Implementation of Model Parallelism Training

Semester Project

Author Kang Peilin

Advisor Lin Tao

Supervisor Prof. Martin Jaggi

Machine Learning and Optimization laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne

Submitted on June 18, 2019
Lausanne, EPFL

Abstract

Recently, several methods have been proposed to implement the model parallelism Training. In this report, first, we analysis the improvements and limitations of these methods. They are Naive sequential, Delayed Gradient(DDG), Feature Replay(FR) and PipeDream. For Naive Sequential model parallelism method, both forward pass and backward pass are sequential, thus under-utilize the GPU resources. We use the Naive Sequential method as the baseline. Delayed Gradient(DDG) uses the delayed gradient to parallel the backward pass, but suffers from weight staleness. The different parts of the model on different GPUs have different degrees of staleness. Feature Replay(FR) recomputes the activation before doing the backward pass and parallel the backward pass, but suffers from forward pass and backward pass inconsistency. PipeDream parallels both the forward pass and backward pass, but suffers from weight staleness as DDG. Then we implement an unified model parallelism training framework using Pytorch and MPI to run these methods. We apply these four parallelism methods on ResNet used for image classification and BERT used for language modeling. The experimental result shows that for resnet20, DDG, FR and PipeDream can make the training process faster. But they will slightly reduce the best top1 accuracy and with the number of partition blocks increases, the best top1 accuracy becomes worse. Under most of the circumstances, FR losses more accuracy than DDG and PipeDream. For BERT model, all of DDG, FR and PipeDream reduce the statistical efficiency which means the number of steps needed to achieve a given LM loss increases compared with naive sequential method. And FR performs worse than DDG and PipeDream.

Contents

1 Introduction

In recent years, deep neural network(DNN) has been used in various real world tasks, such as image classification, language modeling, and has achieved great success. Since the size of the model and at the same time the size of the datasets are increasing dramatically, traditional way of training model on a single machine with limited computation ability and memory size may seem unsuitable.

The most common way of training DNN is data parallelism. Data parallelism is you use the same model for every machine and feed it with different mini-batches of the data. During the backward pass, you need to collect all the gradients and update the parameters with the overall average. This may cause the communication overheads eclipse the computation time, and becomes the main bottleneck.

Another approach is model parallelism. Model parallelism is you split the deep neural network into several blocks and put each block on different machines. Then you feed first block with mini-batches of the data and the latter blocks just receive the output of the former block's forward pass as the input. Compared with data parallelism, the biggest advantage of model parallelism is that the communication overheads are greatly reduced, because you only need to pass the activation and gradient of the last layer of each block. The main drawback of model parallelism is the under-utilization of machines. As shown on Figure 1, we split the model into 4 blocks and put each block on a machine. During the training, both the forward pass and backward pass are sequential, which means there is only one machine working at a time.

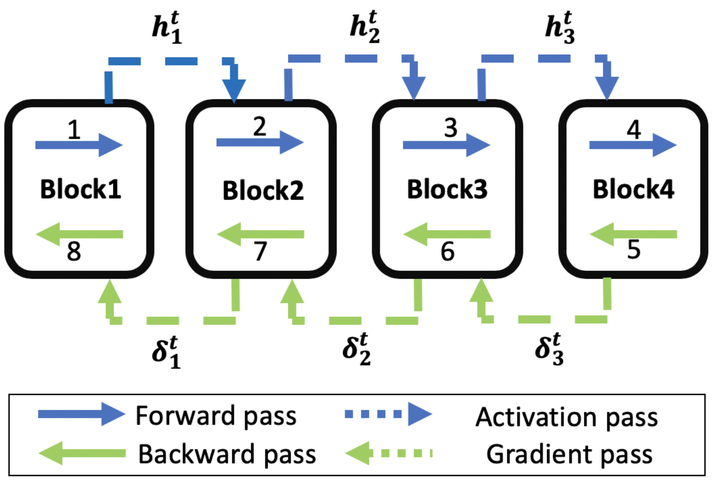


Figure 1: We split the model into 4 blocks and put each block on different machines. During the training, both the forward pass(blue arrow, from 1 to 4) and the backward pass(green arrow, from 5 to 8) are sequential. And the communication procedure passes the activation and the gradient of the last layer of each block.

There have been several methods proposed in order to better utilize the computation ability of machines and increase the degree of the parallelism. [7] proposed a method named DDG which uses the delayed gradients to break the backward dependence and thus parallel the backward pass without losing too much accuracy. [6] proposed to recompute the forward pass again using the latest weight version before doing the backward pass. [4] proposed a method named PipeDream. In the startup phase of the Pipedream, the first machine loads K(K is the number of machines) mini-batches to keep the pipeline full in steady state. Once in steady state, each machine alternates between performing the forward and backward pass for a minibatch. PipeDream parallels both forward pass and backward pass.

In this report, we implement an unified model parallel training framework based on Pytorch and MPI. On this framework we can run naive sequential model parallelism, Delayed Gradient(DDG), Feature Replay(FR) and PipeDream. Then we apply these different parallel methods on ResNet used for image classification and BERT used for language modeling to test advantages and disadvantages of these methods and better understand the model parallelism. Section 2 of this report details the principles, improvements and limitations of different model parallelism methods in theory. Section 3 describes the specific implementation of the unified model parallelism training framework. Section 4 presents the experimental results on ResNet and BERT. The final section is the conclusion.

2 Model Parallelism

We want to train a deep neural network to gain high accuracy using as little time as possible. As concluded in [4], this goal can be captured with two matrices: 1) statistical efficiency: the number of epochs needed to reach a desired level of accuracy; 2) hardware efficiency: the time needed to complete a single epoch. Total time to train a model is simply the product of these two matrices. Here, besides the statistical efficiency and hardware efficiency, we added another two matrices to measure the different model parallelism methods: 3) the best accuracy/smallest loss the model can achieve; 4) the total memory size needed to train the model.

2.1 Naive Sequential

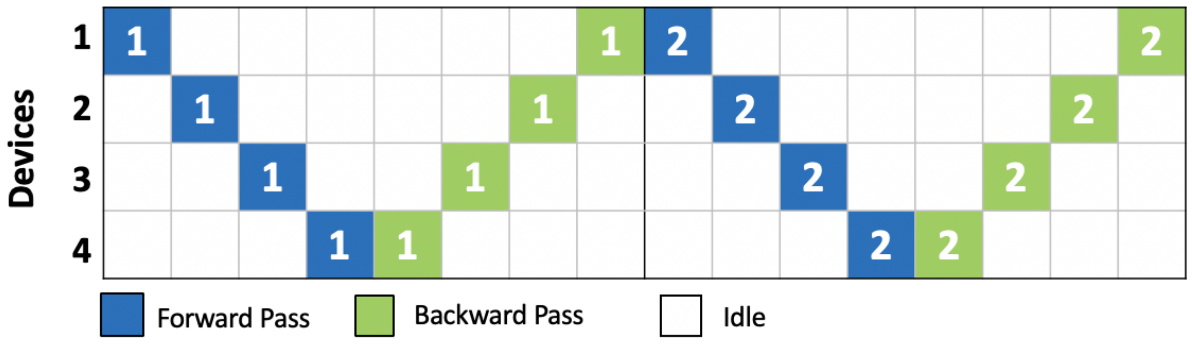


Figure 2: Naive sequential model parallel training with 4 devices, the blue one represents the forward pass and the green one represents the backward pass, white square represents idle. And the number on square means the mini-batch ID

As shown on Figure 2, the workflow of naive sequential model parallelism is very simple. During the forward pass, Device 1 gets the input data, for example mini-batch 1, and then the mini-batch 1 flows from device 1 to device 4 to perform the forward calculation. During the backward pass, device 4 calculates the loss and gradient of mini-batch 1, and then the gradient flows from device 4 to device 1 to perform the weight update for mini-batch 1.

2.1.1 improvements

The naive sequential model parallelism is almost the same as the single machine training, apart from when the parameter size of the model can't be allocated on a single machine and then we can use naive sequential model parallelism method to handle this problem. The statistical efficiency, hardware efficiency, the best accuracy and the total memory size needed to train the model are almost the same as a single machine.

2.1.2 limitations

The limitation of this method is obvious, we use several devices, but we almost don't get any benefits, the computation ability of devices are highly under utilized. This limitation is caused by the sequential training workflow of the deep neural network.

2.2 Delayed Gradients(DDG)

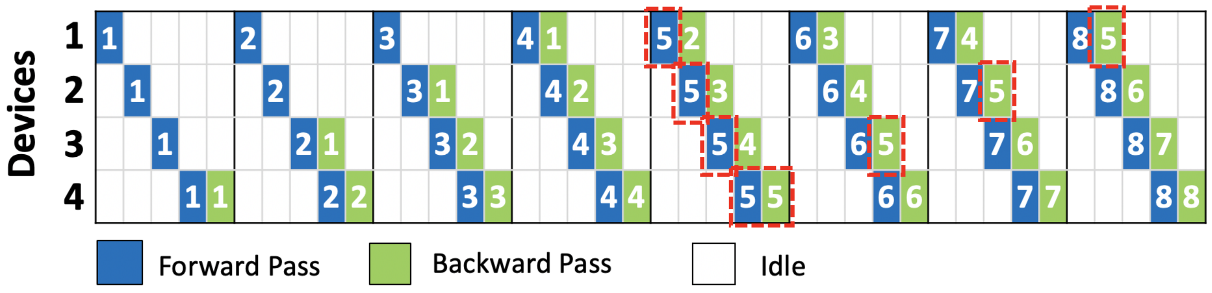


Figure 3: Delayed gradient(DDG) parallel training with 4 devices. Like the naive sequential, the number on square means the mini-batch ID.

We assume the model split into K devices. We denote h_k^t as the activation(forward pass output) of mini-batch t in device k. We denote δ_k^t as the error gradients of mini-batch t in device k. And we denote w_k^t as the weight of device k updating from mini-batch t(means after doing the backward pass and weight update of mini-batch t).

Forward pass of the DDG is the same as naive sequential, input data flows from device $k = 1$ to $k = K$. In each device, we compute the activation in sequential order. In the backward pass, all devices except the last one have delayed error gradients in store such that they can execute the backward computation as soon as they finish the forward computation. The last device updates with the up-to-date gradients, while the devices k updates with the gradient which delays $K - k$ mini-batches(error gradients in store is δ_k^{t-K+k}).

2.2.1 improvements

parallel backward pass DDG uses the delayed gradients to break the backward dependency. In this way the total computation time for a step reduced to $T_F + \frac{T_B}{K}(T_F$

and T_B is the forward pass time and backward time for a mini-batch in Naive sequential method). As a result, we need less time to finish an epoch training and the hardware efficiency increases.

2.2.2 limitations

memory consumption We need to use the same version of the model parameters for the forward and backward pass of a given mini-batch. For example, in figure 3, in the forward pass, the mini-batch 5 in device 1 uses the weight updates from mini-batch 1 (w_1^1), but before the backward pass of mini-batch 5, the weight of the model in devices 1 updates from mini-batch 4 (w_1^4). So, in order to make sure the forward and backward pass of mini-batch 5 uses the same version of the model parameters, we need to store the intermediate state(inputs, activation and weights) of the mini-batch 5 after forward pass. This may cause the memory explosion when the model is huge or we have many blocks. For device k, it needs to store $K-k+1$ intermediate states.

weight staleness Here we suffer from the weight staleness. Different blocks in the DNN model suffer from different degrees of staleness. As shown in equation 1, for the weight updates of mini-batch t, the gradients in device k are computed with weights that are $K-k$ delayed. For example, in figure 3, mini-batch 5 in device 1 uses weight updates from mini-batch 1, while in device 2, 3 and 4 uses weights updates from mini-batch 2, mini-batch 3 and mini-batch 4 separately. The equation is $w^5 = w^4 - lr * \nabla f(w_1^1, w_2^2, w_3^3, w_4^4)$. The weight staleness may cause the statistical efficiency to decrease and at the same time may hurt the best accuracy the trained DNN can achieve.

$$w^t = w^{t-1} - lr * \nabla f(w_1^{t-K}, w_2^{t-K+1}, ..., w_K^{t-1}) \tag{1}$$

2.3 Features Replay(FR)

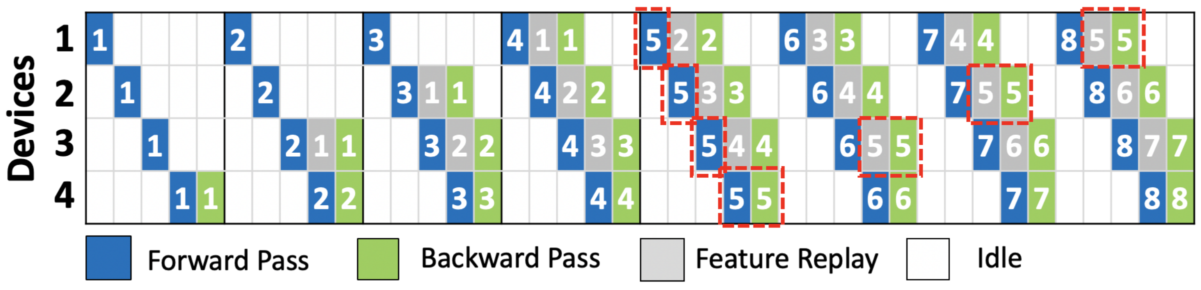


Figure 4: Feature replay(FR) parallel training with 4 devices

For Features Replay, after doing the forward pass of mini-batch t, instead of store all the intermediate states(inputs, activation and weights), we only store the inputs of each device. Then before doing the backward pass of this mini-batch t, we use the delayed inputs and the latest weight to recompute the forward pass to get the activation and then the gradient.

Take the mini-batch 5 in figure 4 as an example. In device 1, we perform the forward pass of the mini-batch 5 using the weight updated from mini-batch 1 (w_1^1), and store the inputs of mini-batch 5. Then after forward pass of mini-batch 8, we need to do the backward pass of the mini-batch 5. We use the latest weight which is updated from mini-batch 4 (w_1^4) and stored inputs of mini-batch 5 to recompute the activation and then calculate the gradient and update the weight.

2.3.1 improvements

parallel backward pass As shown in figure 4, the backward pass of feature replay(FR) is paralleled. So, the total computation time for a step is reduced to $T_F + \frac{T_B}{K}$(T_F and T_B are the forward pass time and backward time for a mini-batch in Naive sequential method)

lower memory consumption Unlike the DDG, where we need to store all the intermediate state of a mini-batch(inputs, activation and weights), Here we only need to store the inputs and recompute before the backward pass. Thus, we have the much lower memory consumption compared to DDG.

2.3.2 limitations

Forward and backward inconsistency We also take the mini-batch 5 as an example. In the forward pass of device 1, mini-batch 5 uses weight updated from mini-batch 1, while doing the backward pass of mini-batch 5, we recompute the forward pass of mini-batch 5 using the latest weight. As shown on Figure 4, for all devices, the latest weight before doing the backward pass of mini-batch 5 is updated from mini-batch 4. In this way, we don't have weight staleness in backward pass. However, the gradient is computed using a different set of weights than the ones used in the corresponding forward pass. We call this phenomenon forward and backward inconsistency.

2.4 PipeDream

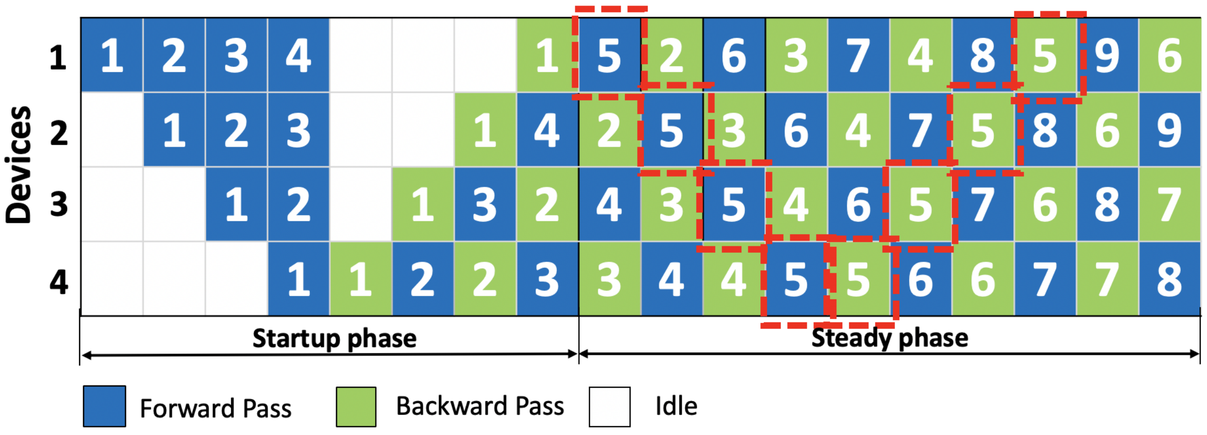


Figure 5: PipeDream parallel training with 4 devices

In the startup phase of the Pipedream, the first device loads K(K is the number of devices) mini-batches to keep the pipeline full in steady state. Once in steady state, each machine alternates between performing the forward and backward pass for a mini-batch.

2.4.1 improvements

Parallel the forward and backward pass At shown on figure 5, both forward pass and backward pass of PipeDream are paralleled. So, the total computation time for a step is further reduced to $\frac{T_F+T_B}{K}$(T_F and T_B are the forward pass time and backward time for a step in Naive sequential method)

2.4.2 limitations

The limitations of the PipeDream are the same as the DDG. First, there is huge memory consumption, because for device k we need to store $K-k+1$ intermediate states(inputs, activation and weights). Second, PipeDream also has weight staleness. We have the different weight versions used for a given mini-batch across blocks. For the mini-batch t, the gradients in device k are computed with weights that are $K-k$ delayed. The detailed analysis is almost the same as DDG and take mini-batch 5 as an example can make things clear, here, we don't rewrite again.

Method	**Computation Time**	**Memory**
Naive sequential method	$T_F + T_B$	1 minibatch
Delayed Gradient(DDG)	$T_F + \frac{T_B}{K}$	K-k+1 (k means device k)
Features Replay(FR)	$T_F + \frac{T_B}{K}$	Only store input for each device
PipeDream	$\frac{T_F+T_B}{K}$	K-k+1 (k means device k)

Table 1: Summary of four model parallelism method

3 Implementation

We implemented an unified model parallelism training framework using Pytorch and MPI [1]. In this framework we can run the above four model parallelism methods(Naive sequential, DDG, FR and PipeDream) on ResNet(used for image classification) and BERT(used for language modeling). While implementing this unified framework, I borrowed some code in these two repositories [9] [8].

3.1 Model partition

In order to apply the above four model parallelism methods on ResNet and BERT, the first step we need to do is the model partition. Based on the number of devices K we have, we need to partition the model into K parts. Figure 6a shows the basic building blocks of ResNet, Figure 6b shows the ResNet with N basic building blocks. According to[5], for example, ResNet20 consists of the first layer(3×3 convolutions), 9 basic building blocks and the final layer(pool and fc). In our partition algorithm, we treat ResNet20 to have 11 total layers(1 first layer, 9 basic blocks, 1 final layer), and do the uniform partition. If the number of devices is 3, device 1 will have layer from 1 to 4, device 2 will have layer from 5 to 8 and device 3 will have layer from 9 to 11.

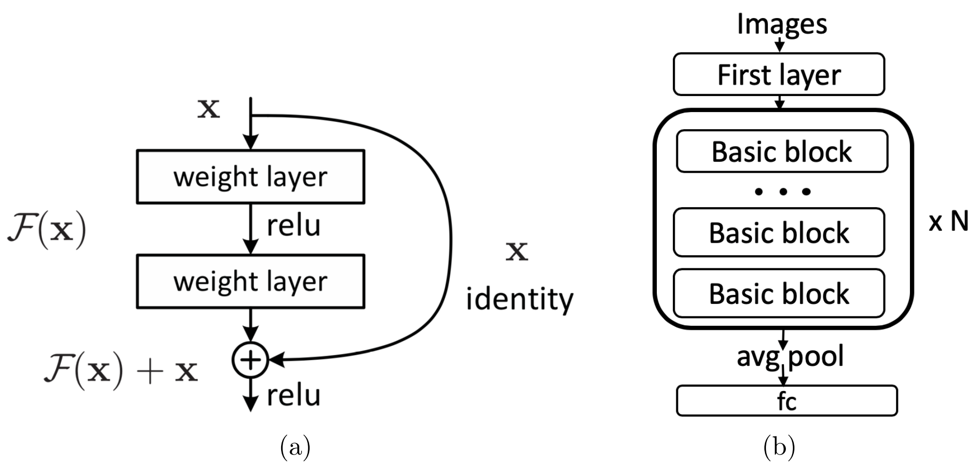


Figure 6: (a)Basic building block of ResNet, (b) ResNet with N basic blocks

Figure 7a shows the basic building blocks of BERT, named Encoder[10], Figure 7b shows the BERT with N Encoders. According to [3], for example the $BERT_{BASE}$ has 12 Encoders. In this case, if the number of devices K=3, device 1 will have the embedding layer and encoder from 1 to 4, device 2 will have encoder from 5 to 8 and device 3 will have encoder from 9 to 12.

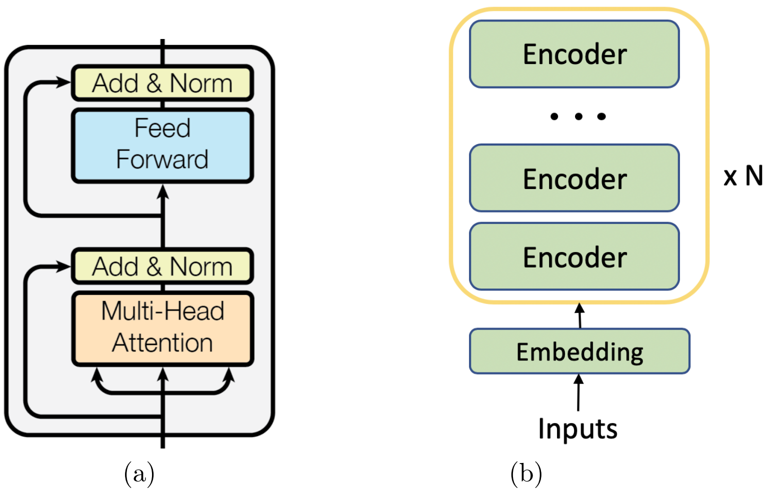


Figure 7: (a)Basic building block of BERT: Encoder, (b) BERT with N encoder layers

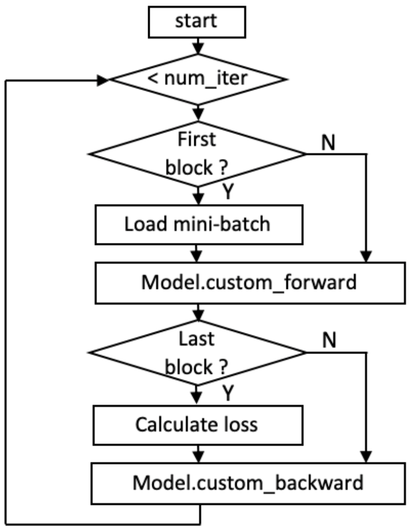
3.2 Activation and gradient passing

As shown on figure 1, in forward pass we need to pass the device output(activation) from the former device to the next device, while in backward pass, we need to pass gradient from the next device to the former device. We use the distributed package included in PyTorch [1] to deal with the message passing between clusters of the machines. One of the most elegant aspects of torch.distributed is its ability to abstract and build on top of different backends. Here we use the backend MPI [11].

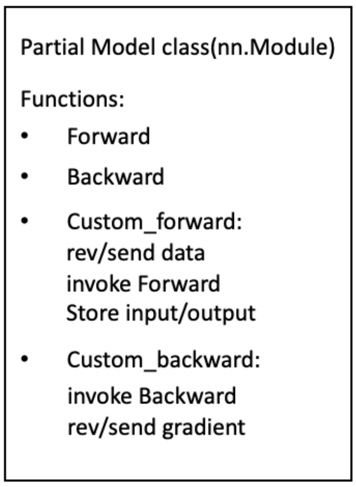
While using the torch.distributed package to receive the message, the receiver needs to know the shape of the message in advance. Since the batch size and the shape of the input data will not change during the different training steps, we can calculate the shape of each message between devices before the training begins. For example, in figure 1, we need to know the shape of the output(activation) of device 1, 2 and 3, and the shape of the gradients of the device 4 ,3 and 2. our framework will run a single mini-batch after creating the model and before the training starts to get the shape of each activation and gradient between devices.

3.3 Training schedule

The most important part of our unified model parallelism framework is to implement the parallel training schedule for the four model parallelism methods. Figure 8a shows the general training schedule, which is same for different models(either ResNet or BERT) and different parallelism methods. Figure 8b shows the partial of the model class. Forward and Backward functions are inherited from the model itself while the Custom_forward and Custom_backward functions are defined by ourselves and invoked by the general training schedule. They are different for each blocks and each model parallelism methods Table 2 3 4 5 show the detailed schedule used in our unified framework.



(a)



Partial Model class(nn.Module)

Functions:

- Forward
- Backward
- Custom_forward:
  rev/send data
  invoke Forward
  Store input/output
- Custom_backward:
  invoke Backward
  rev/send gradient

(b)

Figure 8: (a)General training schedule, (b) Partial model class. Forward and Backward functions are inherited from model itself, while Custom_forward and Custom_backward are defined by ourselves and different for each block.

Training schedule of Naive Sequential		
first device	**middle device** k	**last device**
	For step t **do**	**For** step t **do**
For step t **do**	**custom forward pass:**	**custom forward pass:**
get input h_0^t **from dataloader**	receive inputh_{k-1}^t from former device	get the target from the dataloader
custom forward pass:	do forward pass to get activation h_k^t	receive input h_{K-1}^t from former device
do forward pass to get activation h_1^t	send h_k^t to next device	do forward pass
send h_1^t to next device	**custom backward pass:**	**calculate loss based on target**
custom backward pass:	receive gradient δ_k^t from next device	**custom backward pass:**
receive gradient δ_1^t from next device	do the backward pass to update the weight	do the backward pass to update the weight
do the backward pass to update the weight	use h_{k-1}^t to get gradient δ_{k-1}^t	use h_{K-1}^t to get gradient δ_{K-1}^t
end for	send δ_{k-1}^t to former block	send δ_{K-1}^t to former block
	end for	**end for**

Table 2: The training schedule of the Naive sequential method

Table 2 shows the training schedule of naive sequential method. This schedule is easy to understand.

Training schedule of Delayed Gradient(DDG)		
first device	**middle device k**	**last device**
	Create input_deque = len($K-k+1$)	
	Create output_deque = len($K-k+1$)	
Create output_deque = len(K)	**For** step t **do**	
For step t **do**	**custom forward pass:**	**For** step t **do**
get input h_0^t from dataloader	receive input h_{k-1}^t from former device	**custom forward pass:**
custom forward pass:	store h_{k-1}^t in input_deque	get the target from dataloader
do forward pass to get activation h_1^t	do forward pass to get activation h_k^t	receive input h_{K-1}^t from former device
store h_1^t in output_deque for backward	store h_k^t in output_deque	do forward pass
send h_1^t to next device	send h_k^t to next device	**calculate loss based on target**
custom backward pass:	**custom backward pass:**	**custom backward pass:**
get stored gradient δ_1^{t-K+1}	get stored gradient δ_k^{t-K+k}	do the backward pass to update the weight
pop output h_1^{t-K+1} to do the backward	pop output h_k^{t-K+k} to do the backward	use h_{K-1}^t to get gradient δ_{K-1}^t
receive gradient δ_1^{t-K+2} from next device	pop input h_{k-1}^{t-K+k} to get δ_{k-1}^{t-K+k}	send δ_{K-1}^t to former block
end for	send gradient δ_{k-1}^{t-K+k} to former block	**end for**
	receive gradient $\delta_k^{t-K+k+1}$ from next device	
	end for	

Table 3: The training schedule of the Delayed Gradient(DDG) method

Table 3 shows the training schedule of DDG method. As discussed before, For device k, DDG needs to store $K-k+1$ intermediate states. While using Pytorch, we only need to store the output of the model and all the intermediate state(weight) of the computation graph will be stored automatically.

Training schedule of Feature Replay(FR)		
first device	**middle device k**	**last device**
	Create input_deque = len($K-k+1$)	
Create input_deque = len(K)	**For** step t **do**	
For step t **do**	**custom forward pass:**	**For** step t **do**
get input h_0^t from dataloader	receive input h_{k-1}^t from former device	**custom forward pass:**
custom forward pass:	store input in input_deque	get the target from dataloader
store h_0^t in input_deque	do forward pass to get activation h_k^t	receive input h_{K-1}^t from former device
do forward pass to get activation h_1^t	send h_k^t to next device	do forward pass
send h_1^t to next device	**custom backward pass:**	**calculate loss based on target**
custom backward pass:	get stored gradient δ_k^{t-K+k}	**custom backward pass:**
get stored gradient δ_1^{t-K+1}	pop input h_{k-1}^{t-K+1} recompute activation $\tilde{h}_k^t$ with w_k^t	do the backward pass to update the weight
pop input h_0^{t-K+1} recompute activation $\tilde{h}_1^t$ with w_1^t	use $\tilde{h}_k^t$ to do the backward	use h_{K-1}^t to get gradient δ_{K-1}^t
use $\tilde{h}_1^t$ to do the backward	use input h_{k-1}^{t-K+k} to get δ_{k-1}^{t-K+k}	send δ_{K-1}^t to former block
receive gradient δ_1^{t-K+2} from next device	send gradient δ_{k-1}^{t-K+k} to former block	**end for**
end for	receive gradient $\delta_k^{t-K+k+1}$ from next device	
	end for	

Table 4: The training schedule of the Feature Replay(FR) method

Table 4 shows the training schedule of FR method. Compared with DDG, instead of storing all the intermediate states(while implementing with Pytorch, simply store the output), we only store the input. Before doing the backward pass, we use stored input and latest weight to calculate output(activation) again.

Training schedule of PipeDream		
first device	**middle device k**	**last device**

Create output_deque = len(K)
For step t **do**
  get input h_0^t from dataloader
  custom forward pass:
    do forward pass to get activation h_1^t
    store h_1^t in output_deque for backward
    send h_1^t to next device
  custom backward pass:
    receive gradient δ_1^{t-K+1} from next device
    pop output h_1^{t-K+1} to do the backward
end for
	Create input_deque = len($K-k+1$)
	Create output_deque = len($K-k+1$)
	For step t **do**
	  custom forward pass:
	    receive input h_{k-1}^t from former device
	    send gradient $\delta_{k-1}^{t-K+k-1}$ to former block
	    store h_{k-1}^t in input_deque
	    do forward pass to get activation h_k^t
	    store h_k^t in output_deque
	    send h_k^t to next device
	  custom backward pass:
	    receive gradient δ_k^{t-K+k} from next device
	    pop output h_k^{t-K+k} to do the backward
	    pop input h_{k-1}^{t-K+k} to get and store δ_{k-1}^{t-K+k}
	end for
		For step t **do**
		  custom forward pass:
		    get the target from dataloader
		    receive input h_{K-1}^t from former device
		    send δ_{K-1}^{t-1} to former block
		    do forward pass
		  calculate loss based on target
		  custom backward pass:
		    do the backward pass to update the weight
		    use h_{K-1}^t to get and store grad δ_{K-1}^t
		end for

Table 5: The training schedule of the PipeDream method

Table 5 shows the training schedule of PipeDream method. Compared with DDG, We adjust recv/send schedule in order to parallel the forward pass. But the basic idea is almost the same as the DDG.

4 Evaluation

This section compares the four model parallel training methods(Naive sequential, DDG, FR and PipeDream) by runing on two tasks: 1) ResNet used for image classification, 2) BERT used for language modeling.

4.1 ResNet for image classification

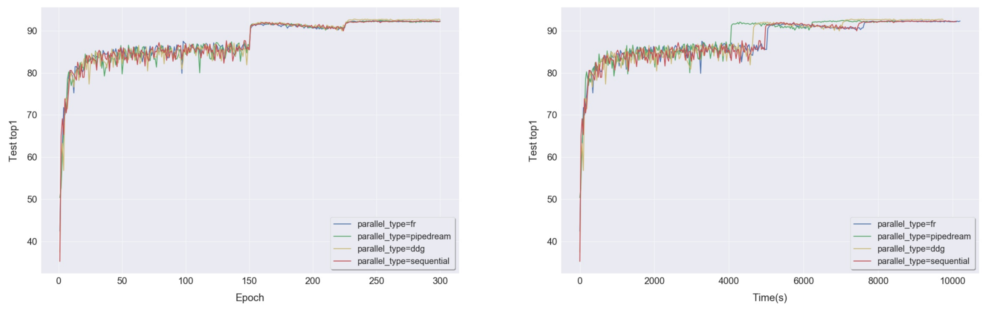
We evaluate different training methods with ResNet20 on image classification benchmark datasets: CIFAR-10. We use SGD with the momentum of 0.9, Each model is trained using batch size 128 for 300 epochs and the stepsize is divided by a factor of 10 at 150 and 225 epochs. The weight decay constant is set to 1×10^{-4}. All our experiment run on GPU GeForce GTX Titan X. Though we can put different blocks into different GPUs, here limited by the GPU resources, we put different blocks into different processes and put all the processes into 1 GPU.

In the experiment, we partition the model into different number of blocks(2 blocks, 3 blocks, 4 blocks, 6 blocks and 11 blocks). For each number of blocks, we train the model separately to see the effect of the partition number on the final results. We train the model under different learning rates in order to find the best one. We run three times for each training parameters and calculate the average and standard deviation of the top1 accuracy under the best learning rate. The result is shown on table 6. When the number of blocks is 2, DDG, FR and PipeDream can reach almost the same top1 accuracy as Naive Sequential method. But when the number of blocks increase, the loss of the top1 accuracy can't be ignored. When the number of block is 3,4,and 6, FR performs worse than DDG and PipeDream. But when the number of block is 11, FR seems to perform better than DDG and PipeDream. This is an interesting phenomenon, and I think it is worth exploring further by using ResNet110 in future work.

num of blocks	**Sequential**		**DDG**		**FR**		**PipeDream**	
	top1	**lr**	**top1**	**lr**	**top1**	**lr**	**top1**	**lr**
2 blocks	92.622 ± 0.176	0.25	92.695 ± 0.113	0.32	92.555 ± 0.181	0.25	92.541 ± 0.012	0.20
3 blocks	92.471 ± 0.187	0.32	92.515 ± 0.031	0.20	$\mathbf{92.364} \pm 0.029$	0.20	92.458 ± 0.041	0.32
4 blocks	92.551 ± 0.144	0.25	$\mathbf{92.258} \pm 0.128$	0.25	$\mathbf{92.388} \pm 0.037$	0.32	92.521 ± 0.233	0.32
6 blocks	92.762 ± 0.151	0.32	$\mathbf{92.258} \pm 0.121$	0.32	$\mathbf{92.157} \pm 0.087$	0.25	$\mathbf{92.288} \pm 0.225$	0.25
11 blocks	92.598 ± 0.164	0.32	$\mathbf{90.728} \pm 1.511$	0.16	$\mathbf{92.241} \pm 0.144$	0.32	$\mathbf{91.727} \pm 0.393$	0.25

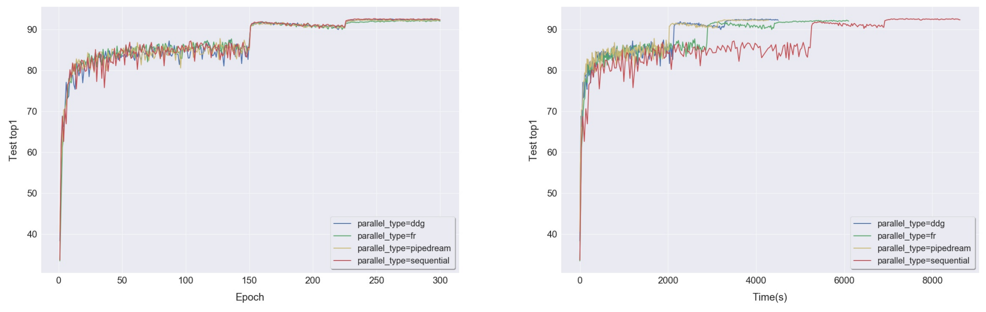
Table 6: The best top1 accuracy of different methods of different blocks. The bold part indicates that the top1 accuracy is lower than others.

The Figure 9 10 11 12 13 shows the top1 vs epochs and top1 vs times when ResNet20 split into 2, 3, 4, 6, 11 blocks separately. And here we fixed the learning rate to be 0.32. We can conclude from these figures:(1) the impact of DDG, FR and PipeDream on statistical efficiency is negligible, which means the number of epoches to reach a given accuracy is almost the same. (2) the hardware efficiency is improved. Under most of the circumstances, PipeDream > DDG > FR > Sequential, which means the time needed to train an epoch decrease. But what we need to pay attention here is that when we train the model on GPU, the GPU also has many other processes run by other colleagues, so the time needed to train the model may not accurate at all. (3) When the number of block is 11, both DDG and PipeDream divergence, while FR still convergence, which is an interesting phenomenon and needed exploring further.



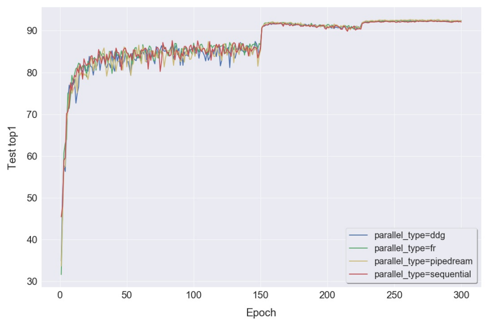
(a) (b)

Figure 9: ResNet20 split to 2 blocks, lr is 0.32, (a) top1 vs. epochs, (b) top1 vs times

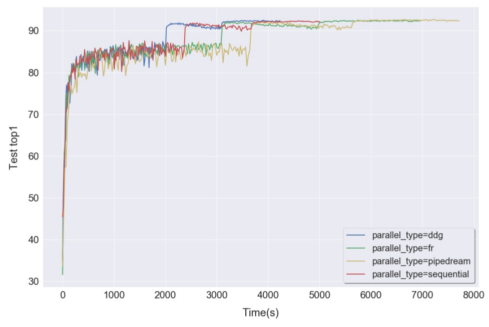


(a) (b)

Figure 10: ResNet20 split to 3 blocks, lr is 0.32, (a) top1 vs. epochs, (b) top1 vs times

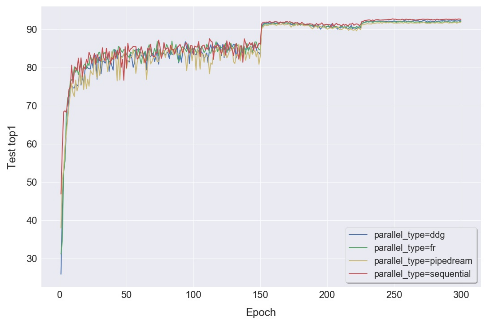


(a)

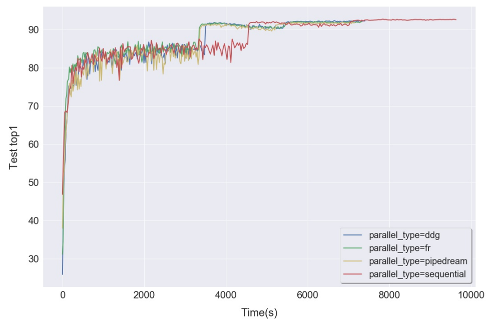


(b)

Figure 11: ResNet20 split to 4 blocks, lr is 0.32, (a) top1 vs epochs, (b) top1 vs times

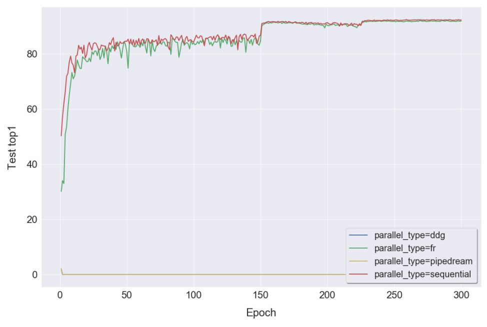


(a)

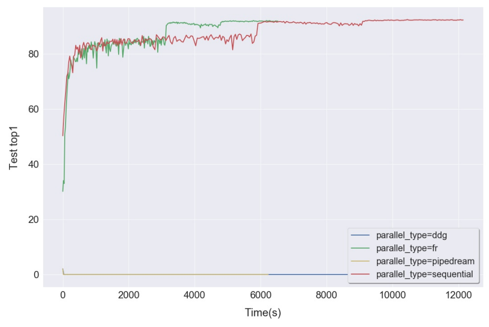


(b)

Figure 12: ResNet20 split to 6 blocks, learning rate is 0.32, (a) top1 vs. epochs, (b) top1 vs times



(a)



(b)

Figure 13: ResNet20 split to 11 blocks, learning rate is 0.32, (a) top1 vs. epochs, (b) top1 vs times

4.2 BERT for language modeling

First we construct the simplified BERT. We set the number of Encoder layers L=3, the hidden size H=768, the number of self-attention heads A=12, the feed-forward size to be 4H. The total parameters of this simplied BERT are about 69MB. We partition this simplifed BERT into 3 blocks, and put each block into a process. We can put 3 processes into 3 GPUs, but limited by our computation resources, we put these 3 processes into 1 GPU. The GPU type we use is GeForce GTX Titan X.

According to [3], we also pretrain this simplified BERT using two novel unsupervised prediction tasks: 1) Masked Language Modeling, we mask some percentage of the input tokens at random, and then predicting only those masked tokens. According to [3], we set this percentage to be 15%. 2) Next Sentence Prediction, we pretrain a binarized next sentence prediction task. When choosing the sentences A and B, 50% of the time B is the truly next sentence, while 50% of the time not.

The dataset we use is the wikipedia training data. We follow the wikipedia data extraction process in this repository [2] to generate our train and test dataset. Limited by our GPU resources, we only choose part of the whole dataset. The dataset size is 11MB.

We train with batch size of 4 sentences(limited by our GPU memory size, in [3] batch size is 256)for 200000 steps, which is about 2 epoches. Each sentence length is 512. We use Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$. The weight decay constant is set to be 1×10^{-2} and the learning rate is set to be 1×10^{-4}. The learning rate warmup over the first 9900 steps, and linear decay of the learning rate. The training loss is the sum of the mean masked LM CrossEntropy and mean next sentence prodiction CrossEntropy.

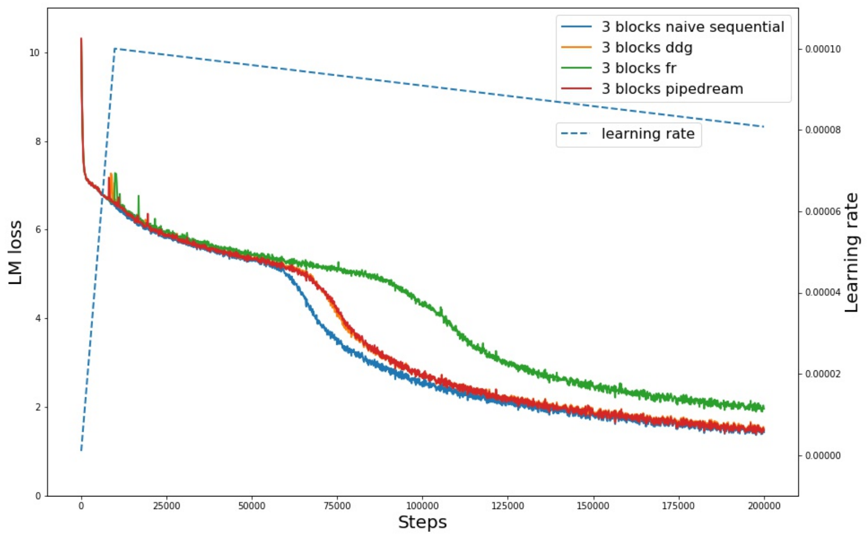
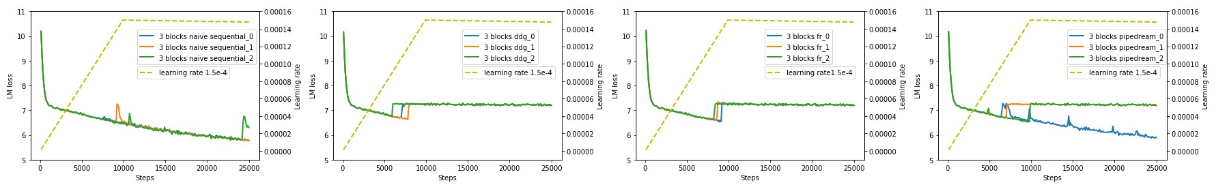


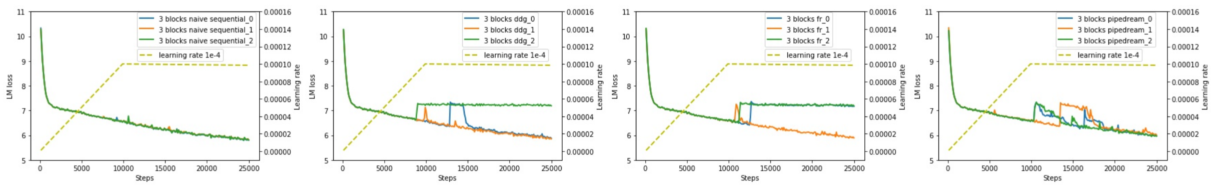
Figure 14: LM Loss vs. Steps for simplifed BERT with learning rate $1e-4$, Between steps 0-25000, the LM loss curve has some shocks

Since the loss of the next sentence is almost 0, we only analysis LM loss here. Figure 14 shows the LM loss vs. steps for simplied BERT with learning rate 1×10^{-4}. We treat the Naive sequential as the baseline. Compared with Naive sequential, both DDG and PipeDream can reach almost the same LM train loss at the end, but the statistical efficiency is reduced, which means in order to reach a given LM train loss, DDG and PipeDream need more steps. FR performs worse than the DDG and PipeDream, FR can't reach the same LM train loss as the Naive Sequential loss at the end and need more steps than DDG and PipeDream to reach a given LM train loss. When We focus on steps from 0-25000 in figure 14, we find that the drop curve of LM loss produced some shocks when the learning rate close to the set learning rate. In order to learn more about this phenomenon, We set our learning rate to be 1.5×10^{-4}, 1×10^{-4}, 0.5×10^{-4}. For each learning rate we run 3 times for our four model parallelism training methods and each time we run 25000 Steps. The result is shown on figure 15,16,17.



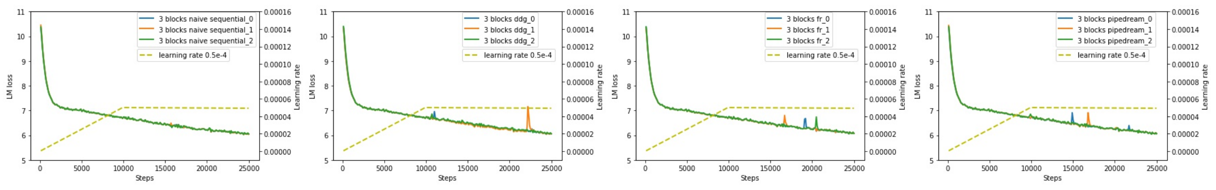
(a) (b) (c) (d)

Figure 15: (a)Naive sequential with learning rate 1.5×10^{-4}, (b)DDG with learning rate 1.5×10^{-4}, (c)FR with learning rate 1.5×10^{-4}, (d)PipeDream with learning rate 1.5×10^{-4}



(a) (b) (c) (d)

Figure 16: (a)Naive sequential with learning rate 1×10^{-4}, (b)DDG with learning rate 1×10^{-4}, (c)FR with learning rate 1×10^{-4}, (d)PipeDream with learning rate 1×10^{-4}



(a) (b) (c) (d)

Figure 17: (a)Naive sequential with learning rate 0.5×10^{-4}, (b)DDG with learning rate 0.5×10^{-4}, (c)FR with learning rate 0.5×10^{-4}, (d)PipeDream with learning rate 0.5×10^{-4}

As shown on figure 15, when learning rate is 1.5×10^{-4}, Naive sequential convergences all three times, DDG and FR divergences all three times, and PipeDream convergence once and divergence twice.

As shown on figure 16, when learning rate is 1×10^{-4}, Naive sequential convergences all three times, DDG convergences twice and divergences once, FR convergences once and divergences twice and PipeDream convergences all three times.

As shown on figure 17, when learning rate is 0.5×10^{-4}, all four parallelism training methods convergence all three times.

This experiment shows that when training BERT with DDG,FR or PipeDream, the learning rate may can't be too big in order to make the loss convergence. FR seems most easily to divergence, which means forward and backward inconsistency will impact the training process most. DDG and PipeDream seem to perform better than FR but worse than Naive sequential method, which means weight staleness also impact the training process.

5 Conclusion

We first analysis the improvements and limitations of four model parallelism training methods. We use Naive Sequential parallelism training method as baseline. DDG and FR parallel the backward pass, PipeDream parallel both forward and backward pass. DDG and PipeDream suffer from weight staleness, while FR suffer from forward and backward inconsistency. In order to futher understand these four methods, we implement an unified framework to run these four methods on ResNet20 used for image classification and simplified BERT used for language modeling. From the experiment, we can conclude:

- For resnet20, DDG, FR and PipeDream will reduce the best accuracy of top1 slightly, and FR is slightly worse than PipeDream and DDG.
- For ResNet20, When the model is divided into more blocks, the loss of top1 accuracy is more serious.
- For ResNet20, the impact of DDG, FR and PipeDream on statistical efficiency is negligible, which means the number of epoches to reach a given accuracy is almost the same.
- DDG, FR and PipeDream increase the hardware efficiency, so we need less time to train an epoch.
- For simplified BERT model, DDG, FR and PipeDream will reduce the statistical efficiency. The number of steps needed to reach a given LM Loss is increased and FR is much worse than DDG and PipeDream.
- For simplified BERT model, when the learning rate is big($\geq 1 \times 10^{-4}$)) or change fast, DDG, FR and PipeDream may cause divergence. And FR is worse than DDG and PipeDream.

References

[1] Séb Arnold. Writing distributed applications with pytorch. `https://pytorch.org/tutorials/intermediate/dist_tuto.html`, 2017.

[2] Giuseppe Attardi. A tool for extracting plain text from wikipedia dumps. `https://github.com/attardi/wikiextractor`, 2019.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv e-prints*, page arXiv:1810.04805, Oct 2018.

[4] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv e-prints*, page arXiv:1806.03377, Jun 2018.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, Dec 2015.

[6] Zhouyuan Huo, Bin Gu, and Heng Huang. Training Neural Networks Using Features Replay. *arXiv e-prints*, page arXiv:1807.04511, Jul 2018.

[7] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. Decoupled Parallel Backpropagation with Convergence Guarantee. *arXiv e-prints*, page arXiv:1804.10574, Apr 2018.

[8] NVIDIA. Ongoing research training transformer language models at scale, including: Bert. `https://github.com/NVIDIA/Megatron-LM`, 2019.

[9] slowbull. A pytorch implementation of the paper "decoupled parallel backpropagation with convergence guarantee". `https://github.com/slowbull/DDG`, 2018.

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *arXiv e-prints*, page arXiv:1706.03762, Jun 2017.

[11] Wikipedia. `https://en.wikipedia.org/wiki/Message_Passing_Interface`.