Prediction & Cost Analysis of Distributed Deep Learning Jobs in a Muli-GPU Setting

Kshitij Bharat Sanghvi
Department of Computer Science
Courant Institute of Mathematical Sciences
NY, USA
kbs391@nyu.edu

***Abstract*—The last decade witnessed a boom in SIMD architecture. Concepts that were just theory at one point are now being deployed in a reasonable amount of time. With the capable hardware and in conjunction with cloud technologies, a lot of new applications are now being used with relative ease. Several new fields of study such as machine learning and deep learning are now prevalent across business domains. With every new generation, the hardware is getting more powerful. But it's still not capable to fulfill the demanding request of its users. To that end, the consumer has a choice. Either opt for a single GPU with high computation capacity or a collection of GPUs of relatively lower computation capacity. The main factors to drive this decision are performance and cost. In this paper, I present a model to capture the cost versus performance for distributed training deep learning jobs by providing a tool that predicts the time and cost to complete a job.**

Keywords: Multi-GPU Training, Deep Learning Jobs, Cost analysis

I. Introduction

Convolution neural networks were first introduced by Yan Lecun with the introduction of LeNet[1] in digit recognition. Which displayed the prowess of neural networks in image classification jobs. Deep convolutional neural networks led a breakthrough in image classification accuracy. In 2012, Alex by using two GTX 580 GPUs was able to break the accuracy barrier which held stagnant for almost a decade, his model being called AlexNet[2]. As the field matured a lot of different architectures were born such as the Google inception model[3], VGG[4] and the ResNets[5]. A lot of experiments have been conducted on the various configurations when the number of layers is increased. It is theoretically and empirically proven that deep neural networks with deeper layers hold more model capacity and better classification. We know that deep neural networks can become very difficult to train especially when the training data is sparse. This can cause high variance in the model. The residual framework eases this training job even when the layers get deeper. Residual networks are made up of a residual block which is kind of a mapping of a function as shown below in Fig 1. ResNets can also be merged with stochastic depth[6] to further improve the training and add regularization.

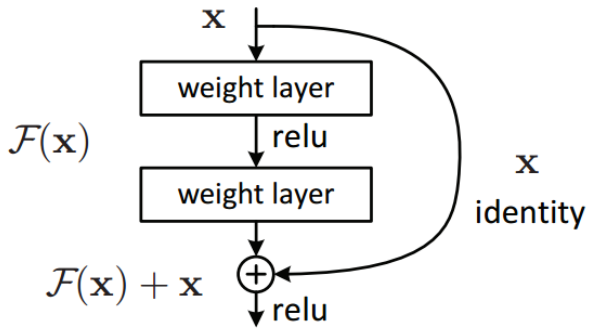


Fig. 1. Residual Block

Training deep neural networks is always been a computationally intensive job. Training can be abstracted to matrix multiplication, because that is the core operation. GPUs are the choice of hardware because of their capability to exploit massive parallelism and offer high bandwidth. Even though GPUs are efficient at performing this computation, they are extremely expensive. Nvidida's latest GPU - A100 retails in the range of \$10000. Cloud services such as GCP, offer Nvidia A100 at the price of \$2145 on monthly use. Thus, it is paramount to determine the optimum strategy. For instance, it is not economical to pay double the value for a faster GPU if the return is only a speed up of 1.2x. In other words, we need to determine the Pareto optimality of the available configurations.

II. Background

Over the years the deep learning frameworks have evolved. In the initial stages, there was no support for multi-GPU systems. This resulted in the training times being exceptionally long. For instance, using the places dataset[7] which contains approximately 2.5M images took 6 days to be trained on a single K40. The main motivation to improve on the hardware is to reduce the training times. Empirically, it is observed that working with FP16 floating points is comparatively faster than FP32 floating points and there is a modest difference in accuracy. Additionally, working with mixed precision[8] can enable larger batch sizes as each weight will occupy only half the size. Batch size is restricted by the GPU global memory. Higher the batch size that fits the GPU memory more is the utilization of the GPU. Thus, one straight advantage of using multi-GPU is larger batch sizes with larger learning rates leading to faster convergence.

A deep learning job can be parallelized in several ways across a multi-GPU. Before that, the GPU topology affects the inter-GPU communication. Depending upon the architecture several different collective techniques have been employed such as Blink[9]. Researchers have played with various topologies depending upon the placement of GPUs - from ring to point-to-point. Depending upon the learning being synchronous or asynchronous different topologies have proven to provide different results. The blatant advantage of the point-to-point system is direct communication, the side effect being increased cost and communication and frequently being bottle-necked at the common contact. Ring topologies are suitable when there is an inherent sequence involved in the computation.

Mostly in the point-to-point architecture, there is a parameter server model implementation where each GPU independently computes the forward pass i.e. computes the loss, goes backward, calculates the gradient for the stochastic gradient descent, and sends the result to the parameter server. The parameter server may or may not wait for all the updates depending upon the policy at hand and broadcast the updated weight to all GPUs, including the ones working on the previous iteration. It is easy to deduce that the synchronous version – the parameter server that waits for all GPUs, will take much longer than the asynchronous to complete the job. The former is bottle-necked by the slowest performing GPU. The accuracy achieved by the asynchronous architecture might never match the synchronous one but will converge quickly. This led researchers to find a middle point between the synchronous and asynchronous versions. K batch synchronous and asynchronous versions were introduced[10] as shown in Fig 2. The K-batch versions of respective schemes wait for at least K updates to be propagated to the parameter server before the aggregation can begin. When K is equal to N it becomes synchronous as before. This leads to the obvious problem of stale gradients. The problem is easily visible in the figure. L represents a particular GPU.

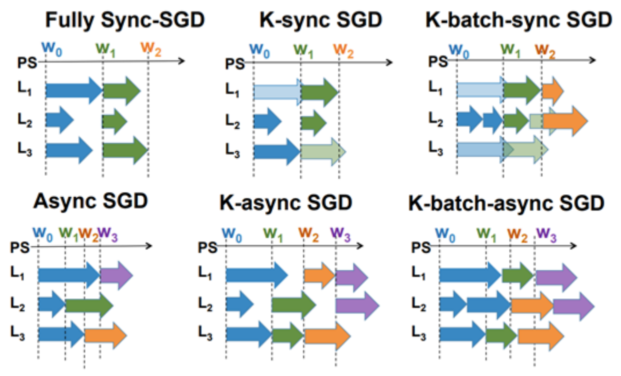


Fig. 2. The Middle Ground, K batch Asyn and Syn.

III. Parallelism

Parallelism can be implemented at various levels in a DL job. Model parallelism is when different layers are located in different GPUs. As an example, Alexnet employs model parallelism. Data level parallelism is when the same copy of the model lives in the GPU but they compute values on different data. This is depicted in the Fig 3. A hybrid approach is when there is both – model and data-level parallelism. The raw advantage that we get from distributed GPUs is that we can work with larger batch sizes. We can work with larger learning rates with larger batch sizes for quick convergence.

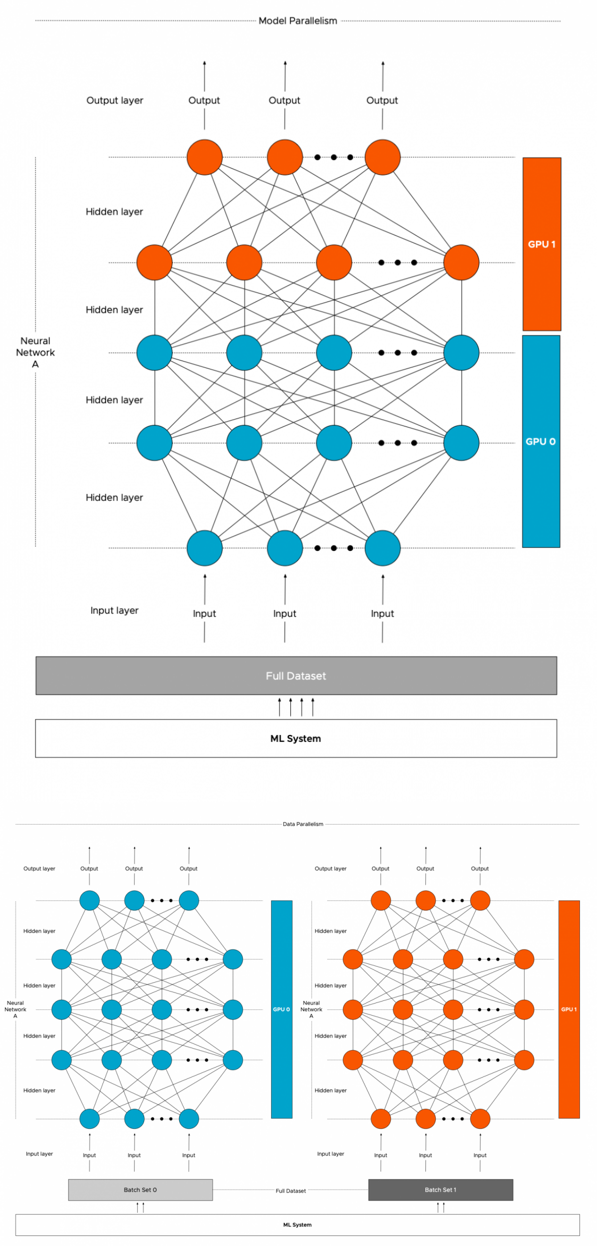


Fig. 3. Differentiating Data and Model Parallelism. Different colors depict the location on different GPUs

However, there is significant communication overhead. That is why the scaling efficiency does not scale equally as the number of GPUs. The synchronization overheads need to be borne in a multi-GPU setting depending upon the topology. For the parameter server model, point-to-point broadcasting is used whereas in a ring topology (typically connected by NVLink) techniques like all reduce are employed in which after a complete cycle all the required values are exchanged among peers.

IV. Performance Metrics

Several performance metrics can be used to identify a particular job. However, we use the following.
• Loss: The loss computed at each iteration. The difference in the predicted and actual values.
• Accuracy: The ratio of the numbers of samples classified correctly.
• Time to Accuracy: The amount of time taken to reach a particular accuracy.
• Scaling efficiency: Ratio of the time to train one iteration on a single GPU to the time taken on multiple GPUs.

V. Model Of Computation

I extend the predictive model based on the Optimus[11] approach. It is based on two sub-models. One that models the training loss as a function of the number of iterations. Second models the training time. The training loss model uses three parameters as shown: $L = \frac{1}{\beta_0 k + \beta_1} + \beta_2$

The series of events in a synchronous parameter server model can be broken down as follows: Let us assume there are g GPUs. The batch size of b. If we split the batch evenly we get $\frac{b}{g}$ batch for each GPU. Let us assume it takes $t_{forward}$ for each forward pass. Each GPU takes $\frac{b}{g}t_{forward}$ for loss computation. The backward pass is independent of the batch size and is equal to $t_{backwards}$. Let S denote the size of the gradients computed. Each GPU sends S gradients to the parameter server. Let $n \leq g$ denote the number of GPUs that a parameter server is responsible for. Let the bandwidth be b. Let T_{agg} be the time taken by the parameter server to accumulate the gradients. Thus, the bandwidth between the server and the GPU is $\frac{B}{n}$. Sending and broadcasting gradients are symmetrical operations, thus over time can be formulated as $2\frac{S}{B/n}$ Let the communication overhead be equal to og.

Thus, the time can be formulated as: $T = max(\frac{b}{g}t_{forward} + t_{back} + 2\frac{S}{B/n} + t_{agg} + og)$ The number of training steps completed by all GPUs can be given by gT^{-1} which can be parameterized as $f(n) = \alpha_0 \frac{M}{g} + \alpha_1 g + \alpha_2$ Now, we have a model that predicts the loss as a function of iteration. We have a model that predicts the number of iterations per unit time. We can now identify the optimal cost of training a job easily. We identify the iterations where the loss is quickly decreasing and given that there is sufficient training samples we can assume that the model is converging quickly. This helps us to distribute iterations, when the loss is converging quickly according to our predictions we can schedule those iterations on the multi-GPU system and when the drop is modest we can schedule those on economical hardware.

VI. Implementation

Hardware: We use 2xK80, P100 on the Google Cloud Platform. Unfortunately A100 had limited availability and higher pricing. The model to train and test with is a variety of ResNets with varying depth.

Dataset: The CIFAR-10 dataset [11] consists of 60000, $32 * 32$ colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. A sample of the dataset is shown in Fig 4.

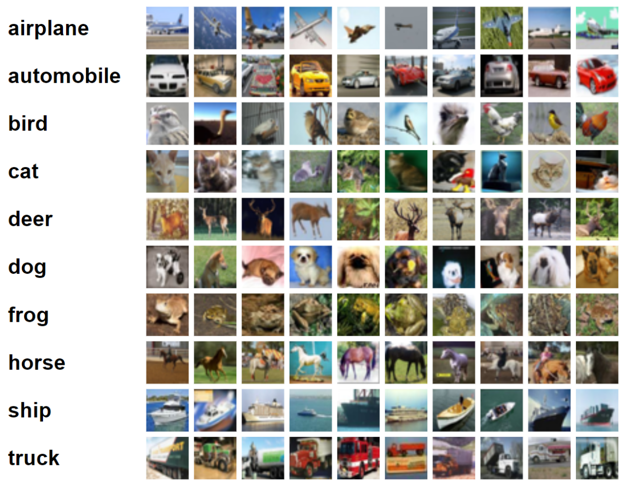


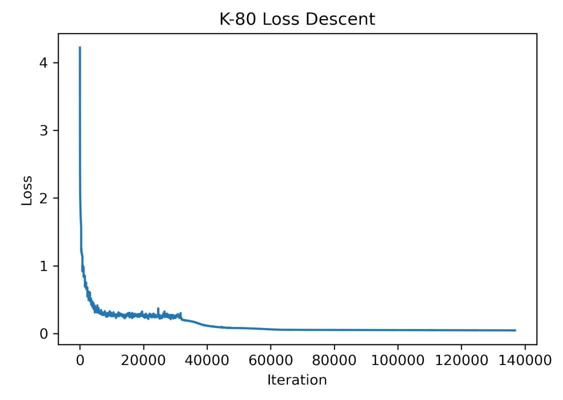
Fig. 4. The CIFAR-10 Dataset

VII. Results & Analysis

In Fig 5 we can see how the loss converges as a function of the number of iterations for ResNet-20 on a single and double K-80.

The loss degradation is ideally independent of the underlying hardware. However, due to in-deterministic nature of the cuDNN for high performance we know that the values will never be a perfect match. The loss degradation for 2xK80 will not be equal. As each GPU will compute only half the batch size and those results will be aggregated which will justify the marginal difference due to optimisations such as BatchNorm[13] and regularisation[14]. We develop a simple linear regression models to estimate the β values as depicted in the optimus approach. This gives us a model to estimate the loss as a function of iteration count.

The task of deep learning jobs is repetitive. In that if we find the time to execute one iteration it will remain, a little more or less, constant throughout. To find the value of one



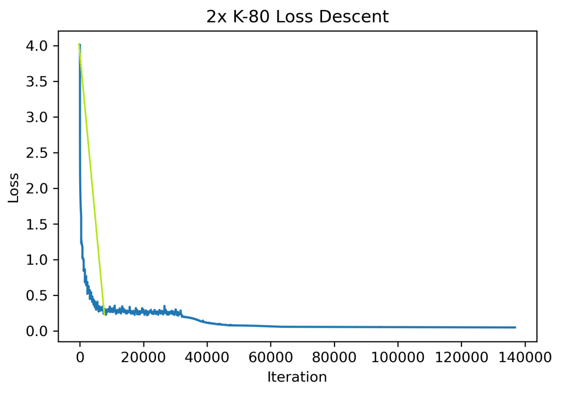


Fig. 5. Loss Degradation on a Single K-80 vs 2x K-80

TABLE I

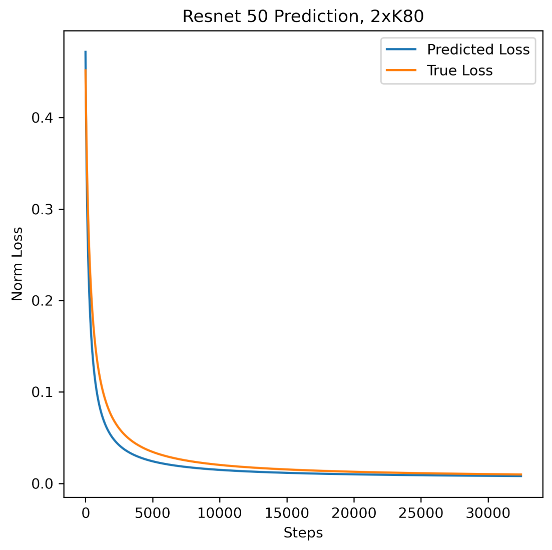
GPU	Instances	Price Per Hour	Time 1 Epoch	\$ for 1 Epoch
K80	1	\$0.45	87s	\$0.01088
K80	2	\$0.90	55s	\$0.01375
P100	1	\$1.46	48s	\$0.01947

iterations, we time 10 successive epochs and take the median. Tensor-flow automatically outputs the time to the stdout. Then we can divide the time by the number of iterations to calculate the time per iteration. These values are showed in Table 1.

Our loss model gives the following predictions for ResNet 50 on P100 and dual K80s as showed in Fig 6. The model predicts that the 2xK80 configuration will reach the target loss or accuracy of 80% at the 51th Epoch and P100 will reach at 94. The prediction is not extremely accurate as both models should reach the target loss in similar iterations.

From the above methodology, we can compute the expected cost this deep learning job incurs. Cost = Number of Epochs * Cost of Each Epoch. P100 = $94*0.01947 = \$1.83018$ 2xK80s = $51 * 0.01375 = \$0.70125$. The target loss is reached at the 25th Epoch. That gives us an error of 276% for P100 and 104% for the 2 K80s.

Predicted time taken by P100 = $48*94 = 4512s$ and 2xK80 = $55*51 = 2805s$ This is due to the high error in modelling the loss and the β parameters. Whereas the actual time for P100 = $25 * 48 = 1200s$ and that for 2K80s = $25 * 55 = 1375s$.



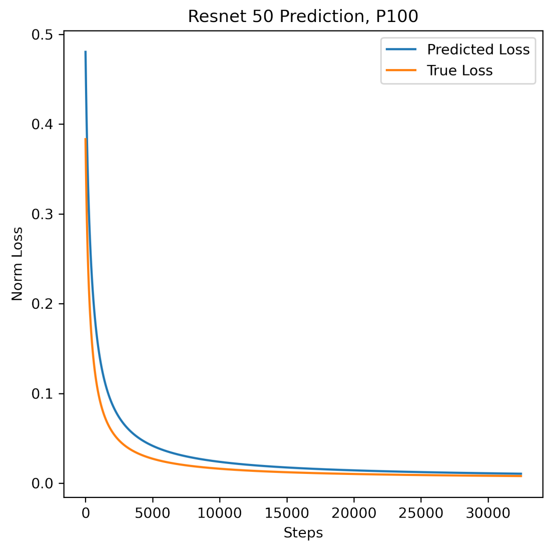


Fig. 6. Prediction Error: P100 43.91% and 2xK80 25.43%

In our prediction P100 under-performs. This can be mitigated by choosing a better regression model for the parameters and can serve as future work.

Choosing between the two approaches is a subjective choice. Calculating the optimum training time depends on how critical the learning is. This subjectivity can be eased by quantifying the speed-up by cost, Δ. Let Δ denote the the speed-up per fold increase in cost. The value of Δ can be observed in Fig 5. Table 1 provides the values that help us compute the same.

Alternatively, for intervals of iterations where there is an expected drastic drop in loss, we can compute those iterations on GPUs that are geared towards reduced training time. This

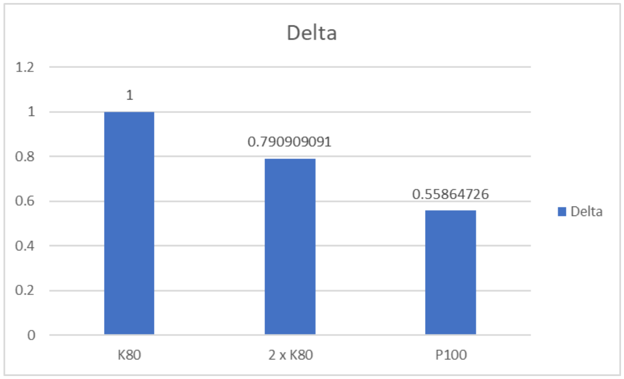


Fig. 7. Delta Values for Single K-80, Dual K-80 and P100

will help the engineer determine the range of accuracy the model will fall into quickly. The region where there is a modest decrease in loss can be computed on lower powered GPU. This methodology will be optimum as switching from one GPU to another or an entire different cluster by saving a model locally can be immediately handled. Not incurring any extra overhead in switching GPU clusters, we can select the nunber and type of GPUs by a list in GCP. This region of high slope of loss is shows in green in diagram 2 of Fig 5.

VIII. Conclusion and Future work

I have provided a model that can give us the expected cost by computing the expected loss at a given iteration, where we can compute the time required to complete an iteration for a given ResNet architecture. This helps us to decide if a cluster of compute capacity GPUs will be cost friendly or a single high performance GPU will provide a better Δ value than 1. We do not include the utilisation of the GPU as a factor because that metrics dictates the selection of the algorithm. Our model inherently takes this into consideration as an algorithm that is under-utilising a GPU will take longer than an efficient one for the same amount of work. Our model helps to decide if the job should be scheduled on a particular configuration. The model does not hint on cues to design an efficient algorithm for a particualr hardware. In future work, I would like to see how this model extends to other architectures such as inception models.

References

[1] LeCun, Y.; Boser, B.; Denker, J. S.; Henderson, D.; Howard, R. E.; Hubbard, W. Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541-551.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. 2012. ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems 25 (NIPS'2012).

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of CVPR, pages 1–9, 2015.

[4] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In ICLR, 2015.

[5] Huang, Gao, et al. "Deep networks with stochastic depth." European conference on computer vision. Springer, Cham, 2016.

[6] http://places.csail.mit.edu/

[7] Haidar, Azzam, et al. "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers." SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018

[8] Wang, Guanhua, et al. "Blink: Fast and generic collectives for distributed ml." Proceedings of Machine Learning and Systems 2 (2020): 172-186.

[9] Dutta, S., Joshi, G., Ghosh, S., Dube, P., Nagpurkar, P. (2018, March). Slow and stale gradients can win the race: Error-runtime trade-offs in distributed SGD. In International Conference on Artificial Intelligence and Statistics (pp. 803-812). PMLR.

[10] Peng, Yanghua, et al. "Optimus: an efficient dynamic resource scheduler for deep learning clusters." Proceedings of the Thirteenth EuroSys Conference. 2018.

[11] J. https://www.cs.toronto.edu/ kriz/cifar.html

[12] Ulyanov, Dmitry, Andrea Vedaldi, and Victor Lempitsky. "Instance normalization: The missing ingredient for fast stylization." arXiv preprint arXiv:1607.08022 (2016).

[13] Gal, Yarin, and Zoubin Ghahramani. "Dropout as a bayesian approximation: Representing model uncertainty in deep learning." international conference on machine learning. PMLR, 2016.