Large scale deep learning optimizations

Povzetek / Abstract

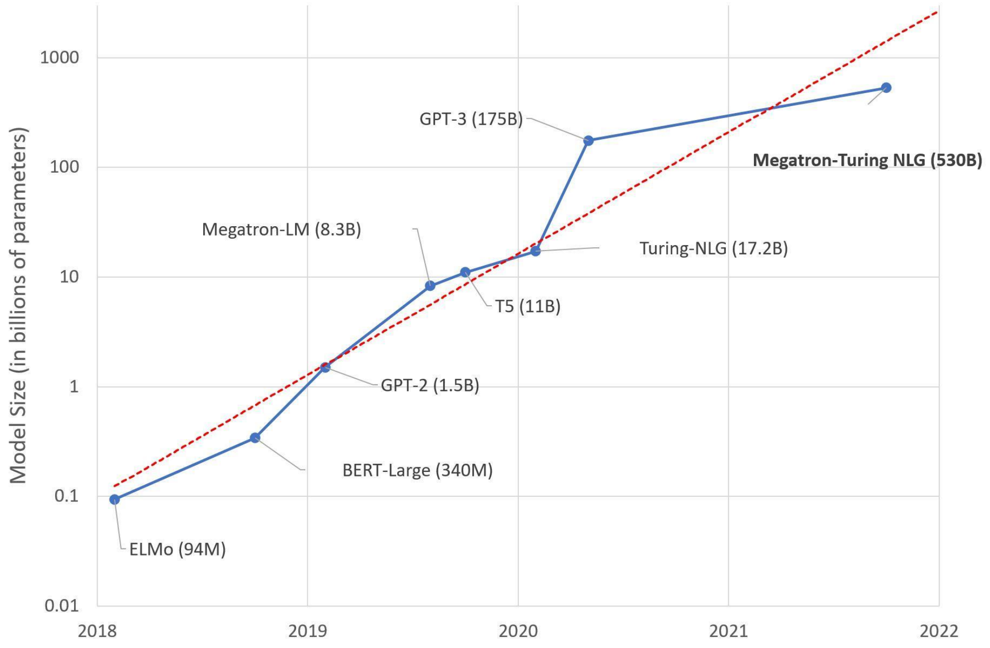
Recent research in large scale and low latency training and inference of deep
learning models showed big potential to drive the next wave of artificial intelligence through
more complex models, faster iterations, lower resource usage and faster predictions.
Distributed training can be split in data and model parallel. Further improvements such as
ZeRO algorithms to reduce memory overhead and communication efficient optimizers are
presented. Concept of lower precision will be explained, commonly used with newer
Nvidia GPUs utilizing tensor cores in the form of mixed precision for training and quantization
for inference. Other inference optimization techniques will also be summarized like
knowledge distillation and pruning. Theoretical knowledge will be applied to popular
architectures such as transformer-based neural network architectures. We will conclude by defining some open questions which will significantly upgrade current algorithms, open new research frontiers or solve some engineering challenges to make it more accessible.

Ključne besede / Keywords: distributed training, inference optimization, low precision, knowledge distillation, pruning

1 Training

1.1 Introduction

Training is the first step after we define model architecture in order to learn the model on a dataset in effort to detect patterns and apply them to new data during inference. Multiple approaches will be presented for training deep learning models, starting with distributed training which helps us speed up training and/or enable training of large models, making it especially useful in natural language processing with the emergence of transformer architecture. We can observe usage of deep learning models like GPT-3(175 billion parameters) (Brown, 2020) and Megatron-Turing NLG(530 billion parameters) (Patwary, 2022) and some even observed a linear trend in the number of parameters which they named the “new Moore’s law” (Simon, 2021). We will continue with introduction of mixed precision training and to that end present bfloat format and different Nvidia GPU architectures along with ways to connect GPUs, either inside or between compute nodes. Connecting GPUs efficiently is important since a lot of times model training is struggling due to connectivity and we will present software layer optimization in this regard in the form of general and model specific communication efficient optimizers.



Picture 1: Model size grows in linear trend (Simon, 2021)

1.2 Distributed deep learning

1.2.1 Modes of parallelism

If the model can be fit on a single GPU and we’d like to speed up training we can use data parallelism where N model replicas are fed N(or its multiplier, at least in the simple form) data partitions. Model parallelism is used when a model layer can’t be fit on a single GPU, also called intra-layer or tensor parallelism where a layer is distributed among multiple GPUs. Pipeline parallel is when layers are distributed on different GPUs in an efficient way to minimize idle resources. We can also combine pipeline, tensor and data parallelism which is called 3D parallelism or PTD-P (Narayanan, 2021).

1.2.2 Data parallel

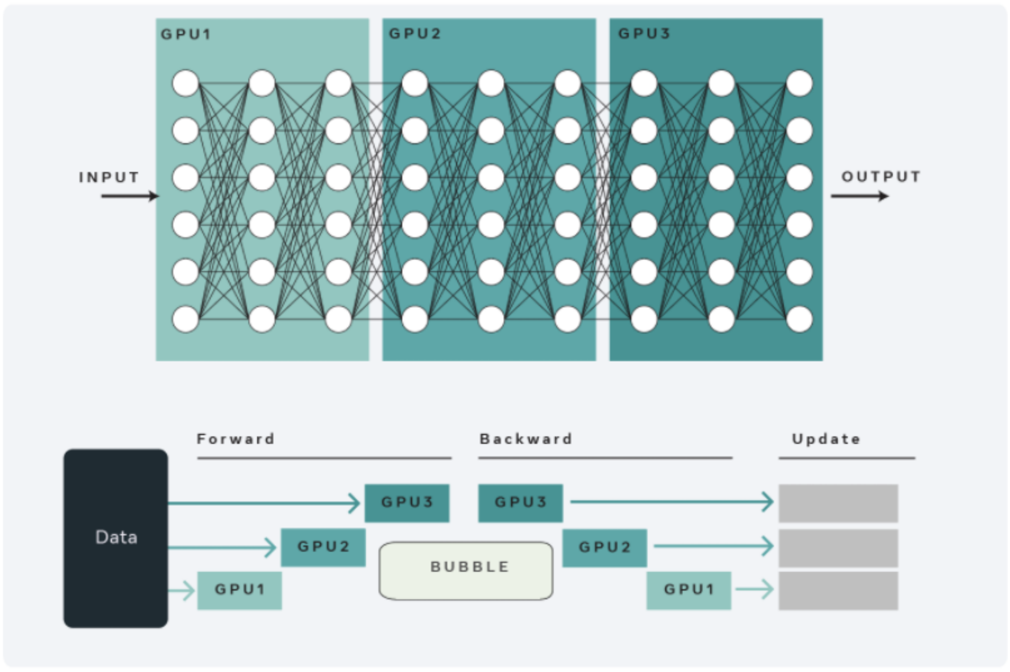
Data parallelism is done by sharding the data and feeding partitions to model replicas which can be synced through allreduce or synchronization server (Li, 2020). The latter is not commonly used as it presents an additional problem of managing the sync server in a highly available manner, but allows for some further customization for synchronization logic(Li, 2014). Most of the time we use tree-based allreduce, ring-based allreduce or CollNet, but this depends on the backend(NCCL, MPIU, GLOO) which chooses the most appropriate algorithm based on configurable thresholds with default values. PyTorch seems to have the most advanced (Li, 2020)implementation with multiple features. For starters it’s synchronizing gradients and not averaging parameters to avoid wrong results with the later technique which are especially noticeable with optimizers relying on past local values such as first and second order momentum in Adam optimizer along with enabling overlap of communication with computation. As many other problems in distributed deep learning there are some improvements in order to make the most out of communication versus computation tradeoff(Li, 2020):

- gradient bucketing: sending small tensors is inefficient so we collect them in a buckets(with default size of 25MB based on empirical data(Li, 2020) since it largely depend on compute environment and model) and send whole bucket
  - smaller bucket means inefficient communication, while larger bucket can mean delayed synchronization which can lead to decreased training results which leads to lower accuracy
- overlap communication with computation: sending data asynchronously
  - PyTorch has separate object for optimizer for practical reasons and optimizer states are saved in a bit map, synchronized in a separate allreduce operation
- gradient accumulation: not only bucketing multiple tensors, but multiple interactions, therefore breaking batch in a microbatch and sending one bucket with multiple tensors per microbatch to further improve communication efficiency
  - very important feature from scalability point of view because it brings linear scalability with negligible performance loss

ProcessGroup is an important concept in PyTorch DDP (Li, 2020) and we can think of it as a group of processes working together, which means a group of CUDA streams in the NCCL backend or a group of threads in GLOO, depends on the backend used. We can also improve performance with composite ProcessGroup where we put ProcessGroups in an array and feed data to them with Round robin[1] load balancing strategy to overcome ProcessGroup level concurrency limit.

1.2.3 Pipeline parallel

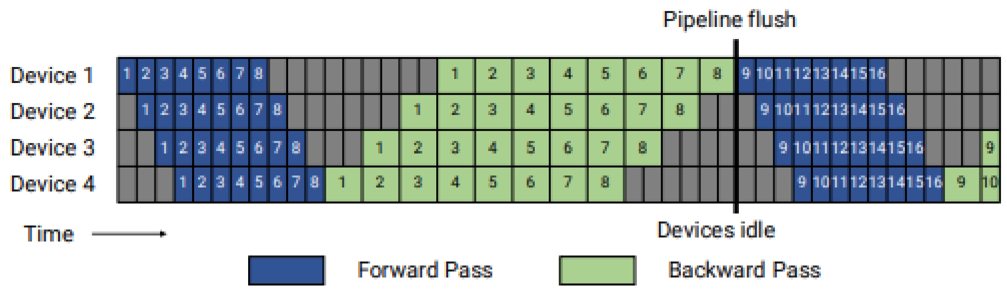
Pipeline parallel and tensor parallel have to be utilized to enable training if the model or even single layer can't be fit on a single GPU. Pipeline parallel places different layers on different GPUs and streams data through them in a way that reduces idle time for GPU resources which means minimizing bubble size(time between forward and backward pass) in distributed training terminology. There are multiple ways to stream data through our partitioned model which are called pipeline schedules.



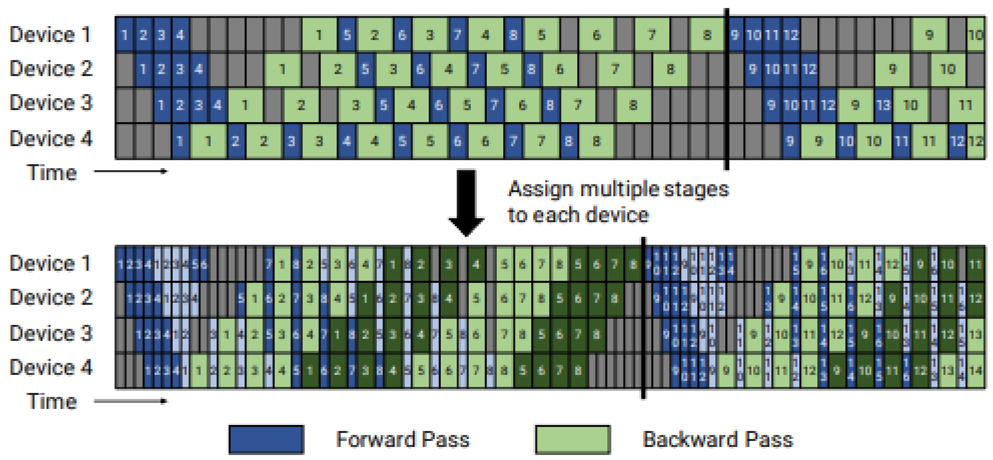
Picture 2: PIpeline parallel visualization where model layers are distributed on different GPUs where model is feed single data partition, while data parallelism is where model replicas are feed different data partitions (Meta, 2022)

GPipe (Narayanan, 2021) and PipeDream (Narayanan, 2019) are considered to be state of the art scheduling algorithms and are visualized in Picture 2 and 3.

[1] https://en.wikipedia.org/wiki/Round-robin_scheduling



Picture 2: GPipe (Huang, 2019) scheduling algorithm(picture taken from Narayanan, 2021)



Picture 3: PipeDream scheduling algorithm. Upper picture presents the default schedule presented in (Narayanan, 2019) and the bottom picture summarizes the extension presented in (Narayanan, 2021). (Narayanan, 2021)

We can observe significant bubble time in GPipe scheduling which is inefficiency improved with the so -called 1 forward 1 backward schedule(1F1B for short) presented in PipeDream (Narayanan, 2019) paper. Scheduling can be made even more efficient by combining(Narayanan, 2021) GPipe and PipeDream idea by adding interleaving, shown on the bottom of Picture 3. We introduce some notations in order to quantify bubble time:

- t_f: time to execute forward pass
- t_b: time to execute backward pass
- t_{pb}: size or time of pipeline bubble
- t_{id}: ideal time per iteration
- m: number of microbatches(batch is split in m microbatches)
- p: number of stages which is number of GPU devices in the pipeline
- v: number of model chunks, introduced in 1.2.3.3 Interleaved schedule
- ×: multiplication

1.2.3.1 GPipe schedule

(Huang, 2019)Pipeline bubble first does all forward passes and then moves on to backwards passes to finish all backward passes before moving to forward passes for the new batch. It contains p-1 forward passes and p-1 backward passes, which motivates equation 1.

$$t_{pb} = (p-1) \times (t_f + t_b) \tag{1}$$

Ideal processing time is defined in equation 2 and intuitively means processing time if the model would be on the single GPU without any concerns and speed downs of parallelism, which is the last missing piece for bubble time fraction or pipeline bubble size(PBS) defined in equation 3.

$$t_{id} = m \times (t_f + t_b) \tag{2}$$

$$\text{PBS} = \frac{t_{pb}}{t_{id}} = \frac{p-1}{m} \tag{3}$$

We want small PBS which means we should strive for m >> p, however based on the GPipe algorithm where all forward passes are done first and then all backward passes, high m means high memory footprint due to stashing of intermediate activations in-between forward and backward pass. This is addressed in the next section, 1.2.3.2 PipeDream-Flush schedule.

1.2.3.2 PipeDream-Flush schedule

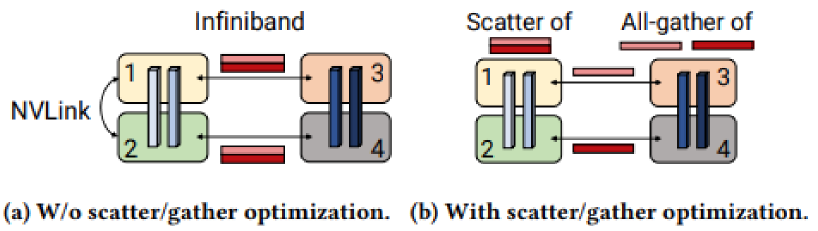
(Narayanan, 2019)PipeDream-Flush is a 1F1B algorithm which starts with few forward passes as part of the warm-up phase. After warm up phase, which differs for each worker, pipeline starts with a 1F1B schedule and at the end all outstanding backward passes are finished. This schedule limits the number of in-flight microbatches(microbatches with outstanding backward pass, which has consequence that activations from forward pass have to be kept or cached for backward pass) to the depth of the pipeline instead of the number of microbatches in a batch like GPipe. This is important since it solves GPipe memory inefficiency with m >> p discussed before, because PipeDream-Flush has maximum of p inflight requests, while GPipe has maximum of m inflight requests. When m >> p it makes PipeDream-Flush the best choice for this scenario, which is important because we tend to use larger batches for large models.
PipeDream-BW2 (Narayanan, 2021) further proposes improvements such as gradient coalescing strategy to further reduce memory usage and automatic model partitioning along with well defined mathematical formulation of some pipeline parallel concepts.

1.2.3.3 Interleaved schedule

In order to reduce pipeline bubble layer to device mapping can be reshuffled(Narayanan, 2021) by moving away from contiguous layer subsets(also called model chunks) to round robin distribution to process multiple layers. For example if we have 4 layers and 2 GPU devices we placed layers 1 and 2 on GPU 1 and layers 3 and 4 on GPU 2 before, but now we use Round robin balancing which moves layer 1 and 3 to GPU 1 and layers 2 and 4 to GPU 2, but with limitation that number of microbatches must be integer multiple of number of GPU devices to achieve optimal efficiency. We use the 1F1B idea from PipeDream-Flush in order to be memory efficient with regards to caching activations. Because we changed how layers are assigned to GPU devices we can divide t_f and t_b with v which in consequence enables us to divide PBS equation introduced in equation 3 with v, which means bubble time is decreased but communication overhead is multiplied by v which can be improved with addons like Nvidia Infiniband, AWS EFA or newer GPU server architectures like Nvidia H100 racks.
We can empirically see in (Narayanan, 2021) that gap between interleaved and default PipeDream-Flush starts to converge as batch size increases and there are 2 possible explanations:

1. increased batch size means smaller bubble time in default schedule
2. the amount of point-to-point communication in pipeline parallel is proportional to batch size which makes interleaved schedule less efficient due to how model chunks are created

Interleaved schedule also requires scatter-gather optimization to outperform default schedule(Picture 4).



(a) W/o scatter/gather optimization. (b) With scatter/gather optimization.

Figure 9: Scatter/gather communication optimization. Light blue blocks are layers in the first pipeline stage, and dark blue blocks are layers in the second pipeline stage. Without the scatter/gather optimization, the same tensor is sent redundantly over inter-node InfiniBand links. Instead, at the sender, we can scatter the tensor into smaller chunks, reducing the sizes of tensors sent over InfiniBand links. The final tensor can then be rematerialized at the receiver using a gather operation.

Picture 4: Scatter-gather optimization where we avoid duplication over Infiniband(point-to-point communication) by sending tensor partitions and performing all-gather to re-create the whole tensor.

1.2.4 Tensor parallelism

Tensor parallelism, also referred to as intra-layer parallelism, largely depends on model architecture. For example with transformer architecture we want to exploit parallelism in the multi-head attention parallelism by partitioning K, Q and V matrices in column-parallel fashion. PyTorch team is currently improving developer experience for tensor sharding[2], while some alternatives already exist[3].

1.2.5 3D parallelism

With larger models we have to combine tensor, pipeline and data parallelism to maximize the effect of distributed training. We present 3 important rules on how to combine parallelism modes effectively taken from (Narayanan, 2021). Notation definitions for the following section is:

- (p, t, d): parallelism dimensions for pipeline, tensor and data
- n: number of GPUs(we require p × t × d)
- B: batch size
- b: microbatch size
- m: number of microbatches per pipeline($\frac{B}{b \times d}$)
- s: sequence length
- h: hidden state
- l^{stage}: applicable to pipeline parallel, number of layers in a given stage
- ×: multiplication

It has been shown (Narayanan, 2021) that correctly configured 3D parallelism can enable almost linear throughput up to over 2000 GPUs for 175 billion GPT-3 transformer(and up to around 1600 billions for 530 billion Megatron LM) compared to ZeRO stage-3 which shows significant drop.

1.2.5.1 Tensor and pipeline parallelism

We will extend PBS from equation 3 with replacing p with assumption that t × p = n which gets us equation 4.

$$\text{PBS} = \frac{\frac{n}{t} - 1}{m} = \frac{(n-t)}{t \times m} \tag{4}$$

We can observe that more tensor partitions(higher t) means larger pipeline bubble and we also know that higher t and p increases communication, but differently since pipeline parallelism uses cheap point-to-point communication on every pair of passes(forward or backward pass) and tensor parallelism uses more expensive allreduce communication for each forward and backward pass effectively doubling communication frequency and increasing throughput. Next step is to quantify our idea with b×s×h. Tensor parallelism leads to equation 5.

$$\text{communication(per layer)} = 8 \times b \times s \times h \times \frac{t-1}{t} \tag{5}$$

Pipeline parallel multiplies tensor parallelism by l^{stage}. We can therefore conclude based on equation 5 that avoiding tensor parallel across nodes is important while it’s more manageable to use pipeline parallels in cross server fashion.
Tools like Nvidia Infiniband, AWS EFA and new Nvidia H100 can help with node-to-node throughput.

1.2.5.2 Model(pipeline and tensor) and data parallelism

Let’s first explore pipeline parallelism, which means we set t=1 and expand PBS equation defined in 1.2.3 with expanding m:

$$\text{PBS} = \frac{n-d}{\frac{B}{d}} = \frac{(n-d) \times d}{B} \tag{6}$$

We can see increasing d brings down PBS, but will increase communication. Based on ring-based all-reduce complexity being $\frac{d-1}{d}$, which can be rewritten to $1 - (\frac{1}{d})$ we can see communication overhead should not be a big concern and therefore an increase of d will increase throughput. Another angle to decrease PBS is by increasing B, which is great since it also makes all-reduce data parallel communication less infrequent, further increasing throughput.
We continue with comparing tensor instead of pipeline parallel by setting p=1 and t>1. The difference in frequency of all-reduce based communication is significant in comparing tensor and data parallel: tensor parallel will require all-reduce every microbatch, while data parallel will require all-reduce every batch.

[2] https://github.com/pytorch/pytorch/issues/55207
[3] https://github.com/KaiyuYue/torchshard

Conclusion we can draw here is to reduce tensor parallelism as much as possible, one potential tool can be memory-centric operator tilling introduced in ZeRO-infinity (Rajbhandari, 2021).

1.2.5.3 Microbatch size

By further expanding on the PBS equation we can see that microbatch size depends on a lot of factors(b, throughput and memory characteristics of the model, p, d, B) and therefore is case specific. Increasing microbatch for example leads to 2 opposite effects:

- decreases m, which leads to larger bubble time therefore considered as negative side effect
- improve GPU utilization which is favorable side effect

1.2.6 Efficient memory management: ZeRO papers

ZeRO stands for zero redundancy optimizer, which hints on minimizing redundancies to minimize memory footprint in effort to fit a larger model on a single GPU.
First paper(Rajbhandari, 2020) in ZeRO sequence of papers splits redundancies in 2 parts and addresses them with ZeRO-DP and ZeRO-R solutions:

- ZeRO-DP(Rajbhandari, 2020) is about model states: shard optimizer state, gradients and parameters which are 3 stages of this algorithm and uses all-reduce(1st stage is optimizer state sharding, 2nd stage is 1st stage along with gradient sharding, 3rd stage is 2nd stage along with parameter sharding).
- ZeRO-R(Rajbhandari, 2020) is about residual states: activation partitioning(identify and remove duplicates), find appropriate constant temporary buffer sizes (to strike balance between memory and computation) and proactively reduce memory fragmentation(memory fragmentation is result of interleaving between short and long lived memory objects - activation checkpoints during forward pass and parameter gradients during backward pass are long lived, while recomputed activation during forward pass and activation gradients during backward pass are short lived; ZeRO pre-allocates contiguous memory chunks)

ZeRO-DP stage 3 and ZeRO-R presented large improvement when it was published, namely 8x increase in model size nad 10x increase in performance over state of the art. They used a 100 billion parameter transformer which was successfully scaled to 400 GPUs.
ZeRO-offload (Ren, 2021) sharded data parallel approach, which is essentially ZeRO-DP stage 3. It's exploring heterogeneous compute architecture to use CPU memory and computation by offloading optimizer memory and computation from GPU to host CPU along with introduction of Adam optimizer which is optimized for CPU and one step delay of parameter update to overlap CPU parameter update with GPU forward and backward pass. They also introduced the concept of saving model states in FP32 on CPU and then casting to FP16 when transferring to GPU by developing float2half algorithm since we usually have a lot more free RAM than GPU memory and by compressing the data sent from CPU to GPU we are reducing communication bandwidth . CPU offload is an important technique, but we have to be careful to not cause communication bottleneck between CPU and GPU with the newest Nvidia GPU H100 architecture significantly improving CPU-GPU bandwidth. ZeRO offload in combination with model parallelism improved model size for Nvidia DGX-2 by 4.5x compared to only using model parallelism.
Last paper in the sequence, ZeRO-infinity (Rajbhandari, 2021) continues with exploiting heterogeneous architecture by using CPU and NVMe memory simultaneously, unencumbered by their limited bandwidth. They developed and open sourced Infinity offload engine to improve partitioning of model states to CPU and NVMe memory, memory-centric tilling which is alternative to tensor parallel that partitions the layer and executes it on single GPU, enable large scale model training without refactoring, bandwidth-centric partitioning to enable parallel access to memory(parameters are partitioned and use all-gather instead of broadcast which makes big difference if they are located on CPU) and DeepNVMe module for writing and reading tensors to NVMe storage with Pytorch asynchronously and at near-peak NVMe bandwidth. They continue to advocate some techniques from previous ZeRO papers such as stage 3 ZeRO-DP partitioning, activation checkpointing(where activation is recomputed on backward pass rather than cached on forward pass), CPU offloading along with overlapping communication and computation. ZeRO-infinity made significant improvement by exceeding linear scalability for up to 1 trillion parameter model, enabled training 40x bigger models than with 3D parallelism and enabled training 1 trillion models on a Nvidia DGX-2 without model parallelism.

1.2.6 Fully sharded data parallel

Fully sharded data parallel was moved to PyTorch 1.11 (Zhao, 2022) from only being available in Fairscale[4] before and is based on ZeRO stage 3 sharding along with optional cpu offloading(see 1.2.6) and backward prefetching where we can prefetch next all parameters for next layer during backward pass which increases efficiency by overlapping communication and computation. Wrapping is also an important topic with FSDP in PyTorch where we can tune how much sharding we want to avoid memory overflow. This can be done either through parameters of automatic wrapping or manually wrap layers(Zhao, 2022).

1.2.7 Lower precision

1.2.7.1 Mixed precision training

Mixed precision training(Micikevicius, 2017) was introduced to utilize the tensor cores more efficiently by using FP16 instead of FP32 wherever possible in order to speed up training and reduce memory usage(enabling larger models or use model memory in more efficient way) while avoiding sacrificing training accuracy.
While weights, activations and gradients are represented as FP16 datatypes we maintain FP32 copy of weights for optimizer state to increase optimizer accuracy and reduce GPU memory usage with FP16.
Second improvement is loss scaling which solves the issue of reducing training efficiency with FP16 and smaller gradients. We can therefore maintain accuracy and reduce memory footprint. Loss factor should be chosen empirically with gradient statistics, since we want maximum scaled value being as close to 65504(maximum for FP16) which maximally negates accuracy loss.
Deep learning contains vector operations and authors of original mixed precision paper(Micikevicius, 2017) discovered it’s better to perform some calculations in FP32 and then cast the result to FP16 than just calculating everything in FP16. This idea enables us to maintain precision of FP32 and use memory savings of FP16. Reduction operations should be carried out in FP32, while point-wide operations seems to be accurate even with FP16 which can speed up calculation with expectation of functions such as exponent, logarithm and power which can benefit from FP32 range (Stosic, 2019). Examples of neural network layers that can be run with FP16 are linear and convolution.
Frameworks like PyTorch enable us to run automatic mixed precision (Stosic, 2019) which tunes all 3 features automatically.

1.2.7.2 Bfloat format

Lower precision also means lowering range and precision, which saves memory and bandwidth but can reduce accuracy. The issue with changing range can also be that it might not be just drop in replacement and will require refactoring. Brain floating point (Kalamkar, 2019) or bloat16 for short changes in power and mantissa ratio in floating point numbers by allowing FP32 range for accuracy similar to FP16 with negligible accuracy loss. This means bloat16 can be a drop in replacement for FP32 due to the same range, but offers memory efficiency of FP16. It makes it also great candidate for replacing FP16 in mixed precision training, outlined above in 1.2.7.1, which is empirically proven (Wang & Kanwar, 2019) to improve performance with geometric mean of 13.9% on architectures like transformer, NMT, ResNet, Mask-RCNN and SSD.

1.2.8 Communication efficient optimizers

1.2.8.1 Algorithm specific

1.2.8.1.1 Adam

Adam optimizer (Kingma, 2015) is one of the most popular building blocks of deep learning projects. It builds upon RMSProp which introduces the idea of adapting learning rate for each of the parameters by running average of the magnitudes of recent gradients for that weight. Adam does not only use first moment(mean), but also second moment(variance) with decay factor for both moments. This allows it to handle sparse and noisy gradients along with efficient handling of non-stationary objectives.
Distributed learning adds a new frontier to the optimizers: communication efficiency between GPUs and nodes, which means how optimizers are synced between replicas or partitions.
1-bit Adam (Tang, 2021) is a 2 stage communication efficient version of Adam optimizer. It is motivated by discovery that Adam’s nonlinear term(variance) becomes stable at early stage of the training and the fact that we already know that error-compensated compression works great, but is limited to only some

[4] https://fairscale.readthedocs.io/en/latest/

algorithms like SGD and Momentum SGD which don't have non-linear dependency on gradient that affects compression mechanism which adds compression error on step i-1 to the compression of step i. While this nonlinearity is believed to be core feature of Adam it makes hard to implement error-based compression because it's hard to define correction factor, especially for variance. 1-bit Adam runs Adam in the first stage, called warmup stage to pre-condition momentum SGD by not updating variance term after warm up phase is done and compression stage starts with 1-bit error-compensated compression. 1-bit compression reduces communication between model partitions by 97% and 94% for FP32 and FP16. We denote the end of training phase once Adam variance becomes stable, which can be tuned manually(they used 23000 steps for BERT-Large pre-training) but can be also tuned automatic since Adam variance is unstable during learning rate warm up and can be quantified by setting threshold(they used ≥0.96 which resulted in 22173, which is close to manually tuned 23000 steps) for:

$$\frac{|v|}{|v-\Delta|}\ ;\ \Delta = \frac{1}{1-\beta_2} \qquad (7)$$

where $|v|1$ stands for l1 norm of Adam variance and β_2 for the decay factor of variance in Adam.

Convergence is also analytically proven for this 2-stage algorithm in the paper.
0/1 Adam (Lu, 2022) builds on top of 1-bit Adam with 2 novel ideas:

1. eliminate requirement to run full expensive precision communication at early stage with adaptive variance state freezing by discovering that Adam steps are close and can therefore be sparsely updated
2. 0/1 Adam(Lu, 2022) authors observed that in 2nd 1-bit Adam stage (with frozen variance) there was linear dependency between change to model parameters and Adam momentum state which implies it can be locally approximated, which removes(or at least reduces) the need for communication which motivates "0" part in the name since it requires 0 communication of 1-bit syncs

1.2.8.1.1 LAMB

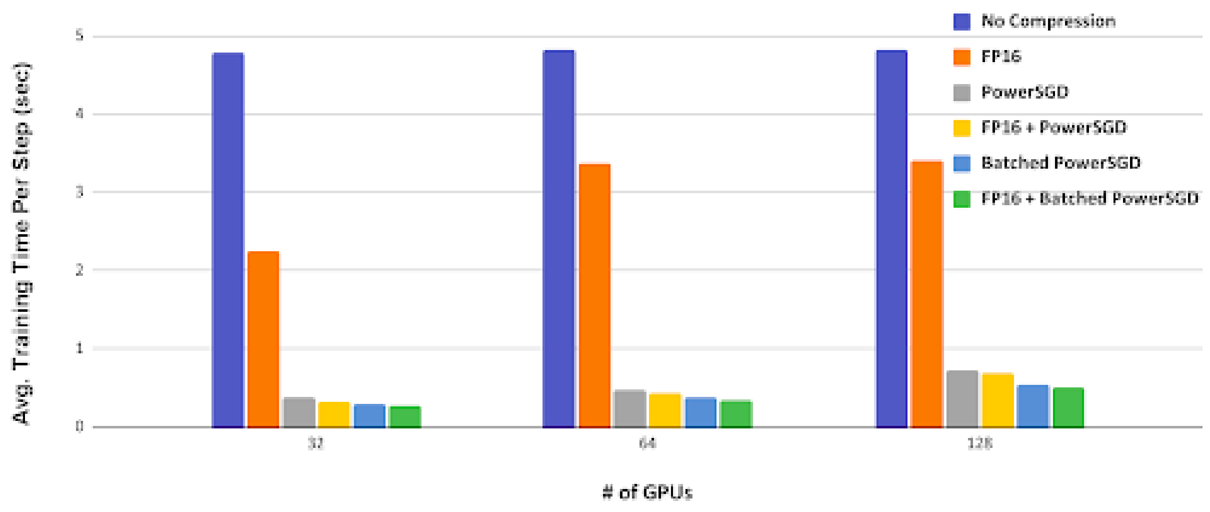
LAMB (You, 2019) is a layerwise adaptive large batch optimizer based on Adam optimizer(Kingma, 2015) and since it was inspired by BERT pre-training it is used for adapting the learning rate in large batch training. LARS (You, 2017) was used in such settings before LAMB, but it had issues with performance on attention models. LARS uses a trust ratio which is the norm of layer weights divided by the norm of layer gradients and it helps stabilize learning rate for big batches by setting learning rate for layer to be global learning rate multiplied by trust ratio. LAMB replaces SGD from LARS with Adam.

1.2.8.2 General

General compression, vector compression and more specifically gradient compression(like quantization, sketching and sparsification) (Hang, 2020) has been studied extensively since it has the advantage of being general and doesn't have to be adapted for each optimizer algorithm. Our focus will be on the most commonly used algorithm for compressing gradients: PowerSGD (Vogels, 2019). This is not optimizer specific communication optimization, but rather a general DDP communication hook in Pytorch which makes it general and applicable to a wide array of optimizers. PowerSGD is based on power iteration factorization and views each non-vector gradient tensor as a matrix. Singular value decomposition is another factorization technique, but PowerSGD is less computationally intensive(more than 5 times faster latency) and can improve matrix approximation with training. PowerSGD should be deferred for some initial training steps since the beginning of training can be sensitive to inaccurate gradients and deferring for 10% is suggested default. Pytorch provides 2 additional optimizations for PowerSGD:

- batching: instead of compressing gradient layer by layer batch them and compress flattened tensor which increases speed, but can reduce accuracy
- low precision: cast gradient to FP16, can sometimes even lead to better accuracy since it prevents potential floating-point underflow during compression

Both optimizations can be used at the same time. PowerSGD is a great tool to use if all-reduce communication is the bottleneck, but this can also point to some potential improvements: communication is not overlapped with computation or communication is not optimized on lower level(AWS EFA, Nvidia InfiniBand,..). One downside of using PowerSGD is that it needs some additional memory, which is already scarce in distributed training and point of optimization.
Just using FP16 with PowerSGD can result in 1.4x-2.1x speed increase which can be increased to almost 180x when using other features of PowerSGD and tuning parameters.



Picture 4: PowerSGD features speedup (Meta and AWS, 2021)

Compression Scheme	Compression Rate
FP16	2
PowerSGD	984
FP16 + PowerSGD	1,969
Batched PowerSGD	1,946
FP16 + Batched PowerSGD	3,892

Picture 5: PowerSGD compression rates[5] (Meta and AWS, 2021)

1.2.9 Operator fusion

Fused operator launches only one CUDA kernel for multiple fused point wise operations, which makes it useful for usage in activation functions (Migacz, 2021). Megatron LM team reported throughout increase of 19% and 11% when operator fusion was applied to the GPT-3 175 billion parameter model and larger 530 billion transformer (Narayanan, 2021).

1.2.12 Activation checkpointing

This technique is mentioned in the ZeRO-offload(Ren, 2021) and Megatron paper(Narayanan, 2021). As with a lot of other techniques in distributed training this technique is about tradeoff between memory and computation. In order to free up some memory we're not caching activation values from the forward pass in the computation graph, but recompute them when we need them in the backward pass.

1.2.13 Profiling

Profiling is a technique that helps us understand bottlenecks or resource(memory) leaks in our code. With distributed deep learning this is an important step because as long as we have a larger model and/or require fast training engineering part, not only modeling is important. We have multiple different profilers with the Nvidia Nsight compute and Native Pytorch profiler(can be imported to Tensorboard) being a popular choice. We'd also like to stress that an important component to monitor and profile is GPU-to-GPU communication, CPU-to-GPU communication and networking in order to find bottlenecks and improve our training.

1.2.13 Libraries

[5] https://en.wikipedia.org/wiki/Compression_ratio

- https://github.com/pytorch/pytorch(https://pytorch.org/docs/stable/fsdp.html)
- https://github.com/microsoft/DeepSpeed - de facto library for latest distributed training research
- https://github.com/facebookresearch/fairscale
- https://github.com/NVIDIA/Megatron-LM
- https://github.com/microsoft/Megatron-DeepSpeed
- https://github.com/nvidia/apex
- https://pytorch.org/torchx/latest/ (previously torch elastic)
- https://github.com/horovod/horovod
- https://github.com/tensorflow/mesh
- https://github.com/bytedance/byteps
- https://github.com/BaguaSys/bagua
- https://github.com/huggingface/accelerate
- https://github.com/ELS-RD/transformer-deploy
- https://github.com/hpcaitech/ColossalAI

2. Inference optimizations

When we're talking about inference optimization we mostly talk about reducing latency, but for some edge deployments we are also very focused on reducing the overall memory footprint of the model since we're working in an environment with minimal resources where we don't have the luxury of bigger GPU instances. Low latency also becomes important with a lot of real world use cases which have to be real time.

2.1 Quantization

The problem of quantization could be defined as how to represent a set of continuous real-valued numbers in order to minimize the number of bits and maximize accuracy.
Lets define a quantization function Q(r) which takes input(r) which is floating point data type and returns either a lower precision float data type or integer.(Gholami, 2021) Popular choice is Q(r)=Integer(r/S) - Z where S is a scaling factor and Z is an integer zero point. To start off we can define multiple dimensions of quantization techniques:

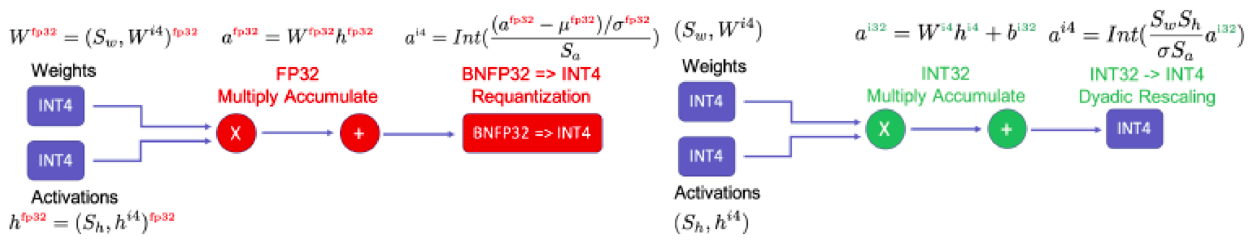
- Quantization function can be uniform(spacing output values uniformly) or non-uniform. Uniformity makes it better overall, non-uniform can better capture the nature of the signal but can be harder to efficiently implement.
- Range of quantization is defined through process called calibration and we express range as $[\alpha, \beta]$ (also called clipping range) where range is between α and β with both ends included. Symmetric quantization means "$-\alpha = \beta$" which is widely adopted since quantization function is simplified by Z=0 and $S = \frac{\beta - \alpha}{2^b - 1}$. Calibration can be carried out by just choosing the minimum and maximum of the signal, but is sensitive to outliers, therefore working with percentiles is more robust. We can also have asymmetric quantization range.
- Dynamic quantization is when clipping range is calculated during runtime for each input which can lower latency due to additional calculations, but can increase accuracy. Static quantization on the other hand is calculated before and while it lowers latency overhead from dynamic quantization it can hurt accuracy.
- Quantization granularity is about level of granularity used to set clipping range
    - layerwise: each convolutional filter range can vary which reduces effectiveness of this simpler approach
    - groupwise: group common channels to keep some of the efficiency of layerwise approach while increasing accuracy(used in Q-BERT (Shen, 2020))
    - layerwise: range is determined for each separate layer(the most popular choice)
    - sub-channel wise: range is determined on group of parameters inside convolution, extreme approach with a lot of overhead but with best accuracy, therefore good technique for benchmarking quantization accuracy

Quantization can be done after training(post-training quantization, PTQ) or during training (quantization-aware training, QAT). Forward and backward passes with QAT are done on a quantized model, but parameters are quantized after the optimizer step to prevent rounding of gradient to zero using Straight Through Estimator(STE). QAT has better accuracy and was found to be able to learn quantization parameters(such as clipping range) very well. PTQ does not require re-training, but can cause reduction of accuracy which can be offset with better quantization statistics for clipping range and improved granularity using techniques such as:

- ACIQ (Banner, 2018) which can be inefficient in practice for deployment

- OSME (Choukroun, 2019) which reduces deployment inefficiency of ACIQ
- adaptive methods such as AdaRound (Nagel, 2020) that restricts difference between quantized and original weights to |1| and AdaQuan (Hubara, 2020)

Zero-shot quantization(ZSQ) is when we apply quantization without knowing anything about the data. If we do some fine tuning after that, where fine tuning is essentially QAT, we decrease accuracy loss but increase latency. There is some interest in using synthetic data for ZSQ by exploring GAN architectures, which may not be representative of real data which motivated usage of BatchNorm statistics to generate better synthetic data such as ZeroQ(Cai, 2020) which leverages mixed-precision quantization using novel Pareto frontier based method.
Simulated(fake) quantization is where parameters are stored in quantized form(4 bit integers), but calculations are carried out in FP32 form, which is a significant issue with accuracy or if we're using integer efficient hardware. HAWQ-V3 (Yao, 2021) solves this with integer only inference and hardware aware mixed-precision quantization approach using integer linear programming problem to find best bit-precision setting which balances the trade-off between efficiency(memory, latency) and model degradation. The method showed promising results on ResNet18, ResNet50 and InceptionV3 architectures. Integer quantization for inference was also tested on transformers and showed good results with some suggestions that integer quantization even improves generalization.


$W^{fp32} = (S_w, W^{i4})^{fp32}$
$a^{fp32} = W^{fp32} h^{fp32}$
$a^{i4} = Int(\frac{(a^{fp32} - \mu^{fp32})/\sigma^{fp32}}{S_a})$
Weights
FP32
Multiply Accumulate
BNFP32 => INT4
Requantization
INT4
X
+
BNFP32 => INT4
INT4
Activations
$h^{fp32} = (S_h, h^{i4})^{fp32}$
(S_w, W^{i4})
$a^{i32} = W^{i4} h^{i4} + b^{i32}$
$a^{i4} = Int(\frac{S_w S_h}{\sigma S_a} a^{i32})$
Weights
INT32
Multiply Accumulate
INT32 -> INT4
Dyadic Rescaling
INT4
X
+
INT4
INT4
Activations
(S_h, h^{i4})
Procure 6: fake/simulated quantization on the left and real quantization on the right (Yao, 2021)

Mixed precision is interesting problem where we have different precision levels for different layers in the network in order to optimize accuracy. It can be formulated as a neural architecture search problem or automatically by using second order density of the model with methods like HAWQ (Dong, 2019) which is faster than the neural architecture search solution mentioned first.
We always want to do some experiments to gather data for empirical analysis and with quantization we can use MQBench (Li, 2021), a framework for quantization benchmarks.

2.2 Knowledge distillation

Knowledge distillation is a method to lower model memory footprint and reduce latency by using the original network called teacher network and a smaller network called student to learn from the teacher network. Distillation techniques can be separated (Gou, 2021) based on how we calculate similarity of student and teacher network:

- response-based knowledge: based on the result(last layer), with common metric being soft targets (Hinton, 2015)
- feature-based knowledge: based on all layers which becomes more important for deeper neural networks with interesting part being how to compare intermediate layers in the common case when student network is shallower than teacher(one interesting theory is cross-layer knowledge distillation, where teacher layers for each student layer are adaptively assigned proper via attention allocation (Chen, 2021))
- relationship-based knowledge further explores the relationship between layers and data

Techniques can be further separated based on different training schemes(Gou, 2021):

- offline: teacher is trained before distillation which makes this method very easy since it has 2 separate phases
- online: 1 phase training when teacher and student learn simultaneously
- self-attention: a special cases of online distillation where the same networks are used for teacher and student where knowledge from the deeper sections of the network is distilled into its shallow sections with some other techniques such as self-attention distillation (Zhang, 2020)(network utilizes the attention maps of its own layers as distillation targets for its lower layers), snapshot distillation (Yang, 2019)(knowledge in the earlier epochs of the network (teacher) is transferred into its later epochs (student) to support a supervised training process within the same network) and many others

Adversarial distillation is an algorithm where GAN generates hard to distill data. There are many more distillation algorithms such as multi teacher, attention or graph based and the reader can reference (Gou, 2021) for more information.

2.3 Pruning

The idea of pruning is to remove parts of the network that are not sufficiently used and by that reduce memory footprint and reduce latency.
Unstructured pruning(Liang, 2021) replaces neurons with zeros in the weight matrix and therefore increases sparsity. Similar to quantization we can split pruning techniques to offline(static) or dynamic and we can further split them based on granularity. Static pruning removes neurons offline after training(but before inference) with optional fine tuning before inference. It offers multiple techniques to define criteria for dropping the weight:

- magnitude based: The basic premise is that weights with larger values are more important. The most basic form is to prune all weights with value 0 and the next step is to do the same for all weights below a certain threshold. Popular approach to choosing what to prune is (Han, 2015) magnitude-based pruning which uses threshold value where we apply pruning layer by layer and threshold is determined by manually setting “quality parameter” which is the multiplied by standard deviation of layer’s weights. This determines the threshold and all weights lower than that are pruned. Model is retrained after pruning is finished.
- penalty based: Goal is to add constraints such as regularization penalty term to the loss function, which pushes some weights to 0 that are pruned.
- dropout: Originally regularization technique to improve generalization is also removing neurons and can be also used as pruning technique.
- redundancies: Idea is to normalize weights, find the similar ones and remove redundancies in effort to minimize accuracy loss compared to magnitude or penalty based pruning.

Dynamic pruning(Liang, 2021) happens at runtime and can use input information to improve pruning. This is conditional pruning where only the optimal part of the network is used and the rest can be considered as pruned. We have 2 approaches to decide on which subgraph can be used:

- reinforcement learning adaptive networks such as cascade network (Odena, 2017)
- differentiable adaptive networks to reduce computational cost such as Dynamic Channel Pruning (Chiliang, 2019)

Structured pruning(Liang, 2021) is about not removing just a single weight, but removing them in groups like entire neurons, filters or channels which reduces or removes sparsity issues from unstructured pruning. The techniques to decide what to remove are similar to unstructured pruning.
While unstructured pruning can’t reduce memory footprint since matrices stay the same, structured pruning can make models more memory efficient.
Structured pruning tends to reduce accuracy and Hessian aware pruning (Yu, 2022), also referred to as HAP, is a method to minimize accuracy loss by improving efficiency of important components rather than removing them. Channels are ordered starting with least sensitive and are pruned in increasing order (of sensitivity) until desired model size(FLOPs) is reached. Relative Hessian trace is used to measure sensitivity(compared to magnitude based method commonly used) and while insensitive components are pruned, sensitive components are replaced with a low rank implant that is more computationally efficient(for example replacing spatial convolution with point-wise convolution or other types of decomposition such as CP/Tucker). Model is fine-tuned after the implant is inserted. This method achieves less than 0.1%/0.5% degradation on PreResNet29/ResNet50 with more than 70%/50% parameters pruned while also showing promise with pruning attention heads from transformer based models by performing better than gradient based methods(Michel, 2019) which indicates that Hessian based sensitivity might be better signal than gradients.

2.4 Libraries

- https://github.com/IntelLabs/distiller
- https://github.com/facebookresearch/bitsandbytes
- https://github.com/neuralmagic/deepsparse
- https://github.com/huggingface/optimum
- https://github.com/apache/tvm (sub int8 quantization)
- https://github.com/nvidia/apex
- https://github.com/openppl-public/ppq
- https://github.com/Zhen-Dong/HAWQ
- https://github.com/yaozhewei/HAP

3 Hardware

Focus will be on Nvidia products due to their wide use in the deep learning community because of popular CUDA and CUDnn libraries along with leadership in the field of GPU architecture.

3.1 GPU

Low precision has been standard in the last years for training and inference. That motivates us to use Volta, Ampere, Tegra and Hopper architecture with tensor cores.

3.2 GPU-to-GPU communication

3.2.1 Inter node communication

We have multiple different solutions:

- Nvidia InfiniBand which can be used anywhere, already built in Nvidia DGX racks.
- AWS EFA is AWS proprietary technology. It's using OS-bypass to bypass TCP/IP stack of EC2 instance to go through Libfabric directly to EFA device (AWS, n.d.) and accelerate TCP communication with scalable reliable datagram (Shalev, 2020) designed for modern data centers with multiple network paths while avoiding congested paths. It sends packets over as many paths as possible without ordering in order to achieve consistent low latency communication.
- NCCL fast socket (Google Cloud, 2021) is Google cloud proprietary technology used to accelerate operations such as all reduce.

3.2.2 Intra node communication

Nvidia NVlink and NVling aggregation(NVswich) are both part of Nvidia product stack that take care of intra-node communication. It's worth mentioning that the newest DGX(DGX H100)(Nvidia, 2020) made the biggest leap here with 7.2 terabytes per second of bidirectional GPU-to-GPU bandwidth, 1.5 times more than previous generation of DGX racks.

3.2.3 Backends

NCCL is from Nvidia, specific for GPUs and does not support CPU communication. It's also the best (Hölzl, 2020) performing GPU backend.
GLOO is from Meta(formerly known as Facebook), the default backend for CPU in Pytorch and doesn't support GPUs. It is however in comparison with MPI the only CPU backend with support for FP16.
MPI is the oldest protocol listed, with beginnings in 1991 (*Message Passing Interface*, n.d.) and supports only FP32 communication on both GPUs and CPUs. Compared to GLOO it has lower latency for smaller tensors and is less sensitive for bigger clusters (Hölzl, 2020).

3.3 CPU-GPU communication

One of the most basic part deep learning workflow, data loading requires loading of data from CPU to GPU if we're using GPUs to train the model. Some software layer optimizations such as pinning memory and increasing workers/thread in Pytorch (Meta, n.d.) or above mentioned DeepNVMe(Rajbhandari, 2021) from Deepspeed can be made but we still hit the hardware limit. This is also especially important for ZeRO CPU-offload (Ren, 2021) algorithms. Nvidia released Grace hopper(Nvidia, 2022) architecture with Nvidia H100 that increases CPU-GPU bandwidth.

4. Application to state of the art architectures

4.1 Transformers

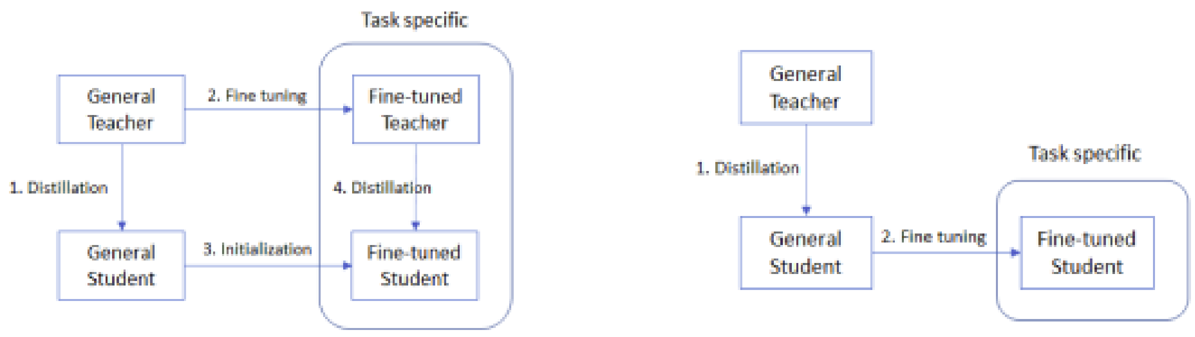
Transformer architecture is known for its 2 stage training schedule (Raffel, 2020):

- pre-training on large and general dataset leveraging unsupervised learning
- fine tuning on small and specific dataset leveraging supervised learning

It has been shown (Raffel, 2020) that accuracy scales with transformer model size which makes training and deployment more challenging, making transformers a great fit for application of theory presented in this paper.

4.1.1 Training

The first optimization we can make is with input where there is already some research(Hou, 2022) on token dropping by keeping only the most important tokens and therefore improving pre-training speed by 25% while not sacrificing accuracy.
Movement pruning (Sahn, 2020) addresses lower performance of magnitude pruning on transfer learning(2nd training stage) of transformers because weights are primarily determined by a general pre-training task and moves from a zero-order to first-order method by not pruning based on absolute values, but movement(change) in transfer learning phase by not keeping non-zero values for example, but values moving away from 0 where we have to note that weights are primarily influenced by pre-training phase. It has a soft version based on threshold for the different and hard version based on choosing top n weights. Compared to second-order based methods involving Hessian (LeCun, 1989) movement pruning is also more computationally efficient. Movement pruning is a deterministic technique and utilizes straight-through estimator. (Bengio, 2013). It was shown (Sahn, 2020) that movement based pruning consistently outperforms method at high sparsity with less than 15% of remaining weights and performing knowledge distillation after movement pruning is finished. Pruning can be also applied to attention heads (Michel, 2019) by masking head(s) then performing forward and backward passes and accumulating absolute gradient which is the importance score, an indication if we should prune attention head based on sorted importance scores. One surprising outcome of that paper was that removing some attention heads even improves BLEU score and that sometimes we can get away with leaving only a single attention head even when starting with 12 or 16 attention heads. It was also empirically shown that reducing attention heads will have the most negative impact in the Encoder-Decoder part of transformer architecture with head importance being determined early in the training. DynaBERT(Hou, 2020) takes this idea further by additionally proposing using width-adaptive BERT, then pruning hidden states of feed-forward layer and rewriting pruned attention modules which introduces depth wise adaptability with a target model size. Adaptive width means adaptive number of attention heads, core component of transformer architecture next to feed forward networks. Importance score is calculated based on variation in the training loss if we remove it. Empirical results have shown DynaBERT has consistently matched performance of the base models(BERT and RoBERTA) with the same or smaller size. Movement pruning was later adapted to block level (Lagunas, 2021) which made the model 2.4 times faster, 74% smaller with only sacrisifing 1% of F1 score.
Knowledge distillation is another technique to lower memory footprint and latency. Due to 2 stage training of transformers we can make distinction (Kim, 2020) between task specific distillation and task agnostic distillation in Picture 6 below. The only time performance issues can be observed is when model types are different.
Self-distillation with no labels has also been researched in the field of computer vision, showing some promise (Caron, 2021).
Transformers are suitable for model parallel training due to natural parallelization of the multi headed attention module.
Another interesting optimization is sparsification among others (Fournier, 2021), especially for attention modules in transformers.



(a) Task specific distillation to general distill models (b) Fine-tuning of general distilled models

Picture 7: task specific vs general quantization scheme (Kim, 2020)

4.1.2. Inference

Some improvements are not about models, but rather about input. Transformer inputs get padded and Bytedance developed (Bytedance, 2020) improvement by dynamically removing padding when possible to reduce computation.

Quantization is common optimization technique which is further developed in Q-BERT(Shen, 2020) model that uses

- group-wise quantization scheme: split matrix in groups with each group having separate quantization range
    - there is trade-off between number of groups: it was empirically shown for BERT in Q-BERT paper that going beyond 128 groups is not efficient due to the performance saturation while larger number of groups are inefficient from hardware point of view because it increases number of Look-up tables necessary for each table multiplication
- Hessian based mixed precision method(Hessian Aware quantization - HAWQ): layer-wide(different layers have different structure and consequently different sensitivity) second order information, due to BERT reacting differently as opposed to computer vision model we used mean and variance of top eigenvalues(the main idea of HAWQ is that layers with higher Hessian spectrum(larger top eigenvalues) are more sensitive to quantization and require higher precision)
    - calculating Hessian spectrum with matrix-free power iteration method (Dong, 2019) is needed in order to keep computation feasible
    - model has to be tuned to local minima in order for HAWQ to be successful

It enables up to 2 bit precision which enables up to 13 times compression of model parameters and up to 4 times compression of embedding table as well as activations by sacrificing at most 2.3% performance degradation.
Q-BERT paper(Shen, 2020) is also looking into which part of Transformer architecture is the most sensitive to quantization based on empirical data:

- embedding layer(10%-20% performance drop, which is very significant)
- position embeddings(2% of additional accuracy loss than word embedding when both are quantized)
- fully connected layer is more sensitive to quantization than self-attention layer(11% vs 7% accuracy drop), but layers don’t have the same architecture therefore we should be cautious with direct comparison

We can also observe some research on combining multiple techniques, like weight pruning, mode distillation and quantization to achieve a compression ratio of 40x with only 1% loss. (Zafrir, 2022)

4.1.3 Performers: alternative architecture that approximates transformer architecture

Performers (Choromanski, 2021) tackle transformer bottleneck(attention matrix has quadratic space and time complexity with respect to input size) by applying fast attention via positive orthogonal random features(FAVOR+) which makes them fully compatible with transformer architecture and brings them linear complexity with strong assumptions: nearly unbiased estimation of attention matrix, uniform convergence and low estimation variance without relying on any priors like sparsity and low rankness. It’s worth mentioning other efficient transformer architectures have been proposed such as Reformer(where dot-product attention is replaced with locality-sensitive hashing going from O(L^2) to O($L \times logL$) where L is length of sequence and replace standard residuals with reversible residual layers which allows storing activations only once rather than N per training process where N is number of layers) where authors show performance stays the same while memory footprint is reduced(Kitaev, 2020) and others (Tay, 2020) (So, 2021).

5. Open questions

5.1 Auto tuning parameters

Parallelism modes have a lot of different parameters such as bucket size in data parallel that have to be empirically tuned. Deepspeed[6] already supports autotuning of some parameters(ZeRO stages, microbatch size and many ZeRO parameters using model information, system information and heuristics) with Pytorch team showing some interest in adding autotuning to Pytorch[7]. There is a lot of space for innovation in automatic tuning of hyperparameters, a great opportunity for future work which interlaps with AutoML. GSPMD (Xu, 2021) is automatic compiler level parallelization optimization where user can annotate tensors with hints and is based on XLA compiler[8], therefore limited to tensorflow as opposed to DeepSpeed which is

[6] https://www.deepspeed.ai/docs/config-json/#autotuning
[7] https://youtu.be/RQfK_ViGzH0?t=537
[8] https://www.tensorflow.org/xla

based on PyTorch. I would be interesting to compare GSPMD with DeepSpeed Autotune and create something better by learning from shortcomings of each approach.

5.2 Improve 3D training

Data parallel increases training speed, while tensor and pipeline parallel enable training larger models. We would also like to train larger models as fast as possible and data parallel helps us to achieve that, hence the motivation for fully sharded data parallel. Megatron ML team started to explore (Narayanan, 2021) interaction between different parallelism modes in order to use them together efficiently and we think this is only the beginning of this complex, but useful question.

5.2 AutoML implications

AutoML methods are tuning hyperparameters and finding optimal architecture based on the loss function of the model. We could add model efficiency(latency and memory footprint) to the loss function to nudge algorithms not only to find best performing, but also most efficient model. Ideally we would also have hyperparameters to tune how important is accuracy versus efficiency for some particular use case. There has also been some work in this area (Wu, 2019).

5.3 Lower precision training

Training with integer level quantization is still unstable(Gholami, 2021), but is promising with larger adoption of edge computing where we can also have the option to train the model, especially for online or real time training. Research could be build upon chunk-based accumulation and floating point stochastic rounding which enables FP8 training having the same accuracy as FP32 training (Wang, 2018).
Tesla recently described proprietary CFloat8 format(Tesla, 2021) where they observe weights and gradients have much smaller range than activation which motivates CFloat8, 8 bit format with configurable mantissa and exponent allocation compared to other standard formats with predefined allocation. This idea could be further explored and adapted for Nvidia GPUs.
Another interesting area of research would be to evaluate a combination of Quantization aware training and parallel training. We failed to find any research papers directly tackling this question.

5.4 Further research in application of distributed training and inference optimization to specific architectures such as transformers

Tools like quantization are theoretical concepts and there is still a lot of room to find customized applications to state of the art architectures such as transformers and graph neural networks. Interesting and newer challenge would be optimizing implants from Hessian aware pruning (Yu, 2022) for transformers. It is worth mentioning Binary neural networks(BNN) which is specific architecture which is already quantized, but we decided to focus on GPUs wherever our estimate is that BNN are interesting for running on FPGA chips. While we focus on transformer architecture, graph neural networks also seem like an interesting and relevant architecture for my research (*Chaitjo/awesome-Efficient-Gnn: Efficient Graph Neural Networks - a Curated List of Papers and Projects*, 2022).

5.5 Engineering challenges

Solving some engineering challenges makes distributed training more accessible. Kubernetes is a popular platform to deploy applications at scale, reliable and cloud agnostic. This makes it great tool for machine learning as can be seen by Kubeflow[9] popularity. Bringin first class support for those algorithms for kubernetes and improving monitoring for multi-gpu environments is a great enabler for more innovation. Deepspeed for now focuses on a more interactive approach, but the team created some tooling to apply it to kubernetes[10] and showed interest[11] in helping the community do the same.

[9] https://www.kubeflow.org/
[10] https://github.com/microsoft/dlworkspace - Jinja based templating of kubernetes yaml manifests
[11] https://github.com/microsoft/DeepSpeed/issues/274

5.6 Further empirical evaluation and tuning focused on transformers

I noticed an opportunity for further empirical testing and tuning that could as well be focused on transformers especially for inference optimization, 3D parallelism modes interaction and distributed optimizers, especially head to head comparison of different papers within one field, for example quantization. It would be interesting to review efficient Transformer architectures Performer and Reformer(where memory bottleneck with attention(QK^T) was addressed) in the light of optimizations presented here. Google recently published (Chowdhery, 2022) combination of their Pathways concept with Transformer, where Pathways architecture enabled them an interesting combination of parallelism(data parallel on TPU pod level along with combination of data and model parallel within), with mixed modes in 2 dimensions (Xu, 2021) which could be further examined in the context of topics presented here.

5.7 Effect on distributed training and inference optimization on scaling laws for large models

It seems like scaling laws for large models are changing (*New Scaling Laws for Large Language Models*, 2022), with shift from “model size is the primary scaling driver” to model size and training data size are equally important. Scaling laws are the answer to questions like: “I have 100 GPU hours, how should I choose the tradeoff between model size and training data size?” where 100 GPU hours mean(1 GPU 100 hours or 10 hours for 10 GPUs). Answer from the OpenAI paper (Kaplan & McCandlish, 2020) was reexamined by the DeepMind team (Hoffmann, 2022) and they concluded that training data is as important as model size, they speculated that different result stems from OpenAI not tuning learning rate for different model sizes and DeepMind training larger models. While this is a tradeoff with the same compute capacity(GPU hours) for training it is a bit different for inference because a smaller model means lesser costs which is an important consideration. It would be interesting to apply some optimization techniques from this paper to better understand the tradeoffs such as if model mixed precision or data parallel affects scaling laws.

References

AWS. (n.d.). *Elastic Fabric Adapter - Amazon Elastic Compute Cloud*. AWS Documentation. Retrieved March 30, 2022, from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html

Banner, r. (2018). Post-training 4-bit quantization of convolution networks for rapid-deployment. *ArXiv*. https://arxiv.org/abs/1810.05723

Bengio, Y. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv*. https://arxiv.org/abs/1308.3432

Brown, T. (2020). Language Models are Few-Shot Learners. *NeurIPS*. https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

Bytedance. (2020). Running BERT without Padding. https://github.com/bytedance/effective_transformer

Cai, Y. (2020). A novel zero shot quantization framework. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, (13169-13178). https://ieeexplore.ieee.org/document/9156720

Caron, M. (2021). Emerging Properties in Self-Supervised Vision Transformers. *ICCV*. https://openaccess.thecvf.com/content/ICCV2021/papers/Caron_Emerging_Properties_in_Self-Supervised_Vision_Transformers_ICCV_2021_paper.pdf

chaitjo/awesome-efficient-gnn: Efficient Graph Neural Networks - a curated list of papers and projects. (2022). GitHub. Retrieved April 1, 2022, from https://github.com/chaitjo/awesome-efficient-gnn

Chen, D. (2021). Cross-Layer Distillation with Semantic Calibration. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(8). https://ojs.aaai.org/index.php/AAAI/article/view/16865

Chiliang, Z. (2019). Accelerating Convolutional Neural Networks with Dynamic Channel Pruning. *Data Compression Conference*, 563–563. https://ieeexplore.ieee.org/document/8712710

Choromanski, K. (2021). Rethinking attention with performers. *ICLR*. https://openreview.net/forum?id=Ua6zuk0WRH

Choukroun, Y. (2019). Low-bit quantization of neural networks for efficient inference. *ICCV Workshop*, 3009-3018.

https://openaccess.thecvf.com/content_ICCVW_2019/papers/CEFRL/Kravchik_Low-bit_Quantization_of_Neural_Networks_for_Efficient_Inference_ICCVW_2019_paper.pdf

Chowdhery, A. (2022). PaLM: Scaling Language Modeling with Pathways. *Google*. https://storage.googleapis.com/pathways-language-model/PaLM-paper.pdf

Dong, Z. (2019). HAWQ: Hessian AWare Quantization of Neural Networks with Mixed-Precision. *ArXiv*. https://arxiv.org/abs/1905.03696

Fournier, Q. (2021). A Practical Survey on Faster and Lighter Transformers. *ArXiv*. https://arxiv.org/abs/2103.14636

Gholami, A. (2021). A Survey of Quantization Methods for Efficient Neural Network Inference.

Google Cloud. (2021, May 19). *How to optimize Google Cloud for deep learning training*. Google Cloud. Retrieved March 30, 2022, from https://cloud.google.com/blog/products/ai-machine-learning/how-to-optimize-google-cloud-for-deep-learning-training

Gou, J. (2021). Knowledge Distillation: A Survey. *International Journal of Computer Vision*, *129*, 1789–1819. https://link.springer.com/article/10.1007/s11263-021-01453-z

Han, S. (2015). Learning both weights and connections for efficient neural networks. *NIPS'15: Proceedings of the 28th International Conference on Neural Information Processing Systems*, *1*, 1135-1143. https://papers.nips.cc/paper/2015/hash/ae0eb3eed39d2bcef4622b2499a05fe6-Abstract.html

Hang, X. (2020). Compressed Communication for Distributed Deep Learning: Survey and Quantitative Evaluation. https://repository.kaust.edu.sa/handle/10754/662495

Hinton, G. (2015). Distilling the knowledge in a neural network. *Arxiv*. https://arxiv.org/abs/1503.02531

Hoffmann, J. (2022). Training Compute-Optimal Large Language Models. *ArXiv*. https://arxiv.org/pdf/2203.15556.pdf

Hölzl, E. (2020, September 8). *Communication Backends, Raw performance benchmarking · MLBench*. MLBench. Retrieved March 30, 2022, from https://mlbench.github.io/2020/09/08/communication-backend-comparison/

Hou, L. (2020). DynaBERT: Dynamic BERT with Adaptive Width and Depth. *NeurIPS*, *33*. https://proceedings.neurips.cc/paper/2020/hash/6f5216f8d89b086c18298e043bfe48ed-Abstract.html

Hou, L. (2022). Token Dropping for Efficient BERT Pretraining. *ArXiv*. https://arxiv.org/pdf/2203.13240.pdf

Huang, Y. (2019). GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *Advances in Neural Information Processing Systems*, *32*, 5998-6008. https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

Hubara, I. (2020). Improving post training neural quantization: Layer-wise calibration and integer programming. *ArXiv*. https://arxiv.org/abs/2006.10518

Kalamkar, D. (2019). A Study of BFLOAT16 for Deep Learning Training. *ArXiv*. https://arxiv.org/abs/1905.12322

Kaplan, J., & McCandlish, S. (2020). Scaling Laws for Neural Language Models. *ArXiv*. https://arxiv.org/pdf/2001.08361v1.pdf

Kim, Y. J. (2020). FastFormers: Highly Efficient Transformer Models for Natural Language Understanding. *ArXiv*. https://arxiv.org/abs/2010.13382

Kingma, D. P. (2015). Adam: A Method for Stochastic Optimization. *International conference on learning representations*. https://arxiv.org/abs/1412.6980

Kitaev, N. (2020). Reformer: The Efficient Transformer. https://arxiv.org/abs/2001.04451v2

Lagunas, F. (2021). Block Pruning For Faster Transformers. *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 10619–10629. https://aclanthology.org/2021.emnlp-main.829

LeCun, Y. (1989). Optimal brain damage. *NIPS*. https://papers.nips.cc/paper/1989/hash/6c9882bbac1c7093bd25041881277658-Abstract.html

Li, M. (2014). Scaling distributed machine learning with the parameter server. *OSDI'14: Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, 583-598. https://dl.acm.org/doi/10.5555/2685048.2685095

Li, S. (2020). PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *International Conference on Very Large Databases (VLDB)*, *13*(12), 3005-3018. https://doi.org/10.14778/3415478.3415530

Li, Y. (2021). MQBench: Towards Reproducible and Deployable Model Quantization Benchmark. *NEURIPS*. https://arxiv.org/abs/2111.03759

Liang, T. (2021). Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing*, *461*(C), 370-403. https://doi.org/10.1016/j.neucom.2021.07.045

Lu, Y. (2022). Maximizing Communication Efficiency for Large-scale Training via 0/1 Adam. *ArXiv*. https://arxiv.org/abs/2202.06009

Message Passing Interface. (n.d.). Wikipedia. Retrieved March 30, 2022, from https://en.wikipedia.org/wiki/Message_Passing_Interface

Meta. (n.d.). *torch.utils.data — PyTorch 1.11.0 documentation*. PyTorch. Retrieved March 30, 2022, from https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

Meta. (2022). *Pipeline Parallelism | FairScale documentation*. FairScale Documentation. Retrieved April 1, 2022, from https://fairscale.readthedocs.io/en/latest/deep_dive/pipeline_parallelism.html

Meta and AWS. (2021, November 4). *Accelerating PyTorch DDP by 10X With PowerSGD*. Medium. Retrieved April 2, 2022, from https://medium.com/pytorch/accelerating-pytorch-ddp-by-10x-with-powersgd-585aef12881d

Michel, P. (2019). Are Sixteen Heads Really Better than One? *NeurIPS*. https://papers.nips.cc/paper/2019/hash/2c601ad9d2ff9bc8b282670cdd54f69f-Abstract.html

Micikevicius, P. (2017). Mixed Precision Training. *ArXiv*. https://arxiv.org/abs/1710.03740

Migacz, S. (2021). *Performance Tuning Guide — PyTorch Tutorials 1.11.0+cu102 documentation*. PyTorch. Retrieved March 30, 2022, from https://pytorch.org/tutorials/recipes/recipes/tuning_guide.html#fuse-pointwise-operations

Nagel, M. (2020). Up or down? adaptive rounding for post-training quantization. *International Conference on Machine Learning*, 7197–7206. https://proceedings.mlr.press/v119/nagel20a.html

Narayanan, D. (2019). PipeDream: generalized pipeline parallelism for DNN training. *SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles*, (27), 1-15. https://dl.acm.org/doi/10.1145/3341301.3359646

Narayanan, D. (2021). Memory-Efficient Pipeline-Parallel DNN Training. *Proceedings of Machine Learning Research*, *139*, 7937-7947. https://proceedings.mlr.press/v139/narayanan21a.html

Narayanan, D. (2021, April 9). Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *ArXiv*. https://arxiv.org/pdf/2104.04473.pdf

New Scaling Laws for Large Language Models. (2022, April 1). LessWrong. Retrieved April 4, 2022, from https://www.lesswrong.com/posts/midXmMb2Xg37F2Kgn/new-scaling-laws-for-large-language-models

Nvidia. (2020, December 3). *DGX Best Practices :: DGX Systems Documentation*. DGX Best Practices :: DGX Systems Documentation. Retrieved March 30, 2022, from https://docs.nvidia.com/dgx/bp-dgx/networking.html

Nvidia. (2022). *NVIDIA Grace CPU and Arm Architecture | NVIDIA*. Nvidia. Retrieved March 30, 2022, from https://www.nvidia.com/en-us/data-center/grace-cpu/

Odena, A. (2017). Changing Model Behavior at Test-Time Using Reinforcement Learning. *International Conference on Learning Representations,*. https://arxiv.org/abs/1702.07780

Patwary, M. (2022). Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *ArXiv*. https://arxiv.org/abs/2201.11990

Pytorch(Meta). (2022). *DDP Communication Hooks — PyTorch 1.11.0 documentation*. PyTorch. Retrieved March 30, 2022, from https://pytorch.org/docs/stable/ddp_comm_hooks.html

Raffel, C. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *JMLR*, *140*. https://jmlr.org/papers/v21/20-074.html

Rajbhandari, S. (2020). ZeRO: memory optimizations toward training trillion parameter models. *SC '20: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1-16. https://dl.acm.org/doi/abs/10.5555/3433701.3433727

Rajbhandari, S. (2021). ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. *SC '21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1-14. https://doi.org/10.5281/zenodo.5156596

Ren, J. (2021). ZeRO-Offload: Democratizing Billion-Scale Model Training. *USENIX Annual Technical Conference*. https://arxiv.org/abs/2101.06840

Sahn, V. (2020). Movement Pruning: Adaptive Sparsity by Fine-Tuning. *NIPS*. https://arxiv.org/abs/2005.07683

Shalev, L. (2020). A cloud-optimized transport protocol for elastic and scalable HPC. *IEEE Micro | Special Issue on Commercial Products 2020*. https://ieeexplore.ieee.org/document/9167399

Shen, S. (2020). Q-BERT: Hessian based ultra low precision quantization of bert. *AAAI*, 8815-8821. https://www.google.com/search?q=Q-BERT%3A+Hessian+based+ultra+low+precision+quantization+of+bert+AAAI&oq=Q-BERT%3A+Hessian+based+ultra+low+precision+quantization+of+bert+AAAI&aqs=chrome..69i57j69i58.2473j0j4&sourceid=chrome&ie=UTF-8

Simon, J. (2021, October 26). *Large Language Models: A New Moore's Law?* Hugging Face. Retrieved March 28, 2022, from https://huggingface.co/blog/large-language-models

So, D. (2021). Primer: Searching for Efficient Transformers for Language Modeling. *ArXiv*. https://arxiv.org/abs/2109.08668

Stosic, D. (2019). *INTRODUCTION TO MIXED PRECISION TRAINING*. NVlabs. Retrieved March 30, 2022, from https://nvlabs.github.io/iccv2019-mixed-precision-tutorial/files/dusan_stosic_intro_to_mixed_precision_training.pdf

Tang, H. (2021). 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. *Proceedings of Machine Learning Research*, *139*. https://proceedings.mlr.press/v139/tang21a

Tay, Y. (2020). Efficient Transformers: A Survey. *ArXiv*. https://arxiv.org/abs/2009.06732

Tesla. (2021). Tesla Dojo whitepaper. https://tesla-cdn.thron.com/static/SBY4B9_tesla-dojo-technology_OPNZ0M.pdf?xseo=&response-content-disposition=inline%3Bfilename%3D%22tesla-dojo-technology.pdf%22

Vogels, T. (2019). PowerSGD: Practical Low-Rank Gradient Compression for Distributed Optimization. *NeurIPS 2019 - Advances in Neural Information Processing Systems*, 14269–14278. https://dl.acm.org/doi/10.5555/3454287.3455565

Wang, N. (2018). Training Deep Neural Networks with 8-bit Floating Point Numbers. *NeurIPS*, *31*. https://proceedings.neurips.cc/paper/2018/hash/335d3d1cd7ef05ec77714a215134914c-Abstract.html

Wang, S., & Kanwar, P. (2019, August 23). *BFloat16: The secret to high performance on Cloud TPUs*. Google Cloud. Retrieved March 30, 2022, from https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

Wu, B. (2019). Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 10734–10742. https://ieeexplore.ieee.org/document/8953587

Xu, Y. (2021). GSPMD: General and Scalable Parallelization for ML Computation Graphs. *ArXiv*. https://arxiv.org/abs/2105.04663

Yang, C. (2019). Snapshot distillation: Teacher-student optimization in one generation. *CVPR*. https://arxiv.org/abs/1812.00123

Yao, Z. (2021). HAWQ-V3: Dyadic Neural Network Quantization. *Proceedings of Machine Learning Research*, *139*(38), 11875-11886. http://proceedings.mlr.press/v139/yao21a.html

You, Y. (2017). Large Batch Training of Convolutional Networks. *ArXiv*. https://arxiv.org/abs/1708.03888

You, Y. (2019). Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. *ICLR 2020 Conference*. https://arxiv.org/abs/1904.00962

Yu, S. (2022). Hessian-Aware Pruning and Optimal Neural Implant. *WACV*. https://openaccess.thecvf.com/content/WACV2022/papers/Yu_Hessian-Aware_Pruning_and_Optimal_Neural_Implant_WACV_2022_paper.pdf

Zafrir, O. (2022). Prune Once for All: Sparse Pre-Trained Language Models. *ArXiv*. https://arxiv.org/pdf/2111.05754.pdf

Zhang, L. (2020). Task-Oriented Feature Distillation. *NeurIPS*. https://papers.nips.cc/paper/2020/file/a96b65a721e561e1e3de768ac819ffbb-Paper.pdf

Zhao, Y. (2022, March 14). *Introducing PyTorch Fully Sharded Data Parallel (FSDP) API*. PyTorch. Retrieved March 30, 2022, from https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/