Priority-based Parameter Propagation for Distributed DNN Training

Anonymous Authors[1]

Abstract

Data parallel training is widely used for scaling distributed deep neural network (DNN) training. However, the performance benefits are often limited by the communication-heavy parameter synchronization step. In this paper, we take advantage of the domain specific knowledge of DNN training and overlap parameter synchronization with computation in order to improve the training performance. We make two key observations: (1) different parameters can afford different synchronization delays and (2) the optimal data representation granularity for the communication may differ from that used by the underlying DNN model implementation. Based on these observations we propose a new mechanism called *Priority-based Parameter Propagation* (*P3*), which, synchronizes parameters at a finer granularity and schedules data transmission in such a way that the training process incurs minimal communication delay. We show that: *P3* can improve the training throughput of ResNet-50, Sockeye and VGG-19 by as much as 25%, 38% and 66% respectively.

1 Introduction

In recent years, deep learning has attracted tremendous attention in the machine learning community and beyond by achieving notable success across a wide spectrum of tasks such as computer vision (He et al., 2015), machine translation (Wu et al., 2016) and speech recognition (Amodei et al., 2015). Training these models, however, take days to weeks or sometimes even months to finish because of the high degree of computational complexity, large number of parameters and large datasets iteratively processed (Silver et al., 2017; Zhu et al., 2018). This high computation cost necessitates distributed training to keep the training time reasonable.

Data parallel distribution with synchronous stochastic gradient descent (SGD) is a popular method for scaling DNN training over a cluster of machines (Chen et al., 2016). In this paradigm, worker machines iteratively train a shared model on different samples of the input dataset, synchronizing by combining parameter updates on every iteration. A training iteration involve three main steps: (1) a *forward propagation* step for calculating the value of a loss on a subset of input dataset function using up-to-date parameter values, (2) a subsequent *backward propagation* step for computing the gradients for every model parameter with respect to the loss calculated, and (3) a *parameter synchronization* step for aggregating local gradients of all the worker machines and updating the parameters with the corresponding aggregated gradient values using the SGD algorithm.

During distributed training, each worker machine generates and synchronizes hundreds of megabytes of gradient values on every iteration (Alan et al., 2018). Handling such huge volume of data require high network bandwidth. This problem is exacerbated with the emergence of larger DNN models and better hardware accelerators, because worker machines can generate more data faster. This leads to more frequent network synchronization often beyond the capabilities of the networking infrastructure in major cloud providers and most academic clusters (Luo et al., 2018). These factors often make distributed DNN training a communication-bounded workload. In this work, we target this problem and propose solutions to scale data parallel training under limited bandwidth conditions.

One way to handle a heavy communication load is to use higher bandwidth networks. There are network solutions like Ethernet (Cunningham et al., 1999) and InfiniBand networks (Shanley, 2002) that can offer over 100Gbps bandwidth capacity networking infrastructure for faster parameter synchronization. However, these technologies are yet to be adopted widely because of the relatively high deployment cost. Moreover, faster networks for distributed DNN training may not be sustainable solution considering the rate of advancements in hardware accelerators and growth in the

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

model complexity (Luo et al., 2018).

An alternative approach is to reduce the communication volume by compressing gradient values (Wen et al., 2017; Lin et al., 2018). Since gradient values are generally represented as floating point numbers, it is extremely challenging to get reasonable compression ratios from lossless compression techniques (Burtscher & Ratanaworabhan, 2009). Instead, recent work in this area propose lossy compression techniques like gradient quantization (Seide et al., 2014; Alistarh et al., 2017; Wen et al., 2017) and sparse parameter synchronization (Aji & Heafield, 2017; Lin et al., 2018). These methods, however, risk affecting the final convergence accuracy of the model because of the information loss that comes with value approximation and stale parameter updates (Khoram & Li, 2018).

An orthogonal approach is to utilize the network bandwidth more efficiently by leveraging domain specific opportunities in DNN training. Because of the iterative nature of deep learning training algorithms, the traffic generated is usually bursty. A common practice used in some distributed machine learning frameworks is to attenuate these traffic bursts by overlapping communication with computation. The training computation is performed as a sequence of operations called layers. During backward propagation each of these operations generate gradients for a subset of parameters of the whole model. Frameworks exploit this sequential layer-by-layer structure in deep learning training algorithms by scheduling independent gradient computation operations and network communication operations together. Frameworks trigger synchronization for a layer as soon as the gradients for that layer is generated and is ready to be propagated (Zhang et al., 2017). Using this approach, parameter synchronization can be effectively overlapped with backward propagation.

In this work, we find new opportunities to reduce the communication bottleneck in distributed DNN training. Our first observation is that domain specific knowledge of DNN training allows us to schedule parameter synchronization not only based on when the data is generated, but also based on *when the data is consumed*. Training computation is a sequence of stages, operating on one or a few layers of the model at a time. During training, the gradients of the layers are generated from final to initial layers and subsequently consumed in the reverse order in the next iteration. Figure 1 shows a snapshot of the training process containing the backward propagation of one iteration and the forward propagation of the next one. The temporal gap between gradients generated and consumed per layer are higher for final layers compared to the initial ones. Scheduling parameter synchronization using this information can help to overlap communication with both backward and forward propagation.

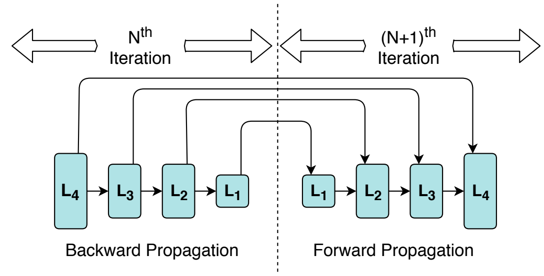


Figure 1. Training iteration

Our second observation is that the layer-wise granularity used by the underlying neural network implementation may not always be optimal for parameter synchronization. In our experiments, we observe that for certain models (e.g., VGG-19, Sockeye), parameter synchronization at finer granularity can improve the network utilization and reduce the communication delay.

Based on these observations we propose a new synchronization method called *Priority-based Parameter Propagation* (*P3*).

1.1 Our Approach: Priority-based Parameter Propagation (*P3*)

There are two main ideas behind *P3*: (1) During parameter synchronization, *P3* splits the gradients of the layers into smaller slices and synchronize them independently. (2) *P3* synchronizes the parameter slices based on their priority, where priority of a parameter is defined by how soon it is going to be consumed in the subsequent iteration. During back propagation, *P3* always allocates network cycles to the highest priority parameters in the queue, preempting synchronization of a previous low priority parameter slice if necessary.

P3 offers following advantages over state-of-the-art mechanisms. (1) *P3* can provide improved training performance under limited bandwidth conditions by better overlapping communication with computation and utilizing the available network bandwidth more efficiently. (2) *P3* is model-agnostic and the implementation only requires minimal effort and is localized within the framework. (3) *P3* always communicates full gradients and does not affect model convergence.

In summary, this paper makes the following contributions:

- We show that parameter synchronization at layer-wise granularity can cause suboptimal resource utilization in some models (e.g., VGG-19, Sockeye). We also show that the parameter synchronization can be scheduled better to efficiently use the available network bandwidth by taking into account not only the information on when the gradients are generated, but also when they are consumed.

- We present a new parameter synchronization mechanism, called *Priority-based Parameter Propagation* (*P3*), which takes advantage of the temporal gap in the generation and consumption of gradients of different layers and propagates the gradients based on their priorities, with initial layers getting higher priority. We demonstrate that *P3* has better resiliency towards bandwidth limitations compared to other parameter synchronization mechanisms.

- We implement *P3*[1] on MXNet (Chen et al., 2015), a popular distributed machine learning framework, and evaluated the performance against standard MXNet implementation as the main baseline. With *P3* , we improve training performance of several state-of-the-art models like ResNet-50 (He et al., 2015), Sockeye (Hieber et al., 2017) and VGG-19 by as much as 25%, 38% and 66% correspondingly when available network bandwidth is limited.

2 Background

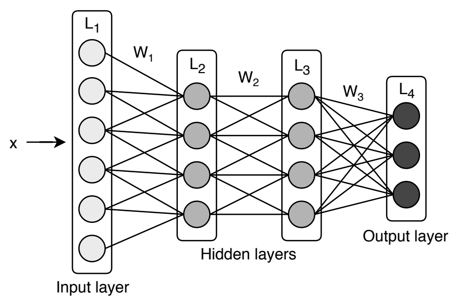


Figure 2. Deep neural network structure

As illustrated in Figure 2, a DNN consists of a hierarchy of parameters arranged as a sequence of layers ranging from as few as 5-10 (Krizhevsky et al., 2012) to as many as 100s (He et al., 2015). Each layer takes an input vector x and emits an output vector based on a transformation function $f(W, x)$, where W is the parameter matrix of the layer. In Figure 2, the initial input layer of the DNN takes the application specific data samples as input and the final output layer produces the value of the DNN's objective function after applying a series of transformation operations defined by the layers on the input vector. The output is generally a scalar value representing the error in the prediction (*loss*).

DNN training is an iterative process for optimizing the objective function defined by the neural network. During training, DNN runs a series of operations on the input vectors sampled from the dataset and calculates the loss associated with the model parameters on the input data. This is called a forward propagation. After that, a backward propagation step is performed that calculates the error contribution of each parameter by computing gradients of all the layers with respect to the loss. The backward propagation method for calculating gradients is based on the chain rule of derivatives and is therefore performed in the reverse order of forward propagation i.e., gradients of the final layers are calculated first and moves backwards to the initial layers, hence the name backward propagation (Rumelhart et al., 1986). Once the gradients are calculated, the parameters are updated using an optimization algorithm, usually Stochastic Gradient Descent (SGD) (Bottou, 2010). This process (forward propagation, backward propagation, and parameter update) is repeated by randomly sampling input from a sufficiently large dataset until the model converges to an acceptable optima.

The training process takes many (e.g., thousands) iterations to converge and is therefore highly computationally expensive. The total training time can be dramatically reduced by distributing the workload into multiple machines by taking advantage of the data parallel nature of the SGD algorithm. Data parallel training (Keuper & Preundt, 2016) involves multiple workers simultaneously working on a shared parameter set with the whole dataset distributed equally among them. Workers calculate gradients on same parameter values but on different input data samples and aggregate these gradients in a synchronous fashion before performing parameter updates. This mechanism is called a *synchronous SGD* algorithm (Chen et al., 2016).

There are many methods used in practice for synchronous parameter update. The parameter server architecture (Li et al., 2014) is one of the most popular methods among them. A parameter server is a distributed shared memory system that keeps track of the up-to-date values of all the model parameters. Before every iteration, each worker machine reads the latest parameter values (θ) from the parameter server and locally computes gradients for the inputs sampled from its data shard. The workers then send the local gradients ($\triangledown$) to the parameter server. The parameter server waits until it receives gradient updates from all worker machines, then aggregates the gradients together and updates the parameters for the next iteration.

Figure 3 shows parameter server-based data parallel training in a four-node cluster. The communication between worker machine and parameter server is usually over a network and often becomes the bottleneck in achieving linear scalability in data parallel training (Zhu et al., 2018; Shi & Chu, 2017).

Popular machine learning frameworks, e.g., MXNet (Chen et al., 2015) and TensorFlow (Abadi et al., 2016) can be distributed over a cluster of machines using the parameter

[1] We will be open-sourcing *P3* implementation soon.

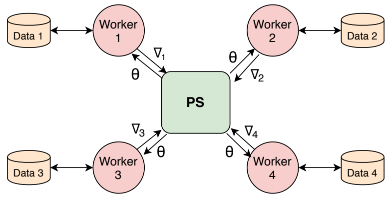


Figure 3. Parameter server architecture

server architecture. MXNet is designed specifically for making data parallel training efficient and easy to execute. It comes with a built-in implementation of parameter server called *KVStore*. In MXNet, worker machines send out gradients of a layer to the *KVStore* as soon as they are calculated and issues a parameter pull request once all the other workers have finished sending gradient updates for that layer. This aggressive parameter synchronization mechanism makes data parallel training on MXNet very efficient.

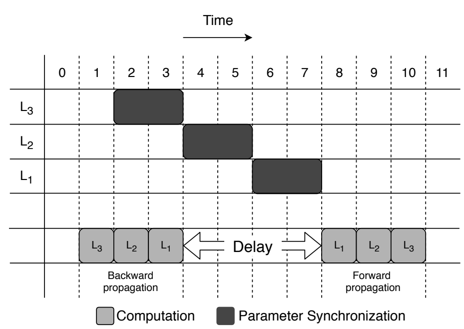
TensorFlow, on the other hand, is designed as a more generic machine learning framework. Hence it does not have an explicit parameter server implementation. However, a parameter server can be implemented on top of the graph computation framework provided by TensorFlow. Since the parameter server is a part of the computation graph, the communication between the worker subgraph and parameter server subgraph is handled by the framework itself. TensorFlow automatically places *Send* and *Receive* operations on the edges of the computation graph that crosses the device boundaries. Similar to MXNet, the worker subgraph executes the send operation as soon as the gradients are computed. However, since every training iteration is a separate graph execution, the parameter pull request is not issued until start of the next iteration. This disconnection in sending gradients and receiving parameter updates could cause underutilization of bidirectional bandwidth of the network.

Despite small differences such as described above, we observe that machine learning frameworks (MXNet, TensorFlow, Caffe2, etc.) follow two common behaviors. For performance reasons, the operations in the DNN implementation usually prefer to perform computations on large data representations and because of this, the gradients for all the parameters in a layer is usually generated in a single shot. We observe that (1) because the gradients are generated in a layer level granularity, frameworks perform parameter synchronization at the same granularity as well. Moreover, since the DNN implementation is written as a dependency graph in these frameworks, (2) the gradients of the layers are sent out to the parameter server over the network as soon as the backward propagation of that layer has completed. In this work, we address the limitation associated with these two observations.

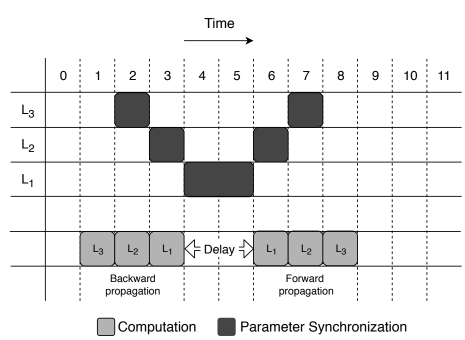
Apart from parameter server architecture, there are other mechanisms used for gradient aggregation. For example, there are many variations of MPI *all_reduce* operation specifically designed for machine learning workloads (Daily et al., 2018; Awan et al., 2017). In this work, we implement *P3* over the parameter server architecture in MXNet. However, *P3* design principles, namely parameter slicing and priority-based propagation, are general enough to be applied to any gradient aggregation methods.

3 Limitations of Parameter Synchronization

Current parameter synchronization mechanisms have major limitations in effectively utilizing available network bandwidth due to two main reasons. The first one comes with the aggressive synchronization performed by the frameworks where the gradients of the layers are sent to the parameter server immediately after finishing the backward propagation of that layer. Since the backward propagation progresses from the final to the initial layer, the gradients are also generated and propagated in that order. However, the next forward propagation can only be started after receiving the parameter updates for the *first layer*. We observe that under limited bandwidth, gradient propagation of the final layers can induce queuing delays on the initial layers and subsequently delay the next iteration. This prevents the communication from being efficiently overlapped with the forward propagation.



(a) Aggressive synchronization



(b) Priority based synchronization

Figure 4. Parameter synchronization

Figure 4(a) shows parameter synchronization of a 3-layered

DNN. In this example, the forward and backward propagation of each layer takes 1 time unit and parameter synchronization takes 2 time units. Since the parameters of the layers are aggressively synchronized, the total delay between the two iterations is twice the time taken for synchronizing the first layer. Moreover, since the order of parameter updates are always maintained throughout the training, during forward propagation the network stays completely idle.

This effect becomes more dominant when the communication time required for individual layers vary due to the presence of dense layers in the DNN, as the synchronization time need for dense layers are relatively higher. Figure 5 shows the parameter distribution of three popular image classification models: ResNet-50, InceptionV3, VGG-19 and a machine translation model Sockeye. The skewed parameter size distribution is a general trend in image classification models where the final fully connected layers are usually heavier and can potentially induce higher queuing delay on to the lighter initial convolution layers.

The second limitation is due to the parameter synchronization being performed at a full layer-wise granularity. The communication cost of parameter synchronization consists of three major components: (1) *gradient propagation* time for the worker machine in order to send the gradients to the parameter server, (2) time taken by the parameter server to aggregate the gradients and perform *parameter update*, and (3) *parameter propagation* time taken by the parameter server to send the updated parameters back to worker machine(s). As we described in Section 2, current distributed machine learning frameworks overlap gradient propagation of one layer with the backward propagation of the next one. On top of this, at parameter server side, the gradient propagation of a layer is overlapped with the parameter update of the previous layer. This type of communication-computation pipelining is effective only if the size of the layers are more or less uniform. Unfortunately, this is usually not the case. For example, Figure 5(c) shows that VGG-19 contains a single fully connected layer which has 71.5% of all the parameters in the entire network. We observe that the disproportionately heavy layers like this could severely affect the effective utilization of network bidirectional bandwidth.

This effect is explained in Figure 6(a) using the previous example of parameter synchronization of a 3-layered DNN. In this case, gradient propagation, parameter update and parameter propagation of the second layer take thrice as much time as that of the first and third layers. Because of this imbalance, the communication delay in this model is mainly dominated by the second layer. The parameter synchronization of the first and the third layer can only be partially overlapped with the second layer. As seen in the example, this severely underutilizes the computing resources and bidirectional bandwidth by spending the last 3 time steps just for receiving parameter updates from the parameter server.

From the above observations we draw two major conclusions. (1) Application domain-specific knowledge of DNNs can be utilized to schedule communication not only based on *when* the data is generated in the backward propagation, but also based on when the data is going to be consumed in the subsequent forward propagation. Scheduling parameter synchronization based on this information and sending gradients conservatively could reduce the delay by better overlapping communication with computation. (2) The optimal granularity required for parameter synchronization may differ from the one used for data representation by the underlying model implementation. Synchronizing parameters at a finer granularity can better utilize the available computing and networking resources.

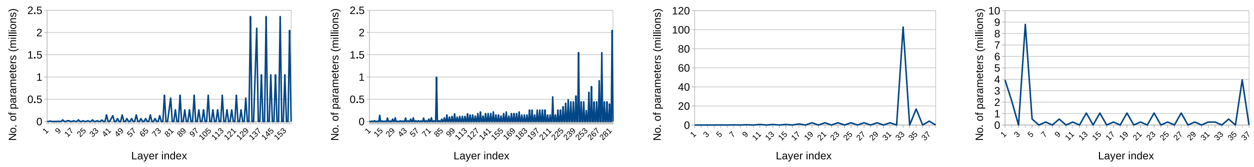
4 *P3*: Design and Implementation

Based on the above observations, we propose a new method for parameter synchronization called *P3*. As explained in Section 1.1, *P3* has two core components: (1) *parameter slicing*, and (2) *priority-based update*.

P3 synchronizes parameters at a finer granularity by slicing the gradient matrix of the individual layers in the DNN into smaller pieces and synchronizing them independently. By doing so, we observe that the network utilization can be improved. In Figure 6(b), splitting the second layer into 3 smaller packets and independently synchronizing them achieves better overlap between data transmission and parameter update. Since the synchronization of all the slices are perfectly pipelined, the network stays busy most of the time. The bidirectional bandwidth is completely utilized during the synchronization of all the intermediate slices. This considerably reduces the communication cost in these types of DNNs. In this example, with parameter slicing the communication cost has reduced by 30%.

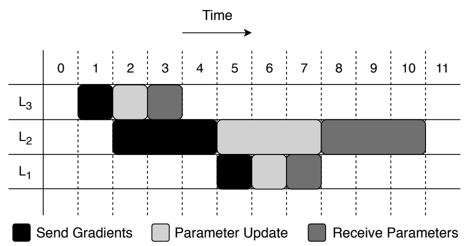
After splitting the layers into smaller pieces, we assign priorities to each slice. These slices inherit their priority values from its parent layer. Priorities of the layers are assigned based on the order in which they are processed in the forward propagation. The first layer gets the highest priority and the priority decreases moving towards the end, with last layer getting the lowest priority. During the parameter synchronization, gradient slices are transmitted based on their priority as shown in Figure 4(b). In this example, with prioritization enabled, the delay between two iterations has reduced by half and the communication is evenly overlapped with both forward and backward propagation.

We implemented *P3* by modifying MXNet parameter server module called KVStore. Below, we explain how the baseline KVStore works and then the modifications we made for *P3*.

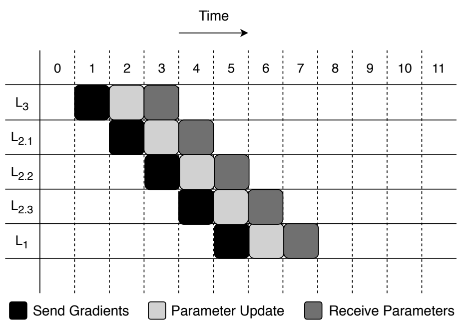


(a) ResNet-50 (b) InceptionV3 (c) VGG-19 (d) Sockeye

Figure 5. Parameter distribution



(a) Layer level granularity



(b) Fine granularity

Figure 6. Coarse and fine granularity

4.1 KVStore: Baseline system

KVStore is a wrapper implemented on top of the light-weight parameter server *ps-lite* (Li et al., 2014). KVStore has two components: KVWorker which runs locally to the worker machine as part of the training process and a separate server process called KVServer. KVWorker is responsible for sending gradients and receiving parameter updates from KVServer. KVServer is responsible for receiving gradients from KVWorkers, aggregating them and updating the parameters while ensuring data consistency. KVServer stores the parameters at layer level as a key-value pair, where key is the index of a layer and value is an array of floating points each corresponding to the parameter values of that layer. For load balancing purposes, more than one KVServers can be used for the training with the parameters equally sharded between them. For better resource utilization, a common practice is to run a KVServer on every machine with a worker process.

Before starting the training process, KVStore initializes and distributes the parameters of all the layers among the KVServers. KVStore follows a simple heuristic for fair distribution of parameters. Layers with size smaller than a fixed threshold are assigned to a randomly chosen KVServer. Parameters of larger layers are split equally among the KVServers. This is different from parameter slicing used in *P3* (explained in Section 4.2). The threshold is a configurable parameter and is set to 10^6 parameters by default.

KVServer exposes two main interfaces to each KVWorker for sending gradients and requesting updated parameters: a *Push* request and *Pull* request. During training, MXNet issues a parameter synchronization request for a layer to the KVServer through the KVWorker as soon as the back propagation of that layer has finished. KVWorker serializes (and fragments in case of large layers) the gradient matrix and issues a Push request to the corresponding KVServer(s). KVServer keeps a counter for the number of updates received on a key-value pair for maintaining data consistency. KVServer aggregates gradients until it has received updates from all the workers. Once all the gradient updates have been received, KVServer updates the parameters using the aggregated gradient values.

Once the parameters are updated KVServer notifies all the workers and resets the counter for that key-value pair. When KVWorker receives a notification, it immediately issues a Pull request to the KVServer(s) for the corresponding updated parameter values. KVServer then sends the latest parameter values in response and KVWorker (reconstructs for large layers) updates the local parameter values for the next iteration. MXNet overlaps the parameter synchronization of the layers by asynchronously issuing Push requests for the layers whose gradients are ready to be propagated.

4.2 *P3*: Implementation

In order to implement *P3*, we modify KVWorker and KVServer into *P3*Worker and *P3*Server. On the worker side, when a parameter synchronization is issued, *P3*Worker splits the gradient matrix of the layer based on a predefined size threshold (choice of this threshold is explained in Section 5.6). Unlike KVStore, this threshold defines the maximum granularity with which layers are split. This is the parameter slicing part in *P3*. For load balancing purposes, each of these slices are assigned to a *P3*Server in a round robin fashion.

The priority-based gradient propagation is implemented using a producer-consumer mechanism communicating through a priority queue. After parameter slicing, the producer part of *P3*Worker assigns priorities to the individual slices and pushes slices onto the priority queue all at once. A separate consumer thread in the *P3*Worker continuously polls the highest priority slice from the queue and sends the slice to the *P3*Server through the network with its priority stamped on the packet header. The consumer thread uses blocking network calls, so the rate at which the priority queue is polled is automatically adjusted based on the networking delays of the data transmission. This simple producer-consumer model makes sure that the network does not experience bursty traffic flow from the *P3*Worker at the same time the backward propagation is not hindered at the worker side. Also the slice with the highest priority in the priority queue always gets the first preference for transmission.

We also add a producer-consumer mechanism at the receiving end of the *P3*Server in order to deal with in-network delays. The packets received at the *P3*Server are pushed onto a priority queue with the priority assigned by the *P3*Worker as the key. A server consumer thread then polls from this queue and processes the packet the same way as in a KVServer. Prioritization at the *P3*Server ensures highest priority parameters are processed first.

Apart from these modifications, we remove the explicit update notification and pull requests from the KVServer. *P3*Server immediately broadcasts the updated parameters to all workers once it has received all of the updates. Since workers always issue a pull request after every push, this change does not affect the correctness of the training algorithm. This modification was necessary because MXNet only issues a pull request once it has received the update notification for all the slices of a layer. Eliminating this helped to improve the bidirectional bandwidth utilization. Since individual slices are synchronized independent to each other, sending gradients for a slice can be overlapped with the parameter updates received by another.

5 Evaluation

5.1 Methodology

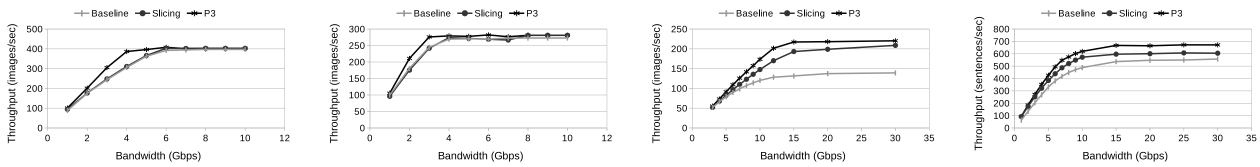
We have evaluated the *P3* implementation on three image classification models: ResNet-50 (He et al., 2015), InceptionV3 (Szegedy et al., 2015), VGG-19 (Simonyan & Zisserman, 2014) and on an LSTM-based model, Sockeye (Hieber et al., 2017). In all performance evaluation experiments we chose the standard MXNet KVStore implementation described in Section 4.1 as the baseline. Since *P3* implementation does not interfere with the model implementation or the training algorithm, the model convergence is not affected in any way. This means the baseline and *P3* would follow the same training curve for a given hyper parameter set. Under this condition, the improvement in training performance is completely determined by the rate at which input data is processed. Therefore the primary performance comparison metric we use is the training throughput, which is the number of total training samples processed by the worker machines in one second. The throughput measurements are taken after training the models for a few iterations until the throughput has become stable and averaged over 1000 iterations. In all the experiments we set the number of KVServers/*P3*Servers equal to the number of worker machines.

We conduct performance evaluation of *P3* in three different experiments. Section 5.2 shows how resilient *P3* is towards bandwidth limitations in the network. We perform this experiment by training the model on a four machine cluster each equipped with one Nvidia P4000 GPU (NVIDIA Corporation) and interconnected with a 100Gbps InfiniBand network (Shanley, 2002). We measure throughput variation while artificially limiting the network interface transmission rate using Linux's *tc qdisc* utility. Section 5.3 shows how well *P3* utilizes the available bandwidth and reduces the network idle time. The network utilization is measured per interface level using Linux's *bwm-ng* tool at a 10 millisecond granularity. Finally, in Section 5.4 we test the scalability of *P3* on different cluster sizes. This experiment is conducted on AWS using g3.4xlarge machine instances on a 10Gbps network.

In Section 5.5, we compare the convergence accuracy for models trained using *P3* and compression based techniques. For this comparison study, we picked the state-of-the-art compression technique Deep Gradient Compression (DGC) (Lin et al., 2018). We implemented DGC on top of the baseline MXNet based on the details provided in the original paper and the information collected from the authors. In addition to these experiments, we have also evaluated the effects of different parameter slice sizes on the training throughput in Section 5.6.

5.2 Bandwidth v.s. throughput

In this experiment, we analyze how much improvement *P3* can provide on throughput compared to the baseline implementation when the network bandwidth is not sufficient enough for training. We measure the training throughput of ResNet-50, InceptionV3, VGG-19 and Sockeye on a tightly controlled four-machine cluster by setting different transmission rates on the network interface on all the machines. Figure 7 compares the throughput from *P3* with the baseline system for different network bandwidths. We also measured the performance benefits achieved from only using the parameter slicing optimization.



(a) ResNet-50 (b) InceptionV3 (c) VGG-19 (d) Sockeye

Figure 7. Bandwidth v.s. Throughput

In Figure 7(a) and 7(b), both baseline and *P3* give similar training performance when the network bandwidth is sufficient enough for scaling these models on 4 machines. However, the baseline throughput starts to drop in ResNet-50 below 6Gbps. At the same time, *P3* maintains the linear throughput until the bandwidth drops below 4Gbps. This is because *P3* reduces the peak bandwidth required for the model by efficiently overlapping communication with the computation. At 4Gbps, *P3* provides 26% more throughput than the baseline. For InceptionV3, the maximum speed up obtained is 18%. It is interesting to note that these models does not benefit from parameter slicing, as the layer sizes are relatively small in these DNNs (Figure 5(a) and 5(b)).

Figure 7(c) and 7(d) shows the throughput of VGG-19 and Sockeye. These models contain one or two very large layers (Figure 5(c) and 5(c)), and because of the presence of these large layers, the parameter slicing optimization alone is giving considerable improvement in performance. At 30Gbps, parameter slicing can provide 49% speedup on VGG-19. The speedup is further improved with *P3* by as much as 66% at 15Gbps. Sockeye is a special case among other models. Unlike image classification models, the heaviest layer in this model is the initial layer. In Figure 7(d), Sockeye performance has improved by a maximum of 38% with *P3*. We observe that *P3* always performs better than the baseline with higher performance benefits under limited bandwidth conditions. Performance benefits of *P3* diminish when the network bandwidth is lower. This is because the communication time is significantly higher and there is little room for improvement by overlapping communication with computation.

5.3 Network utilization

This experiment compares the network utilization of *P3* with the baseline system. We conduct this experiment for ResNet-50, VGG-19 and Sockeye and measure the traffic generated and received by one of the four worker machines. Figure 8 shows the network utilization of baseline system. The baseline implementation has bursty network traffic generated with regular peaks and crests across all models. This pattern is observed in TensorFlow as well. Figure 8(b) shows the network utilization of ResNet-50 on TensorFlow over 4Gbps network. Similar to MXNet, TensorFlow also underutilizes the available network bandwidth. For the Sockeye model, the network idle time of ResNet-50 and VGG-19 is extremely dominant because of the heavy initial layer. Moreover, the inbound and outbound traffics are not overlapped as the baseline fails to fully utilize bidirectional bandwidth.

Figure 9 shows the network utilization graph with *P3*. We observe that *P3* improves the network utilization compared to baseline. In Figure 9(a) and 9(b), the network idle time has been considerably lowered with *P3*. Especially for Sockeye in Figure 9(c), *P3* utilizes bidirectional bandwidth more effectively than baseline system . This is one of the key reasons for the speedup observed for Sockeye model despite having a heavy initial layer.

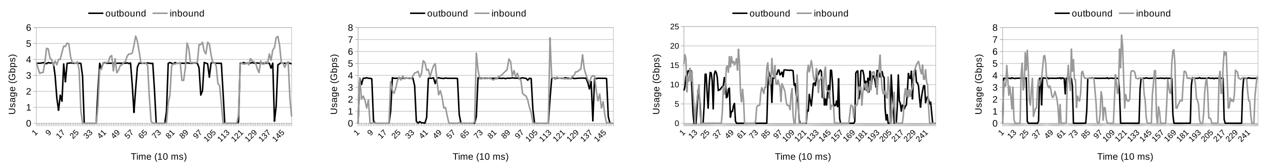
5.4 Scalability

We perform scalability analysis on ResNet-50, VGG-19 and Sockeye in order to show how well *P3* can perform on large clusters compared to the baseline system. We conducted this experiment by distributing models on clusters of different sizes (2, 4, 8 and 16) over a 10Gbps network. Figure 10(a) shows, for ResNet-50 both the baseline and *P3* perform similarly. As shown in Section 5.2, 10Gbps network is more than enough for linearly scaling ResNet-50. The throughput of VGG-19 has been considerably improved with *P3* by as much as 61% on an eight machine cluster (Figure 10(b)).

Figure 10(c) shows the scalability of Sockeye. LSTM-based models are very hard to scale over multiple machines, because of the heavy initial layers and difference in iteration time in worker machines due to the variable sequence length of input data. With *P3*, we improve Sockeye throughput by as much as 18% on eight-machine cluster.

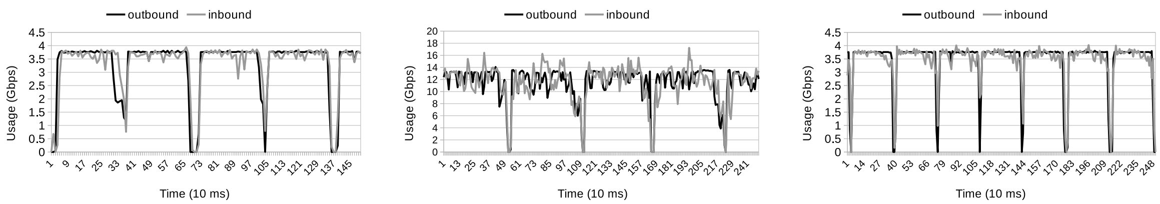
5.5 Training accuracy

As described in the Section 1, there are many compression techniques proposed for improving data parallel training performance. These methods can provide higher performance gains compared to *P3* , however, at the cost of loss in the final convergence accuracy. In this section, we compare convergence accuracy of *P3* with the state-of-the-art compression technique Deep Gradient Compression (DGC) (Lin et al., 2018).



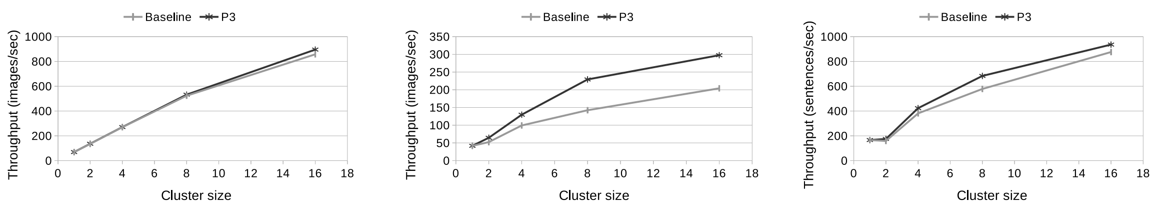
(a) ResNet-50 at 4Gbps (b) ResNet-50 at 4Gbps on TensorFlow (c) VGG-19 at 15Gbps (d) Sockeye at 4Gbps

Figure 8. Network utilization of the baseline system



(a) ResNet-50 at 4Gbps (b) VGG-19 at 15Gbps (c) Sockeye at 4Gbps

Figure 9. Network utilization of *P3*



(a) ResNet-50 (b) VGG-19 (c) Sockeye

Figure 10. Throughput scaling with different number of machines

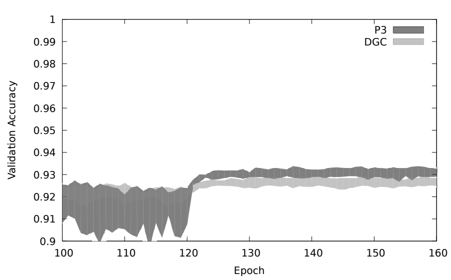


Figure 11. *P3* v.s. DGC

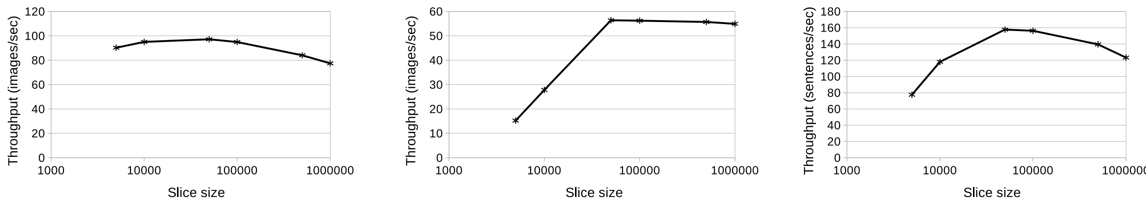
We trained ResNet-110 on the CIFAR-10 dataset over a 4 machine cluster with both *P3* and DGC using 5 different hyper parameter settings for 160 epochs. Figure 11 shows the validation accuracy range of *P3* and DGC from these experiments. The dark bands represent the gap between the worst and best accuracy on the 5 hyper parameter setting. We observe that the final accuracy obtained with *P3* is always better than DGC. We calculate an average accuracy drop of 0.004 with DGC.

Unlike compression based mechanisms like DGC, *P3* always communicate the full gradients with other worker machines and does not make any modification in the original SGD algorithm. Because of this the performance benefits from *P3* comes without any penalty on model accuracy.

5.6 Parameter slice size selection

As we showed in Section 4, a small gradient packet size can improve the network utilization and, in turn, can improve overall training throughput. In this section, we show how the size of the parameter slice affects training performance. Figure 12 shows the throughput obtained for ResNet-50 and VGG-19 with *P3* on different parameter slice sizes.

Initially, throughput increases as size decreases, and reaches a peak at $50,000$ and then it start dropping. This happens because if the size is made too small, the overhead of synchronizing packets at very small granularity is higher and dwarfs the benefits of parameter slicing. In all our experiments, we used a maximum granularity of $50,000$ parameters per slice.



(a) ResNet-50 (b) VGG-19 (c) Sockeye

Figure 12. Granularity v.s. Throughput

6 RELATED WORK

In this paper, we describe the key limitations in the data parallel deep learning distribution techniques used in popular machine learning frameworks (e.g., TensorFlow and MXNet), and propose solutions to mitigate these limitations by taking advantage of domain specific characteristics of deep learning models. To the best of our knowledge, this is the first work to summarize and address these issues.

One notable prior work which proposes domain specific optimizations for data parallel deep learning workloads is Poseidon (Zhang et al., 2017). This work introduced the idea of wait-free-back-propagation (WFBP) which hides the communication overhead behind back propagation by independently synchronizing individual layers in the neural network. In our work, we built upon this idea, and show that we can overlap computation with both forward and backward propagation. We further improve this idea by using parameter slicing that utilizes network bandwidth better.

Most recent work in this area tries to reduce communication overhead by sending fewer gradients. One popular method to reduce data transmission is gradient quantization (representing the gradient values using fewer bits). For example, 1-bit SGD (Seide et al., 2014) represents a 32-bit floating point gradient value in a single bit. In order to account for the information loss that comes with the value approximation, 1-bit SGD also add an error feedback in the SGD algorithm. 1-bit SGD can provide up to $10\times$ speed up. QSGD (Alistarh et al., 2017) and TernGrad (Wen et al., 2017) use similar methods but also provide mathematical guarantees on convergence.

Another approach is sparse parameter synchronization. The idea is to synchronize only a few parameters on every iteration instead of the whole model. Gradient dropping method only synchronizes parameters which have gradient values more than a threshold. The threshold is calculated based on a fixed compression ratio (Aji & Heafield, 2017). AdaComp (Chen et al., 2017) automatically tunes the compression ratio depending on the local gradient activity and achieves up to $200\times$ compression.

All the above techniques make trade offs between training performance and model accuracy because of the information loss introduced by value approximation or stale parameter updates (Khoram & Li, 2018). *P3* on the other hand, does not introduce any information loss since it always sends full gradient matrix on every iteration.

Recent work, called Deep Gradient Compression(DGC) (Lin et al., 2018), offers up to $600\times$ compression and around $5\times$ speedup in low bandwidth networks while maintaining the same baseline accuracy on several DNN models. DGC use local gradient accumulation and momentum correction techniques to maintain the same accuracy. Even though the authors report no accuracy loss with DGC, there is no formal proof on the convergence guarantees cited in the paper. And as shown in Section 5.5, we find it difficult to reproduce their results despite substantial effort[2]. In our experiments, *P3* always gives better accuracy than the DGC. We conclude that our mechanism is a safer approach, as *P3* does not introduce information loss in the training algorithm and therefore there is no potential risk of accuracy loss. Moreover, our proposal is an orthogonal approach to the compression techniques and can be used on top of compression mechanisms to further improve performance.

7 CONCLUSION

In this paper, we analyze the data parallel distributed training methods used in current machine learning frameworks and observe that they fail to fully utilize available network bandwidth and induces high penalty on training performance under bandwidth limitations. Based on this observation we propose a new parameter synchronization method called *P3* which improves the training performance by better utilizing the available network bandwidth. We implemented *P3* over MXNet and demonstrate it to have higher resiliency towards bandwidth constraints and better scalability than the baseline MXNet implementation. With *P3*, we improved training throughput of ResNet-50 by as much as 25%, Sockeye 38% and VGG-19 66%.

[2]This includes personal communication with the authors in order to get all their experiments correctly.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Aji, A. F. and Heafield, K. Sparse communication for distributed gradient descent. *CoRR*, abs/1704.05021, 2017. URL http://arxiv.org/abs/1704.05021.

Alan, M., Panda, A., Bottini, D., Jian, L., Kumar, P., and Shenker, S. Network evolution for dnns. *SysML*, doc/182, 2018. URL http://www.sysml.cc/doc/182.pdf.

Alistarh, D., Grubic, D., Li, J., Tomioka, R., and Vojnovic, M. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 1709–1720. Curran Associates, Inc., 2017.

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J., Fan, L., Fougner, C., Han, T., Hannun, A. Y., Jun, B., LeGresley, P., Lin, L., Narang, S., Ng, A. Y., Ozair, S., Prenger, R., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, Y., Wang, Z., Wang, C., Xiao, B., Yogatama, D., Zhan, J., and Zhu, Z. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015. URL http://arxiv.org/abs/1512.02595.

Awan, A. A., Chu, C., Subramoni, H., and Panda, D. K. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: MPI or nccl? *CoRR*, abs/1707.09414, 2017. URL http://arxiv.org/abs/1707.09414.

Bottou, L. Large-scale machine learning with stochastic gradient descent. In *in COMPSTAT*, 2010.

Burtscher, M. and Ratanaworabhan, P. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.*, 58(1):18–31, January 2009. ISSN 0018-9340. doi: 10.1109/TC.2008.131. URL http://dx.doi.org/10.1109/TC.2008.131.

Chen, C., Choi, J., Brand, D., Agrawal, A., Zhang, W., and Gopalakrishnan, K. Adacomp : Adaptive residual gradient compression for data-parallel distributed training. *CoRR*, abs/1712.02679, 2017. URL http://arxiv.org/abs/1712.02679.

Chen, J., Monga, R., Bengio, S., and Józefowicz, R. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016. URL http://arxiv.org/abs/1604.00981.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274.

Cunningham, D., Lane, B., and Lane, W. *Gigabit Ethernet Networking*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1999. ISBN 1578700620.

Daily, J., Vishnu, A., Siegel, C., Warfel, T., and Amatya, V. Gossipgrad: Scalable deep learning using gossip communication based asynchronous gradient descent. *CoRR*, abs/1803.05880, 2018. URL http://arxiv.org/abs/1803.05880.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Hieber, F., Domhan, T., Denkowski, M., Vilar, D., Sokolov, A., Clifton, A., and Post, M. Sockeye: A toolkit for neural machine translation. *CoRR*, abs/1712.05690, 2017. URL http://arxiv.org/abs/1712.05690.

Keuper, J. and Preundt, F.-J. Distributed training of deep neural networks: Theoretical and practical limits of parallel scalability. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, MLHPC '16, pp. 19–26, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-5090-3882-4. doi: 10.1109/MLHPC.2016.6. URL https://doi.org/10.1109/MLHPC.2016.6.

Khoram, S. and Li, J. DNN model compression under accuracy constraints, 2018. URL https://openreview.net/forum?id=By0ANxbRW.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 583–598, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685095.

Lin, Y., Han, S., Mao, H., Wang, Y., and Dally, B. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

Luo, L., Nelson, J., Ceze, L., Phanishayee, A., and Krishnamurthy, A. Parameter hub: a rack-scale parameter server for distributed deep neural network training. *CoRR*, abs/1805.07891, 2018. URL http://arxiv.org/abs/1805.07891.

NVIDIA Corporation. Nvidia quadro p400. URL https://www.pny.com/nvidia-quadro-p4000.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature*, 323:533–, October 1986. URL http://dx.doi.org/10.1038/323533a0.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, September 2014.

Shanley, T. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321117654.

Shi, S. and Chu, X. Performance modeling and evaluation of distributed deep learning frameworks on gpus. *CoRR*, abs/1711.05979, 2017. URL http://arxiv.org/abs/1711.05979.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017. URL http://dx.doi.org/10.1038/nature24270.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 1509–1519. Curran Associates, Inc., 2017.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., ukasz Kaiser, Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P., and Xing, E. P. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pp. 181–193, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-38-6.

Zhu, H., Akrout, M., Zheng, B., Pelegris, A., Phanishayee, A., Schroeder, B., and Pekhimenko, G. TBD: benchmarking and analyzing deep neural network training. *CoRR*, abs/1803.06905, 2018. URL http://arxiv.org/abs/1803.06905.