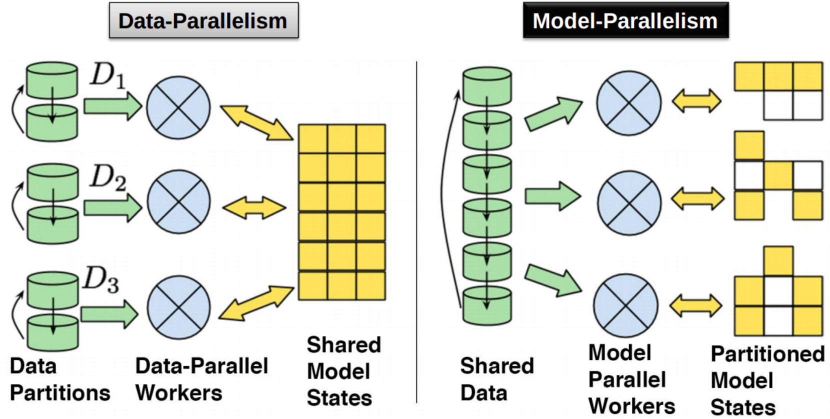
Finding an efficient way to parallelise Pytorch deep-learning models

Supervisor: Arya Mazaheri M.Sc.



Outline

Parallel Programming

Introduction

- Data Parallelism and Model Parallelism in deep learning.
- Data Parallelism is well-supported by typical deep learning frameworks.
- Model Parallelism is yet widely to be addressed, except for manual strategies.
- Training on heterogeneous systems, e.g. Clusters of servers.
- Some frameworks does this, but require rewriting old models.

Contributions

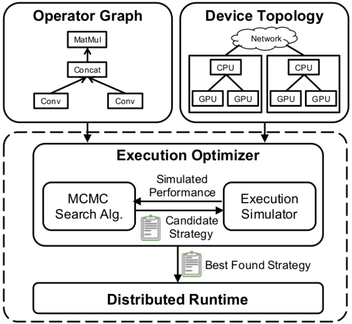
Making use of automatic strategy finding process and applying this result to implement a corresponding parallelised model with PyTorch distributed library.

Related Work

- AlexNet [1] - manual, low-level data parallelism with CUDA in C/C++.
- DistBelief [2] - manual, low-level model parallelism on different dedicated machines with C/C++.
- Mesh-Tensorflow [3] - automatic, requires rewriting models in a Tensorflow-like framework.
- Megatron-LM [4] - automatic, uses PyTorch, restricts to language models.
- FlexFlow [5] - automatic, uses Legion (CUDA, cuDNN in C/C++), under development, can determine the best parallel strategy.

Execution Optimiser of FlexFlow

- An *operator graph* $\mathcal{G}$ to describe all operators and state in a deep neural network. Each node $o_i \in \mathcal{G}$ is an operator.
- A *device topology* $\mathcal{D} = (\mathcal{D}_N, \mathcal{D}_E)$ describing all available hardware devices and their interconnections.
- A parallelisation strategy $\mathcal{S}$ describes one possible parallelisation of an application.

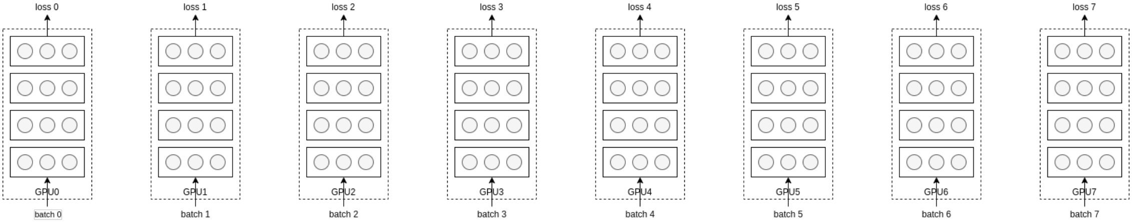


FlexFlow overview.

Data and Model Parallelism with PyTorch: Three different libraries from PyTorch

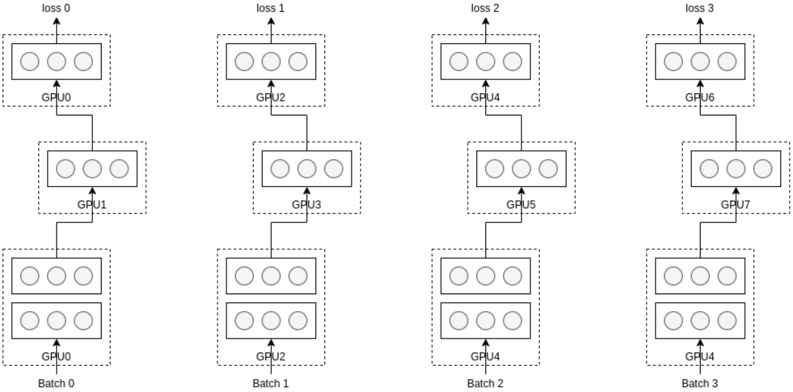
	Multi-GPU	Multi-machine	Customised Gradient Flow
`DataParallel` [6]	yes	no	no
`DistributedDataParallel` [7]	yes	yes	no
`torch.distributed` [8]	yes	yes	yes

Data and Model Parallelism with PyTorch: Data Parallelism in `DataParallel`



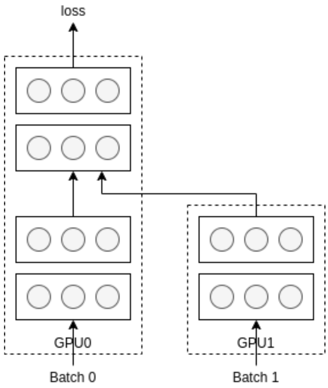
Multi-GPU which `DataParallel` only support.

Data and Model Parallelism with PyTorch: Model Parallelism in `DistributedDataParallel`



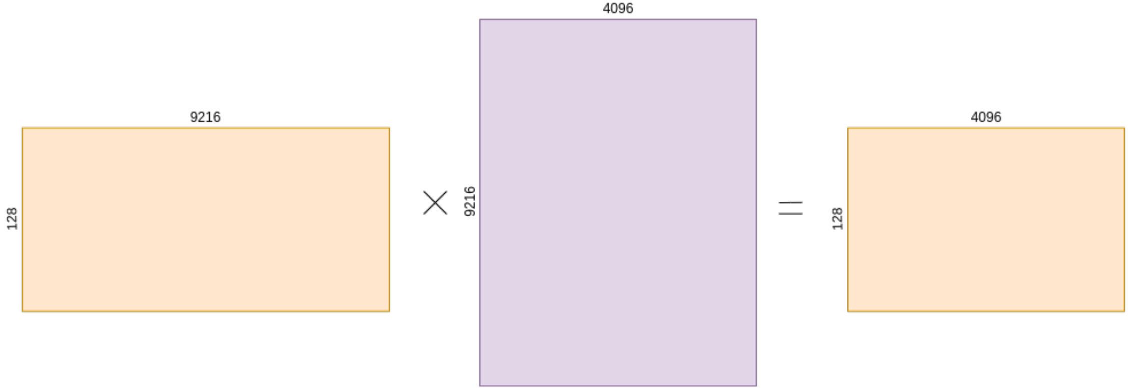
Multi-machine which `DistributedDataParallel` only support.

Gradient Flow Customisation for the combination of Model and Data parallelism from `torch.distributed`



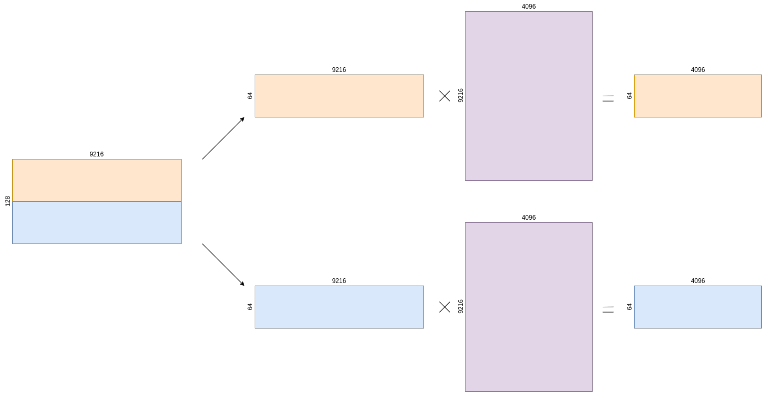
`torch.distributed` supports everything flexibly.

Parallelisation of Fully-connected Layer: Fully-connected Layer as Matrix Multiplication



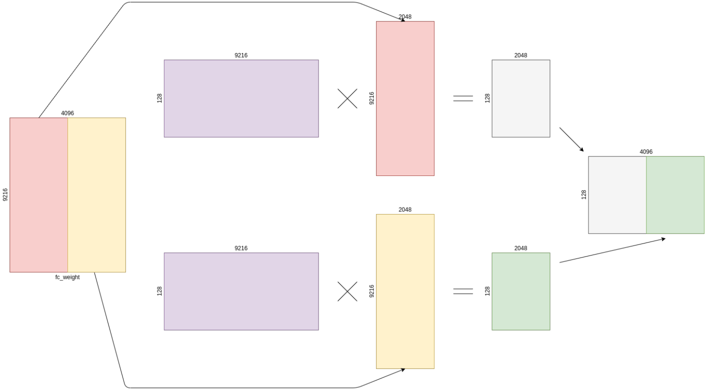
Fully-connected layer computation is interpreted as matrix multiplication.

Parallelisation of Fully-connected Layer: Data Parallelism with the first dimension



Data parallelism in fully-connected layer.

Parallelisation of Fully-connected Layer: Model Parallelism with the second dimension



Model parallelism in fully-connected layer.

Best Found Strategy: Parallelisation Strategy for AlexNet by FlexFlow’s Execution Optmiser

```
op[0] conv1:    dim(1 ) gpu(0 )
op[1] pool2:    dim(1 ) gpu(0 )
op[2] conv3:    dim(1 ) gpu(0 )
op[3] pool4:    dim(1 ) gpu(0 )
op[4] conv5:    dim(1 ) gpu(0 )
op[5] conv6:    dim(1 ) gpu(0 )
op[6] conv7:    dim(1 ) gpu(0 )
op[7] pool8:    dim(1 ) gpu(0 )
op[8] flat:     dim(1 ) gpu(0 )
op[9] linear9:  dim(1 1 ) gpu(0 )
op[10] linear10:        dim(1 1 ) gpu(0 )
op[11] linear11:        dim(1 1 ) gpu(0 )
```

Layer placement for one GeForce 1080Ti GPU only.

```
op[0] conv1:     dim(2 ) gpu(0 1 )
op[1] pool2:     dim(2 ) gpu(0 1 )
op[2] conv3:     dim(2 ) gpu(0 1 )
op[3] pool4:     dim(2 ) gpu(0 1 )
op[4] conv5:     dim(2 ) gpu(0 1 )
op[5] conv6:     dim(2 ) gpu(0 1 )
op[6] conv7:     dim(2 ) gpu(0 1 )
op[7] pool8:     dim(2 ) gpu(0 1 )
op[8] flat:      dim(2 ) gpu(0 1 )
op[9] linear9:   dim(2 1 ) gpu(0 0 )
op[10] linear10:        dim(2 1 ) gpu(0 0 )
op[11] linear11:        dim(2 1 ) gpu(0 0 )
```

Layer placement for two identical GeForce 1080Ti GPUs.

Strategy with Customised Gradient Flow

Conv2D layers are replicated on two GPUs, but full-connected layers are on one GPU0 only!

Implementation with `torch.distributed`: Layer (Parameter) placement

```
Rank:  1 , name:  conv1.weight , device:  1
Rank:  0 , name:  conv1.weight , device:  0
Rank:  0 , name:  conv1.bias , device:  0
Rank:  1 , name:  conv1.bias , device:  1
Rank:  0 , name:  conv2.weight , device:  0
Rank:  1 , name:  conv2.weight , device:  1
Rank:  1 , name:  conv2.bias , device:  1
Rank:  0 , name:  conv2.bias , device:  0
Rank:  1 , name:  fc1.weight , device:  1
Rank:  0 , name:  fc1.weight , device:  0
Rank:  0 , name:  fc1.bias , device:  0
Rank:  1 , name:  fc1.bias , device:  1
Rank:  1 , name:  fc2.weight , device:  1
Rank:  0 , name:  fc2.weight , device:  0
Rank:  0 , name:  fc2.bias , device:  0
Rank:  1 , name:  fc2.bias , device:  1
```

Parameter placement for typical data parallelism.

```
Rank:  0 , name:  conv1.weight , device:  0
Rank:  0 , name:  conv1.bias , device:  0
Rank:  0 , name:  conv2.weight , device:  0
Rank:  0 , name:  conv2.bias , device:  0
Rank:  0 , name:  fc1.weight , device:  0
Rank:  0 , name:  fc1.bias , device:  0
Rank:  0 , name:  fc2.weight , device:  0
Rank:  0 , name:  fc2.bias , device:  0
Rank:  0 , count_param:  8
Rank:  1 , name:  conv1.weight , device:  1
Rank:  1 , name:  conv1.bias , device:  1
Rank:  1 , name:  conv2.weight , device:  1
Rank:  1 , name:  conv2.bias , device:  1
Rank:  1 , name:  fc1.weight , device:  0
Rank:  1 , name:  fc1.bias , device:  0
Rank:  1 , name:  fc2.weight , device:  0
Rank:  1 , name:  fc2.bias , device:  0
```

Parameter placement for customised parallelism.

Fully-connected layer placement in strategy with Customised Gradient Flow

Conv2D layers are replicated on two GPUs, but full-connected layers are on one GPU0 only!

Implementation with `torch.distributed`: Backpropagation and Loss

```
(env) nguyen@gpuserver:~$ python train_dist_orig.py
Rank  0 , epoch  0 :  tensor(1.3128, device='cuda:0')
Rank  1 , epoch  0 :  tensor(1.3088, device='cuda:1')
Rank  1 , epoch  1 :  tensor(0.5501, device='cuda:1')
Rank  0 , epoch  1 :  tensor(0.5536, device='cuda:0')
Rank  0 , epoch  2 :  tensor(0.4317, device='cuda:0')
Rank  1 , epoch  2 :  tensor(0.4219, device='cuda:1')
Rank  0 , epoch  3 :  tensor(0.3529, device='cuda:0')
Rank  1 , epoch  3 :  tensor(0.3471, device='cuda:1')
Rank  1 , epoch  4 :  tensor(0.3128, device='cuda:1')
Rank  0 , epoch  4 :  tensor(0.3188, device='cuda:0')
Rank  1 , epoch  5 :  tensor(0.2913, device='cuda:1')
Rank  0 , epoch  5 :  tensor(0.2901, device='cuda:0')
Rank  0 , epoch  6 :  tensor(0.2697, device='cuda:0')
Rank  1 , epoch  6 :  tensor(0.2650, device='cuda:1')
Rank  0 , epoch  7 :  tensor(0.2507, device='cuda:0')
Rank  1 , epoch  7 :  tensor(0.2498, device='cuda:1')
Rank  1 , epoch  8 :  tensor(0.2344, device='cuda:1')
Rank  0 , epoch  8 :  tensor(0.2343, device='cuda:0')
Rank  0 , epoch  9 :  tensor(0.2266, device='cuda:0')
Rank  1 , epoch  9 :  tensor(0.2222, device='cuda:1')
```

Loss convergence in typical data parallelism.

```
(env) nguyen@gpuserver:~$ python train_dist.py
Rank  0 , epoch  0 :  tensor(1.3312, device='cuda:0')
Rank  1 , epoch  0 :  tensor(1.3234, device='cuda:0')
Rank  1 , epoch  1 :  tensor(0.5594, device='cuda:0')
Rank  0 , epoch  1 :  tensor(0.5695, device='cuda:0')
Rank  1 , epoch  2 :  tensor(0.4242, device='cuda:0')
Rank  0 , epoch  2 :  tensor(0.4403, device='cuda:0')
Rank  0 , epoch  3 :  tensor(0.3590, device='cuda:0')
Rank  1 , epoch  3 :  tensor(0.3645, device='cuda:0')
Rank  0 , epoch  4 :  tensor(0.3245, device='cuda:0')
Rank  1 , epoch  4 :  tensor(0.3161, device='cuda:0')
Rank  0 , epoch  5 :  tensor(0.2962, device='cuda:0')
Rank  1 , epoch  5 :  tensor(0.2851, device='cuda:0')
Rank  1 , epoch  6 :  tensor(0.2654, device='cuda:0')
Rank  0 , epoch  6 :  tensor(0.2758, device='cuda:0')
Rank  0 , epoch  7 :  tensor(0.2530, device='cuda:0')
Rank  1 , epoch  7 :  tensor(0.2565, device='cuda:0')
Rank  1 , epoch  8 :  tensor(0.2373, device='cuda:0')
Rank  0 , epoch  8 :  tensor(0.2389, device='cuda:0')
Rank  0 , epoch  9 :  tensor(0.2310, device='cuda:0')
Rank  1 , epoch  9 :  tensor(0.2217, device='cuda:0')
```

Loss convergence in customised parallelism.

Loss calculation in strategy with Customised Gradient Flow

Both tensors which perform loss calculation are resided on only GPU0!

Conclusions and Future Work

Conclusions:

- Proposed a process how to take into account the parallelisation strategy when applying parallelism in training PyTorch models.
- FlexFlow's execution optimiser could be used to determine the best parallelisation strategy.
- `torch.distributed` suits the most to implement the best found strategy.

Future Work:

- Further investigation into FlexFlow's execution optimiser: initiallisation strategy, optimisation step, etc.
- Implementing more complicated models with `torch.distributed`.
- Developing a Python wrapper for such execution optimiser to turn any PyTorch model into its customised version.

References I

[1] A. Krizhevsky, I. Sutskever and G. E. Hinton, ‘Imagenet classification with deep convolutional neural networks’, in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, ‘Large scale distributed deep networks’, in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[3] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, ‘Mesh-tensorflow: Deep learning for supercomputers’, in *Advances in Neural Information Processing Systems*, 2018, pp. 10 414–10 423.

References II

[4] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper and B. Catanzaro, 'Megatron-lm: Training multi-billion parameter language models using gpu model parallelism', *arXiv preprint arXiv:1909.08053*, 2019.

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

[6] S. Kim and J. Kang, *Optional: Data Parallelism — PyTorch Tutorials 1.5.1 documentation*, [Online]. Available: `https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html` (visited on 06/07/2020).

Parallel Programming

References III

[7] S. Li, *Getting Started with Distributed Data Parallel — PyTorch Tutorials 1.5.1 documentation*, [Online]. Available: `https://pytorch.org/tutorials/intermediate/ddp_tutorial.html` (visited on 06/07/2020).

[8] S. Arnold, *Writing Distributed Applications with PyTorch — PyTorch Tutorials 1.5.1 documentation*, [Online]. Available: `https://pytorch.org/tutorials/intermediate/dist_tuto.html` (visited on 06/07/2020).