Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

Anonymous Authors[1]

Abstract

Recent work in language modeling demonstrates that training large transformer models advances the state of the art in Natural Language Processing applications. However, very large models can be quite difficult to train due to memory constraints. In this work, we present our techniques for training very large transformer models and implement a simple, efficient intra-layer model parallel approach that enables training transformer models with billions of parameters. Our approach does not require a new compiler or library changes, is orthogonal and complimentary to pipeline model parallelism, and can be fully implemented with the insertion of a few communication operations in native PyTorch. We illustrate this approach by converging transformer based models up to 8.3 billion parameters using 512 GPUs. We sustain 15.1 PetaFLOPs across the entire application with 76% scaling efficiency when compared to a strong single GPU baseline that sustains 39 TeraFLOPs, which is 30% of peak FLOPs. To demonstrate that large language models can further advance the state of the art (SOTA), we train an 8.3 billion parameter transformer language model similar to GPT-2 and a 3.9 billion parameter model similar to BERT. We show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased performance as the model size grows. Using the GPT-2 model we achieve SOTA results on the WikiText103 (10.8 compared to SOTA perplexity of 15.8) and LAMBADA (66.5% compared to SOTA accuracy of 63.2%) datasets. Our BERT model achieves SOTA results on the RACE dataset (88.8% compared to single model SOTA accuracy of 86.5%). We release our code and the weights of our models for reproducibility.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Natural Language Processing (NLP) is advancing quickly in part due to an increase in available compute and dataset size. The abundance of compute and data enables training increasingly larger language models via unsupervised pretraining (Devlin et al., 2018; Radford et al., 2019). Empirical evidence indicates that larger language models are dramatically more useful for NLP tasks such as article completion, question answering, and natural language inference (Lan et al., 2019; Raffel et al., 2019). By finetuning these pretrained language models on downstream natural language tasks, one can achieve state of the art results as shown in recent work (Devlin et al., 2018; Peters et al., 2018; Howard & Ruder, 2018; Radford et al., 2018; 2017; Ramachandran et al., 2016; Liu et al., 2019b; Dai et al., 2019; Yang et al., 2019; Liu et al., 2019a; Lan et al., 2019).

As these models become larger, they exceed the memory limit of modern processors, and require additional memory management techniques such as activation checkpointing (Chen et al., 2016). Widely used optimization algorithms such as ADAM require additional memory per parameter to store momentum and other optimizer state, which reduces the size of models that can be effectively trained. Several approaches to model parallelism overcome this limit by partitioning the model such that the weights and their associated optimizer state do not need to reside concurrently on the processor. For example, GPipe (Huang et al., 2018) and Mesh-Tensorflow (Shazeer et al., 2018) provide frameworks for model parallelism of different kinds. However, they require rewriting the model, and rely on custom compilers and frameworks that are still under development.

In this work, we implement a simple and efficient model parallel approach using intra-layer model-parallelism. We exploit the inherent structure in transformer based language models to make a simple model-parallel implementation that trains efficiently in PyTorch, with no custom C++ code or compiler required. This approach is orthogonal to pipeline-based model parallelism as advocated by approaches such as GPipe (Huang et al., 2018).

To demonstrate the scalability of our approach, we establish a baseline by training a model of 1.2 billion parameters

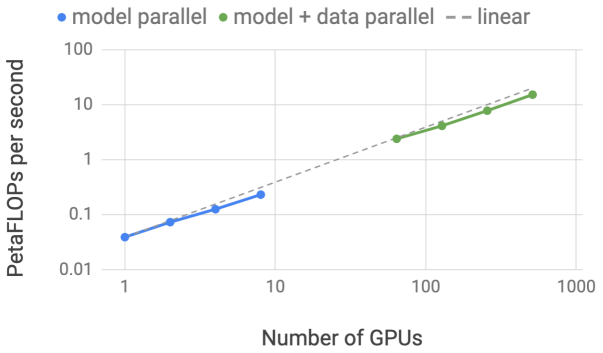


Figure 1. Model (blue) and model+data (green) parallel FLOPS as a function of number of GPUs. Model parallel (blue): up to 8-way model parallel weak scaling with approximately 1 billion parameters per GPU (e.g. 2 billion for 2 GPUs and 4 billion for 4 GPUs). Model+data parallel (green): similar configuration as model parallel combined with 64-way data parallel.

on a single NVIDIA V100 32GB GPU, that sustains 39 TeraFLOPs. This is 30% of the theoretical peak FLOPS for a single GPU as configured in a DGX-2H server, and is thus a strong baseline. Scaling the model to 8.3 billion parameters on 512 GPUs with 8-way model parallelism, we achieve up to 15.1 PetaFLOPs per second sustained over the entire application. This is 76% scaling efficiency compared to the single GPU case. Figure 1 shows more detailed scaling results.

To analyze the effect of model size scaling on accuracy, we train both left-to-right GPT-2 (Radford et al., 2019) language models as well as BERT (Devlin et al., 2018) bidirectional transformers and evaluate them on several downstream tasks. We show that the existing BERT architecture results in model degradation as the size increases. We overcome this challenge by rearranging the layer normalization and residual connection in the transformer layers and show that with this change, results for the downstream tasks on development sets improve monotonically as the model size increases. In addition, we show that our models achieve test set state of the art (SOTA) results on WikiText103, cloze-style prediction accuracy on LAMBADA, and reading comprehension RACE datasets.

In summary, our contributions are as follows:

- We implement a simple and efficient model parallel approach by making only a few targeted modifications to an existing PyTorch transformer implementation.
- We perform an in-depth empirical analysis of our model and data parallel technique and demonstrate up to 76% scaling efficiency using 512 GPUs.
- We show that careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased accuracies as the model grows.
- We demonstrate that scaling the model size results in improve accuracies for both GPT-2 (studied up to 8.3 billion parameters) and BERT (studied up to 3.9B parameters) models.
- We showcase that our models achieve state of the art results on test sets: perplexity on WikiText103 (10.8 ppl), accuracy on LAMBADA (66.5%), and accuracy on RACE (88.8%).
- We open source our code along with the training and evaluation pipelines at `https://github.com/ANONYMIZED-URL`

2. Background and Challenges

2.1. Neural Language Model Pretraining

Pretrained language models have become an indispensable part of NLP researchers' toolkits. Leveraging large corpus pretraining to learn robust neural representations of language is an active area of research that has spanned the past decade. Early examples of pretraining and transferring neural representations of language demonstrated that pretrained word embedding tables improve downstream task results compared to word embedding tables learned from scratch (Mikolov et al., 2013; Pennington et al., 2014; Turian et al., 2010). Later work advanced research in this area by learning and transferring neural models that capture contextual representations of words (Melamud et al., 2016; McCann et al., 2017; Peters et al., 2018; Radford et al., 2017; 2019). Recent parallel work (Ramachandran et al., 2016; Howard & Ruder, 2018; Radford et al., 2018; Devlin et al., 2018; Liu et al., 2019b; Dai et al., 2019; Yang et al., 2019; Liu et al., 2019a; Lan et al., 2019) further builds upon these ideas by not just transferring the language model to extract contextual word representations, but by also finetuning the language model in an end to end fashion on downstream tasks. Through these works, the state of the art has advanced from transferring just word embedding tables to transferring entire multi-billion parameter language models. This progression of methods has necessitated the need for hardware, systems techniques, and frameworks that are able to operate efficiently at scale and satisfy increasing computational needs. Our work aims to provide the tools necessary to take another step forward in this trend.

2.2. Transformer Language Models and Multi-Head Attention

Current work in NLP trends towards using *transformer* models (Vaswani et al., 2017) due to their superior accuracy and compute efficiency. The original transformer formula-

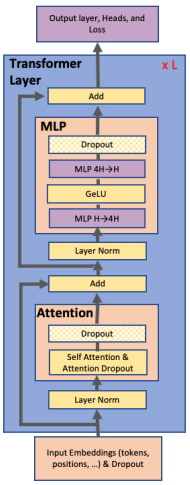


Figure 2. Transformer Architecture. Purple blocks correspond to fully connected layers. Each blue block represents a single transformer layer that is replicated N times.

tion was designed as a machine translation architecture that transforms an input sequence into another output sequence using two parts, an *Encoder* and *Decoder*. However, recent work leveraging transformers for language modeling such as BERT (Devlin et al., 2018) and GPT-2 (Radford et al., 2019) use only the *Encoder* or *Decoder* depending on their needs. This work explores both a decoder architecture, GPT-2, and an encoder architecture, BERT.

Figure 2 shows a schematic diagram of the model we used. We refer the reader to prior work for a detailed description of the model architecture (Vaswani et al., 2017; Devlin et al., 2018; Radford et al., 2019). It is worthwhile to mention that both GPT-2 and BERT use GeLU (Hendrycks & Gimpel, 2016) nonlinearities and layer normalization (Ba et al., 2016) to the input of the multi-head attention and feed forward layers, whereas the original transformer (Vaswani et al., 2017) uses ReLU nonlinearities and applies layer normalization to outputs.

2.3. Data and Model Parallelism in Deep Learning

There are two central paradigms for scaling out deep neural network training to numerous hardware accelerators: data parallelism (Valiant, 1990) where a training minibatch is split across multiple workers, and model parallelism in which the memory usage and computation of a model is distributed across multiple workers. By increasing the minibatch size proportionally to the number of available workers (i.e. *weak scaling*), one observes near linear scaling in training data throughput. However, large batch training introduces complications into the optimization process that can result in reduced accuracy or longer time to convergence, offsetting the benefit of increased training throughput (Keskar et al., 2017). Further research (Goyal et al., 2017; You et al., 2017; 2019) has developed techniques to mitigate these effects and drive down the training time of large neural networks. To scale out training even further, parallel work (Chen et al., 2016) has combined data parallelism with activation checkpointing: recomputing activations in the backward pass without storing them in the forward pass to reduce memory requirements.

However, these techniques have one fundamental limitation in the problem size they can tackle: the model must fit entirely on one worker. With language models of increasing size and complexity like BERT and GPT-2, neural networks have approached the memory capacity of modern hardware accelerators. One solution to this problem is to employ parameter sharing to reduce the memory footprint of the model (Lan et al., 2019), but this limits the overall capacity of the model. Our approach is to utilize model parallelism to split the model across multiple accelerators. This not only alleviates the memory pressure, but also increases the amount of parallelism independently of the microbatch size.

Within model parallelism, there are two further paradigms: layer-wise pipeline parallelism, and more general distributed tensor computation. In pipeline model parallelism, groups of operations are performed on one device before the outputs are passed to the next device in the pipeline where a different group of operations are performed. Some approaches (Harlap et al., 2018; Chen et al., 2018) use a parameter server (Li et al., 2014) in conjunction with pipeline parallelism. However these suffer from inconsistency issues. The GPipe framework for TensorFlow (Huang et al., 2018) overcomes this inconsistency issue by using synchronous gradient decent. This approach requires additional logic to handle the efficient pipelining of these communication and computation operations, and suffers from pipeline bubbles that reduce efficiency, or changes to the optimizer itself which impact accuracy.

Distributed tensor computation is an orthogonal and more general approach that partitions a tensor operation across multiple devices to accelerate computation or increase model size. FlexFlow (Jia et al., 2018), a deep learning framework orchestrating such parallel computation, provides a method to pick the best parallelization strategy. Recently, Mesh-TensorFlow (Shazeer et al., 2018) introduced a language for specifying a general class of distributed tensor computations in TensorFlow (Abadi et al., 2015). The parallel dimensions are specified in the language by the end user and the resulting graph is compiled with proper collective primitives. We utilize similar insights to those leveraged in Mesh-TensorFlow and exploit parallelism in computing the transformer's attention heads to parallelize our transformer model. However, rather than implementing a framework and compiler for model parallelism, we make only a few targeted modifications to existing PyTorch transformer implementations. Our approach is simple, does not

require any new compiler or code re-writing, and can be fully implemented by inserting a few simple primitives, as described in the next section.

3. Model Parallel Transformers

We take advantage of the structure of transformer networks to create a simple model parallel implementation by adding a few synchronization primitives. A transformer layer consists of a self attention block followed by a two-layer, multi-layer perceptron (MLP) as shown in Figure 2. We introduce model parallelism in both of these blocks separately.

We start by detailing the MLP block. The first part of the block is a GEMM followed by a GeLU nonlinearity:

$$Y = \text{GeLU}(XA) \tag{1}$$

One option to parallelize the GEMM is to split the weight matrix A along its rows and input X along its columns as:

$$X = [X_1, X_2],\ A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}. \tag{2}$$

This partitioning will result in $Y = \text{GeLU}(X_1A_1 + X_2A_2)$. Since GeLU is a nonlinear function, $\text{GeLU}(X_1A_1 + X_2A_2) \neq \text{GeLU}(X_1A_1) + \text{GeLU}(X_2A_2)$ and this approach will require a synchronization point before the GeLU function.

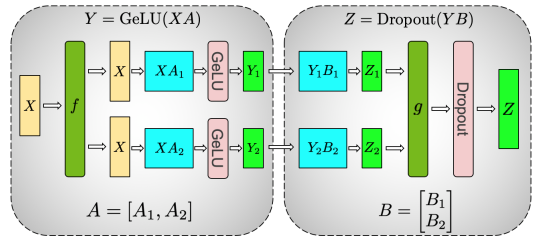
Another option is to split A along its columns $A = [A_1, A_2]$. This partitioning allows the GeLU nonlinearity to be independently applied to the output of each partitioned GEMM:

$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)] \tag{3}$$

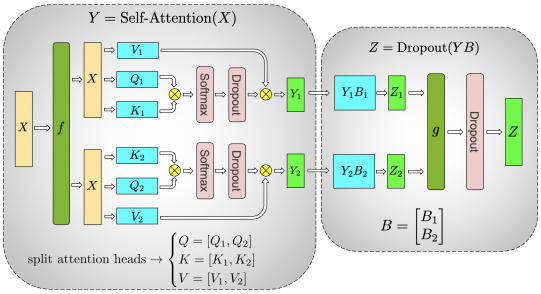
This is advantageous as it removes a synchronization point. Hence, we partition the first GEMM in this column parallel fashion and split the second GEMM along its rows so it takes the output of the GeLU layer directly without requiring any communication as shown in Figure 3a. The output of the second GEMM is then reduced across the GPUs before passing the output to the dropout layer. This approach splits both GEMMs in the MLP block across GPUs and requires only a single all-reduce operation in the forward pass (g operator) and a single all-reduce in the backward pass (f operator). These two operators are conjugates of each other and can be implemented in PyTorch with only a few lines of code. As an example, the implementation of the f operator is provided below:

```
class f(torch.autograd.Function):
    def forward(ctx, x):
        return x
    def backward(ctx, gradient):
        all_reduce(gradient)
        return gradient
```

Code 1. Implementation of f operator. g is similar to f with identity in the backward and all-reduce in the forward functions.



(a) MLP



(b) Self-Attention

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

As shown in Figure 3b, for the self attention block we exploit inherent parallelism in the multihead attention operation, partitioning the GEMMs associated with key (K), query (Q), and value (V) in a column parallel fashion such that the matrix multiply corresponding to each attention head is done locally on one GPU. This allows us to split per attention head parameters and workload across the GPUs, and doesnt require any immediate communication to complete the self-attention. The subsequent GEMM from the output linear layer (after self attention) is parallelized along its rows and takes the output of the parallel attention layer directly, without requiring communication between the GPUs. This approach for both the MLP and self attention layer fuses groups of two GEMMs, eliminates a synchronization point in between, and results in better scaling. This enables us to perform all GEMMs in a simple transformer layer using only two all-reduces in the forward path and two in the backward path (see Figure 4).

The transformer language model has an output embedding with the dimension of hidden-size (H) times vocabulary-size (v). Since the vocabulary size is on the order of tens of thousands of tokens for modern language models (for example, GPT-2 used a vocabulary size of 50,257), it is beneficial to parallelize the output embedding GEMM. However, in transformer language models, the output embedding layer shares weights with the input embedding, requiring modifications to both. We parallelize the input embedding weight matrix $E_{H\times v}$ along the vocabulary dimension $E = [E_1, E_2]$ (column-wise). Since each partition now only

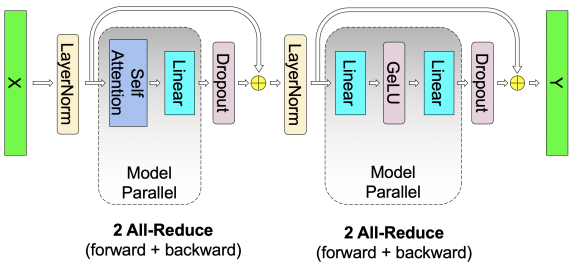


Figure 4. Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

contains a portion of the embedding table, an all-reduce (g operator) is required after the input embedding. For the output embedding, one approach is to perform the parallel GEMM $[Y_1, Y_2] = [XE_1, XE_2]$ to obtain the logits, add an all-gather $Y = \text{all-gather}([Y_1, Y_2])$, and send the results to the cross-entropy loss function. However, for this case, the all-gather will communicate $b \times s \times v$ elements (b is the batch-size and s is the sequence length) which is huge due to vocabulary size being large. To reduce the communication size, we fuse the output of the parallel GEMM $[Y_1, Y_2]$ with the cross entropy loss which reduces the dimension to $b \times s$. Communicating scalar losses instead of logits is a huge reduction in communication that improves the efficiency of our model parallel approach.

Much of our model parallel approach can be characterized as techniques aimed at reducing communication and keeping the GPUs compute bound. Rather than having one GPU compute part of the dropout, layer normalization, or residual connections and broadcast the results to other GPUs, we choose to duplicate the computation across GPUs. Specifically, we maintain duplicate copies of layer normalization parameters on each GPU, and take the output of the model parallel region and run dropout and residual connection on these tensors before feeding them as input to the next model parallel regions. To optimize the model we allow each model parallel worker to optimize its own set of parameters. Since all values are either local to or duplicated on a GPU, there is no need for communicating updated parameter values in this formulation.

We present further details about the hybrid model and data parallelism and handling random number generation in the supplementary material for reference. In summary, our approach as described above is simple to implement, requiring only a few extra all-reduce operations added to the forward and backward pass. It does not require a compiler, and is orthogonal and complementary to the pipeline model parallelism advocated by approaches such as (Huang et al., 2018).

4. Setup

Pretrained language understanding models are central task in natural language processing and language understanding. There are several formulations of language modeling. In this work we focus on GPT-2 (Radford et al., 2019), a left-to-right generative transformer based language model, and BERT (Devlin et al., 2018), a bi-directional transformer model based on language model masking. We explain our configurations for these models in the following section and refer to the original papers for more details.

4.1. Training Dataset

To collect a large diverse training set with longterm dependencies we aggregate several of the largest language modeling datasets. We create an aggregate dataset consisting of Wikipedia (Devlin et al., 2018), CC-Stories (Trinh & Le, 2018), RealNews (Zellers et al., 2019), and OpenWebtext (Radford et al., 2019). To avoid training set leakage into our downstream tasks we remove the Wikipedia articles present in the WikiText103 test set (Merity et al., 2016). We also remove unnecessary newlines from the CC-Stories corpus introduced by preprocessing artifacts. For BERT models we include BooksCorpus (Zhu et al., 2015) in the training dataset, however, this dataset is excluded for GPT-2 trainings as it overlaps with LAMBADA task.

We combined all the datasets and then filtered out all the documents with content length less than 128 tokens from the aggregated dataset. Since similar content might appear multiple times in the aggregated datasets, we used locality-sensitive hashing (LSH) to deduplicate content with a jaccard similarity greater than 0.7. The resulting aggregate corpus contains 174 GB of deduplicated text.

4.2. Training Optimization and Hyperparameters

To train our models efficiently we utilize mixed precision training with dynamic loss scaling to take advantage of the V100's Tensor Cores (Micikevicius et al., 2017; NVIDIA, 2018). We start by initializing our weights W with a simple normal distribution $W \sim \mathcal{N}(0, 0.02)$. We then scale weights immediately before residual layers by $\frac{1}{\sqrt{2N}}$ where N is the number of transformer layers comprised of self attention and MLP blocks. For our optimizer we utilize Adam (Kingma & Ba, 2014) with weight decay (Loshchilov & Hutter, 2019) $\lambda = 0.01$. Additionally, we use global gradient norm clipping of 1.0 to improve the stability of training large models. In all cases, a dropout of 0.1 is used. Lastly, to better manage our memory footprint we utilize activation checkpointing (Chen et al., 2016) after every transformer layer.

For GPT-2 models, all training is performed with sequences of 1024 subword units at a batch size of 512 for 300k itera-

tions. Our learning rate of 1.5e-4 utilizes a warmup period of 3k iterations before following a single cycle cosine decay over the remaining 297k iterations. We stop the decay at a minimum learning rate of 1e-5.

For BERT models, we largely follow the training process described in (Lan et al., 2019). We use the original BERT dictionary with vocab size of 30,522. In addition, we replace the next sentence prediction head with sentence order prediction as suggested by (Lan et al., 2019) and use whole word n-gram masking of (Joshi et al., 2019). For all cases, we set the batch size to 1024 and use a learning rate of 1.0e-4 warmed up over 10,000 iterations and decayed linearly over 2 million iterations. Other training parameters are kept the same as (Devlin et al., 2018).

5. Experiments

All of our experiments use up to up to 32 DGX-2H servers (a total of 512 Tesla V100 SXM3 32GB GPUs). Our infrastructure is optimized for multi-node deep learning applications, with 300 GB/sec bandwidth between GPUs inside a server via NVSwitch and 100 GB/sec of interconnect bandwidth between servers using 8 InfiniBand adapters per server.

5.1. Scaling Analysis

To test the scalability of our implementation, we consider GPT-2 models with four sets of parameters detailed in Table 1. To have consistent GEMM sizes in the self attention layer, the hidden size per attention head is kept constant at 96 while the number of heads and layers are varied to obtain configurations ranging from 1 billion to 8 billion parameters. The configuration with 1.2 billion parameters fits on a single GPU whereas the 8 billion parameter model requires 8-way model parallelism (8 GPUs). The original vocabulary size was 50,257, however, to have efficient GEMMs for the logit layer, it is beneficial for the per-GPU vocabulary size to be a multiple of 128. Since we study up to 8-way model parallelism, we pad the vocabulary such that it is divisible by $128 \times 8 = 1024$, resulting in a padded vocabulary size of 51,200. We study both model and model+data parallel scaling. For the model parallel scaling, a fixed batch size of 8 is used across all configurations. Data parallel scaling is necessary for training many state of the art models which typically use a much larger global batch size. To this end, for the model+data parallel cases we fix the global batch size to 512 for all experiments which corresponds to 64-way data parallelism.

5.1.1. Model and Data Parallelism

Throughout this section, we will showcase weak scaling with respect to the model parameters for both model parallel and model+data parallel cases. Weak scaling is typically done by scaling the batch-size, however, this approach does not address training large models that do not fit on a single GPU and it leads to training convergence degradation for large batch sizes. In contrast, here we use weak scaling to train larger models that were not possible otherwise. The baseline for all the scaling numbers is the first configuration (1.2 billion parameters) in Table 1 running on a single GPU. This is a strong baseline as it achieves 39 TeraFLOPS during the overall training process, which is 30% of the theoretical peak FLOPS for a single GPU in a DGX-2H server.

Table 1. Parameters used for scaling studies. Hidden size per attention head is kept constant at 96.

Hidden Size	Attention heads	Number of layers	Number of parameters (billions)	Model parallel GPUs	Model +data parallel GPUs
1536	16	40	1.2	1	64
1920	20	54	2.5	2	128
2304	24	64	4.2	4	256
3072	32	72	8.3	8	512

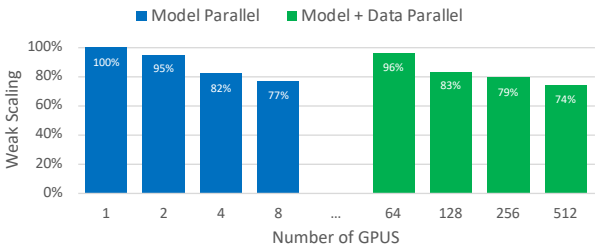


Figure 5. Model and model + data parallel weak scaling efficiency as a function of the number of GPUs.

Figure 5 shows scaling values for both model and model+data parallelism. We observe excellent scaling numbers in both settings. For example, the 8.3 billion parameters case with 8-way (8 GPU) model parallelism achieves 77% of linear scaling. Model+data parallelism requires further communication of gradients and as a result the scaling numbers drop slightly. However, even for the largest configuration (8.3 billion parameters) running on 512 GPUs, we achieve 74% scaling relative to linear scaling of the strong single GPU baseline configuration (1.2 billion parameters).

5.2. Language Modeling Results Using GPT-2

To demonstrate that large language models can further advance the state of the art, we consider training GPT-2 models of the sizes and configurations listed in Table 2. The 355M model is equivalent in size and configuration of BERT-Large model (Devlin et al., 2018). The 2.5B model is bigger than the previous largest GPT-2 model, and the 8.3B model is larger than any left-to-right transformer language model ever trained, to the best of our knowledge. To train and evaluate our language models we use the procedure described in

Table 2. Model configurations used for GPT-2.

Parameter Count	Layers	Hidden Size	Attn Heads	Hidden Size per Head	Total GPUs	Time per Epoch (days)
355M	24	1024	16	64	64	0.86
2.5B	54	1920	20	96	128	2.27
8.3B	72	3072	24	128	512	2.10

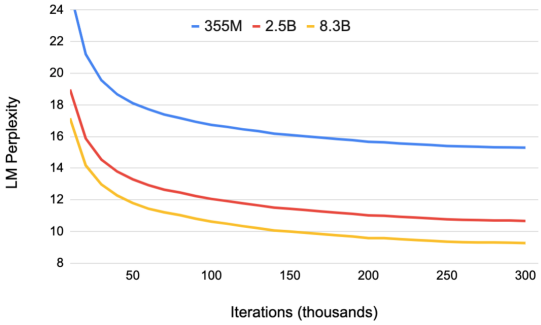


Figure 6. Validation set perplexity. All language models are trained for 300k iterations. Larger language models converge noticeably faster and converge to lower validation perplexities than their smaller counterparts.

section 4. Table 2 also lists the time it takes to advance one epoch which is equivalent to 68,507 iterations. For example, for the 8.3B model on 512 GPUs, each epoch takes around two days. Compared to the configurations used for our scaling studies in Table 1, the 2.5B model is the same, the 8.3B model has 24 attention heads instead of 32, and the 355M is much smaller than any seen previously while still using 64 GPUs to train, leading to the much lower time per epoch.

Figure 6 shows validation perpelixity as a function of number of iterations. As the model size increases, the validation perpelixity decreases and reaches a validation perplexity of 9.27 for the 8.3B model. We report the zero-shot evalution of the trained models on the LAMBADA and WikiText103 datasets in Table 3. We observe the trend that increasing model size also leads to lower perplexity on WikiText103 and higher cloze accuracy on LAMBADA. Our 8.3B model achieves state of the art perplexity on the WikiText103 test set at a properly adjusted perplexity of 10.81. At 66.51% accuracy, the 8.3B model similarly surpasses prior cloze accuracy results on the LAMBADA task. Samples generated from the model are in the supplementary material.

To ensure we do not train on any data found in our test sets, we calculate the percentage of test set 8-grams that also appear in our training set as done in previous work (Radford et al., 2019). The WikiText103 test set has at most 10.8% overlap and the LAMBADA test set (Paperno et al., 2016) has at most 1.4% overlap. We should note that the WikiText103 test set has already 9.09% overlap with the WikiText103 training set (Radford et al., 2019). As these are consistent with previous work, we are confident that no documents from our test data are inadvertently included in our training data.

Table 3. Zero-shot results. SOTA are from (Khandelwal et al., 2019) for Wikitext103 and (Radford et al., 2019) for LAMBADA.

Model	Wikitext103 Perplexity ↓	LAMBADA Accuracy ↑
355M	19.31	45.18%
2.5B	12.76	61.73%
8.3B	**10.81**	**66.51%**
Previous SOTA	15.79	63.24%

Table 4. Model configurations used for BERT.

Parameter Count	Layers	Hidden Size	Attention Heads	Total GPUs
336M	24	1024	16	128
1.3B	24	2048	32	256
3.9B	48	2506	40	512

5.3. Bi-directional Transformer Results Using BERT

In this section, we apply our methodology to BERT-style transformer models and study the effect of model scaling on several downstream tasks. Prior work (Lan et al., 2019) found that increasing model size beyond BERT-large with 334M parameters results in unexpected model degradation. To address this degradation, the authors of that work (Lan et al., 2019) introduced parameter sharing and showed that that their models scale much better compared to the original BERT model.

We further investigated this behaviour and empirically demonstrated that rearranging the order of the layer normalization and the residual connections as shown in Figure 7 is critical to enable the scaling of the BERT-style models beyond BERT-Large. The architecture (b) in Figure 7 eliminates instabilities observed using the original BERT architecture in (a) and also has a lower training loss. To the best of our knowledge, we are the first to report such a change enables training larger BERT models.

Using the architecture change in Figure 7(b), we consider three different cases as detailed in Table 4. The 336M model has the same size as BERT-large. The 1.3B is the same as the BERT-xlarge configuration that was previously shown to get worse results than the 334M BERT-large model (Lan et al., 2019). We further scale the BERT model using both larger hidden size as well as more layers to arrive at the 3.9B parameter case. In all cases, the hidden size per attention

Table 5. Development set results for MNLI, QQP, SQuAD 1.1 and 2.0 and test set results for RACE. The trained tokens represents consumed tokens during model pretraining (proportional to batch size times number of iterations) normalized by consumed tokens during model pretraining for our 336M model.

Model	trained tokens (ratio)	MNLI m/mm accuracy	QQP accuracy	SQuAD 1.1 F1 / EM	SQuAD 2.0 F1 / EM	RACE m/h accuracy
RoBERTa (Liu et al., 2019b)	2	90.2 / 90.2	92.2	94.6 / 88.9	89.4 / 86.5	86.5 / 81.3
ALBERT (Lan et al., 2019)	3	90.8	92.2	94.8 / 89.3	90.2 / 87.4	89.0 / 85.5
XLNet (Yang et al., 2019)	2	90.8 / 90.8	92.3	95.1 / **89.7**	90.6 / 87.9	88.6 / 84.0
Megatron-336M	1	89.7 / 90.0	92.3	94.2 / 88.0	88.1 / 84.8	86.9 / 81.5
Megatron-1.3B	1	90.9 / 91.0	92.6	94.9 / 89.1	90.2 / 87.1	90.4 / 86.1
Megatron-3.9B	0.75	**91.3 / 91.6**	**92.8**	**95.4** / 89.6	**91.0 / 88.0**	**91.2 / 87.8**

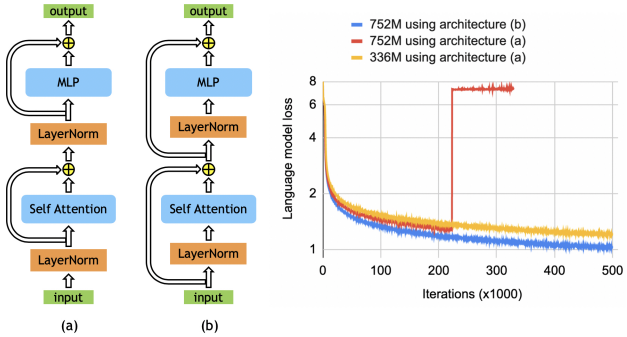


Figure 7. Training loss for BERT model using the original architecture (a) and the rearranged architecture (b). Left figure shows the training loss for 336M and 752M BERT model. While the original architecture performs well on the 336M model, the modifications in (b) enable stable training with lower training loss.

head is kept constant at 64. 336M and 1.3B models are trained for 2 million iterations while the 3.9B model is trained for 1.5 million iterations and is still training.

334M, 1.3B, and 3.9B models achieve validation set perplexity of 1.58, 1.30, and 1.20, respectively, a monotonic decrease with the model size. We finetune the trained models on several downstream tasks including MNLI and QQP from the GLUE benchmark (Wang et al., 2019), SQuAD 1.1 and SQuAD 2.0 from the Stanford Question answering dataset (Rajpurkar et al., 2016; 2018), and the reading comprehension RACE dataset (Lai et al., 2017). For finetuning, we follow the same procedure as (Liu et al., 2019b). We first perform hyperparameter tuning on batch size and learning rate. Once we obtain the best values, we report the median development set results over 5 different random seeds for initialization. The hyperparameters used for each model and task are provided in the supplementary material. Table 5 shows the development set results for MNLI, QQP, SQuAD 1.1, and SQuAD 2.0 and test set results for RACE. For the test set results of RACE, we first use the development set to find the checkpoint that gives us the median score on the 5 random seeds and we report the results from that checkpoint on the test set. From Table 5 we observe that (a) as the model size increases, the downstream task performance improves in all cases, (b) our 3.9B model establishes state of the art results on the development set compared to other BERT based models except for SQuAD 1.1 exact match, and (c) our 3.9B model achieves the single model SOTA results on RACE test set.

6. Conclusion and Future Work

In this work, we successfully surpassed the limitations posed by traditional single-GPU-per-model training by implementing model parallelism with only a few modifications to the existing PyTorch transformer implementations. We efficiently trained transformer based models up to 8.3 billion parameter on 512 NVIDIA V100 GPUs with 8-way model parallelism and achieved up to 15.1 PetaFLOPs sustained over the entire application. We also showed that for BERT models, careful attention to the placement of layer normalization in BERT-like models is critical to achieving increased accuracies as the model size increases. We study the effect of model size on down-stream task accuracy and achieve far superior results on downstream tasks and establish new SOTA for WikiText103, LAMBADA, and RACE datasets. Finally, we open sourced our code to enable future work leveraging model parallel transformers.

There are several directions for future work. Continuing to increase the scale of pretraining is a promising line of investigation that will further test existing deep learning hardware and software. To realize this, improvements in the efficiency and memory footprint of optimizers will be needed. In addition, training a model with more than 16 billion parameters will demand more memory than is available within 16 GPUs of a DGX-2H box. For such models, a hybrid intra-layer and inter-layer model parallelism along with inter-node model parallelism would be more suitable. Three other directions of investigation include (a) pretraining different model families (XLNet, T5), (b) evaluating performance of large models across more difficult and diverse downstream tasks (e.g. Generative Question Answering, Summarization, and Conversation), and (c) using knowledge distillation to train small student models from these large pretrained teacher models.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layernorm. *CoRR*, abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

Chen, C.-C., Yang, C.-L., and Cheng, H.-Y. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv:1809.02839*, 2018.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174.

Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., and Salakhutdinov, R. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019. URL http://arxiv.org/abs/1901.02860.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv:1806.03377*, 2018.

Hendrycks, D. and Gimpel, K. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016. URL http://arxiv.org/abs/1606.08415.

Howard, J. and Ruder, S. Fine-tuned language models for text classification. *CoRR*, abs/1801.06146, 2018.

Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018. URL http://arxiv.org/abs/1811.06965.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv:1807.05358*, 2018.

Joshi, M., Chen, D., Liu, Y., Weld, D. S., Zettlemoyer, L., and Levy, O. Spanbert: Improving pre-training by representing and predicting spans. *arXiv:1907.10529*, 2019.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. On large- batch training for deep learning: Generalization gap and sharp minima. *ICLR*, 2017.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. *arXiv:1911.00172*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Lai, G., Xie, Q., Liu, H., Yang, Y., and Hovy, E. Race: Large-scale reading comprehension dataset from examinations. *arXiv:1704.04683*, 2017.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., and Soricut, P. S. R. Albert: A lite bert for self-supervised learning of language representations. *arXiv:1909.11942*, 2019.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server, 2014.

Liu, X., He, P., Chen, W., and Gao, J. Multi-task deep neural networks for natural language understanding. *CoRR*, abs/1901.11504, 2019a. URL http://arxiv.org/abs/1901.11504.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019b. URL http://arxiv.org/abs/1907.11692.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

McCann, B., Bradbury, J., Xiong, C., and Socher, R. Learned in translation: Contextualized word vectors. *CoRR*, abs/1708.00107, 2017.

Melamud, O., Goldberger, J., and Dagan, I. context2vec: Learning generic context embedding with bidirectional lstm. In *Proceedings of The 20th SIGNLL Conference on Computational Natural Language Learning*, pp. 51–61, 01 2016.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016. URL http://arxiv.org/abs/1609.07843.

Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

NVIDIA. Mixed precision training: Choosing a scaling factor, 2018. URL https://docs.nvidia.com/deeplearning/sdk/mixed-precision-training/index.html#scalefactor.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. *CoRR*, abs/1606.06031, 2016. URL http://arxiv.org/abs/1606.06031.

Pennington, J., Socher, R., and Manning, C. D. Glove: Global vectors for word representation, 2014. URL https://www.aclweb.org/anthology/D14-1162.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. *CoRR*, abs/1802.05365, 2018. URL http://arxiv.org/abs/1802.05365.

Radford, A., Józefowicz, R., and Sutskever, I. Learning to generate reviews and discovering sentiment. *CoRR*, abs/1704.01444, 2017.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training, 2018. URL https://blog.openai.com/language-unsupervised/.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Better language models and their implications, 2019. URL https://openai.com/blog/better-language-models/.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv:1910.10683*, 2019.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. Squad: 100,000+ questions for machine comprehension of text. *EMNLP*, 2016.

Rajpurkar, P., Jia, R., and Liang, P. Know what you dont know: Unanswerable questions for squad. *ACL*, 2018.

Ramachandran, P., Liu, P. J., and Le, Q. V. Unsupervised pretraining for sequence to sequence learning. *CoRR*, abs/1611.02683, 2016. URL http://arxiv.org/abs/1611.02683.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

Trinh, T. H. and Le, Q. V. A simple method for commonsense reasoning. *CoRR*, abs/1806.02847, 2018. URL http://arxiv.org/abs/1806.02847.

Turian, J., Ratinov, L., and Bengio, Y. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pp. 384–394, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

Valiant, L. G. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103-111, 1990.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding. *ICLR*, 2019.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

You, Y., Gitman, I., and Ginsburg, B. Large batch training of convolutional networks. *arXiv:1708.03888*, 2017.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., and Hsieh, C.-J. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv:1904.00962*, 2019.

Zellers, R., Holtzman, A., Rashkin, H., Bisk, Y., Farhadi, A., Roesner, F., and Choi, Y. Defending against neural fake news. *CoRR*, abs/1905.12616, 2019. URL http://arxiv.org/abs/1905.12616.

Zhu, Y., Kiros, R., Zemel, R. S., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. *CoRR*, abs/1506.06724, 2015.