Parallelism Is All You Need

songhuiming@gmail.com

Abstract

Deep learning models are getting bigger and bigger. When BERT model was introduced in 2018, it has 110 million parameters for base and 345 million for large model. Cross lingual model XLM-R has 550 million parameters. GPT-2-xl has 1.5 billion while GPT-3 has 175 billion parameters. Although models' performance has been improving, it brings new problems to train or fine tune the deep neural network models: it takes much more time to retrain the big model; The model may be too big to be loaded into one GPU or the number of batch size will be limited due to limited GPU memory. Because of these, parallelism and the distributed computation is the necessary tool to improve the efficiency. In this paper, we have presented different parallelism technology and proposed the accelerating approach for parallel training with mixed precision and gradients accumulation. We find that parallel training with mixed precision and gradients accumulation will not only increase the training speed but also improve the model performance. Based on our experiments, while achieving better evaluation metrics, time for training process is reduced about 72% with 4 GPUs in parallel and 85% with mixed precision and gradient accumulations.

1 Introduction

Deep learning has gained a lot of popularity in the last several years. The algorithms are inspired by the structure and function of the brain called neural networks. Deep Neural Networks (DNN) have been employed in a wide spectrum of applications, such as image recognition [6], natural language understanding [7], language translation [7], deep graph modeling [17], anomaly detection [8], content recommendation [9], drug discovery [10], art generation [11], game play [12], and self-driving cars [13] and so on. To achieve better performance, many applications pursue higher intelligence by optimizing larger models using larger datasets. When BERT [1] model was introduced in 2018, it has about 110 million parameters for base and 345 million for large model. Cross lingual model XLM-R [2] has 550M parameters. GPT-2-xl [3] has 1.5 billion parameters while GPT-3 [4] has about 175 billion parameters. Text-To-Text Transfer Transformer (T5) [5] has 11 billion parameters. As the model becomes bigger and bigger, it brings new challenges: it takes more and more time to train or fine tune the model; some models are too big to be loaded into one GPU because of limited GPU memory; or even if it can be loaded into GPU, the batch size has to be set as very small since the parameters and gradients need to be saved for parameter update and it will result in lacking of GPU memory.

All of these challenges make it craving advances in distributed training systems. To accelerate the training speed, people have thought about the data parallelism to distribute data to different GPUs to train the model parallelly. To overcome the big model loading issue, people have thought of splitting the model into different GPUs. Among existing solutions, distributed data parallel is a dominant strategy due to its minimally intrusive nature. Training a DNN model usually repeatedly conducts three steps, the forward pass to predict and to compute loss, the backward pass to compute gradients, and the optimizer step to update parameters based on the current parameters value and the computed gradients. The concept of data parallelism is universally applicable to such frameworks. Data parallelism is to parallelize the computation of the gradient for the batch across batch elements.

In this paper, we have explored the different parallelism technology and the other accelerating methods. We found that parallel training with mixed precision and gradients accumulation will not only increase the training speed but also improve the model performance. Based on our experiments, time for training process was reduced about 72% with 4 GPUs to 85% with other techs combined together and the AUC increased slightly compared to the model with single GPU with no gradient accumulations.

2 Preliminary

2.1 Model parallel and data parallel

Model parallel is splitting the model to different devices while data parallel is splitting the development data into different GPUs to calculate. A simple example is given below about how to do it and how many data needs to transfer.

When there are huge amount of parameters, data communication will become the bottle neck to run the model. Suppose there is a data matrix X with size 64×1000, a parameter matrix A with size 1000×1000, and a parameter matrix B with size 1000×500, The loss is assumed to be $||XAB - Y||$. Suppose we have 2 devices (GPU) and we need to calculate forward and back-propagation to update A and B. If we use model parallel: we can split A into 2 matrices with size 1000×500 like $A = [A_1, A_2]$. B is divided into two 500×500 matrices $B^T = [B_1^T, B_2^T]$, then $XAB = XA_1B_1 + XA_2B_2$. When forward, each device needs to transmit/receive 64×500 floating-point numbers (pass XA_iB_i to other devices). If we use data parallel: we can place two A and B with the same initial value on two devices, and split the X into two 32×1000 matrices $X = [X_1, X_2]$ on two devices respectively for forward and backpropagation. Ideally, forward does not need to transmit data, and each device needs to transmit/receive $1000 \times 1000 + 1000 \times 500$ floating-point numbers during backprop (transfer the A and B gradients calculated by X_i to other devices). In this paper we will focus on data parallel since we care more about training speed and the models we used can be fitted into the single GPU.

2.2 Data parallel and AllReduce

when updating models, what we care most are the parameters Θ of the models. We call each device as worker. Parameter update is from the updated gradients of the parameters on each worker, and the worker needs to get the updated model parameter Θ to calculate the gradients. To update parameters Θ, one method is the master-slave model: in which the device will play two roles. The parameter server (PS, Figure 1) will maintain a copy of the latest parameters, and worker will read the parameters from the parameter server to calculate the gradients and send the request to the server to update the gradients and parameters. Before deep learning, the big model usually means there are huge amount of feature vectors but the model itself is shallow. When the feature array increases (e.g., millions of records), it usually becomes sparse. In this situation each worker only needs a small number of features to do the calculation so the data communication between the workers to the parameter server is not big. However, as model becomes much deeper in deep learning, the number of parameters becomes much huger than before and it requires huge amount of data communication between the workers and parameter server.

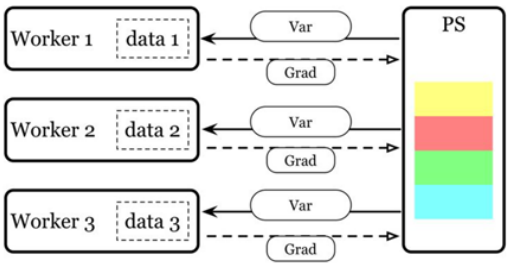

Figure 1. Parameter Server mode

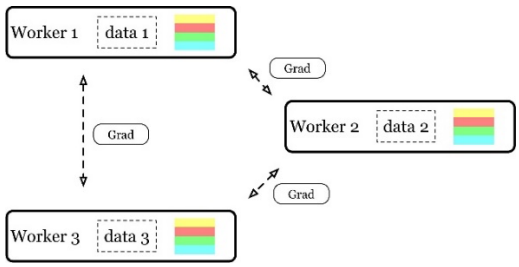

Figure 2. AllReduce mode

To solve the bottleneck of data communication, one of the solutions is called AllReduce (Figure 2). The AllReduce operation expects each process to provide equally-sized tensors, collectively applies a given arithmetic operation (e.g., sum, prod, min) to input tensors from all processes, and returns the same result tensor to each participant. The update of the parameter Θ comes from the gradient calculated by each device. If the parameter Θ obtained by all devices is synchronized during initialization, and the gradient is synchronized before the parameter update, then the parameter copies Θ_i on each device is consistent at any time. So we can indirectly realize the synchronization of the parameter Θ_i based on the gradient. The synchronization of the gradient depends on the efficient implementation of the AllReduce operation [14]. In the AllReduce mode, all devices act as parameter servers and workers at the same time. Each device can directly use the local parameter copy Θ_i to perform forward and back-propagation calculations. After

all the devices have calculated the gradient, AllReduce operation is executed, and each device will get the gradient average of all devices. Finally, each device uses the obtained gradient average to update the local Θ_i (Figure 3).

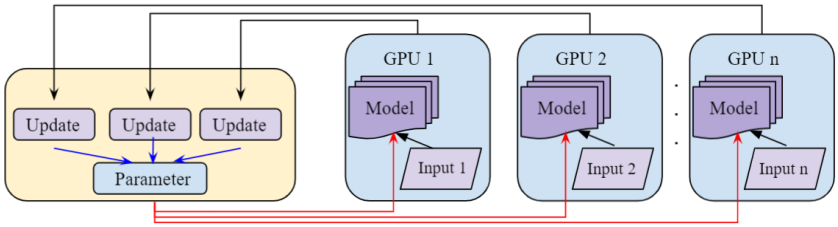

Figure 3. Parallel training and parameter update

2.3 Gradients average

In multi-GPU distributed training, multi-level averaging of gradients is usually used: it first calculates the average of the gradient generated by the mini-batch of each device, then calculates the average of each device. Assuming that the number of samples of the mini-batch of the i_{th}device is n_i, the gradient generated by each sample is G_k and the number of devices is m, the average gradient calculated by all devices is

$$G_{avg} = \frac{1}{m}\sum_{i=1}^{m}\frac{\sum_{k=1}^{n_i} G_k}{n_i}$$

If the number of samples of each device is equal, then

$$G_{avg} = \frac{1}{m}\sum_{i=1}^{m}\frac{\sum_{k=1}^{n_i} G_k}{n_i} = \frac{1}{mn}\sum_{i=1}^{m}\sum_{k=1}^{n_i} G_k = \frac{1}{N}\sum_{i=1}^{m}\sum_{k=1}^{n_i} G_k$$

where N is the total sample used in one update.

2.4 FP16 acceleration

FP16 (Figure 4) is another popular speed-up method. It will bring two aspects of speed increase: GPU computing speed (there are lots of FP16 computing units in Nvidia GPU P100, V100, Titan V) and variable transmission speed. In a bandwidth constrained scenario, after calculating the gradient using FP32, you can convert the gradient to FP16 and then synchronize between GPUs. With the FP16 computing unit, this speed increasement is considerable.

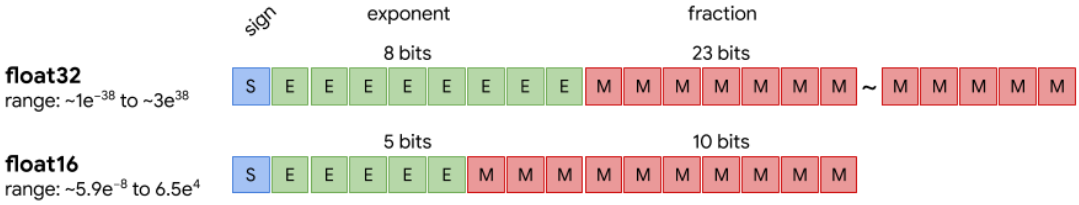

Figure 4. FP32 and FP16

3 Approaches

There are many ways to launch the multi-process jobs in python. Some will start from a single process to process data and then spawn an independent process for each GPU, while the other will launch the multiple processes from the beginning. Here we apply different distributed methods, different launch processes and compare their performance in the public data. The mixed precision of FP32 and FP16 is also tested and compared. The model we used is the transfer learning model based on XLM-R which is a multilingual model that can handle 100+ languages. The multilingual model will improve our efficiency and reduce the cost since we can build one model to process data from different locales. In transfer learning, we dropped the classification layer from XLM-R and replaced it by multiple dropout and fully connected (FC) layers combinations to make the model more general for classification.

3.1 Accelerated by multiprocess spawn

Algorithm 1 Distributed model training: multiprocessing.spawn

Input: Input X, Y and the pretrained model for transfer learning
Output: Distributed calc the forward and backpropagation to update parameters Θ

Suppose there are n GPUs, that is nnodes = n
Call multiprocessing to spawn n processes *mp.spawn(main_worker, nprocs)*
Initialize the process for each GPU with proper backend
Copy model to each GPU
Scale down batch size to batch size/n
Distributed sample to split training data X equally on n GPUs
Initialize the parameters Θ and copy it to all GPUs
for batch i **do**
    Parallelly computing for all GPUs, for GPU j
        On GPU j, collect X_{ij}, Y_{ij}
        Forward and calculate loss L_j
    Till all GPU are done
    Gather loss from all GPU $L = L(L_1, \cdots, L_n)$
    Backward to calculate the gradients G_i
    Update the parameters to $\Theta_{i+1} = f(\Theta_i, G_i)$
    Sync parameters Θ_{i+1} to all GPUs
    back to step i+1 with the new parameter Θ_{i+1}
end for

The first method we tried is process spawn. PyTorch multiprocessing is a wrapper around the native python multiprocessing module. It registers custom reducers that use shared memory to provide shared views on the same data in different processes. Once the tensor/storage is moved to shared memory, it will be possible to send it to other processes without making any copies. Spawning a number of subprocesses can be done by creating process instances and calling join to wait for their completion. In details: First we need to initialize the process for each GPU with proper backend. Then the model with the same initialized value of parameters will be copied to each GPU. Next the data will be randomly and equivalently split to each GPU. For each batch, the model on each GPU will run the forward step to get the predictions and the loss and then run the back propagation step to calculate the gradients. Then the gradients will be aggregated from different GPUs and synced back to each GPU to update the parameters. Since all the model parameters start from the same initial value, and they are updated with the same gradients in every step, it will guarantee that the parameters on each GPU are the same.

3.2 Accelerated by torch.distributed DistributedDataParallel

Algorithm 2 Distributed model training: distributed.launch

Input: Input X, Y and the pretrained model for transfer learning
Output: Distributed calc the forward and backpropagation to update parameters Θ

Suppose there are n GPUs, that is nnodes = n
Launch *torch.distributed.launch* launcher to assign local_rank for each GPU
Initialize for each GPU communication *dist.init_process_group(backend='nccl')*
Scale down batch size to batch size/n
Distributed sample X equally on n GPUs: *distributed.DistributedSampler(X)*
Initialize the parameters Θ and copy it to all GPUs
Wrap the model: *parallel.DistributedDataParallel(model, device_ids=local_rank)*
for batch i **do**
    Similar to Algorithm 1 to update Θ in each batch
end for

PyTorch provides the launcher called torch.distributed.launch to execute the python code parallelly. Rather than using one process to control all the GPUs, with distributed, we only need to write one copy of code and PyTorch will automatically assign it to n processes and run it on n different GPUs. During the execution, the launcher will send the current process index (the corresponding GPU) to python. The model will be wrapped by the DistributedDataParallel

(DDP) class to do AllReduce job. The rest will be similar to spawn above: copy model and split data to each GPU, calculate forward step predictions and calculate the gradients to update the parameters for each batch.

3.3 Accelerated by Nvidia apex with mixed precision

Algorithm 3 Distributed training: apex distributed training with mixed precision

Input: Input X, Y and the model
Output: Distributed calc the forward and backpropagation to update parameters Θ

Suppose there are n GPUs, that is nnodes = n
Initialize the model and optimizer as *amp.initialize(model, optimizer)*
Distributed sample to split training data X equally on n GPUs
Initialize the parameters Θ and copy it to all GPUs
Wrap model: *apex.parallel.DistributedDataParallel(model, device_ids=local_rank)*
for batch i **do**
    Wrap the loss with *amp.scale_loss(loss, optimizer)* to auto precision for loss
    Similar to above to update Θ in each batch
end for

Apex is the open source library by Nvidia for mixed precision and distributed training [15]. The mixed precision is wrapped to dramatically reduce the GPU memory usage and fasten the process. It also optimizes the NCCL communication. Apex will automatically manage the model parameters and the precision in the optimizer, and it will load the different configuration parameters based on the precision requirement. Different from PyTorch distributed function, Apex can manage some of the parameters precision automatically. When calculate the loss, Apex needs to be wrapped by the scale_loss to automatically manage the precision based on the loss value.

3.4 Accelerated by Horovod with ring AllReduce

Algorithm 4 Distributed training: horovod with ring communication

Input: Input X, Y and the pretrained model for transfer learning
Output: Distributed calc the forward and backpropagation to update parameters Θ

Suppose there are n GPUs, that is nnodes = n
Launch with *horovodrun* launcher to assign local_rank for each GPU
Initialize the process for each GPU with proper backend *hvd.init()*
Scale down batch size to batch size/n
Distributed sample X equally on n GPUs: *distributed.DistributedSampler(X)*
Broadcast the parameters Θ from GPU 0 to all GPUs, wrap the optimizer
hvd.broadcast_parameters(model.state_dict(), root_rank=0)
hvd.DistributedOptimizer(optimizer, named_parameters)
for batch i **do**
    Similar to above to update Θ in each batch
end for

Horovod leverage the experience from Facebook "Training ImageNet In 1 Hour" and the "Ring Allreduce" from Baidu which syncs the gradients with ring communication [16]. Similar to the launcher in torch, hovorodrun launcher will automatically dispatch the code to n process for n GPUs. After initialization, it attaches AllReduce hooks to the parameters. During the execution, the launcher will send the current process index into horovod so that we can get the index of each process. The parameters of the model will be broadcasted to each GPU from the root rank GPU. It also has the built-in DistributedSampler to split the data of each batch to different partitions. In each process, the GPU related to that process only needs to fetch that partition of data for that rank of GPU. After loss is calculated in each GPU, it has the distributed optimizer wrapper to do the reduce calculation of the gradients which equals to the average of the gradients from all the GPUs.

3.5 Accelerated by Huggingface accelerate

Algorithm 5 Distributed model training: huggingface accelerate wrapper

Input: Input X, Y and the model
Output: Distributed calc the forward and backpropagation to update parameters Θ

Suppose there are n GPUs, that is nnodes = n
Launch *torch.distributed.launch* launcher to assign local_rank for each GPU
Initialize the process *accelerator = Accelerator(fp16, gpu)*
Copy model to each GPU *model = model.to(accelerator.device)*
Prepare the model, optimizer and dataloader to accelerator
    accelerator.prepare(model, optimizer, train_load, eval_load)
for batch i **do**
    Similar to Algorithm 1 to update Θ in each batch
    Backpropagate with *accelerator.backward(loss)*
end for

Huggingface accelerate is a high level wrapper for distributed computing in PyTorch [18]. What you need to do are just two things: first initialize the accelerator, and then prepare the objects of data loader, model and optimizer for the accelerator. During the initialization step, accelerate will collect the information like what hardware to use, what accelerate state variables are assigned and it will create process for each GPU with different process index. The data loader preparation is responsible for breaking the dataset into subsets based on the process_index inside the process group, making sure that each GPU only gets a subset of the data. The model and optimizer preparation will wrap the model inside the DistributedDataParallel class and pass in a list of devices by the index of each process for each GPU.

3.6 Accelerated by gradients accumulate and parallelism

Algorithm 6 Accumulative gradients

Input: X_i, Θ_i for batch i
Output: updated parameters Θ_{i+1} after batch i

for X_i in batch i **do**
    split X_i by equally accumulative steps to X_{ij}, $j = 1 \cdots n$
    for X_{ij} in accumulative step j **do**
        calculate forward step $f(X_{ij}, \Theta_i)$
        backward step to update the gradients G_{ij}
        if j == num of accumulative steps n:
            get the updated gradietns $G_i = (G_{i,1}, \ldots, G_{i,n})$
            update the parameters based on the accumulative gradients G_i
            $\Theta_{i+1} = f(\Theta_i, G_i)$
    end for
    back to step i+1 with the new parameter Θ_{i+1}
end for

Due to limited GPU memory, a lot of the time the batch size has to be set as very small to avoid GPU out of memory. For example, when we set max sequence length as 100, on M60 GPU, the maximum allowable batch size is 8 for XLM model. This will affect the model stability since the data variance will be high with a small batch size. To solve this, we leverage the accumulative gradients technology to calculate the forward steps for n accumulation steps and aggregate the gradients together. After that we will update the parameters based on the n loops. In this way, under the same condition as above, we can enlarge the batch size from 16 to 128 with $n = 8$. Gradients accumulate will make the model converge better and increase the accuracy. From our experiments it shows that parallel training with mixed precision and gradients accumulation will not only train faster but also gives best performance.

3.7 Others to improve model performance

Other skills we have used to improve the model performance includes: 1) Adjustable learning rate and early stop. After some epochs, the model metrics may be stuck or diverge because the learning rate is high for small gradient change. If the evaluation dataset performance did not improve in the 3 consecutive epochs, the learning rate will be

multiplied by ratio = 0.1. If there is no change after adjustment, we will stop the training to avoid overfitting. 2) Gradients clip. To prevent the abnormal gradients causing the model diverge, gradients clip is applied to truncate the abnormal gradients. 3) Weight decay. To prevent the model from overfitting, the parameters except bias and layernorm weights are penalized by adding the weighted sum of square of parameters into the loss function.

4 Experiments

In this section we did two experiments with the models and methods introduced in Section 3.

4.1 Hate speech / Toxic data classification with cross lingual model XLM-R

The first is the hate speech data collected from internet. It includes: 1) The datasets from hatespeechdata.com, with 62 data sets from 15 languages; 2) The toxic comments data from kaggle.com; 3) The hate speech data published on aclweb.com including abusive, offensive and insulting. The data is cleaned to remove noises like url, ip address, @username, date/time, article id and so on. Words like 'idioooot' are normalized to 'idiot' ('i[d]+io[t]+', '(i)([^a-z]*)(d)([^a-z]*)(i)([^a-z]*)(o)([^a-z]*)(t)' -> idiot, …). The final cleaned data has 2,066,821 sentences and is split to train/valid/test with ratio 70:15:15 respectively.

Since the data includes different languages like En, Fr, De, we developed the cross-lingual transfer learning model based on XLM-R as introduced in section 3 that can encode different language tokens. It process all languages with the same shared vocabulary created through Byte Pair Encoding (BPE). BPE replace the common shared bytes by another byte. Because of this, it can capture the tokens sharing the similar or close patterns. It splits on the concatenation of sentences sampled randomly from the monolingual corpora. Sentences are sampled according to a multinomial distribution with probabilities $\{q_i\}$ and frequency $\{n_i\}$, $i = 1, \dots N$ for token i, where:

$$q_i = \frac{p_i^a}{\sum_{j=1}^{n} p_j^a} \quad with \quad p_i = \frac{n_i}{\sum_{k=1}^{N} n_k}$$

On the training and validation data, we have compared the results of one GPU to the results with 4 GPUs. Figure 5 shows the training time for different approaches. The accuracy plot of the validation data is in Figure 6. Compared to single GPU, the distributed training based on process spawn or distributed launcher trains about 3.3 times faster (4883 vs. 1464 minutes). Since the data communication takes some time so we cannot reach the linear scale-up with 4 GPUs compared to only one GPU. Horovod (2203) runs about 2.2 times faster than the single GPU but it is slower than pytorch DDP. The reason is that Horovod uses negotiation to fusion communication workflow, while DDP using multiple cuda streams to call nccl AllReduce concurrently. Huggingface accelerate is close to DDP since it's mainly the wrapper for DDP. The fastest is from Apex mixed precision training which is about 6.8 times faster than the single GPU. That is because Apex not only has parallelism, but also combines FP16 and FP32 to expedite the training and reduce the memory usage. Based on this, we trained the best model which is mixed of Apex and gradients accumulate. It is about 7.1 times faster than the single GPU.

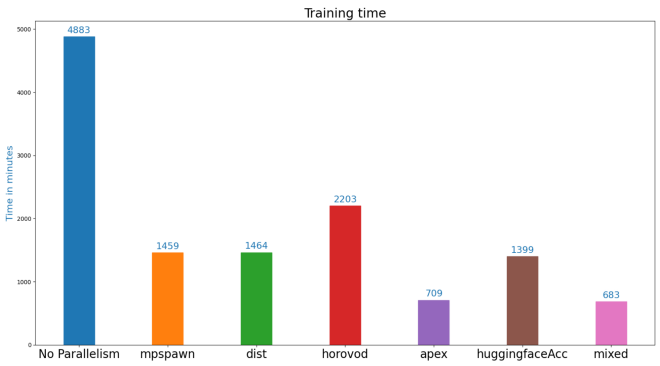

Figure 5. Model training time

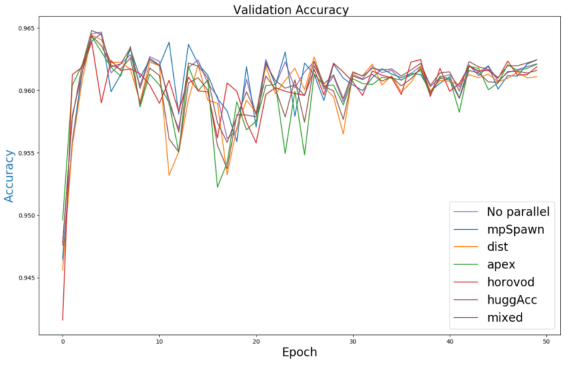

Figure 6. Accuracy on the validation data

Table 1 is the summary of the model performance on test data. As is shown, all the models have very close performance by the F1 score, AUC and accuracy on the test data. The model with gradients accumulate performs lightly better by

AUC which is independent of the threshold. The possible reason is that with gradients accumulate we can use bigger batch size and thus the data variance of the samples will be reduced with bigger batch size.

Table 1: Model training tine, performance and batch size

Hardware Specs	Models	Avg Train Time	F1 - Test	AUC - Test	Accuracy - Test	batch_size
Single GPU	No Parallism	4883	0.81966	0.97315	0.96519	32
Multi GPU (4)	Multiprocessing Spawn	1459	0.81846	0.97249	0.96532	128
	Distributed Parallel launch	1464	0.81638	0.97292	0.96516	128
	Apex Mixed Precision	709	0.81432	0.97029	0.96512	128
	Horovod Ring Comm	2203	0.81528	0.97527	0.96524	128
	Huggingface Accelerate	1400	0.81621	0.97103	0.96518	128
	Parallism + Gradients Accumulate	683	0.81876	0.97316	0.96539	256

References

[1] Devlin, Jacob; Chang, Ming-Wei; Lee, Kenton; Toutanova, Kristina. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805v2, 2018

[2] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Edouard Grave, Myle Ott, Luke Zettlemoyer, Veselin Stoyanov. Unsupervised Cross-lingual Representation Learning at Scale, arXiv:1911.02116, 2019

[3] Better Language Models and Their Implications. OpenAI, 2019.

[4] Brown, Tom B.; Mann, Benjamin; Ryder, Nick; Subbiah, Melanie; Kaplan, Jared; Dhariwal, Prafulla; Neelakantan, Arvind; Shyam, Pranav; Sastry, Girish; Askell, Amanda; Agarwal, Sandhini; Herbert-Voss, Ariel; Krueger, Gretchen; Henighan, Tom; Child, Rewon; Ramesh, Aditya; Ziegler, Daniel M.; Wu, Jeffrey; Winter, Clemens; Hesse, Christopher; Chen, Mark; Sigler, Eric; Litwin, Mateusz; Gray, Scott; Chess, Benjamin; Clark, Jack; Berner, Christopher; McCandlish, Sam; Radford, Alec; Sutskever, Ilya; Amodei, Dario. Language Models are Few-Shot Learners, arXiv:2005.14165, 2020

[5] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu, Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer, arXiv:1910.10683, 2019

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770-778, 2016

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018

[8] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1285-1298, 2017

[9] A. Van den Oord, S. Dieleman, and B. Schrauwen. Deep content-based music recommendation. In Advances in neural information processing systems, pages 2643{2651, 2013

[10] B. Ramsundar, P. Eastman, P. Walters, and V. Pande. Deep learning for the life sciences: applying deep learning to genomics, microscopy, drug discovery, and more. " O'Reilly Media, Inc.", 2019.

[11] H. Mao, M. Cheung, and J. She. Deepart: Learning joint representations of visual arts. In Proceedings of the 25th ACM international conference on Multimedia, pages 1183{1191, 2017.

[12] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In Advances in neural information processing systems, pages 3338-3346, 2014

[13] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316, 2016

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In Advances in Neural Information Processing Systems 32, pages 8024-8035. Curran Associates, Inc., 2019.

[15] NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl, 2019

[16] Horovod distributed training framework, https://github.com/horovod/horovod

[17] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, Maosong Sun, Graph Neural Networks: A Review of Methods and Applications, arXiv:1812.08434, 2018

[18] Huggingface accelerate framework, https://github.com/huggingface/accelerate

[19] Open MPI: A High Performance Message Passing Library. https://www.open-mpi.org/