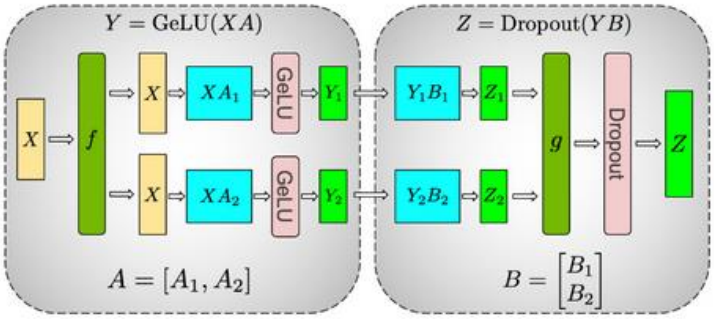
Adventures with BERT and GPT2

BERT and GPT2 are in many ways very similar being networks made mostly of stacks of transformer layers. The basic model has 24 layers of transformers. Each transformer has 16 self-attention heads and a hidden size of 1024. There are some other differences associated with the layer normalization, the details of activation functions and how tokenization of the pretraining data is be done.
In the Tensorflow distribution of BERT from Nvidia (DeepLearningExamples aka DLE, https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/LanguageModeling/BERT) the tokenization is done during the data downloading and preparation. In the Pytorch distributions (Megatron-LM, https://github.com/NVIDIA/Megatron-LM) for both BERT and GPT2 it is done on the first run of pretraining and a second data file is stored to disk.
It should be noted that the source base for DeepLearningExamples was updated after the data shown here was collected, but the authors access to the required hardware was lost before new runs could be made. So, the data (Tensorflow bert on HP 1270) may not reflect the current state of the code.
Of importance here is that the raw files from Wikipedia dumps that are processed in the three cases are all different. For the DLE/TF version of bert, a sharded binary data file is created. For the Pytorch Bert the wikipedia file is first dumped as json records (1 per article) and then a "\n" is added at the end of each sentence with a script that is part of the distribution. For the Pytorch GPT2 code the file just has a single json record/article with nothing beyond the "." between sentences.
The reason for the difference in how the data files are prepared is due the sequence length that the two codes process during pretraining. This is the largest difference in the algorithms, with a large change in memory usage due to the increase in activations that results in GPT2.
There is a lot of confusion about how Bert structures the input sequence. In some distributions the version of Bert creates a sequence of 2 sentences. In such a case, 90% of these are less than 128 tokens in length.
If one looks at the original paper carefully however, what is called a "sentence" is actually a token string that fills approximately half the 512 token input sequence. Thus, two such "sentences", A and B, may actually include multiple sentences. The section below is taken from https://arxiv.org/abs/1810.04805

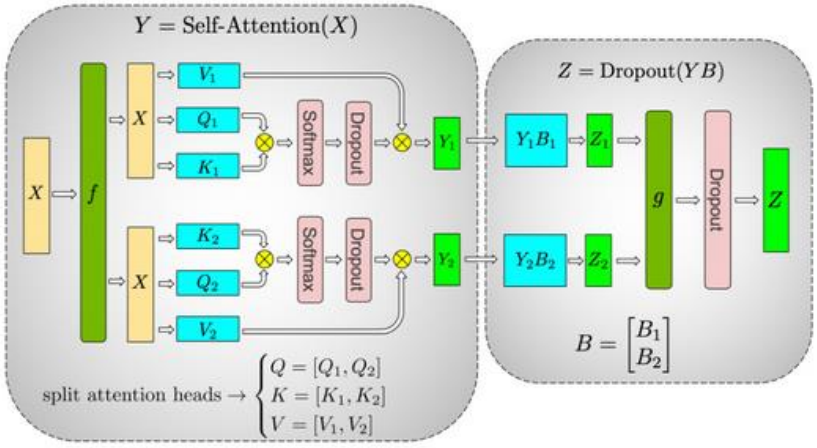
> To generate each training input sequence, we sample two spans of text from the corpus, which we refer to as "sentences" even though they are typically much longer than single sentences (but can be shorter also). The first sentence receives the A embedding and the second receives the B embedding. 50% of the time B is the actual next sentence that follows A and 50% of the time it is a random sentence, which is done for the "next sentence prediction" task. They are sampled such that the combined length is $\leq$ 512 tokens. The

The Megatron-lm code base used here follows the above definition and the 512 token sequence range is filled accordingly. From reading the preprocessing scripts it appears this is also done in the DeepLearningExamples Tensorflow version of Bert.
In the case of GPT2 the input sequence is 1024 tokens. This means the sequence may start or end with a split sentence. The larger sequence length results a lot more activations. To save space and allow a larger minibatch size, this GPT2 code base checkpoints the activations by default. This means that the only activations that are stored during the forward pass are the activations from the last operation of

each layer. The intermediate activations within the layer are then recalculated during the backward propagation. This space savings allows the code to run with a batch size that is 10X larger than if the activation checkpointing is disabled (through a flag).
For the very large versions of Bert that one can construct (ex: 72 layers, 24 attention heads, hidden size of 3204), checkpointing the activations also becomes required.
For the very large models supported by the Megatron-LM code base, model parallelism must also be introduced. The code base supports a new approach for model parallelism of transformer based networks developed by Nvidia (https://nv-adlr.github.io/MegatronLM). Results are presented in the tables below. The Nvidia approach is a per layer model parallel decomposition, with the parallel stream merged at the end of the layer. The use of the Nvidia approach requires the introduction of 2 functions (f and g) to the basic layer structure that is to be run in parallel across GPUs. This is illustrated in the figure below taken from the online documentation (fig 2a/2b https://nv-adlr.github.io/MegatronLM). Basically, the self-attention heads are spread across the parallel processes. The handling of the feed forward operation (FCN) is done by dividing the weight matrix horizontally, minimizing communications.



(a) MLP



(b) Self attention

Figure 2: (a): MLP and (b): self attention blocks of transformer. ***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 forward and ***identity*** in backward.

This decomposition requires a few all-reduce operations as illustrated in the figure below.

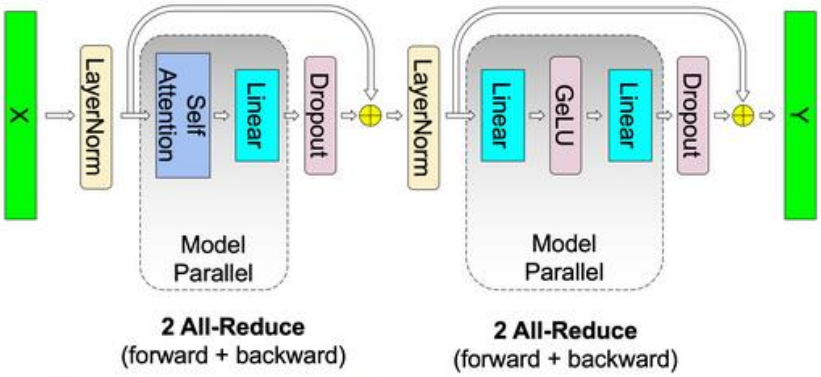


Figure 3: Model parallelism for a GPT-2 transformer layer.

The result shows very acceptable scaling, and perhaps more importantly provides a straightforward mechanism to handle very large models with very long input sequences. (https://nv-adlr.github.io/MegatronLM)

HPE 1270 8 X 32GB PCIe V100 measurements

Initially the codes were run on a HP1270 with 8 32GB PCIe V100s, using the 19.07 NGC image with Pytorch 19.05 for the Megatron-LM code base. This yielded the following results. These were all based on 24 layer models with 16 attention heads and hidden size of 1024. Thus, both models had approximately 340 million parameters. There were issues with the newer versions of Pytorch at the time I downloaded the code base. These have since been addressed.

HPE 1270 8 X 32GB PCIe V100s				
DeepLearningExamples TF Bert				
num_gpu	batch size	Throughput		
8	10	83.2		
Megatron-LM Pytorch Bert_distrib				
num_gpu	batch size	time/iteration (ms)	samples/sec	
8	14	832	134.62	
Megatron-LM Pytorch GPT2_distrib				
num_gpu	batch size	time/iteration (ms)	samples/sec	Checkpnt_activations
8	24	3470	55.33	yes
8	30	4236	56.66	yes
8	32	4526	56.56	yes
8	3	655	36.64	no

The increase in throughput from the larger batch size more than makes up for the recalculations required for the activation checkpointing.

DGX2 (16 32GB V100s with NVLink interconnect) measurements

GPT2:

At this point the experiments moved to a DGX2 machine with 16 32GB V100s interconnected with NVLink. This enabled exploration of models with up to 8.4 billion parameters and run across 16 nodes. As the model sizes were changed an exploration of batch size had to be made along with the number of model parallel processes. In other words, the 16 nodes were broken into a product of model parallel X data parallel.

First, some initial measurements simply increasing the model size and lowering the batch size and using the activation checkpointing algorithm to accommodate the finite memory of the GPUs. Thus no model parallelism is invoked.

This resulted in:

GPT2					
DGX2	data parallel				
16 heads, 24 layers, hidden=1024				cached	model size
num_gpu	batch size	time/iteration (ms)	samples/sec	29.9	354871296
8	32	3794	67.47	29.9	354871296
16	32	3842	133.26		
16 heads, 40 layers, hidden=1536				cached	model size
16	24	OOM			
16	8	2533	50.53	31.3	1212103680
20 heads, 54 layers, hidden=1920				cached	model size
16	2	OOM			2488688640

The scaling from 8 to 16 GPUs = 133.26/(2*67.47) = 98.8%, showing the efficacy of the NVlink. The throughput increase for 8 gpus between the PCIe connected devices and the NVlink connected devices can be evaluated for the GPT2 run with batch size 32 equal to 67.47/56.56, for a gain of 19.3%.
The cached size is reported in stdout and reported here in GBs, making clear why the larger batch sizes run out of memory.

There were then a series of runs take with 16 GPUs and a fixed batch size of 8 and varying the model size (layers and hidden size), the number of attention heads(which does not change the model size) and the width (rank) of the model parallelism.

layers	hidden size					
54	1920					
Attention heads	time/iteration (ms)	samples/sec	model size	rank	size/rank	cached
20	2697	23.73	2488688640	2	1244344320	31.2
layers	hidden size					
64	2304					
Attention heads	time/iteration (ms)	samples/sec	model size	rank	size/rank	cached
4	2226	14.38	4207675392	4	1051918848	27.6
24	2569	12.46	4207675392	4	1051918848	28.7
layers	hidden size					
72	3072					
Attention heads	time/iteration (ms)	samples/sec	model size	rank	size/rank	cached
32	2672	5.99	8348393472	8	1043549184	27.2
24	2603	6.15	8348393472	8	1043549184	27.2
16	2538	6.30	8348393472	8	1043549184	27.2
8	2412	6.63	8348393472	8	1043549184	27.2

Using the model parallel algorithms enables running a GPT2 model with 8.3 billion parameters. The batch size at the largest 2 models could probably be increased given the cached sizes of 27 to 28.7 GBs. Some performance increase is seen with decreasing the number of heads. The rate of samples/sec drops approximately linearly with the model size. This is illustrated more clearly in the table below starting with the smallest model (24 layers, 1024 hidden size) where the normalized values are all ~ 1 over a range of over 20 in model size.

samples/sec	model size	model_size*samples/sec	normalized to smallest model
133.263925	354871296	47291541789	1
50.53296486	1212103680	61251192673	1.295182825
23.73007045	2488688640	59056756752	1.248780533
12.45620864	4207675392	52411682578	1.10826758
5.988023952	8348393472	49990380072	1.057068097

A similar set of runs with the largest model were then done invoking the TORCH_DDP algorithm. This simply required changing the value of USE_TORCH_DDP to True in the pretrain_gpt2.py script. Thus 16 GPUs were used, the rank was fixed at 8 as was the batch size. The results are very similar to the Nvidia DDP and shown below. I learned later that this option addresses data parallelism in multi machine execution. Thus in these single machine runs no change should have been expected and really none is seen.

layers	hidden size					
72	3072					
Attention heads	time/iteration (ms)	samples/sec	model size	rank	size/rank	cached
32	2674	5.98	8348393472	8	1043549184	28.2
24	2581	6.20	8348393472	8	1043549184	28.2
16	2489	6.43	8348393472	8	1043549184	28.2
8	2414	6.63	8348393472	8	1043549184	28.3

BERT (Very Large):

The code base supports adjusting these parameters with Bert based Pytorch models as well. What follows are data collected running the bert encoding within Megatron-LM. This allows the exploration of significantly larger models that still use the 512 token input sequence. The input data set directory must be swapped so that the Megatron-LM/data/wikipedia tree holds the Bert formatted Wikipedia data. This way no changes to the code base (Megatron-LM/data_util/corpora.py) are required.
Running a single rank (no DDP) with no activation checkpointing rapidly runs out of memory as the model size is increased.

Bert data parallel (16 heads)							
batch size	time/iteration (ms)	samples/sec	model size	rank	cached	layers	hidden size
14	541	414.05	336297858	1	29.1	24	1024
2	476	67.23	1185801090	1	30	40	1536

The throughput rate of 414 samples/sec on 16 GPUs in the DGX2 can be compared to the 135 samples/sec observed with the 8 PCIe based GPUs. It is clear the NVLink interconnect greatly increases the scaling and thereby the throughput of the Bert standard large model (16 heads, 24 layers, 1024 hidden size).

The experiments then moved to adding Nvidia DDP to the runs, but not activation checkpointing. The result was that only very small batch sizes could be supported within the memory constraint. The results shown below illustrate the tradeoff between increasing the rank to allow larger batch size and the change in throughput that results. The table is a bit confusing as both the size and rank are changed to see what can be made to fit and run. Clearly some of the batch sizes could have been increased further after increasing the rank.

bert model parallel (nvidia model parallel, no activation checkpnt)								
layers	hidden size	Attention heads	batch size	time/iteration (ms)	samples/sec	model size	rank	cached
24	1024	16	30	667	359.82	339208196	2	29.8
40	1536	16	8	640	100.00	1191886852	2	27.4
54	1920	20	2	600	26.67	2466520324	2	31.3
54	1920	20	8	798	40.10	2484514568	4	28.2
64	2304	24	2	625	12.80	4199522312	4	28.8
64	2304	24	8	871	18.37	4250339344	8	25.1
72	3072	32	2	713	5.61	8424118288	8	28.7

Finally, the activation checkpointing is enabled on the largest of these bert models.
The impact of the activation checkpointing on increasing the batch size and with that the throughput is dramatic, going from 5.6 samples/sec to over 13. As less that 30GBs was cached it would be reasonable to suspect that an even larger batch size could be supported.

bert model parallel (nvidia ddp with activation checkpnt)				
layers	hidden size	Attention heads	rank	
72	3072	32	8	
batch size	time/iteration (ms)	samples/sec	cached	model size
2	1059	3.78	23.5	8424118288
16	2788	11.48	27.2	8424118288
32	4754	13.46	29.8	8424118288