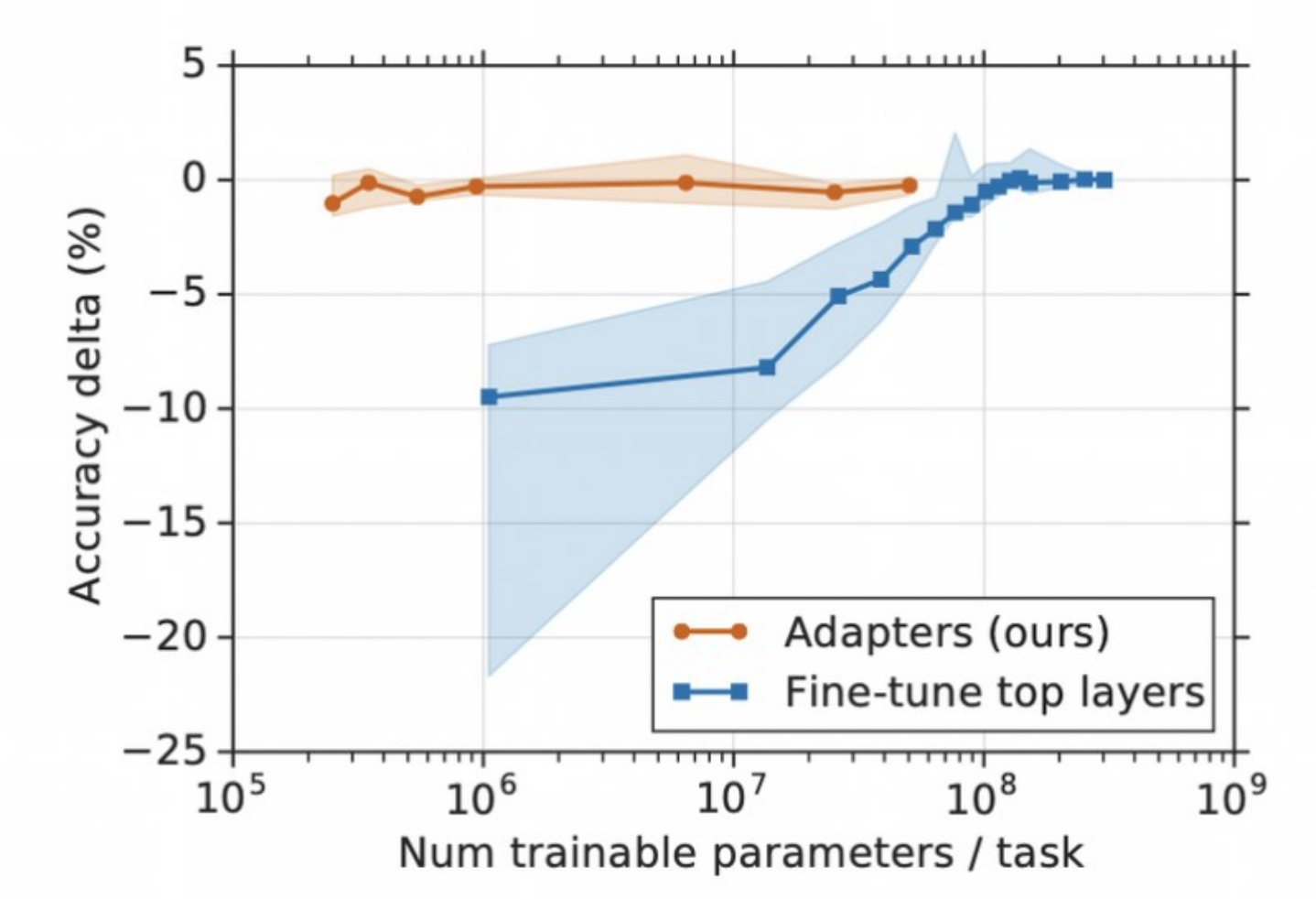
- Parameter-Efficient Fine-Tuning (PEFT) methods enable efficient adaptation of pre-trained language models (PLMs) to various downstream applications without fine-tuning all the model's parameters.
- Supported methods:
  1. LoRA: LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS
  2. Prefix Tuning: Prefix-Tuning: Optimizing Continuous Prompts for Generation, P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks
  3. P-Tuning: GPT Understands, Too
  4. Prompt Tuning: The Power of Scale for Parameter-Efficient Prompt Tuning
  5. AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning

Parameter-Efficient Transfer Learning for NLP

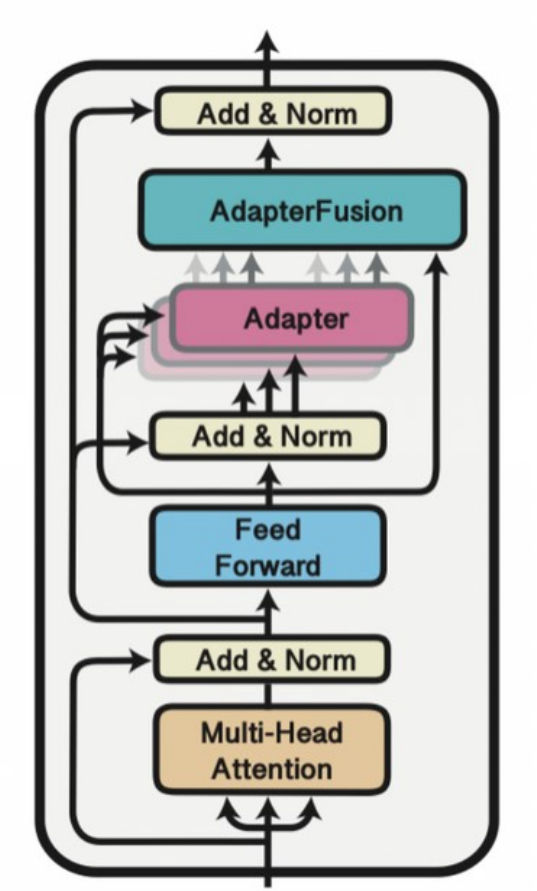


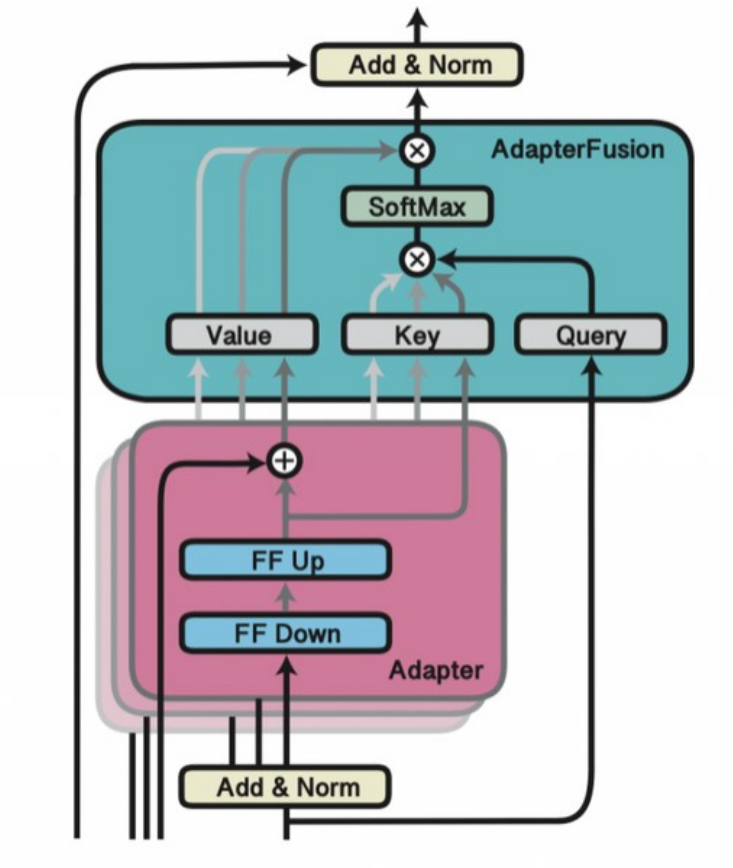
„Parameter-Efficient Transfer Learning for NLP" - Houlsby et al. (https://arxiv.org/pdf/1902.00751.pdf)

Adapter Layers



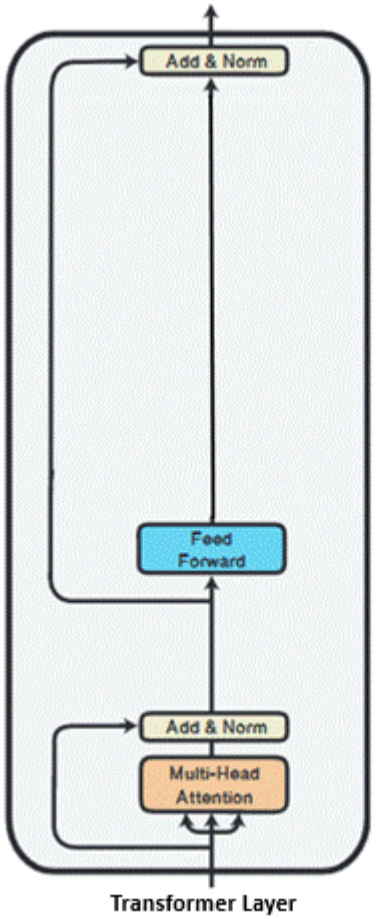
AdapterFusion: Non-Destructive Task Composition for Transfer Learning





Pfeiffer et al. (https://arxiv.org/pdf/2005.00247.pdf)

AdapterHUB



https://adapterhub.ml/blog/2020/11/adapting-transformers-with-adapterhub/

LoRA: Low-Rank Adaptation of Large Language Models

- Low-Rank Adaptation of Large Language Models (LoRA) is a training method that accelerates the training of large models while consuming less memory.
- It adds pairs of rank-decomposition weight matrices (called update matrices) to existing weights, and only trains those newly added weights.

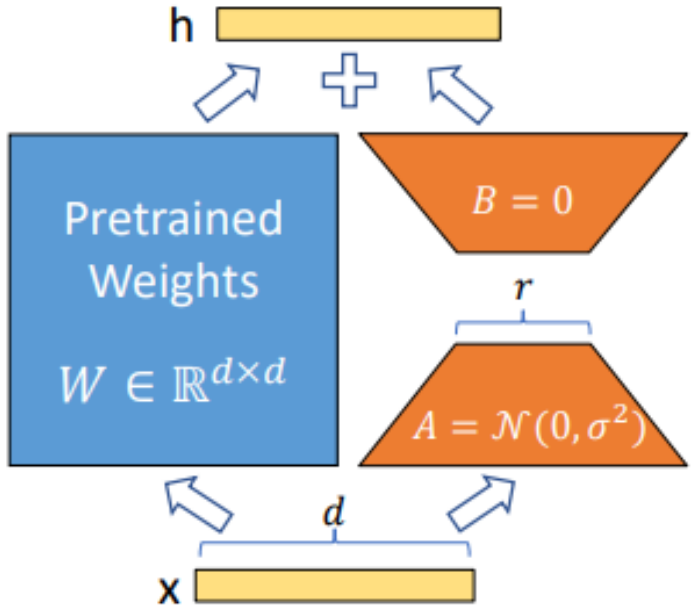
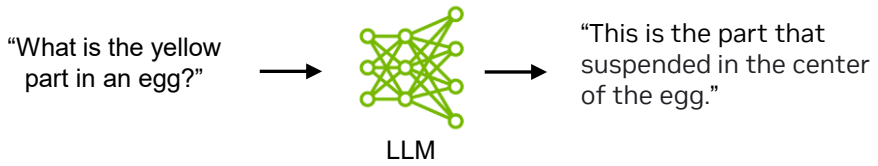


Figure 1: Our reparametrization. We only train A and B.

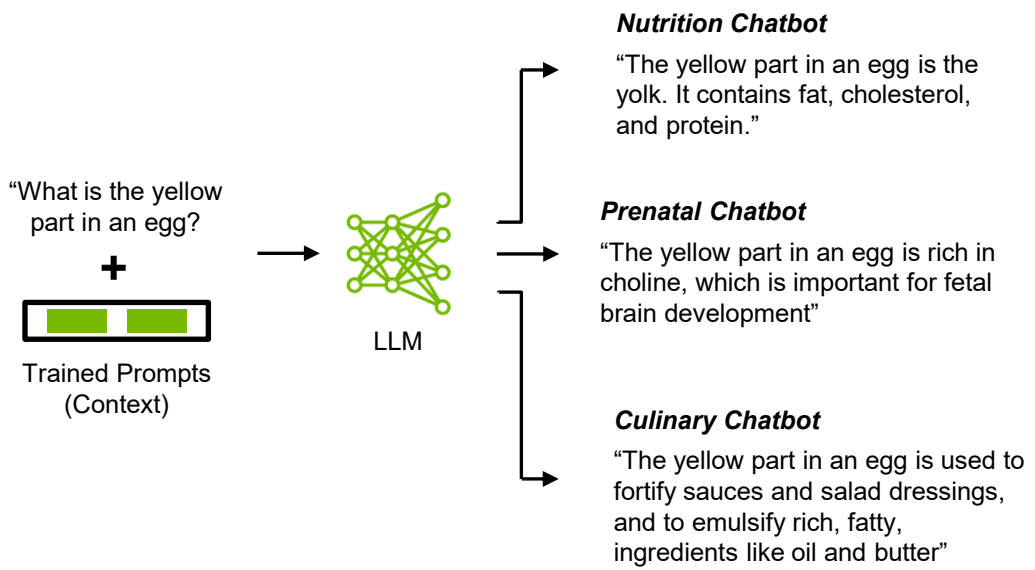
https://arxiv.org/abs/2106.09685
https://github.com/microsoft/LoRA

Customization is Required to Address Business-specific Tasks

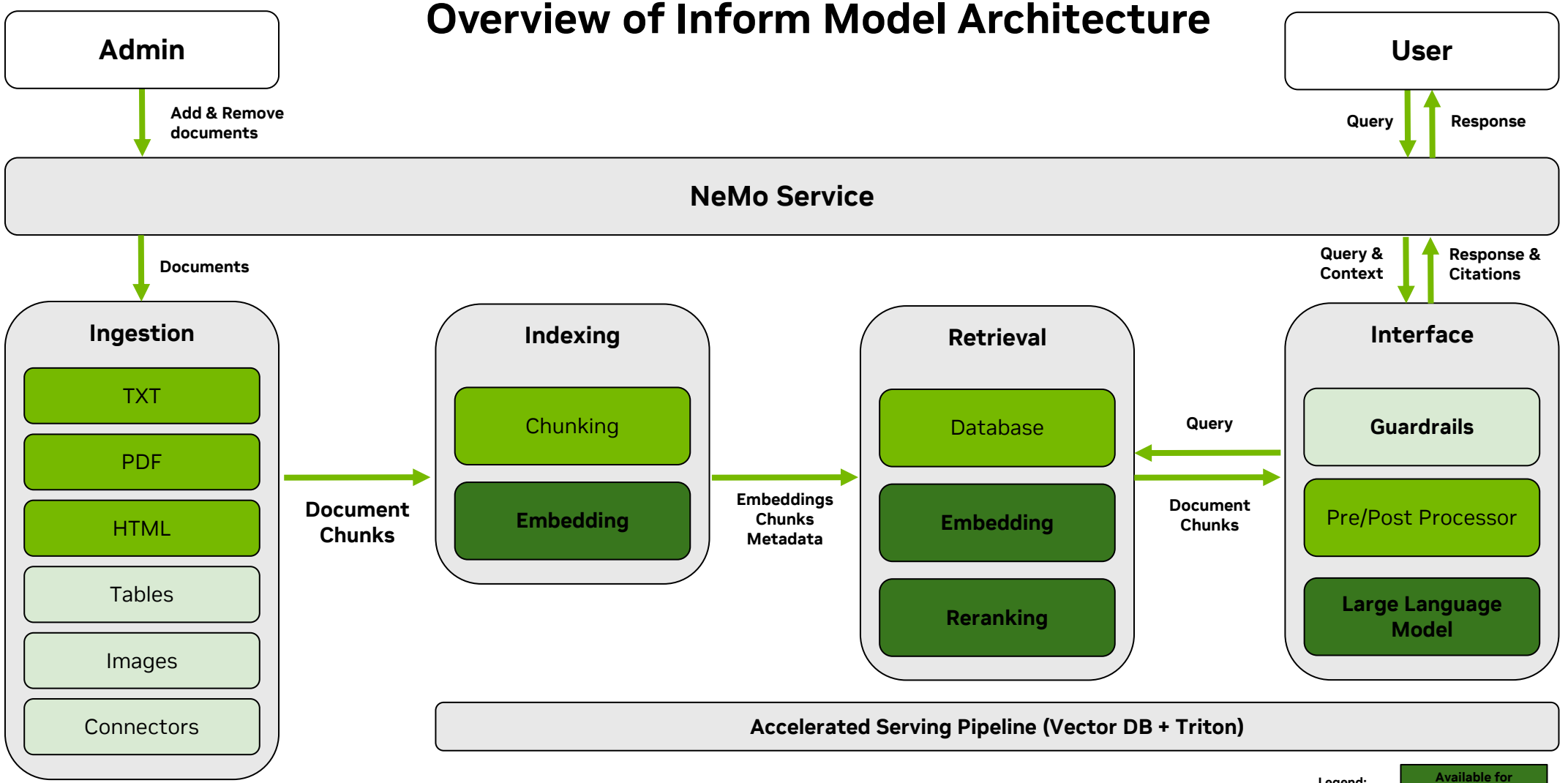
Zero-Shot Response



P-Tuned Response

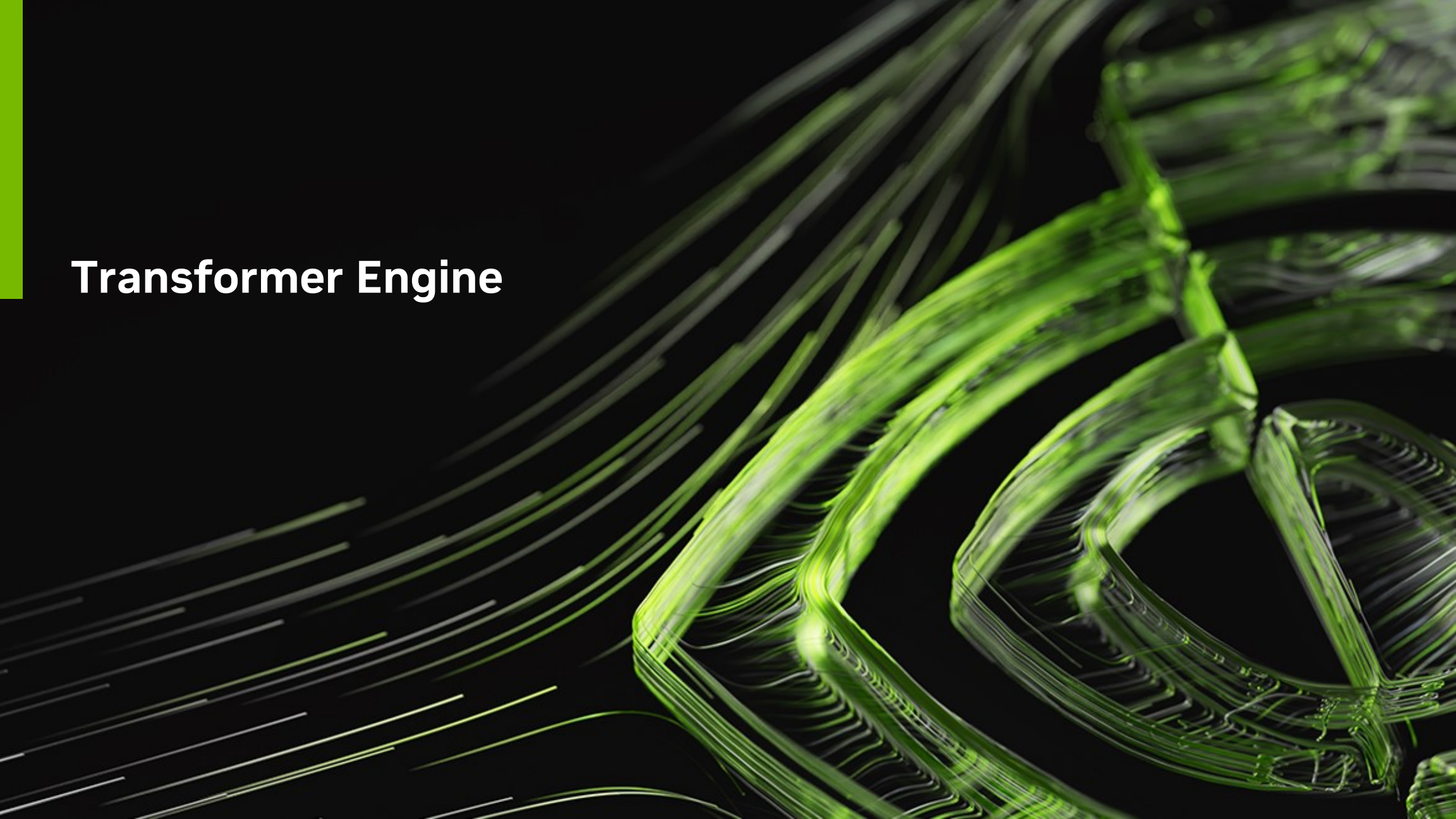


Overview of Inform Model Architecture



https://github.com/NVIDIA/NeMo-Guardrails

Transformer Engine



Transformer Engine

https://github.com/NVIDIA/TransformerEngine

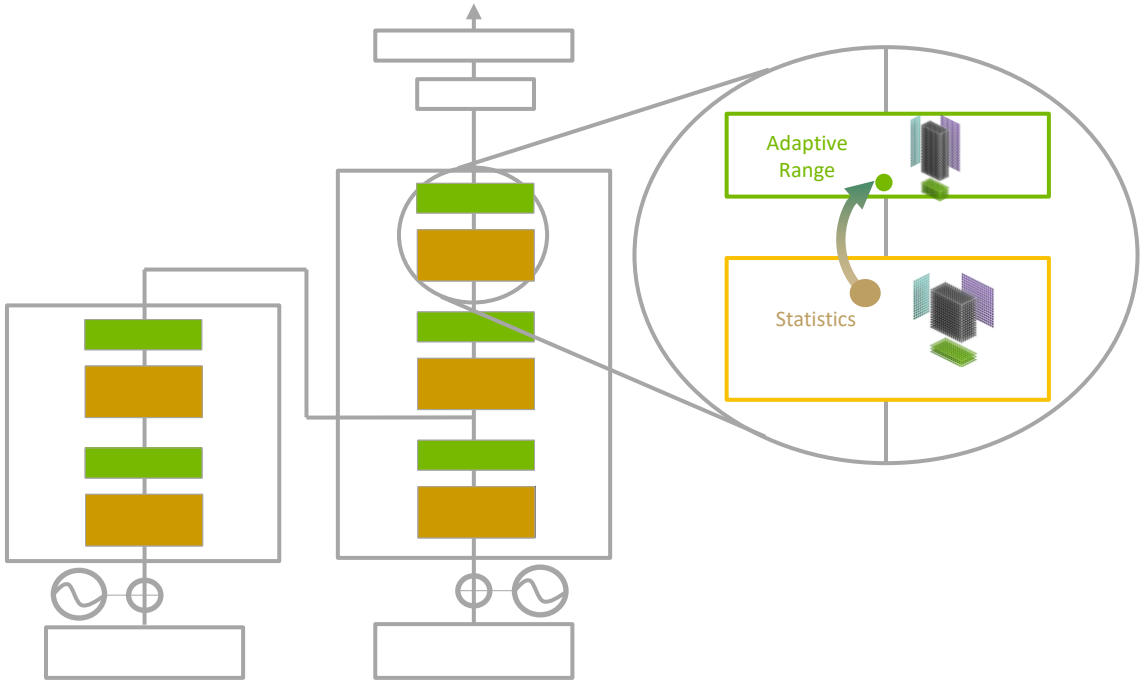
- Transformer Engine (TE) is a library for accelerating Transformer models on NVIDIA GPUs, including using 8-bit floating point (FP8).
- TE provides a collection of highly optimized building blocks for popular Transformer architectures and an automatic mixed precision-like API.

```
import transformer_engine.pytorch as te

class BasicTEMLP(torch.nn.Module):
    def __init__(self,
                 hidden_size: int,
                 ffn_hidden_size: int) -> None:
        super().__init__()
        self.linear1 = te.Linear(hidden_size, ffn_hidden_size, bias=True)
        self.linear2 = te.Linear(ffn_hidden_size, hidden_size, bias=True)

    def forward(self, x):
        x = self.linear1(x)
        x = torch.nn.functional.gelu(x, approximate='tanh')
        x = self.linear2(x)
        return x
```

Transformer Engine

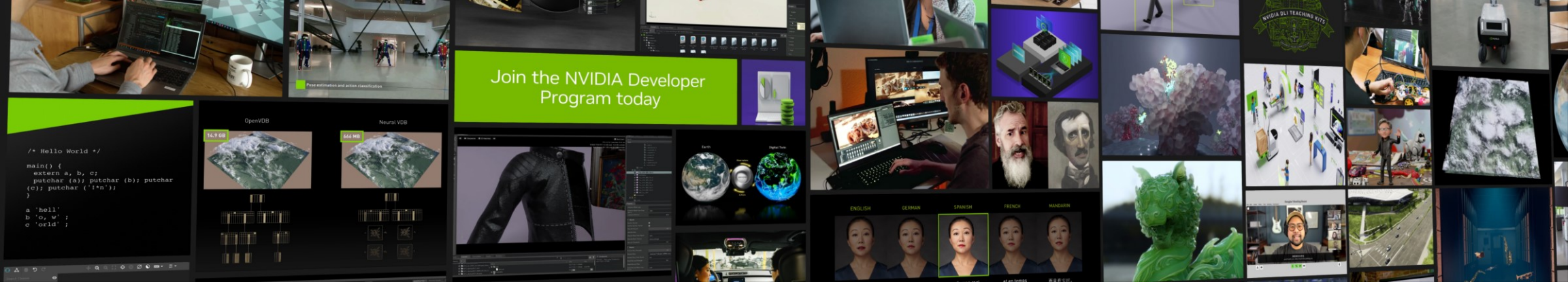
Tensor Core Optimized for Transformer Models

- 6X Faster Training and Inference of Transformer Models
- NVIDIA Tuned Adaptive Range Optimization Across 16-bit and 8-bit Math
- Configurable Macro Blocks Deliver Performance Without Accuracy Loss



Statistics and Adaptive Range Tracking

16-bit 8-bit



NVIDIA Developer Program

The Community that Builds

Program Benefits:

Tools

- 550+ exclusive SDKs and models
- GPU-optimized software, model scripts, and containerized apps
- Early access programs and the NVIDIA Academic Hardware Grant Program*

Training

- Research papers, technical documentation, webinars, blogs, and news
- Technical training and certification opportunities
- 1,000s of technical sessions from industry events On-Demand

Community

- NVIDIA developer forums
- Exclusive meetups, hackathons, and events

Join the Community

** The Hardware Grant Program is available to qualified researchers and educators.*

Many thanks!