Benchmarking Distributed Training and Inference on NVIDIA's A6000 GPUs

Tanushree Banerjee[*]
tb21@princeton.edu

Vedant Shah[*]
vds@princeton.edu

1 Introduction

Today's deep learning models are being trained on increasingly large datasets, making single-GPU training infeasible [1]. For example, GPT-3 [2], OpenAI's large language model released in 2020, consists of 175 billion parameters requiring 34 days to train using 1024 expensive GPUs [3]. Thus, training today's deep learning models requires us to develop efficient distributed/parallel training techniques.

There are several existing paradigms to parallelize training, with data parallelism being the most common paradigm of parallelism due to its simplicity. Under this paradigm, the dataset is split into several shards and placed on each available device, such as a single GPU or CPU. Each device holds a copy of the model replica and trains on the dataset shard allocated. After every training iteration, the results are aggregated and averaged across the devices after each training iteration for the model parameters to remain synchronized.

Project Goal. Our project aims to benchmark distributed training performance on the Computer Science Department's ionic cluster GPUs. Our project consists of two distinct parts, detailed as follows.(i) We train out-of-the-box Computer Vision models from `torchvision` using PyTorch's native `nn.DataParallel` module for distributed training. We compare the computation and communication times averaged over several thousand iterations along with the total data transfer size in each epoch for different batch sizes, backbone models, and parallelism degrees. We then attempt to reason about how different batch sizes, backbone models, and parallelism degrees affect the system performance, as measured by the average computation and communication time in each epoch, as well as the total data transfer size in each epoch. (ii) We implement our own `VanillaDataParallel` module using communication primitives implemented in PyTorch and compare its performance against PyTorch's native `nn.DataParallel` as measured by the average computation and communication time in each epoch, as well as the no.of epochs needed to reach a minimum threshold loss amount. Based on our observations, we attempt to infer what additional optimizations PyTorch might have implemented.

Other Project Deliverables. Our project code can be found in this GitHub repository. Instructions for running our code are provided in README.md. A video for our project is provided here, with slides used in the video provided here.

2 System design

Benchmarking Native PyTorch `DataParallel`. We train models defined in **`src/models.py`**. The construction of these models is detailed in section 3) on the MNIST classification task [4].

We implement a `DistributedTrainer` class in **`src/distributed_trainer.py`** to implement the training script, with Python's `time` package used to record the time taken for computation and

[*]denotes equal contribution.

communication. The operations timed in each of the computation and communication steps are detailed as follows. (i) **Computation time**: this includes (i) resetting the gradients for all the gradients computed with respect to the trainable model params (in this case, only the classifier module), (ii) Running inference, i.e. a forward pass over the current batch in the iteration, (iii) computing the loss with respect to the prediction and the ground truth labels for the data in the current iteration, (iv) running the backward pass to compute the gradients with respect to the loss computed in the previous step, (v) updating the parameters of the model with respect to the loss (ii) **Communication time**: this step includes running `torch.distributed.barrier()` [5] to synchronize the weights of the model with updated parameters across all devices based on the most recent batch losses.

In addition, we record the total loss in each iteration. The data transfer size in each epoch is given by the number of trainable parameters in the model multiplied by the number of times gradients are synchronized in each epoch between the copies of the models on each GPU node. Thus, we compute the total data transfer size in each epoch as follows, where N is the total number of images in the training split, B is the batch size, and k is the number of GPU nodes used is given as follows: Total Data transfer size in each epoch $= \frac{N}{B \times k} \times$ number of trainable parameters (see Appendix 6a for more detail)

We run our experiments, using the main script, **`benchmark.py`**. This script allows us to vary the batch size (as a command line argument), number of GPUs (by specifying device IDs as a command line argument), backbone feature extractor models (by modifying the model used in the script manually, as described int he instructions to run our code), and number of trainable parameters (by modifying the size of the hidden layer in the classifier module of each model specified as a command line argument).

We use the PyTorch module `torch.distributed.run` [5], which is a helper utility function implemented by PyTorch to spawn multiple distributed training processes per training node. This helper function assigns worker `RANK` and `WORLD_SIZE` automatically. We set the number of processes spawned per training node to 2. The exact command template used to run our main training benchmarking script is given in our instructions to run our code, and the `README.md` of our GitHub repository.

Comparing `VanillaDataParallel` against PyTorch's native `DataParallel` module. We implement our own custom `VanillaDataParallel` module in `src/data_parallel.py`. Our custom implementation explicitly calls the following communication primitives, as implemented by PyTorch in the `nn.parallel` package [6], for data parallel distributed training: (i) `nn.parallel.replicate` The model is duplicated across the multiple GPUs; (ii) `nn.parallel.scatter`: Distributes the input in the first dimension across several GPUs; (iii) `nn.parallel.gather`: Gathers and concatenates the input in the first dimension the outputs across the GPUs; (iv) `nn.parallel.parallel_apply`: Apply a set of already distributed inputs to a set of already distributed models.

We compare the performance of our VanillaDataParallel against PyTorch's native DataParallel using the script `vanilla_vs_native_comparison.py`. In this script, we only benchmark the distributed inference performance (rather than the training performance) and compare performance against different batch sizes (32, 64, 128, 256). We measure performance using computation time (time taken for each forward pass) and communication time (time taken to run torch.cuda.synchronize() to make sure all inputs/outputs are appropriately moved from GPU to CPU and combined.).

3 Implementation details

Dataset and task. We train our models on the MNIST (Modified National Institute of Standards and Technology) dataset [4]. This dataset consists of several thousand images of handwritten digits, with 60,000 images in the training split and 10,000 images in the testing split [4]. We train our out-of-the-box models on the MNIST classification task, i.e. classifying the images of handwritten digits. We use PyTorch's implementation of Stochastic Gradient Descent (SGD) [7] with a learning rate of 0.01 and a momentum of 0.9. We use Cross Entropy loss as our loss function.

Dataset preprocessing. We first apply PyTorch transforms, `transforms.ToTensor()` followed by `transforms.Normalize()` to convert images to tensors and normalize pixel values. We then resize these grayscale images (28x28 pixels) to match the expected input dimensions of each model

GPUs and Models We use a variable number of Nvidia's A6000 GPUs [8] from the Princeton University Department of Computer Science ionic cluster. We evaluate the performance of PyTorch's native `DataParallel` module [9] over several out-of-the-box computer vision backbone models from the `torchvision` model zoo. To train models on the MNIST classification task [4], we use these out-of-the-box models as fixed feature extractors. Thus, we load the pretrained weights for these models in a *fixed feature extractor* module and fix these parameters (i.e. do not update these parameters). We then append a *classifier* module on top of this fixed feature extractor module, whose weights we update based on gradients computed in each iteration.

Fixed feature extractor. We evaluate using the following models as are fixed feature extractor: (i) ResNet18 [10], (ii) AlexNet [11], (iii) VGG16 [12], (iv) DenseNet121 [13], and (v) SqueezeNet10 [14]. These models are standard baseline models used as backbones for several modern computer vision training pipelines, and hence serve as good models to benchmark the performance of our system.

Trained classifier module. The classifier module consists of three blocks: (i) a linear layer taking in the features outputted by the feature extractor module, with output size 4096, followed by a ReLU layer and a dropout layer; (ii) another linear layer of input and output size 4096, followed by a ReLU layer and a dropout layer; (iii) a final output linear layer of input size 4096 and output size 10 to match the classification label space of the MNIST dataset.

Distributed training backend used for benchmarking the performance of PyTorch's native DataParallel module. We use the NCCL (NVIDIA Collective Communication Library) backend for inter-GPU communication, as supported by the PyTorch distributed package [5]. NCCL is a library of standard communication routines for GPUs, implementing all-reduce, all-gather, reduce, broadcast, and reduce-scatter, as well as any send or receive-based communication patterns. These routines are explained briefly as follows: (i) **all-reduce** performs reductions on data across devices, writing the result in receive buffers of every rank; (ii) in the **all-gather** operation, each of the K processors aggregates all N values from each processor into an output of dimension $K \times N$; (iii) **reduce** performs the same operation as all-reduce, but writes the result only in the receive buffers of a specified root rank. (iv) **broadcast** copies an N-element buffer on the root rank to all ranks (v) **reduce-scatter** performs the same operation as reduce, except that the result is scattered in equal blocks between ranks, with each rank getting a chunk of data based on its rank index.

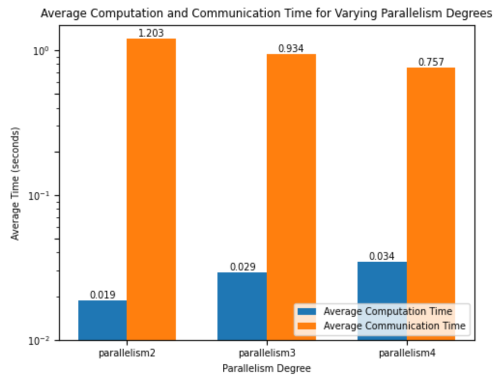
4 Evaluation

Experiment design. We evaluate the performance of the native Pytorch DataParallel module against our own Vanilla DataParallel module. We do this by varying the following training parameters: (i) batch size (**1024**, 512, 256, 128, 64); (ii) Parallelism degree (i.e. number of GPUS: 1, 2, 3, and **4**); (iii) fixed feature extractor architecture: SqueezeNet10 (5130 params), AlexNet (54575114 params), ResNet18 (5130 params), DenseNet121 (21020682 params), VGG16 (119586826 params). (iv) Number of trainable parameters (We vary this by changing the hidden layer size for VGG16: **119586826**, 55599114, 26750986 trainable parameters respectively)). Only one factor is changed each time while keeping all other factors fixed at default values, bolded above.

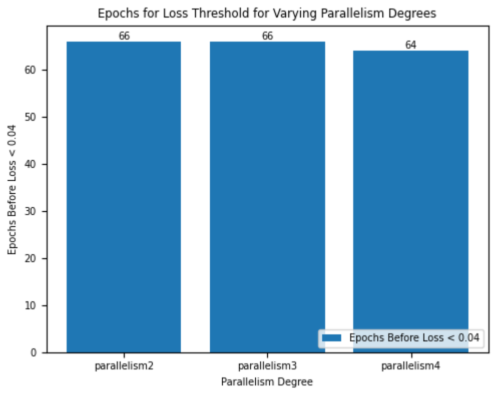
4.1 Performance Metrics

We measure the average computation time, communication time for each iteration, and loss at each iteration and compute the total data transfer in each epoch as detailed in the system design in section 2.

5 Results

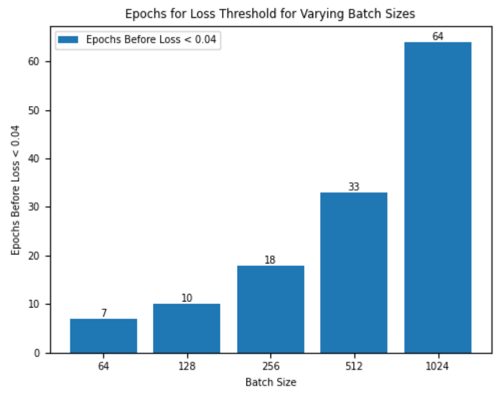


(a) Variant Parallelism Degree, Batch_Size = 1024, Model = VGG16, Parameters = 119586826

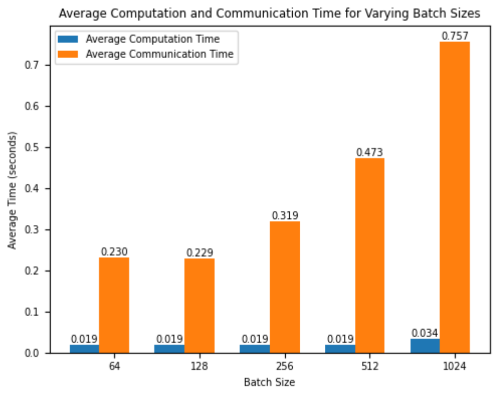


(b) Variant Parallelism Degree, Batch_Size = 1024, Model = VGG16, Parameters = 119586826

5.1 Parallelism Performance Assessment under Controlled Conditions 1a above suggests that average communication time goes down as parallelism increases due to the reduced data volume that needs to be conveyed. Also, the concept of faster convergence due to higher parallelism is evident in Figure 1b above.



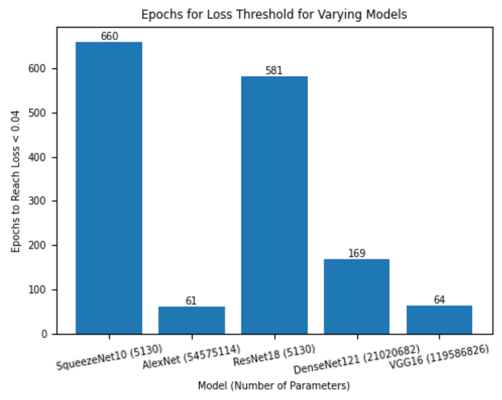
(a) Variant Batch_Size, Model = VGG16, Parallelism = 4, No.of Params = 119586826



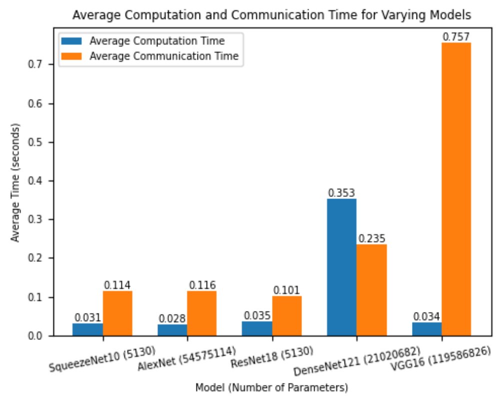
(b) Variant Batch_Size, Model = VGG16, Parallelism = 4, No.of Params = 119586826

5.2 Performance evaluation for different batch_sizes In Figure 2b above, there's evidence that higher batch size corresponds with higher average communication time. This is compliant with the idea that with a larger batch size, more data needs to be transferred between GPUs, and hence this increased data volume and synchronization overhead can result in longer communication times. Figure 2a above appears counter-intuitive at first as larger batch size is associated with more stable and faster convergence, however this could be a case of overfitting the relatively simpler MNIST classification task. Similarly, average communication time goes down as parallelism increases due to the reduced data volume. Given space constraints, the graphs and analysis for varying parallelism degrees are available in the Appendix 1a

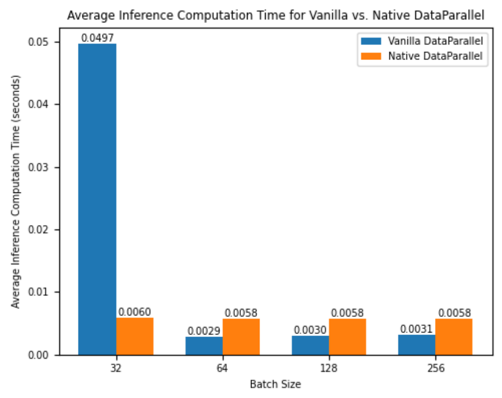
5.3 Performance evaluation for different out-of-the-box models used as the fixed feature extractor Figure 3a below suggests SqueezeNet10 and ResNet18 converge to the low-error threshold 500 epochs after VGG16 and AlexNet, suggesting their simplistic architecture and low parameters underfit the training dataset. Further evidence for this is when VGG16's model parameters are varied (see Appendix Figure 5a). Similarly, in Figure 3b below, the higher the number of parameters and model complexity, the greater the average communication time, as it takes longer to synchronize. DenseNet's average computation time appears remarkably high, perhaps suggesting it requires higher computational power with higher-end GPUs to execute.



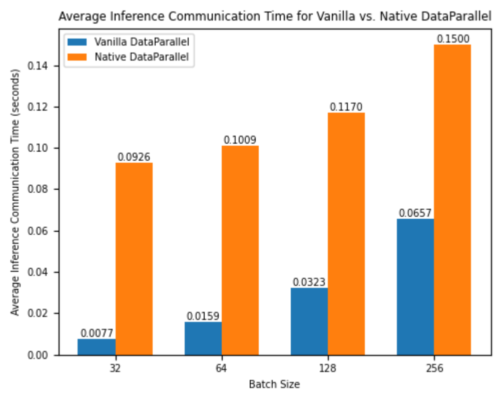
(a) Variant Model, Batch_Size = 1024, Parallelism = 4



(b) Variant Model, Batch_Size = 1024, Parallelism = 4



(a) Vanilla vs PyTorch Native: Average Inference Computation Time



(b) Vanilla vs PyTorch Native: Average Inference Communication Time

Vanilla vs Native DataParallel: Inference Performance Comparison The figure 4a above shows a steep spike in computation time for the Vanilla DataParallel for the small batch size of 32. This indicates inefficiencies in handling smaller workloads (e.g. the overhead associated with setting up parallel processing). Perhaps surprisingly, Vanilla outperforms Native Data Parallel in both computation and communication metrics. On the other hand, the Native DataParallel has consistency in computation time across the batch sizes which could indicate it is effectively optimized for varying workloads. In figure 4b above, the gap in communication times decrease as the batch size increases, indicating that while custom data parallel implementations can be competitive at smaller scaes, PyTorch's native mechanisms likely provide optimizations that become increasingly effective with larger batch sizes and scalability.

6 Conclusion

In conclusion, we reasoned how different backbone models, batch sizes and parallelism degrees affected the system performance by measuring the average computation and communication time for each epoch, as well the number of epochs ran until the loss function fell below a certain threshold. We then implemented our own `VanillaDataParallel` module using communication primitives implemented in PyTorch and compare its performance against PyTorch's native `nn.DataParallel`, albeit for smaller batch sizes of 32 to 256. We computed the average computation and communication time in each epoch and based on our observations, we attempt to infer why the PyTorch Native implementation might be more consistent in the long-term due to its optimizations.

References

[1] S. M. Shenggui Li. Paradigms of parallelism. Accessed on 12/15/23. [Online]. Available: https://colossalai.org/docs/concepts/paradigms_of_parallelism/

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[3] G. von Dulong. (Year of Publication) Gpt-4 will be 500x smaller than people think. here is why. Accessed on 12/15/23. [Online]. Available: https://medium.com/codex/gpt-4-will-be-500x-smaller-than-people-think-here-is-why-3556816f8ff2

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

[5] PyTorch. (2022) Pytorch distributed. Accessed on 12/15/23. [Online]. Available: https://pytorch.org/docs/stable/distributed.html

[6] ——, "Data parallelism tutorial," 2023, accessed on 12/15/23. [Online]. Available: https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html

[7] P. Documentation. (2023) torch.optim.sgd. Accessed on 12/15/2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

[8] N. Corporation. (2023) Nvidia rtx a6000. Accessed on 12/15/2023. [Online]. Available: https://www.nvidia.com/en-us/design-visualization/rtx-a6000/

[9] P. Documentation. (2023) torch.nn.dataparallel. Accessed on 12/15/2023. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html

[10] 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 (CVPR)*. IEEE, 2016, pp. 770–778.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

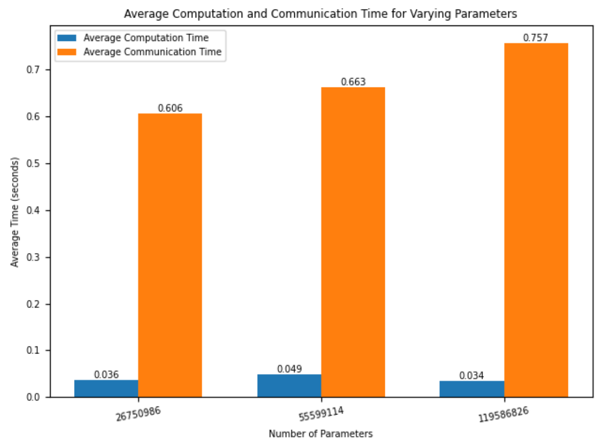
[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

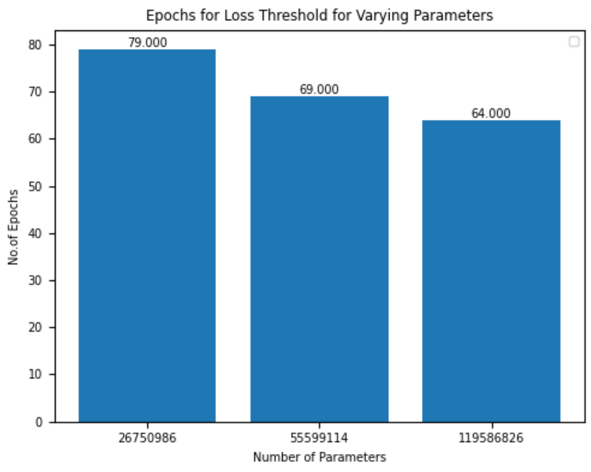
[14] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

7 Appendix

7.1 Parameter Performance Assessment under Controlled Conditions



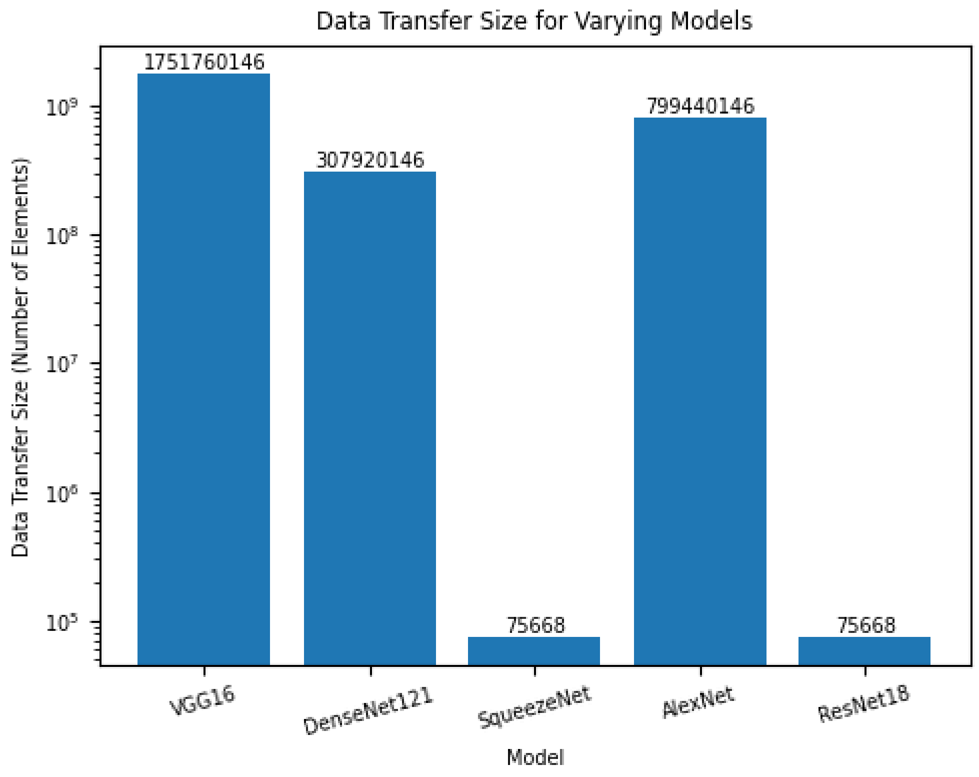
(a) Variant Parameters, Model = VGG16, Batch_Size = 1024, Parallelism = 4



(b) Epochs for Loss Threshold for Varying Parameters

Analysis: These figures further establish how model complexity and more parameters correlates with higher computation in Figure 5a and communication overheads, but greater accuracy in meeting low error thresholds, as in Figure 5b.

7.2 Evaluating Data Transfer Size



(a) Total Model Data Transfer Size