Multi-Node
Stochastic Gradient Descent

Tom Neckermann

Master of Science

School of Informatics
University of Edinburgh

2017

Abstract

This dissertation presents the implementation of a multi-node Stochastic Gradient Descent (SGD) algorithm for neural network training. We combine parameter sharding, overlapping computation with communication, and gradient dropping to achieve fast training in clusters of multi-GPU machines. We show how each of these extensions impacts both training throughput and convergence rate for a variety of different multi-node configurations. We find that the slow communication between machines (nodes) is a major bottleneck. Without compressed communication, we find it difficult to scale training efficiently over multiple nodes. With sparse communication, however, we achieve impressive convergence rates, beating popular methods like SimuParallelSGD (Zinkevich et al., 2010) and DistBelief (Dean et al., 2012). Future work includes optimising the hyperparameters of our framework to achieve more optimal convergence rate and implementing various tweaks that our experiments indicate might further improve performance.

`https://github.com/dr-alan-turing/marian-nmt-distributed`

Acknowledgements

Thanks to Dr. Kenneth Heafield for giving me the opportunity to work on this exciting project and for the advice and general guidance provided throughout the course of my dissertation. Thanks to the Marian team for building the single-node framework that was used as the starting point of my implementation. Thanks to my friends and family for supporting me through the successful completion of this project; in particular Elias, Manolis, Dimitris, Rick and Morty.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Tom Neckermann

Table of Contents

Chapter 1

Introduction

1.1 Context

Stochastic Gradient Descent (SGD) is an iterative optimisation method for minimising an objective function. In a neural network, a training sample propagated through the network results in an error which can be used to compute an error gradient for each parameter/weight of the network. The most basic SGD method involves repeating this process for each training input and updating after every iteration the parameters by adding a fraction of the error gradient to each of them. Thus, its goal is find the optimal weights of a neural network given the training samples. Methods like AdaGrad and Adam are combined with SGD to add modifiers like momentum and adaptive learning rate to speed up this process and avoid finishing in a local instead of global optimum.

1.2 Problem and Goals

Because neural networks are often complex and can contain over a hundred million parameters, it can take SGD weeks to converge to an optimum. Thus, there is an incentive to speed up this process by running it on several machines (nodes) simultaneously. Since SGD typically uses mini-batches, i.e. small subsets of the training samples, it is possible to replicate the network over multiple nodes and feed different mini-batches into each of them. This results in different gradients at each node, which can be aggregated and then used to update the parameters of all networks. In practice, there are many issues relating to the synchronisation between machines. In particular, communication between nodes is very slow and thus it is infeasible to keep all networks' parameters continuously synchronised with each other. This project involves

the implementation of an asynchronous approach where nodes continuously compute gradients locally and synchronise with other nodes in a delayed and out-of-sync fashion. To maximise the utilisation of the processing and communication resources of each machine, we overlap the network's computations with the inter-node communication of gradients and parameters. Furthermore, we implement a sparse communication scheme to reduce the communication time between nodes.

1.3 Structure

We start by introducing Stochastic Gradient Descent and its application in the training of neural networks. Subsequently, we give an overview of existing research in the parallelisation of SGD. Specifically, we describe data parallelism, model parallelism, GPU training and communication compression techniques. We then introduce our implementation, describing how we implemented various concepts from the background section and the trade-offs in our choices. We then present experiments where we test multiple configurations of our implementation on various combinations of machines and GPUs. We briefly compare our results with those found in related works. Finally, we finish the dissertation by discussing the limitations of our framework and providing an outline for future improvements.

Chapter 2

Background

2.1 Artificial Neural Networks

2.1.1 The Neuron

The fundamental unit of an Artificial Neural Network (ANN) is the neuron. Depicted in Figure 2.1, a neuron receives one or more inputs along with a constant value (called *bias*), re-scales them according to parameters (weights), and passes them as arguments into an activation function. The neuron's output can thus be described by the equation:

$$a = f(\sum_{i=1}^{n} w_i \cdot x_i)$$

where f represents the activation function and $w_i \cdot x_i$ is the dot product of the ith weight and the ith input (consider bias a constant input with its own weight). Because any function can be used as activation, an ANN is very flexible and can model complex non-linear relationships in data. Traditional functions used are Sigmoid, which "squashes" real values into a range between 0 and 1 (Williams, 1983), and ReLU, which returns the input thresholded at zero (Dahl et al., 2013).

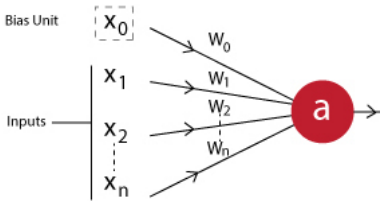


Figure 2.1: Artificial Neuron[1]

[1] https://www.analyticsvidhya.com/blog/2016/03/introduction-deep-learning-fundamentals-neural-networks/

2.1.2 Feed-Forward Networks

In feed-forwards ANNs, neurons (nodes) are connected in an acyclic graph where the outputs of neurons are fed as inputs to other neurons, as shown in **Figure 2.2**. The nodes in the input layer receive features from a data sample (e.g. image) and each sends the output from their activation function to every node in the first hidden layer. The nodes in a hidden layer send their outputs to one or more nodes in the next layer. The number of hidden layers and nodes per layer is not limited. The final layer is the output layer, where the nodes produce a final result, e.g. a classification of the input data sample.

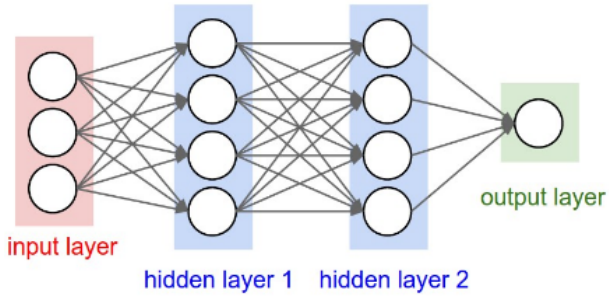


Figure 2.2: Feed-Forward ANN[2]

There are many other types of neural networks, some vastly more complex than feed-forward networks. We only introduce a simple network here because it is sufficient for the purpose of this dissertation.

2.2 Gradient Descent

2.2.1 Overview

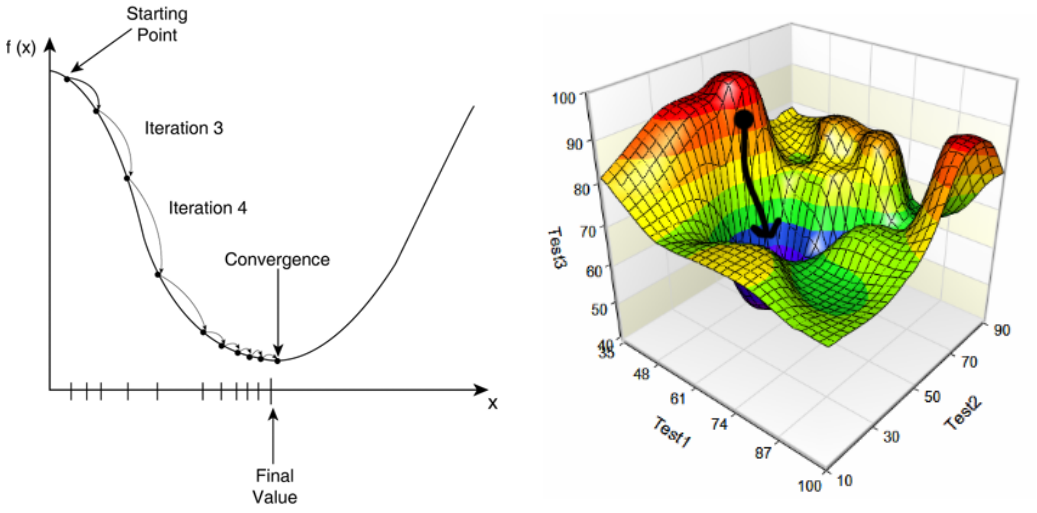
The goal of Gradient Descent (GD) is to find the arguments of a function that result in its global minimum. This is achieved by starting at any point and iteratively moving the parameters by a fraction of their gradients with respect to the function's output. Figures 2.3 (a) and (b) illustrate the path GD might take on a function with 1 dimensional input and 2 dimensional inputs, respectively.

In its most basic form, Gradient Descent can be described by the equation:

$$x_{t+1} = x_t - \eta \nabla F(x_t)$$

[2]http://cs231n.github.io/neural-networks-1/

where x_{t+1} represents parameters x at iteration $t+1$ and $\nabla F(x_t)$ the gradient of the function F for parameters at iteration t. The step size (or learning rate), denoted by η, determines the amount by which the coefficients should change on this update. A large step size can minimise the function faster but risks overshooting and bringing the algorithm to a non-optimal point. A small step size can avoid this but converges slower and might end up in a local minimum.



(a) 1 dimensional input[3]

(b) 2 dimensional input [4]

Figure 2.3: Gradient Descent on a function

2.2.2 Gradient Descent in Machine Learning

In machine learning, the function that is being minimised is usually the cost (error) function, which is based on the difference between the value predicted by the model and the actual (expected) result for a given training sample. The model is initialised with random or predefined parameters and Gradient Descent iteratively adjusts their values in a way that decreases the cost for that same data.

Notable challenges in GD include dealing with local minima and saddle points. Effectively, the gradient points to the path of steepest descent, which does not necessarily lead to the global minimum. A saddle point is where the gradient is zero. While methods that address these problems exist – e.g. adaptive learning rate for local minima (Zeiler, 2012) and Taylor's expansion for saddle points (Narendra and Mukhopadhyay,

[3] http://www.yaldex.com/game-development/1592730043_ch18lev1sec4.html

[4] https://thoughtsahead.com/2017/01/27/machine-learning-series-introduction-to-machine-learning-linear-regression-and-gradient-descent

1997) - they remain an active area of research.

An important consideration is which samples to use for adjusting the parameters in each GD iteration. In Batch Gradient Descent (BGD), all samples from the training dataset are used to compute the gradient. The previous equation can thus be rewritten as follows:

$$w_{t+1} = w_t - \eta \sum_{i=1}^{n} \nabla L_i(w_t)/n$$

where w_t represents the weights (parameters) at iteration t and $\nabla L_i(w_t)$ the gradient of the loss function L at the i-th example w.r.t the weights w_t. Essentially, the gradients of all n samples are averaged to determine the update direction. BGD has been shown to work well for convex functions and those with relatively smooth error manifolds (Wilson and Martinez, 2003).

In Stochastic Gradient Descent (SGD), the true gradient is approximated by that of a single, usually random, sample. The result is significantly faster computation and a much noisier gradient. The latter is beneficial in error manifolds with many local minima because it tends to "jerk" the model out of them (Bottou, 2010). In practice, SGD is typically used with mini-batches (small subsets) of training examples. This averages out the noise while maintaining the desirable properties of SGD, allowing problem-specific tuning via the batch size.

2.2.3 Updating the Weights in a Neural Network

Updating the weights (parameters) of a neural network can be described in four main steps. The first is the forward pass, in which one or more training samples are run through the network to produce a result in the output layer. The second step involves computing the network's error for the input sample(s) by comparing its output with the expected result. This is achieved via a cost (or loss) function such as Mean Squared Error (Wang and Bovik, 2009) or Cross-Entropy (Deng, 2006). In the third step, the error is propagated backwards through the network to obtain at each neuron a cost that represents its contribution to the total error. The Backpropagation method (Hecht-Nielsen et al., 1988) uses these values to compute the partial derivative (gradient) of the cost function with respect to each weight. These gradients are used in the fourth and final step to update the network's parameters using SGD as described above.

2.2.4 Extensions to Basic SGD

Various extensions to basic SGD have been proposed to increase the speed at which the algorithm converges to a minimum. In Momentum (Rumelhart et al., 1988), a fraction of the previous update is added to the current update. This results in an increased velocity for consecutive updates in the same direction. Adaptive Gradient Algorithm, or AdaGrad (Duchi et al., 2011), implements per-parameter learning rates that adapt based on the gradient changes of previous iterations. It performs larger updates for infrequently updated parameters, and smaller updates for frequently updated parameters. Adaptive Moment Estimation, or Adam (Kingma and Ba, 2014), builds on AdaGrad by adding an adapted gradient on top of an adapted learning rate. It essentially stores momentum changes for each of the parameters.

2.3 Related Works

The iterative nature of Stochastic Gradient Descent makes it an inherently sequential algorithm. This means that training a single network can take weeks depending on its complexity. Most approaches therefore run SGD on a multiple networks simultaneously and combine the gradients in a way that speeds up training overall. We present some of the most common approaches below. Note that any subsequent use of the word 'node' refers to a processor, GPU or machine, not a neuron of a neural network.

2.3.1 Data Parallelised Training

The parallelisation of iterative algorithms over multiple compute nodes has been around for many years (e.g. Arrow and Hurwicz, 1958; Baudet, 1978). Tsitsiklis et al. (1986) describe a distributed model wherein individual processors run stochastic optimisation algorithms side-by-side and occasionally exchange their function's outputs with each other. Upon receiving a message, a processor performs a convex combination of its old vector and the new values to evaluate its new parameters. Färber (1997) applies this approach to the training of neural networks in what he calls "pattern parallelism", commonly known as data parallelism today. Each processor (node) has an independent replica of the model (neural network) which is trained on a different subset of the training data. The nodes sum up their computed gradients locally, until a fixed number of samples have been processed (batch size) and a master node reads the accumulated gradients, updates the global weights and broadcasts them to the nodes. This is syn-

chronous in that the nodes wait for each other and are in complete synchronisation after every master update. A major limitation of Färber's approach is that the communication overhead scales linearly with the number of nodes in the cluster, i.e. with twenty nodes the communication takes ten times as long as with two nodes. Therefore, achieving a constant communication to computation ratio requires the batch size to increase with the number of nodes. This results in fewer parameter updates and thus a lower convergence rate for the same number of iterations.

A more recent and commonly cited algorithm is Zinkevich et al.'s (2010) SimuParallelSGD. As before, the full model is replicated on multiple nodes – in this case, machines – and each receives a subset of the training data. The main difference is that there is no communication between nodes during training. Instead, each node has its own parameters that it updates by running SGD over a randomly selected training sample in each iteration. After a fixed number of iterations, a master routine aggregates and averages the weights from all nodes, giving a final weight vector. The main advantage of this algorithm is that it doesn't suffer from communication overhead because there is no synchronisation until the end. Naturally, Zinkevich et al.'s algorithm scales linearly with the number of nodes in terms of samples processed. Its convergence rate, i.e. the wall clock time to achieve a certain error on a test (non-training) data set, also shows significant speed-ups with multiple nodes. In their experiments using binary features from an email corpus, 10 machines converge around five times faster than a single machine. However, 100 machines only converge two times faster than 10 machines. Thus, each machine gives diminishing returns and the "scalability ceiling" is reached relatively quickly. Furthermore, this approach doesn't work for all kinds of neural networks. For example, Recht et al. (2011) point out that SimuParallelSGD does not outperform a serial scheme for neural networks involving matrix completion problems.

2.3.2 Model Parallelism

Färber (1997) presents one of the first model-parallelised training architectures, dubbed "network parallelism". Instead of replicating the model N times for N nodes, every node is responsible for 1/N neurons – and thus roughly 1/N parameters – of the neural network. Because activation outputs for many neurons are communicated across processors, the communication overhead scales linearly with the number of nodes in the cluster. The major advantage of this approach is that very large networks are possible

because the memory available also scales with the number of nodes. Due to this architecture's high communication overhead, it is most suitable for models with a large number of parameters or where the computations on the neurons are expensive.

2.3.3 Full Parallelism

In recent years, distributed SGD implementations usually make use of (variants of) both model and data parallelisation. Dean et al. (2012) describe Distbelief, a framework for training large neural networks using thousands of machines in a cluster. They employ model parallelism by partitioning the neurons over the machines and using inter-node communication at neurons where the edges cross partition boundaries. As part of the framework, Dean et al. introduce Downpour SGD, a data and model parallel algorithm. Like SimuParallelSGD (Zinkevich et al., 2010), Downpour distributes the training samples into subsets that are run in parallel across model replicas. The first distinction is that the model replicas are also distributed across multiple machines (i.e. model parallelism). The second distinction is that after every forward pass by a model (c.f. section 2.2.3), the resulting gradients are sent to a parameter server. The server then updates the global parameters and sends them back to that model. In synchronous SGD, the server waits for all models to have sent their gradients before performing the update and sending back the latest parameters to all nodes. Downpour SGD, on the other hand, is asynchronous in that the server updates and sends back parameters immediately after having received gradients. Thus there is loose synchronisation between model replicas. A major advantage is that nodes do not have to wait for each other, which also means that slower nodes can positively contribute to the training. An additional side-effect is increased stochasticity in the optimisation procedure, which has shown to increase convergence rate for various applications (e.g. Chilimbi et al., 2014). To maximise the scalability of Downpour, a sharded server is employed, i.e. the global parameters are distributed across multiple nodes, where each is responsible for updating and communicating 1/N of the parameters.

The experiments conducted by Dean et al. feature the training of networks for object recognition and acoustic processing tasks. In terms of number of samples processed for a given time interval, they achieve 2x speed-up on 8 machines (vs 1 machine) and 12x speed-up on 81 machines. This metric alone, however, does not indicate the true performance of the parallelisation. For example, on SimuParallelSGD (where communication only occurs at the end), the speed-ups would be roughly 8x

and 81x for 8 and 81 machines respectively. Ultimately, we care about the speed of convergence. For that metric, Downpour running on 600 CPU-based machines was only twice as fast as a single GPU in reaching 16% accuracy on a test set. Thus the main advantage of Downpour is its capability for training huge model sizes, like the 1.7 billion parameter network presented in their paper.

2.3.4 GPU Training

The results in Dean et al.'s experiments highlight the suitability of GPUs for training neural networks. Raina et al. (2009) present an architecture where model-parallelism is achieved by distributing the neurons and matrix computations over hundreds of cores on a single GPU. Their results show a single GPU achieving up to 72x speed-up compared to a top-of-the-line dual-core CPU for training image classifiers. These results help explain why most training is now performed on GPUs (Schmidhuber, 2015).

Noel and Osindero (2014) describe 'Dogwild!', an extension to 'Hogwild!' for CPUs and GPUs. In Hogwild (Recht et al., 2011), data parallelism across CPUs is employed with weak consistency. Specifically, multiple CPUs read from and write to the same parameter vector in shared memory simultaneously, without the use of locks. Thus updates from one processor can be overwritten by another processor. They show that for sparse optimisation problems – i.e. when an update only affects a small part of the parameters – a near optimal rate of convergence can be achieved despite the overwrites. Dogwild extends this approach by scaling it over multiple machines (i.e. multi-node). The global/server parameters are sharded across the machines, and each shard loops over its designated weights, broadcasting them to all nodes. When a node receives updated parameters, it overwrites those of its model and updates them with its gradients (summed since the last communication), also sending them to the server shard, which in turns applies them to the global parameters. Dogwild reduces communication overhead by using unreliable communication protocols, which show up to 10% packet loss in their tests but overall faster convergence rate than reliable transport like TCP or MPI. A major limitation of Noel and Osindero's paper is that they present non-conclusive results. Their primary experiments are for a particularly simple MNIST [5] task, where they achieve linear speed-ups in convergence rate. However, this training took less than 10 seconds to converge on a single CPU and is thus not a reliable indicator for performance on a larger task. They present an "early test" on a

[5] MNIST is a database for training digit detection in images.

more complex ImageNet task, yet they only compare one GPU with two GPUs, both running on the same node.

2.3.5 Reducing Communication Overhead

Gradient and parameter exchanges between processors (or GPUs) are expensive, especially when they are distributed across machines. Thus it is useful to summarise some techniques that result in decreased communication overhead.

Seide et al. (2014) introduce an SGD data-parallel method in which gradients are heavily quantised to a single bit per value. Usually, '1' implies a positive gradient and '0' a negative gradient. This bit-vector can be accompanied by a small number of floats, such as the average positive and negative gradients. This allows the receiving end to approximately reconstruct the original gradients. An important detail is that the quantisation error – i.e. the difference between the reconstructed and original gradients – is remembered and added to the next mini-batch's gradients before they are quantised. This "error feedback" ensures that all gradients are eventually applied to the receiving model. Seide et al.'s experiments with a 46 million parameter model indicate a convergence-rate speed-up of over 4X when using 4 nodes with 2 GPUs each compared to a single GPU on its own. Moreover, they show no accuracy loss despite the 1-bit quantisation. On a 160 million parameter model, a cluster of 20 dual-GPU nodes achieved a 10X speed-up compared to a single GPU, albeit at a small accuracy loss.

Alistarh et al. (2016) extend this concept with QSGD, a quantisation scheme where a gradient vector is quantised by randomised rounding each component to a discrete set of values that preserve statistical properties of the original. It is similar to 1-bit SGD except that the number of bits quantised to can be tuned with a parameter. Their experiments confirm Seide et al.'s findings that quantisation barely impacts accuracy. In terms of convergence rate, 2-bits QSGD appears the fastest while 8-bits QSGD maintains full accuracy (even beating non-quantised SGD in some tests) while also providing significant decreases in communication costs.

Aji and Heafield (2017) present an alternative method of reducing communication payloads. They note that most gradient updates are near zero (i.e. sparse), thus they propose mapping the 99% smallest values to zero and only communicating the remaining 1% across nodes. Because finding the threshold at which to keep or drop values is expensive (Alabi et al., 2012), they sample a small part of the gradient vec-

tor and determine an approximate threshold from there. Furthermore, to reduce the impact on convergence, dropped gradients are remembered and added as residuals to the next computed gradients. This concept is used for parameter synchronisation as well. In short, a server shard stores for each node the parameters that were last sent to it, which the server compares with the latest global parameters. The resulting deltas (difference between the vectors) are dropped in the same way as gradients and only the biggest changes are sent to the requesting node. Aji and Heafield show that even with 99.9% drop-rate, a model can still converge, albeit at a slower rate and worse accuracy. The results for 99% drop-rate show minimal impact on convergence rate and accuracy, despite a 50X decrease in communication payload. Unfortunately, the results do not include tests in a multi-node environment, where the potential benefits are much greater due to significantly more expensive communication costs.

2.3.6 Double Buffering

In data parallelised training, it can be inefficient for the computation of a model to wait for the communication of its gradients and parameters. This is especially true in a multi-node environment where communication between machines is particularly expensive. Hence the introduction of Double Buffering, a technique in which each mini-batch is divided into two pieces, such that while the gradients of the first half are being communicated the computation can proceed with the next half (Seide et al., 2014). This increases the utilisation of the processing units and communication channels. In Seide et al.'s tests, Double Buffering in data-parallel training on 8 GPUs does not impact accuracy and in fact gives over 20% speed-up in convergence rate compared to without it. Combined with 1-bit SGD, they achieve a 5.5X speed-up on 4 nodes with 2 GPUs each compared to a single node with 1 GPU.

2.3.7 The Effect of Delayed Updates

An important observation on Double Buffering is that because a node does not wait until it has received the latest parameters, it is essentially computing gradients using an outdated model (stale weights). This is in fact an implied property of data-parallelism as well: by the time a node has computed new gradients, the server parameters have likely been updated by other nodes. Thus nearly all updates to the global parameters can be considered delayed updates. Moreover, the more nodes there are in a cluster the more frequently the parameters are updated and thus the greater the delay of each

node.

Langford et al. (2009) compare the effects of different delays in updates to convergence rate. On text classification training, they find that even applying gradients that were computed on 1000 iteration-old parameters results in convergence. In fact, a delay of 100 updates performs nearly as well as no delays, while a delay of 10 is nearly unnoticeable. Chen et al. (2012) make similar conclusions about delayed updates, noting that momentum (c.f. section 2.2.4) and mini-batching (c.f. section 2.2.2) have similar effects. They conclude that while reasonable delays should barely impact convergence rate (if at all), the greater the parallelisation the longer the delays and thus the greater the risk of damaging convergence.

Chapter 3

Methods

3.1 Single node training

Our initial implementation was executed on a CPU port of the Marian framework, which by default features single-node training as follows. A mini-batch of training samples is passed to the neural network and the relevant computations are run in parallel across all CPU cores. The resulting gradients are applied to the parameters using an SGD method like AdaGrad or Adam. This is repeated for many mini-batches until convergence.

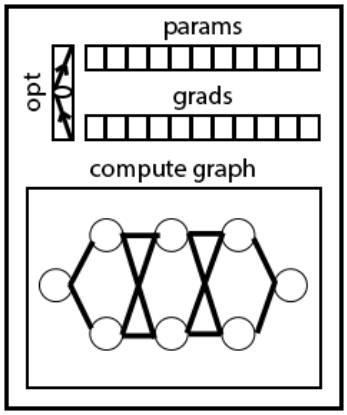


Figure 3.1: Single node structure

Algorithm 1: Single node training

```
initialise graph, opt, batches;
while not converged do
    batch = batches.pop();
    build(graph, batch);
    graph.forward();
    graph.backward();
    opt.update(params, grads);
end
```

Figure 3.1 and **Algorithm 1** detail how this is implemented. Each iteration makes use of a different mini-batch; the specifics of how a mini-batch is constructed is irrelevant at this stage. The input layers are then prepared to consume the samples in the current batch. The node's parameters (*params* in diagram) are used by the neural network (*compute graph*) to predict an output for each sample in the forward pass

(c.f. section 2.2.3). The gradients resulting from the backward pass are placed in the gradient vector (*grads*). The optimiser (*opt*) then applies the gradients to the node's parameters in a specific way, e.g. in basic SGD a fraction of each gradient is simply subtracted from each corresponding parameter (c.f. section 2.2.1). The next iteration will thus use the updated parameters in its forward pass.

3.2 Central parameter server

The simplest way to implement data parallelism (c.f. section 2.3.1) is through a central parameter server. While a node can be assigned the sole role of server, we run the required server functionality alongside the computational tasks of a given node. This enables each node, including the server, to contribute to the graph computations.

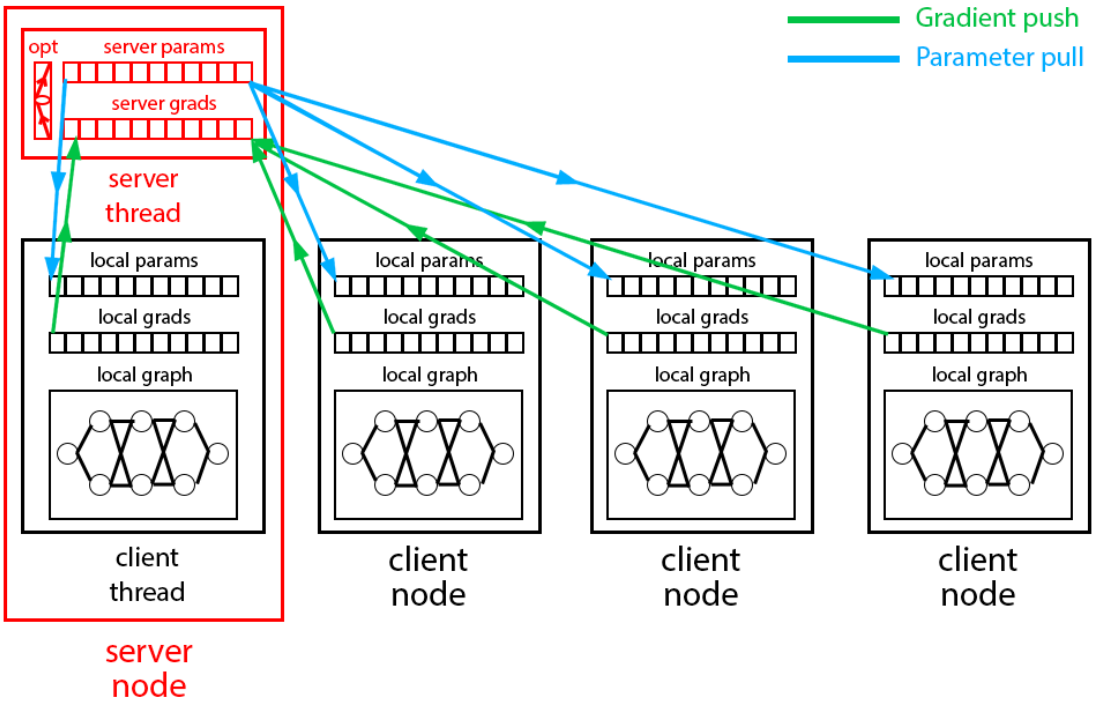


Figure 3.2: Central parameter server

Figure 3.2 illustrates how this is achieved in our program. The "server node" runs a client thread which behaves exactly like the other "client nodes" – i.e. it performs the computations relating to the node's neural network. An additional server thread runs in parallel and is responsible for interacting with nodes communicating their gradients. **Algorithm 2** outlines the specific behaviour of this thread. It continuously polls for a message from any source until it finds a matching send, in which case it receives the

Algorithm 2: Server thread

```
initialise params, opt;
while not converged do
    grads, client = recv(anySource);
    opt.update(params, grads);
    send(client, params);
end
```

Algorithm 3: Client thread

```
initialise graph, batches;
while not converged do
    build(graph, batches.pop());
    graph.forward();
    graph.backward();
    send(grads, server);
    params = recv(server);
end
```

gradients and remembers the sender (client). It then updates the global parameters with the new gradients, which it sends back to the same client. **Algorithm 3** describes the updated behaviour of a client. The only difference compared to **Algorithm 1** is that instead of updating the parameters by running the optimiser locally, the client sends its gradients to the central server and replaces its parameters with those it receives back.

The obvious limitation of this approach is that the server is communicating with a single node at any given time. This means that if a client is communicating with the server, all other clients that have paused their computation in order to push their newly computed gradients are stuck waiting. A naive solution is to increase the number of server threads running simultaneously. Besides the additional complexity that this would introduce (in particular, handling shared write access to the server's parameters), there is a limitation that uncovers the fundamental flaw of the central parameter implementation: the communication speed of one node. While the server is receiving gradients or sending parameters, its network interface is likely operating at full capacity. Therefore the only way to increase the number of simultaneous communications is to decrease the communication speed of each server thread. This will have the same effect as the single threaded server approach. Therefore, a central server node implementation does not scale with the size of the cluster.

3.3 Sharded (distributed) server

Our approach to the over-subscription of the server's communication channel is to use parameter sharding, as described by Dean et al. (2012). This involves distributing the server over all nodes, where each is responsible for a different part of the global

parameters.

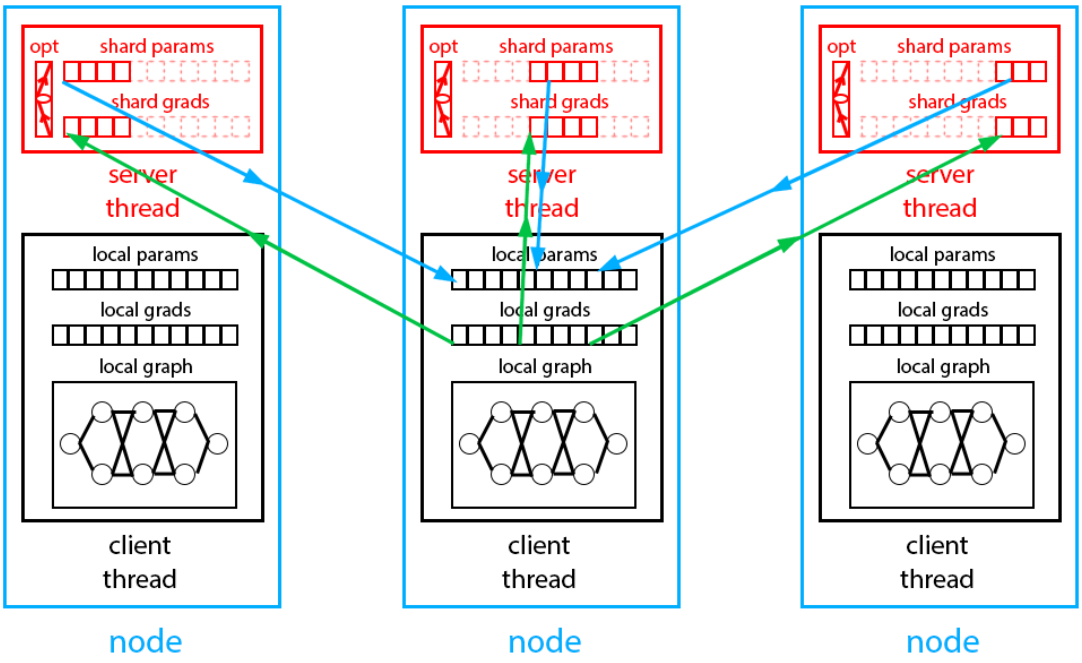


Figure 3.3: Sharded parameter server. Communication of middle node shown.

Our implementation achieves this by running on each node a client thread for computations and a server thread for its server shard responsibilities, as illustrated by **Figure 3.3**. The server thread behaves identically to that of the central parameter server in **Algorithm 2**, except that it only receives 1/N gradients and updates and sends back 1/N parameters, where N is the number of nodes in the cluster. The client thread's behaviour is described by **Algorithm 4**. Instead of synchronising with one server node, it splits its gradients and parameters into N pieces, synchronising each with the appropriate server shard. Note that a client only communicates with one shard at a time, instead of synchronising with all of them simultaneously. The latter would be less performant because it would be bottlenecked by the node's network speed and block other nodes from communicating with the same shards for an unnecessarily long time (there is only one server thread per shard – c.f. section 3.2).

This method scales efficiently as the size of the cluster grows. With double the number of nodes, there are twice as many clients communicating with each server shard, but the payload of each message is also halved because the parameters are now sharded over double as many machines. Thus the amount of communication per node is constant.

Parameter sharding does not address the problem that multi-noded communication

Algorithm 4: Client thread with parameter sharding

```
initialise graph, batches, offsets, sizes;
while not converged do
    build(graph, batches.pop());
    graph.forward();
    graph.backward();
    foreach shard ∈ cluster do
        subGrads = grads.sub(offsets[shard], sizes[shard]);
        send(subGrads, shard);
        subParams = recv(shard);
        params.sub(offsets[shard], sizes[shard]) = subParams;
    end
end
```

can be much slower than the computations of the model. Thus, a node might still spend more than half of its time waiting for messages to be sent and received.

3.4 Overlapping communication and computation

Essentially, we want each node to continually compute gradients with minimal pause between iterations. Thus, we overlap the model computations with the inter-node communication, based on the Double Buffering technique described in section 2.3.6. After a node has computed its gradients, it copies them to a communication buffer, from where they are communicated with the server shards while this node's computation proceeds with the next mini-batch. When the updated parameters have been received in another buffer, they are copied to the model to be used for the next mini-batch.

Our implementation is illustrated by **Figure 3.4**. Each node runs an additional "client communication" thread, which waits until its gradient buffer is filled and then sends its contents to the server shards, receiving updated parameters into another buffer (**Algorithm 5**). The "client computation" thread, depicted by **Algorithm 6**, performs the usual forward and backward passes, and then checks if the communication buffer is ready to exchange data, i.e. the communication thread is ready and waiting. If data is still being exchanged, the computation thread resumes to the next iteration and thus adds its latest gradients to a running sum of gradients. This ensures that

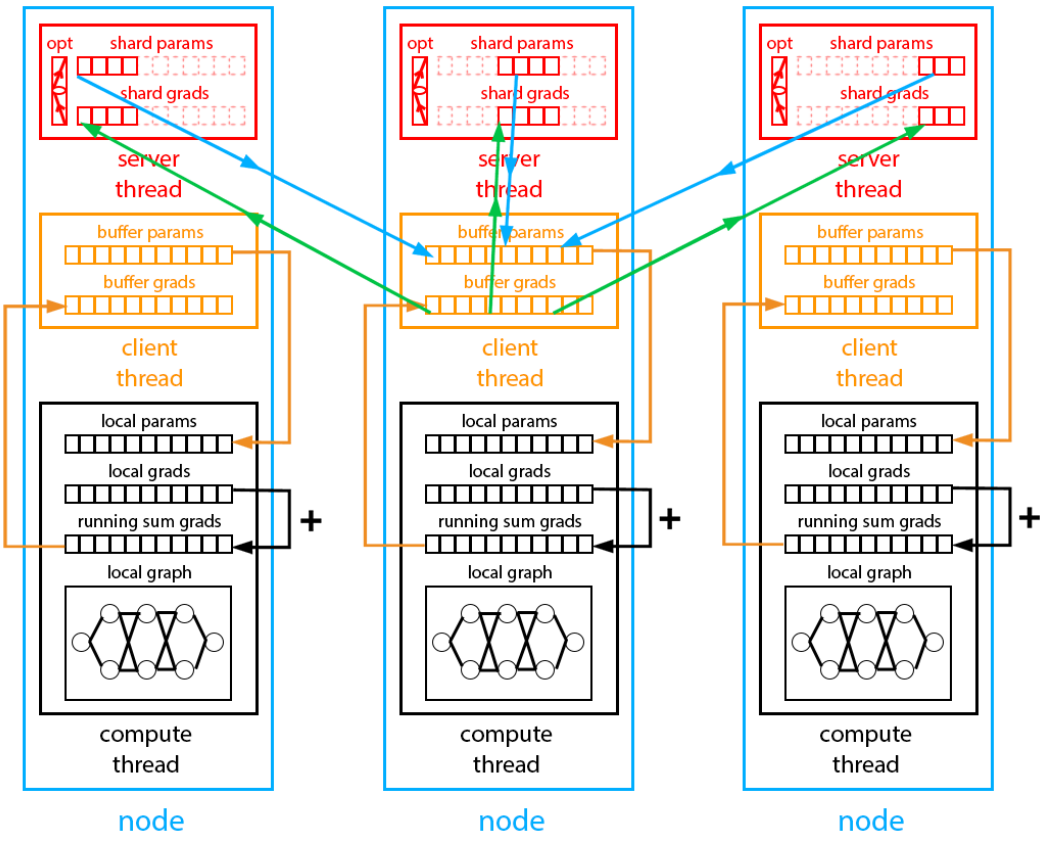


Figure 3.4: Overlapping communication and computation. Communication of middle node shown.

Algorithm 5: Client communication thread

initialise bufferParams, bufferGrads;
while *not converged* **do**
    wait(bufferGrads.filled());
    grads = bufferGrads.empty();
    foreach *node* ∈ *cluster* **do** synchronise(node, grads, params) ;
    bufferParams.fill(params);
end

Algorithm 6: Client computation thread

```
initialise graph, batches, sumGrads, ...;
while not converged do
    // do build, forward, backward - then:
    sumGrads += grads;
    if bufferParams.filled() then
        params = bufferParams.empty();
        bufferGrads.fill(sumGrads);
        sumGrads = 0;
    end
end
```

all gradients of iterations where communication was skipped are remembered. If the communication thread is ready, the computation thread replaces its parameters with those in the communication buffer. It then copies its running sum of gradients into the buffer, notifying the communication thread that it can synchronise with the server shards. Finally, the compute thread clears the summed gradients locally and proceeds to the next iteration.

We implement various tweakable settings that result in different run-time behaviours. The first is the option to apply the summed gradients to the swapped-in (updated) parameters. This attempts to reduce the staleness of the out-of-date parameters (c.f. section 2.3.7). The second setting limits each node to having a single client communication thread actively sending or receiving at any time. This prevents the case where many clients significantly slow down a node's server shard speed, which can negatively impact the communication speeds of other nodes. Another setting is the max number of iterations between every remote synchronisation. If a node's network access to the server shards is somehow blocked for a long duration, it might be undesirable for it to continue summing up gradients locally using increasingly stale parameters and then finally sending "bad" gradients to the shards. The same principle applies to clusters with low network speeds. In effect, un-bounded communication overlap is a type of dynamic batch-sizing (c.f. 2.2.2), where less frequent communication results in a greater number of compute iterations using the same parameters, similar to using large batch sizes. Conversely, more frequent communication results in more frequent parameter updates and is thus analogous to using smaller batch sizes. Too large batch

sizes can hurt convergence rate (Bottou, 2010), which this option intends to mitigate. A final setting is a variation inspired by Elastic Averaging SGD (Zhang et al., 2015a), or EASGD, where nodes update their parameters locally and exert an elastic force on the server parameters. While our variation does not implement the elasticity aspect of EASGD, which is a fundamentally different approach to updating the server parameters, we allow nodes to update their local parameters in between synchronisations, encouraging them to diverge from the global parameters and thus better explore the gradient space. Unlike in EASGD, our communication period is not fixed but rather dynamically determined by the communication time of the client thread in each synchronisation. This encourages maximum utilisation of the communication channel, resulting in more frequent updates to the server shards. Our variation could easily be turned into EASGD by implementing an optimiser that incorporates the update rule of Asynchronous EASGD and making nodes send elastic differences instead of error gradients (c.f. Zhang et al., 2015a).

3.5 Multi-GPU Nodes

We now turn to multi-GPU nodes, where a node contains multiple GPUs, each running an independent neural network. This is a common set-up for nodes with multiple GPUs (e.g. Noel and Osindero, 2014; c.f. section 2.3.4). A typical data-parallel architecture on a single machine is illustrated by **Figure 3.5**. This is essentially the same structure as in the sharded server described in section 3.2. The main difference is that each node is now a GPU and the network communication is replaced by inter-GPU memory copies.

Each GPU runs in its own thread and contains a full model that synchronises with all GPU shards. The per-GPU thread is in fact a CPU-level control thread that sends commands to that GPU and executes related functions. When starting a new iteration, it builds the GPU's graph by copying the input samples to the GPU's memory and then instructing it to do the required computations. When synchronising with a shard, its control thread copies the sub-gradients to the memory of the appropriate GPU shard, instructing that GPU to run its shard optimiser. This results in new sub-parameters on that GPU shard, which are copied back to the original GPU's full model. After synchronising with all shards, the full parameters for this GPU have been updated and the next iteration can commence.

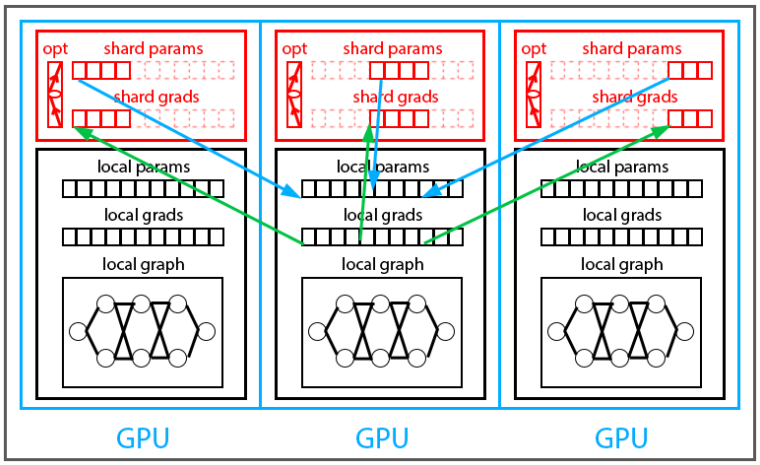


Figure 3.5: Single node multi-GPU structure. Communication of middle GPU shown

3.6 Multi-Node Multi-GPU

When synchronising in a multi-node multi-GPU environment (**Figure 3.6**), each model can be interpreted as an independent node with the same behaviour as a node in the sharded implementation from section 3.3. That is, it computes and then pushes gradients to the server shards, replacing its parameters with those it receives in response. Similarly, each machine is responsible for 1/N parameters. The distinction is that we now have multiple processing units per machine, which enables us to shard the node's server parameters as well (**Algorithm 7**). For example, in a two node cluster where each node has 3 GPUs, every GPU will be responsible for updating 1/6th of the parameters. The main advantage of this approach is that a node's update computations run in parallel across all unit shards, reducing their time to 1/D where D is the number of devices (GPUs) on the node. The reduction in total update time, however, is less substantial because copying gradients and parameters between main memory and GPUs is significantly slower than intra-GPU copies.

It is important to consider two different approaches to parameter sharding. The first involves partitioning the parameters uniformly over the total number of GPUs, regardless of the distribution of GPUs over the nodes. If one node has 5 GPUs and another has 1 GPU (in a two node cluster), each GPU is responsible for 1/6th of the parameters. Thus the update computations are divided equally over the GPUs. However, this also implies that the first node (machine) is responsible for 5/6ths of all parameters, resulting in an unbalanced communication distribution - i.e. the first node exchanges 5x more data than the second in each gradient/parameter exchange. The second approach is uniformly distributing the parameters across the nodes (not GPUs), and then

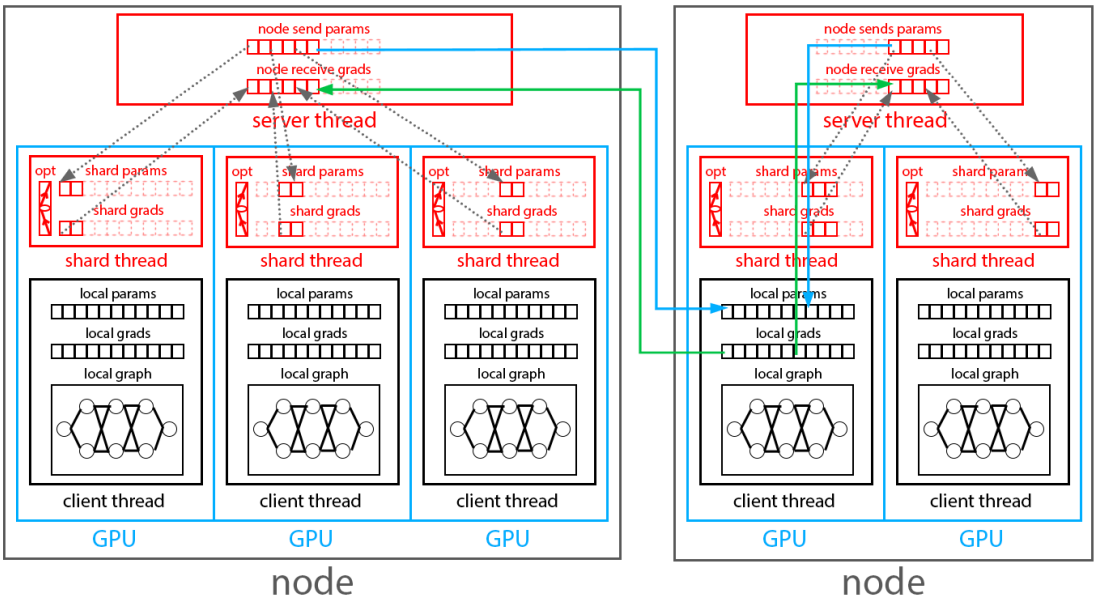


Figure 3.6: Multi-node multi-GPU structure where second node's first client's communication is shown

Algorithm 7: Server thread with device (GPU) sharding

initialise params, offsets, sizes;
while *not converged* **do**
  grads, client = recv(anySource);
  foreach *device* ∈ *node* **do in parallel**
    subGrads = grads.sub(offsets[device], sizes[device]);
    copy(subGrads, device.grads); `// copy sub-grads to device`
    device.opt.update(device.params, device.grads); `// do update`
    copy(device.params, subParams); `// copy sub-params from device`
    params.sub(offsets[device], sizes[device]) = subParams;
  end
  send(client, params);
end

uniformly distributing each node's part of the parameters across its GPUs. In the two-nodes example, the imbalance now shifts to the update computations: the first node has 5 GPUs over which to parallelise the update while the second node only has a single GPU. In a multi-node environment, and in particular when using fast GPUs on each node, communication speed is usually a greater bottleneck than computation speeds. Thus our implementation uses the second approach. A potential optimisation is to distribute the parameters according to a scoring function which finds an optimal compromise between the two approaches. However, this function depends on the cluster's network speed and the computation power of the GPUs. Because we want our solution to work well regardless of the hardware it runs on, we did not opt for the scoring function.

3.7 Multi-Node Multi-GPU with Communication Overlap

Overlapping computations and communication on multi-GPU nodes is similar to the single-model approach described in section 3.4. The main difference is that every GPU now has its own client thread and communication buffers that are used to synchronise with the server shards while the model's computations proceed with the next iteration(s), as illustrated in **Figure 3.7**.

An alternative approach involves equipping each node with only one client thread and a single buffer to which all GPUs push their summed gradients. Thus, when a client has pulled the latest parameters, all other clients on the same node can access the updated parameters without communicating externally. In practice, this is significantly more complicated because each GPU has its own memory. Thus, the shared buffer has to be located on RAM, drastically increasing the time it takes for a GPU to increase the summed gradients because of the relatively slow copies to main memory. Furthermore, summing gradients (floats) on a CPU is significantly more expensive than on GPUs, especially for neural networks with a large number of parameters. Sparse communication (c.f. section 3.8) further complicates this. Our implementation features a variant with a single communication thread that shards a master communication buffer across the GPUs, accumulating the summed gradients of each GPU into it. This data is then copied onto main memory and sent to the server shards. The received parameters are copied to an extra buffer on each GPU, where they are finally "consumed" by the clients upon finishing their current iteration's computations. The benefit of this approach is that the communication of a single node does not increase with the number

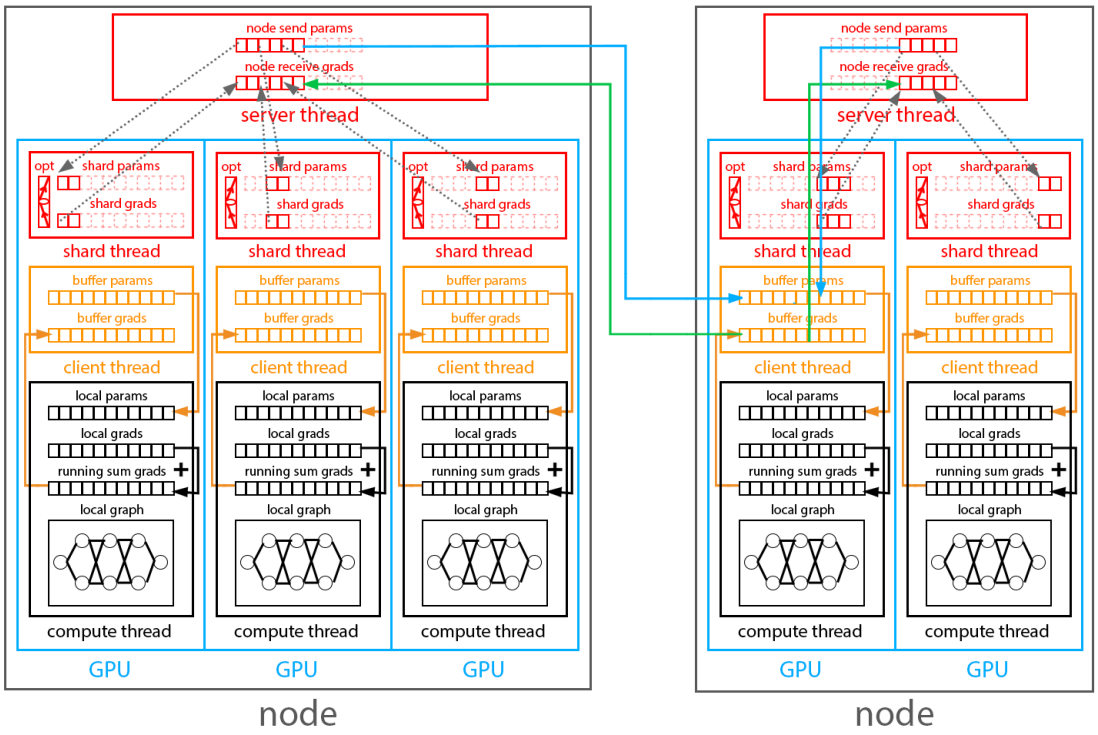


Figure 3.7: Multi-node multi-GPU structure with overlapping communication and computation. Communication of node 2 GPU 1 shown.

of GPUs it contains. The drawback is the significant increase in intra-node communication overhead and the potential for increased staleness in parameters because of the reduced communication per node.

3.8 Sparse Communication

Multi-GPU training on a single node scales well because communication between GPUs is fast, while multi-node training is more difficult because transfer rates between machines are much slower. For example, we measured inter-GPU transfer rates at over 7GB/s with Nvidia 1080s on the same machine, while our inter-node speed tests on a Google Cloud cluster maxed out at 375MB/s. Therefore, we implement gradient dropping as described by Aji and Heafield (2017) (c.f. section 2.3.5). We favour this approach because it can safely reduce message sizes by 98% while Seide et al.'s (2014) 1-bit SGD achieves up to 68% reduction (1 bit per 4 byte float). The dropping was implemented in the Marian framework by Alham Fikri Aji for single node multi-GPU training. We extend this for multi-node multi-GPU as follows.

A client's perspective on gradient dropping is illustrated in **Figure 3.8** and de-

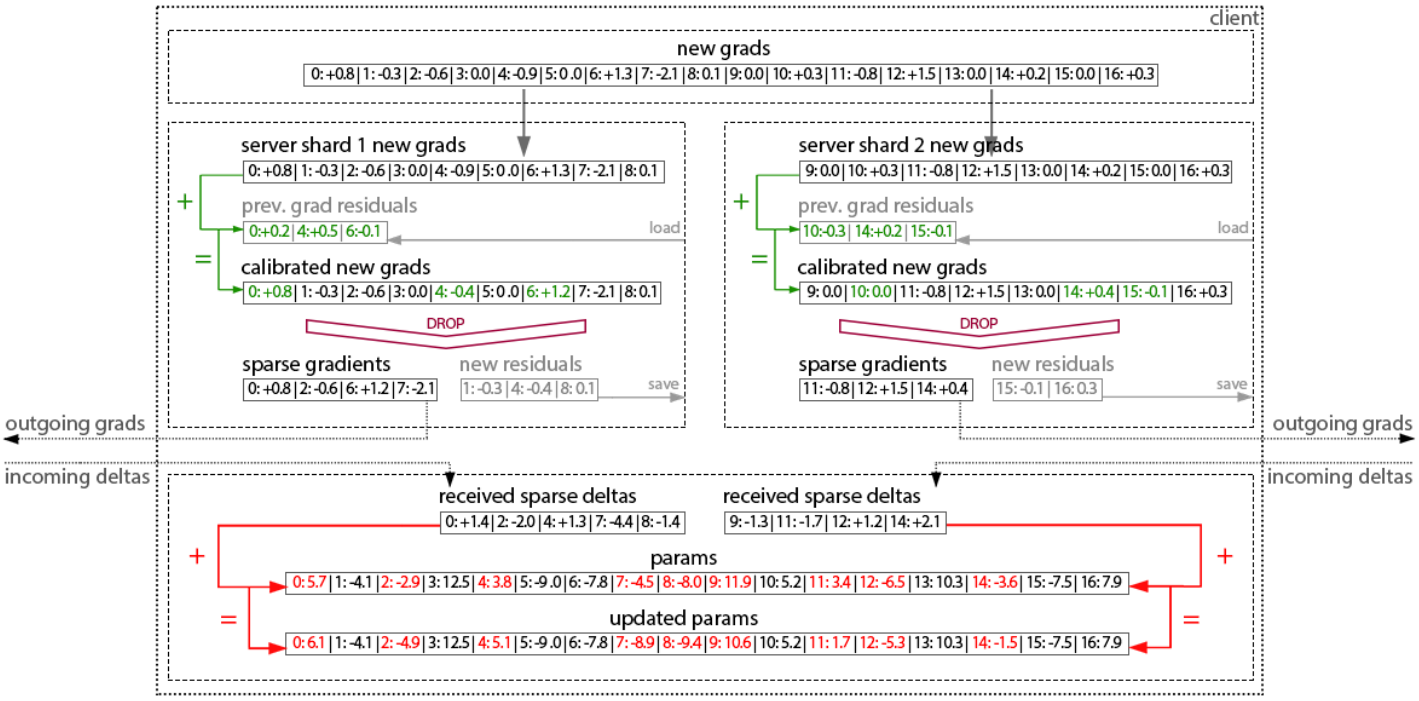


Figure 3.8: Client view of sparse communication (no compute/communication overlap)

Algorithm 8: Client view of sparse communication (no comp/comm overlap)

initialise graph, batches, offsets, sizes;
while *not converged* **do**
    `// do build, forward, backward - then:`
    foreach *shard* ∈ *cluster* **do**
        subGrads = grads.sub(offsets[shard], sizes[shard]);
        subGrads += residuals;
        sparseGrads, residuals = subGrads.drop(dropRate);
        send(sparseGrads, shard);
        sparseDeltas = recv(shard);
        subParams = params.sub(offsets[shard], sizes[shard]);
        subParams += sparseDeltas;
    end
end

scribed by **Algorithm 8**. After having computed new gradients (or having received new gradients in the client thread buffer if using communication overlap), the client divides the gradients according to the partitioning of the server shards. Recall that dropped gradients, called residuals, are remembered and added to the next iteration's gradients (c.f. section 2.3.5). Thus, the client adds the previous residuals to each of the sub-gradients, giving "calibrated" new gradients. The dropping algorithm approximately selects the largest gradients and copies them to a sparse vector, storing the other values in a residuals vector. Each of the sparse sub-gradients are then sent to the appropriate server shard, which uses them to update the global parameters. The server shards respond with sparse deltas, which are essentially gradients that represent the difference between this client's current parameters and the shard's updated parameters. These deltas are applied to the client's parameters to give an approximate version of the updated global parameters.

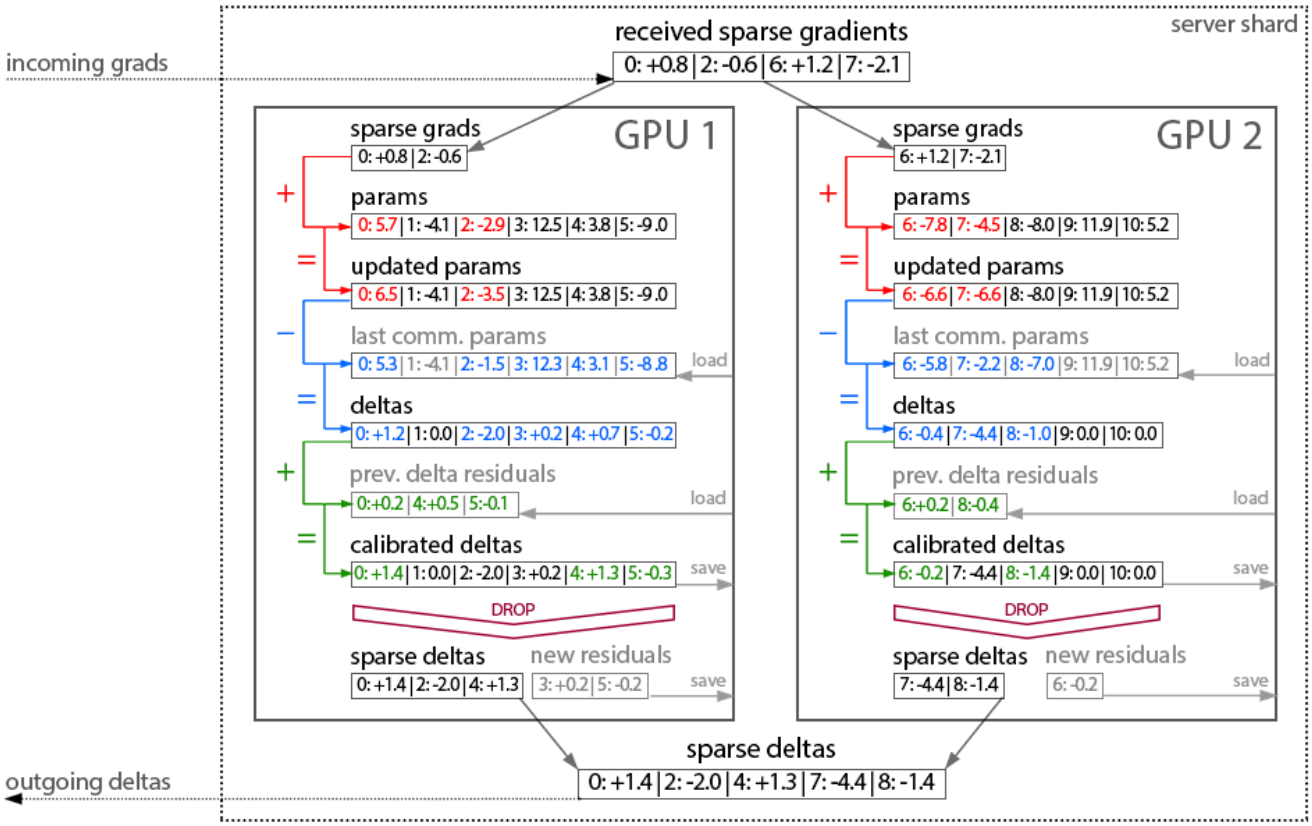


Figure 3.9: Server view of sparse communication with device sharding

A server shard's perspective is given by **Figure 3.9** and **Algorithm 9**. Because the parameters on a shard are further divided across its GPUs, the received sparse gradients are split into sub-gradients that are copied to the appropriate sub-shards (GPUs). Each

Algorithm 9: Server view of sparse communication with device sharding

```
initialise params, offsets, sizes;
while not converged do
    sparseGrads, client = recv(anySource);
    foreach device ∈ node do in parallel
        subSparseGrads = sparseGrads.sub(offsets[device], sizes[device]);
        device.opt.update(device.params, subSparseGrads);
        deltas = device.params - device.lastCommParams[client];
        deltas += residuals;
        sparseSubDeltas, residuals = deltas.drop(dropRate);
        sparseDeltas.sub(offsets[device], sizes[device]) = sparseSubDeltas;
        device.lastCommParams[client] = device.params;
    end
    send(client, sparseDeltas);
end
```

sub-shard then updates its parameters by running the sparse gradients through their optimiser. The sub-shards also store the parameters that were last communicated to each client, which are subtracted from the updated parameters to give the deltas (c.f. section 2.3.5). Sparse deltas are obtained in the same way as sparse gradients on the client-side. These are accumulated on the server shard's main memory and then sent back to the client that sent the sparse gradients.

Note that if a cluster contains 32 nodes with 4 GPUs each, every sub-shard will store 32*4=128 copies of its parameters, because each GPU is a client that has to synchronise with all sub-shards. However, since every GPU is also a server shard (c.f. section 3.6), the parameters allocated to each GPU represent only 1/128th of the full parameters. Thus, the memory usage is constant if the GPUs are evenly distributed across the nodes. A potential problem appears if a two-node cluster contains one node with many GPUs and another with only a single GPU. In that case, the second node's GPU is responsible for half of the global parameters, and has to store a copy for each GPU (client) of the other node. This can become problematic if the first node contains a large number of GPUs, but we consider this a non-issue because the maximum number of GPUs on a single machine is usually very limited (Walton, 2016) and any additional node to the cluster reduces the size of the parameters allocated to each machine.

Chapter 4

Experiments and Results

4.1 Setup

We use our multi-node extension of Marian to train Neural Machine Translation (NMT) networks. Each network's parameters consists of around 110.7 million floats, amounting to 442.7MB of data that has to be exchanged in each (full) gradient push or parameter pull. To give unbiased comparisons, we use the same options for all experiments, except where an option is a new feature of our implementation. Thus, we configure each node to use 64 samples per mini-batch, layer normalisation, a fixed scaling dropout along layers, adam optimisers and an initial learning rate of 0.0002.

To ensure that our results are representative of real-world usage, we run our experiments on high-speed clusters containing machines equipped with powerful GPUs. Specifically, we use the Google Cloud Compute Engine (GCE) service where we configure each machine with 4 high-speed virtual CPU cores running on an Intel platform and a variable number of Nvidia Tesla K40 GPUs (depending on the task). The average network speed between machines was tested at 375 MB/s. The major advantage of using GCE is that it allows us to easily add more nodes to a cluster for scalability experiments, without worrying about the underlying hardware configuration. The drawback is that GCE instances are virtual machines and their CPU cores are on shared (instead of dedicated) processors, which can negatively impact the training performance. However, since we run our baseline configurations on the same instances, our results should generalise to non-virtual machines as well.

4.2 Results

4.2.1 Single-Node

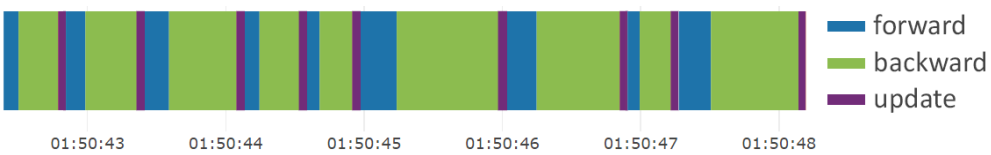


Figure 4.1: Wall-clock time window of a single node with a single GPU

We first evaluate single-node training, which provides a baseline against which we can compare our multi-node experiments. **Figure 4.1** represents a time window (slice) of the main tasks of a single node with one GPU. In parallel to section 2.2.3, *forward* represents the forward pass, *backward* the cost computation and subsequent backpropagation that results in gradients, and *update* the parameter update through the optimiser (in our experiments, Adam). Notice that the backward step is the most expensive, while the update step only takes a small fraction of time. The inconsistent duration of the forward and backward passes is due to the varying lengths of input training sentences.

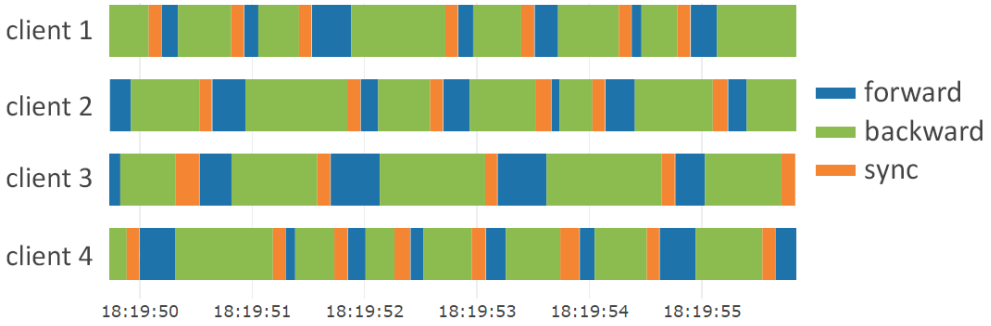
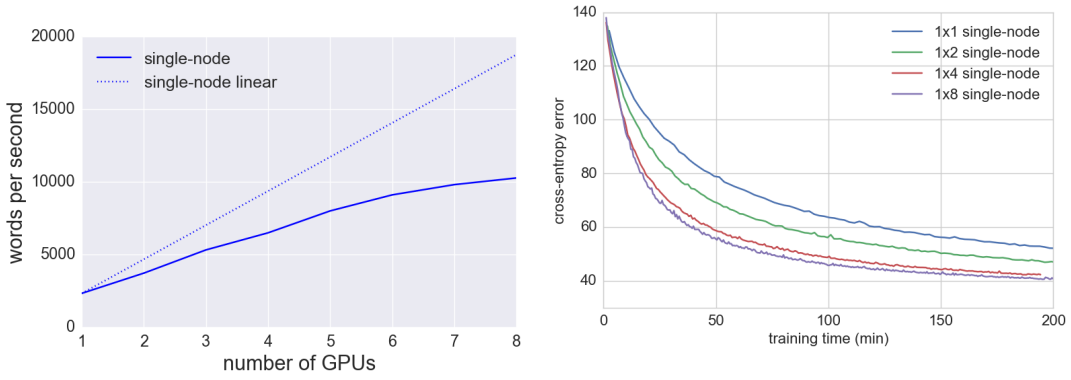


Figure 4.2: Wall-clock time window of a single node with 4 GPUs

Figure 4.2 shows the main tasks for single-noded training with 4 GPUs, where each is a client with its own neural network. The *update* steps are now synchronisation (*sync*) steps, which involve copying the newly computed gradients to the GPU shards, invoking them to update the global parameters, and pulling those back to the client to replace the model's local parameters (c.f. section 3.5). Observe that *sync* takes longer than *update*, because each client now pushes and pulls data to and from the other GPUs, which takes longer than moving data within the client's local (GPU) memory.

In addition, the locks on GPU shards that prevent clients from overwriting each other's gradients and parameters introduce further slow-downs.



(a) Training speed in words/sec

(b) Validation loss on a node with 1, 2 and 4 GPUs

Figure 4.3: Training speed and convergence rate for single-node training

Figure 4.3a shows the average training speed-up in terms of words processed per second. Every additional GPU has a less significant speed-up due to the increased overhead from syncing (as described above) and because of extra strain on the CPU cores and increased complexity in resource allocation and scheduling. An 8 GPU configuration is only 4.3x as fast as a single-GPU node. **Figure 4.3b** shows the validation loss for various GPU configurations with respect to wall-clock time. We measure this by periodically testing the global parameters on a separate validation dataset, noting the cross-entropy error (Deng, 2006) for each test. Note that because we do not train the models on the validation data, this error represents the expected performance on new data. Thus we consider the reduction in cross-entropy error over time the convergence rate. The convergence rate scaling over GPUs for a single node is practically the same as the speed-up in training throughput (words/sec).

4.2.2 Vanilla Multi-Node

The speed-ups in convergence rate on a single-node are significant but intra-node communication is significantly cheaper and less complex than across nodes. **Figure 4.4** represents a time window of "vanilla" multi-node training on 2 nodes, each equipped with a single GPU. Our vanilla configuration implements the sharded parameter server described in section 3.3, without any tricks such as communication overlap or sparse communication. Each node runs a client thread for the computations, doing *forward*,

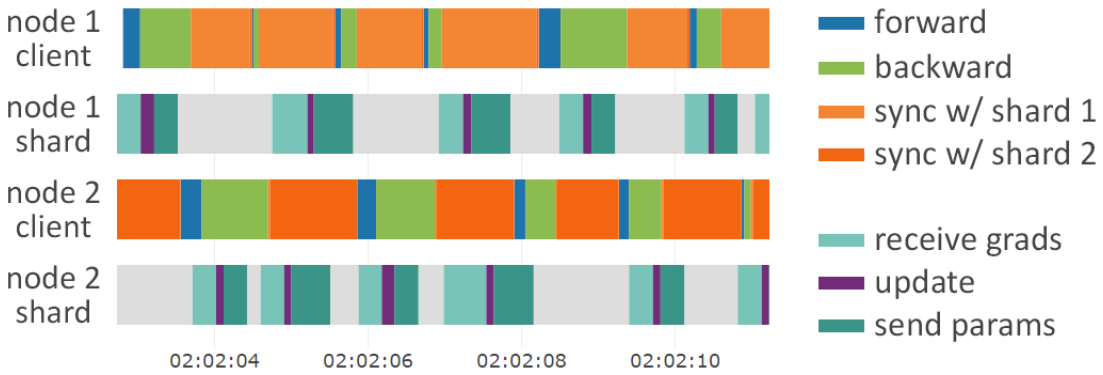
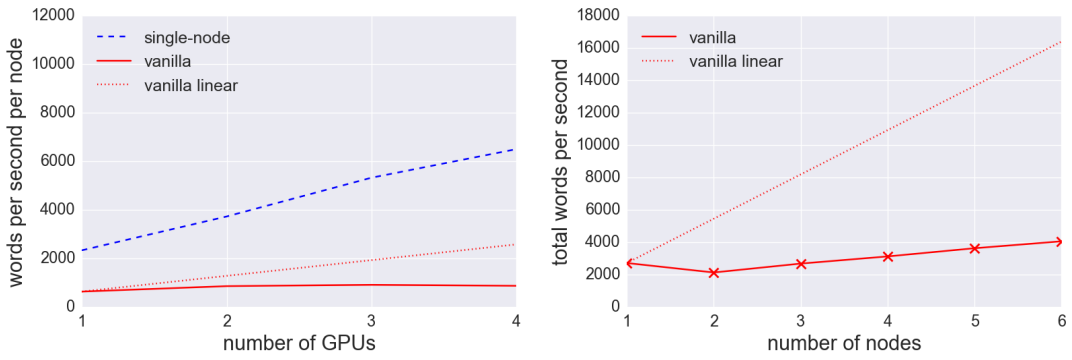


Figure 4.4: Timing of 2 nodes with each 1 GPU

backward and *sync* with each shard, and a server thread which *receive[s] grads*, *update[s]* its parameters, and *send[s] params* back. Notice that each client appears to synchronise with only a single shard (e.g. *node 1 client* with *node 2 shard*). This is due to an optimisation in which a client bypasses its server shard if the sub-parameters that have to be updated are located on the same node. This removes the overhead of unnecessarily copying gradients to the CPU, which is only useful if they have to be sent to a different node. Because this example involves only a single GPU per node, the local parameter updates do not involve any inter-GPU data transmission either, and are thus so fast that they are nearly invisible on the figure. Clearly, the most expensive task for a client is synchronising with remote shards. For a server shard, exchanging data with remote clients is the most expensive, followed by memory copies between RAM and GPU, and lastly running the optimiser on the GPU (both grouped under *update*).



(a) Scaling over GPUs in a 4-node cluster (b) Scaling over nodes with 2 GPUs each

Figure 4.5: Training speed in words/sec

Figure 4.5a shows how the number of GPUs per node affects the performance of vanilla multi-node training. Specifically, we run 4 nodes in a cluster with the same

number of GPUs per node, and measure the average speed of each node in terms of words processed per second. We run this test multiple times, once for each of the following settings: 1, 2, 3 and 4 GPUs per node. Unlike the single-node configuration from section 4.2.1 (shown in **Figure 4.5a** for reference), increasing the number of GPUs provides practically no benefit to training speed. This is because every GPU competes for the same communication resource, which we previously established as the biggest bottleneck. Thus every additional GPU slows down the communication speed, and thus training speed, of all other GPUs.

Figure 4.5b shows the total training speed of all nodes as we increase the size of the cluster, while using a constant number of GPUs per node (in this case 2). With a single node, the communication is very fast (intra-node) and thus we get similar speeds to the single-node configuration from 4.2.1. With 2 nodes, communication becomes a big bottleneck and the training speed per node is more than twice as slow, giving a combined (total) speed that is even less than on one node. As we increase the number of nodes, the communication overhead is conceptually constant due to parameter sharding. In practice, however, a larger cluster means that each client has to communicate a greater fraction of its gradients remotely (slow), instead of locally (fast). Thus every additional slows down the average speed per node, giving a diminishing return in the total training speed of the cluster.

Figure 4.6 shows the convergence rate for various setups of vanilla multi-node: 2 single-GPU nodes (*2x1*), 2 dual-GPU nodes (*2x2*), 4 single-GPU nodes (*4x1*) and 4 dual-GPU nodes (*4x2*). The first observation is that the single-node single-GPU baseline (*1x1*) converges significantly faster than all vanilla multi-node setups, including the *4x2* setup which has a higher total training speed (c.f. **Figure 4.5b**). This is because on a single node each client updates and receives the latest parameters on a very frequent basis. In our vanilla multi-node test with a cluster size of 4, each node is about 3x as slow (in words/sec) as 1 single-GPU node (c.f. **Figure 4.5a**). Therefore, every client updates its parameters about 1/3rd as frequently as they would on a single node in the same time-span. In addition, because there are 8 clients in a *4x2* setup, the gradients sent by each client are computed using parameters that are on average 7 updates behind the global parameters (because there are 7 other clients that push updates to the servers). The second observation is that convergence rate barely increases beyond 2 dual-GPU nodes, even when doubling the number of nodes. This can be attributed to the additional per-node slow-down for every extra node added to the cluster, leading to the same problems that we described for the *1x1* vs *4x2* case.

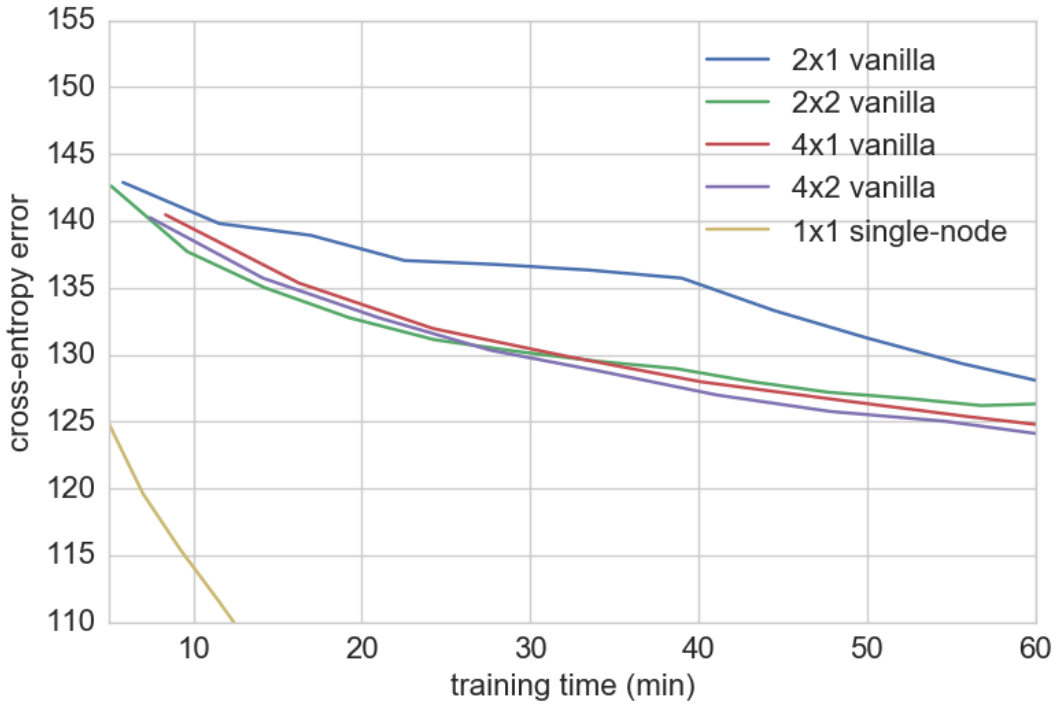


Figure 4.6: Convergence rate for vanilla mult-node

4.2.3 Communication Overlap

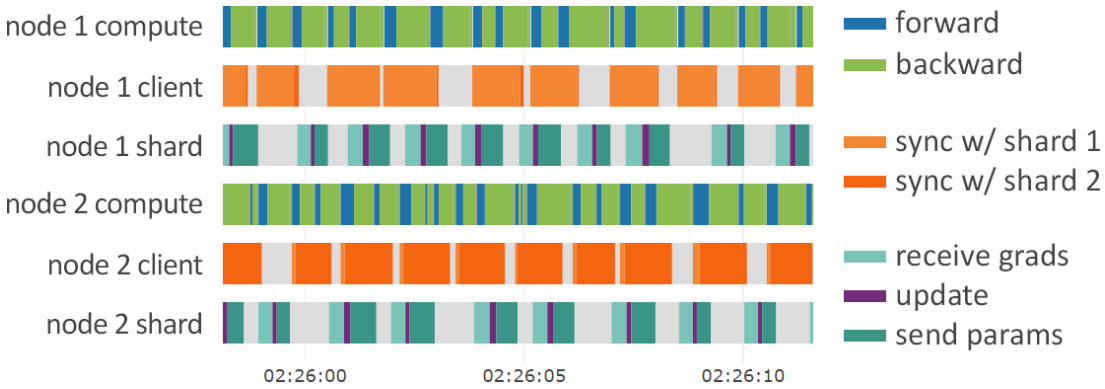
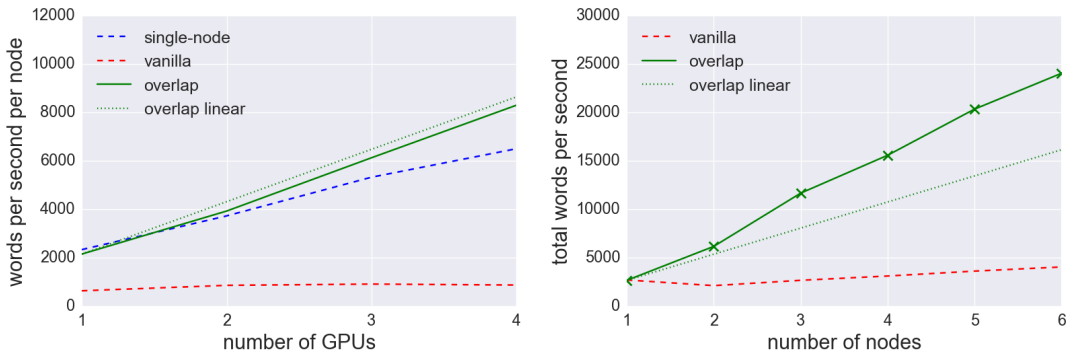


Figure 4.7: Timing of 2 nodes with each 1 GPU with communication overlap

Our communication overlap implementation, described in section 3.7, maximises the utilisation of the GPU by running the communication in a separate thread. **Figure 4.4** shows a time window of the main tasks occurring in a 2 node single-GPU "overlap" setup. In contrast to vanilla multi-node, the computation thread spends over 99% of its time doing *forward* and *backward* passes. The remaining time is spent on summing gradients locally when the communication thread is busy and exchanging its gradients

and parameters with the communication buffer (not visible in figure). The client thread synchronises with the server shards whenever its gradient buffer is filled, after which it waits for the compute thread to empty its parameters buffer and replace its gradients with the latest summed gradients. The server (shard) thread's behaviour is exactly the same as in vanilla multi-node: it receives gradients, updates its parameters and sends them back to the client. Notice that the communication thread spends a considerable amount of time waiting for the compute thread. Decreasing the batch size would reduce the duration of the compute thread's forward and backward passes, thereby resulting in shorter wait times and seemingly better utilisation of the client thread. However, this is more complicated in practice because the client thread shares the communication channel with the server thread, and thus any increase in client activity will impact the server's activity. Furthermore, when using multiple GPUs in a node, there are multiple client threads that all share the same communication resource. Similarly, the server shard thread looks underutilised, but its activity increases with the size of the cluster. In fact, in our overlap experiments 3 nodes was sufficient to continuously keep each server shard busy.



(a) Scaling over GPUs in a 4-node cluster (b) Scaling over nodes with 2 GPUs each

Figure 4.8: Training speed in words/sec

Figure 4.8a illustrates for overlap configuration the correlation between the training speed and the number of GPUs per-node in a 4-node cluster. With more GPUs, there are more clients competing for the same communication channel and thus the communication speed per client slows down. However, unlike in vanilla multi-node, the computations run independently from the remote synchronisation. Thus we achieve near-linear scaling in terms of words processed per second. This is also faster than the single-node baseline, which pauses its computations during every synchronisation.

Figure 4.8b shows how the number of nodes impacts training speed when using

2 GPUs per node. As described for vanilla multi-node, a larger cluster size results in every client having to push and fetch a larger proportion of the gradients and parameters remotely. Thus every additional node results in greater communication overhead, which in the case of overlap multi-node leads to fewer gradient and parameter exchanges between the compute thread and the client communication buffers. Perhaps more importantly, each client communication thread performs reads and writes on the same GPU as the compute thread runs on (c.f. section 3.7), which can impact the speed of the forward and backward passes. While exchanging data with other nodes, however, the client thread is only concerned with the CPU. Thus, longer communication times result in less overall strain on the GPUs. We believe these reasons explain why overlap multi-node achieves superlinear scaling in terms of words processed per second.

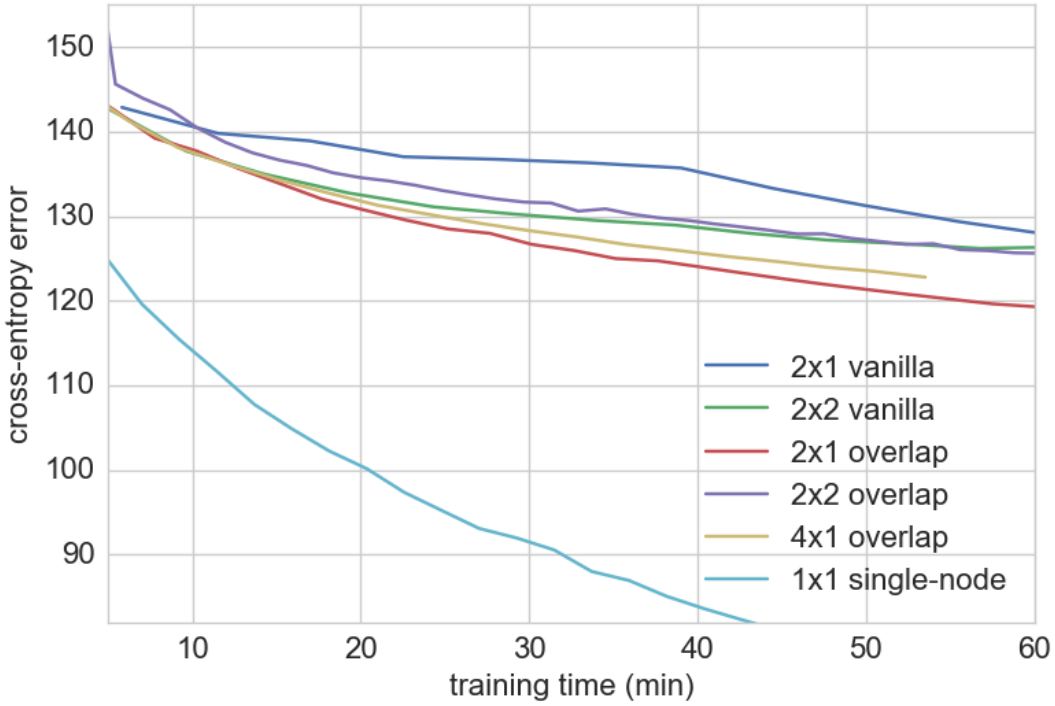


Figure 4.9: Convergence rate for overlap multi-node

However, as confirmed by **Figure 4.9**, a greater training throughput does not necessarily result in a higher convergence rate. In fact, *2x1 overlap* converges faster than both *2x2 overlap* and *4x1 overlap*. Recall from section 3.4 that more compute iterations per synchronisation is similar to using larger batch sizes. Thus, the extra communication overhead in *2x2* and *4x1* means that they are effectively using batches that are sufficiently large to damage their convergence rate compared to *2x1*. We prove this the-

ory through **Figure 4.10**, where we show the effect of limiting the number of compute iterations per synchronisation to 2 for *2x1 overlap* and *2x2 overlap*. As shown, *2x1 bound 2* performs better than its non-bounded counterpart, while *2x2 overlap bound 2* converges nearly 40% faster than its counterpart. Despite multi-node overlap converging significantly faster than vanilla multi-node, it is still exceedingly worse than single-node training. Combined with the fact that it does not scale with the number of GPUs or the cluster size, we do not attempt to further optimise the overlap configuration described here.

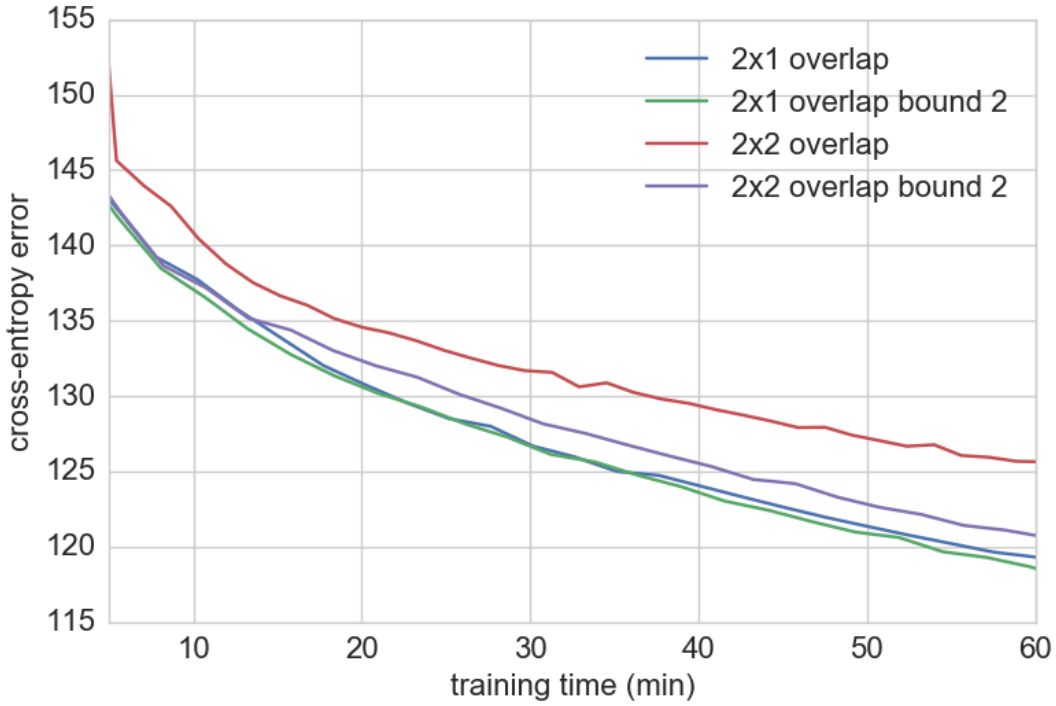


Figure 4.10: Convergence rate for bounded overlap multi-node

4.2.4 Sparse Communication

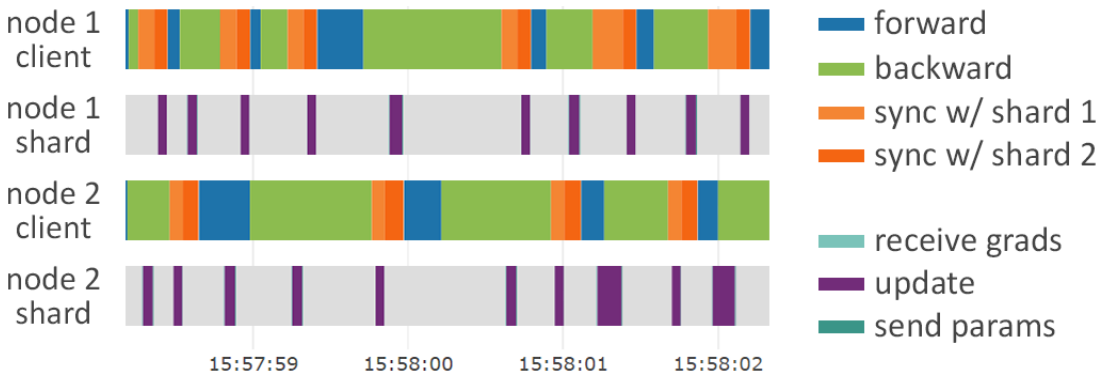
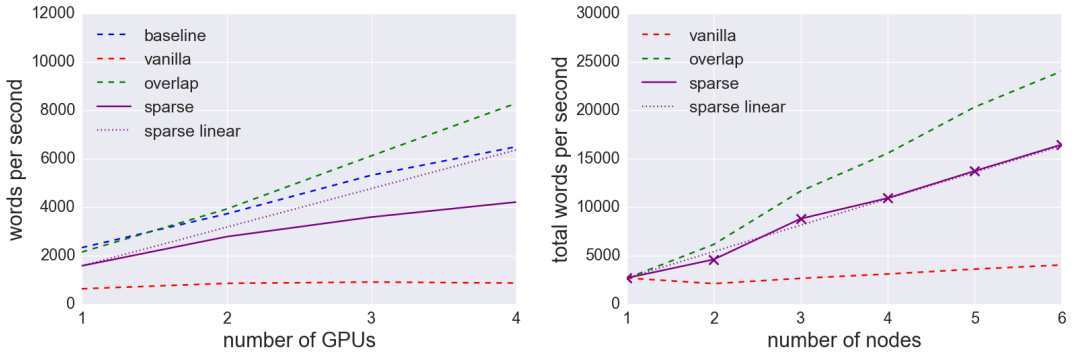


Figure 4.11: Timing of 2 nodes with each 1 GPU with sparse communication

Figure 4.7 shows a time window of training where we apply 99.9% sparse communication to vanilla multi-node, which we refer to as *sparse multi-node*. To simplify comparisons of costs, we force each client in this example to communicate with all server shards, including the one located on the same node (unlike the bypassing from previous sections). The most important observation is that the server shards appear to no longer exchange any data with clients. However, this is simply because each message payload is now roughly 0.02% of the original size from vanilla multi-node. Thus *receive grads* and *send params* are only visible if you sufficiently enlarge the diagram. As a result, each client synchronisation is now significantly faster. In fact, synchronising remotely is now barely slower than syncing locally. However, the client side's synchronisation is considerably slower than in single-node training (c.f. section 4.2.1). This is because gradient dropping induces additional overhead. Similarly, the update step on a server shard takes longer because of the delta computations and subsequent delta dropping (c.f. section 3.8).

Figure 4.12a shows that sparse multi-node scales in a similar sublinear fashion as single-node when increasing the number of GPUs per node. This is explained by the same reasons, i.e. every additional GPU increases complexity and resource contention (c.f. section 4.2.1). Because each client's synchronisation takes longer than in single-node training, and because we do not employ overlapping communication here, the GPUs spend less time on the forward and backward passes, resulting in a lower per-node training throughput than single-node.

Figure 4.12b shows that we achieve linear scaling in training throughput when we scale sparse multi-node to 3 or more nodes. Because communication is now very



(a) Scaling over GPUs in a 4-node cluster (b) Scaling over nodes with 2 GPUs each

Figure 4.12: Training speed in words/sec

cheap, its increased overhead in larger clusters barely impacts training speed. We hypothesise that 2 nodes performs sublinearly because there is less granularity between the number of servers and the number of clients than with 3 or more nodes – e.g. 4 clients and 2 server shards provides less timing flexibility than 6 clients and 3 server shards.

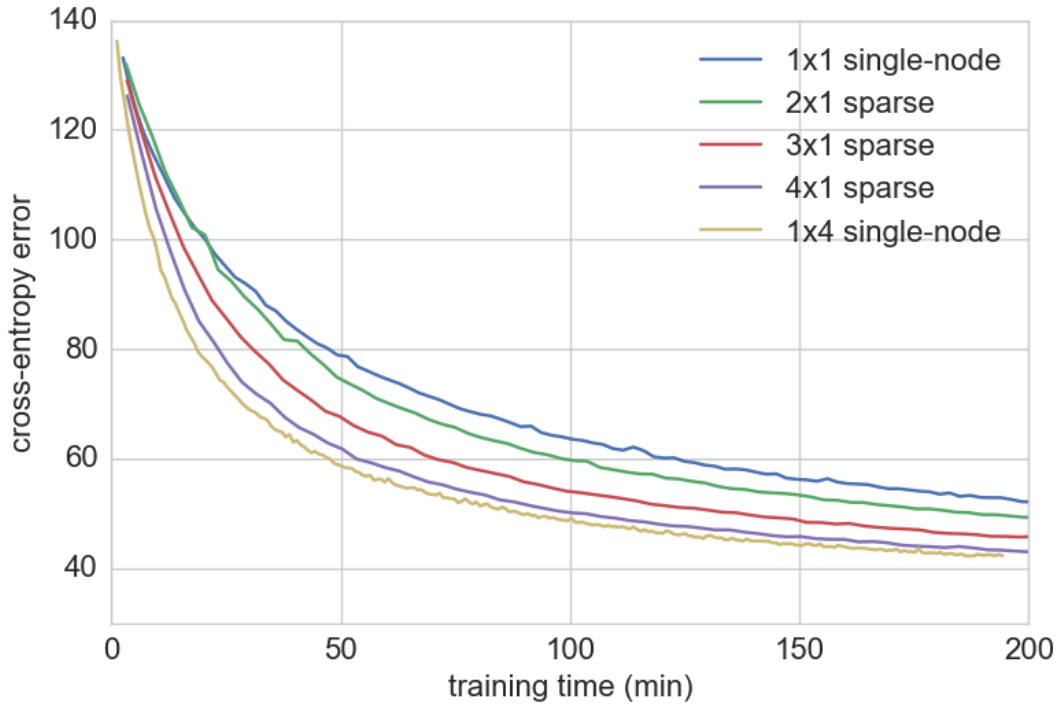


Figure 4.13: Convergence rate for sparse multi-node (single-GPU nodes)

Figure 4.13 shows the validation loss over time for sparse multi-node where each node has a single GPU. *2x1 sparse* converges only roughly 25% faster than 1 single-

GPU node, while *3x1 sparse* converges about 80% faster. We associate the relatively poor performance of 2 node configurations with their lower per-node training throughput described previously. *4x1 sparse* converges around 40% faster than *3x1 sparse* and impressively nearly matches the performance of a single-node with 4 GPUs, converging roughly 150% faster than *1x1 single-node*. This can be attributed to the linear scaling in throughput compared to a single node, thanks to the immense reduction in communication costs.

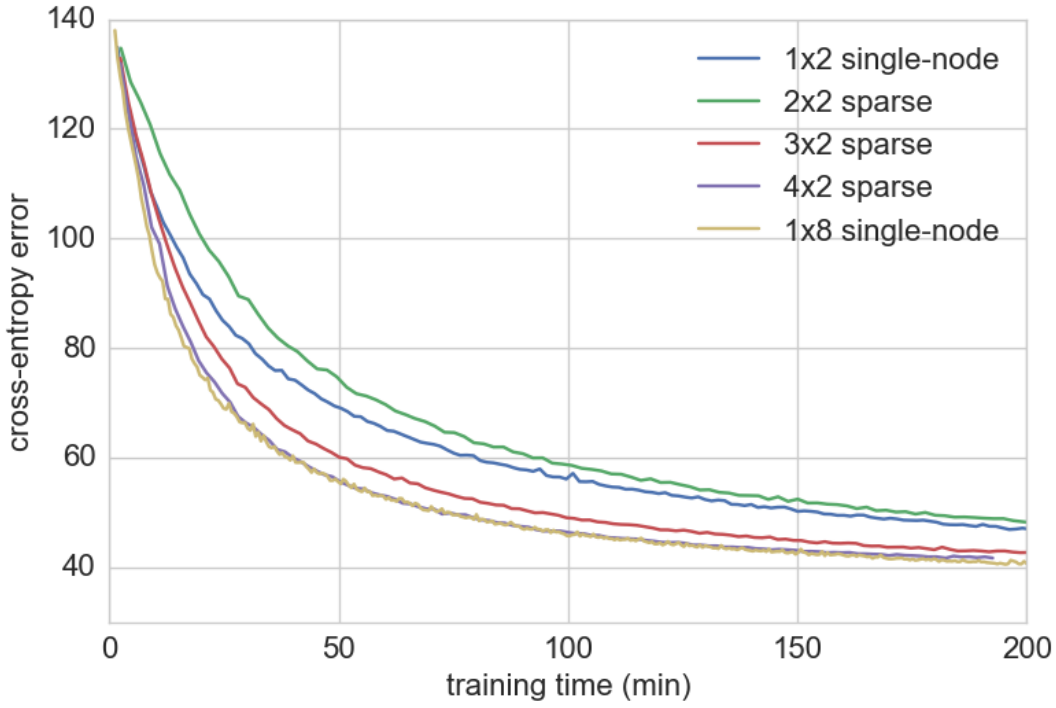


Figure 4.14: Convergence rate for sparse multi-node (dual-GPU nodes)

Figure 4.14 shows the same test but for dual GPU nodes. Here, *2x2 sparse* is actually slower than a single dual-GPU node. Again, we believe this relates to the lower per-node throughput of 2 node clusters, but both issues are worth investigating in future works. *3x2 sparse*, on the other hand, converges about 70% faster than its single node counterpart, which is slightly smaller speed-up compared to the 80% speed-up we saw for *3x1 sparse* vs *3x2 sparse*. This is likely because now there are double as many clients on each node competing for the same resources (in particular, the single shard thread on each node). Surprisingly, *4x2 sparse* converges practically just as fast as *1x8 single-node*, even though the latter has the same number of GPUs all stored on the same node.

4.2.5 Sparse Communication with Overlap

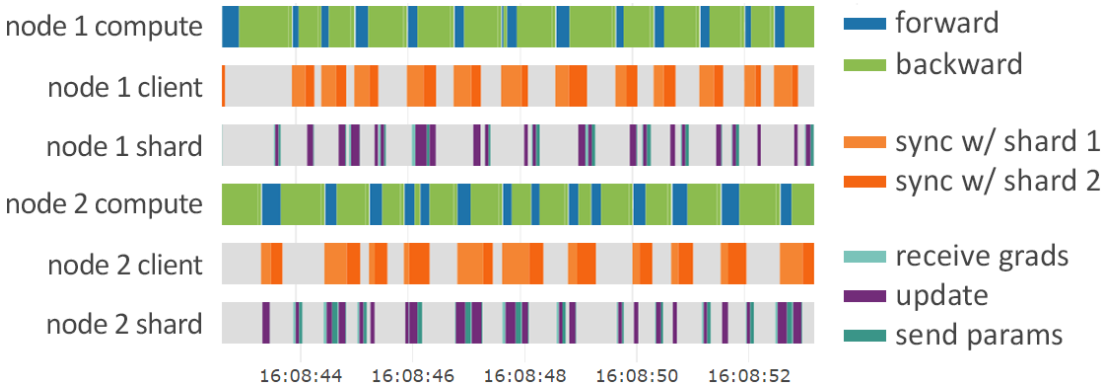
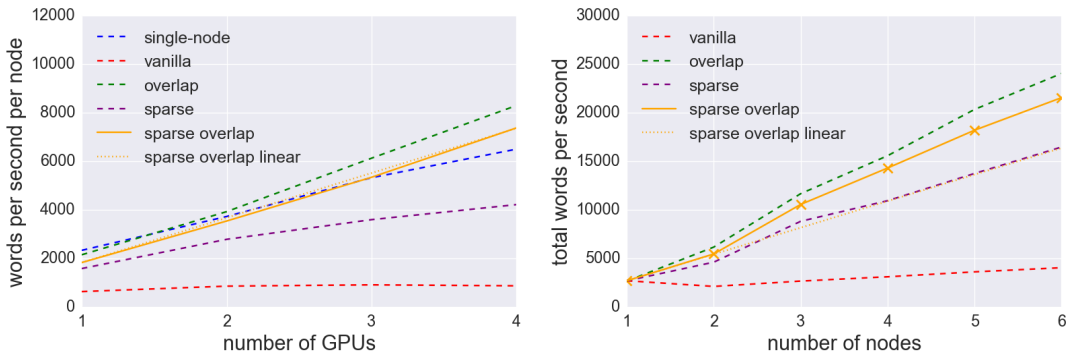


Figure 4.15: Timing of 2 nodes with each 1 GPU with communication overlap

Figure 4.15 shows a time window of 2 single-GPU nodes training with both sparse data exchange and communication overlap enabled, which we dub *sparse overlap*. Note that the client thread is now also responsible for dropping gradients and applying deltas to the parameters, which explains why each of its synchronisations takes longer than each server *update* step. This approach combines the main benefits that we saw in sections 4.2.3 and 4.2.4: very fast inter-node communication and a fully utilised GPU that continuously computes new gradients.



(a) Scaling over GPUs in a 4-node cluster (b) Scaling over nodes with 2 GPUs each

Figure 4.16: Training speed in words/sec

Like before, **Figure 4.16a** shows that we get linear scaling in training throughput as we increase the number of GPUs per node (in a 4 node cluster). Because the client thread has to perform the gradient/delta dropping computations on the same GPU as the compute thread, the training throughput of sparse overlap is slightly less than that of our previous overlap-only configuration.

Figure 4.16b shows that sparse overlap achieves superlinear scaling in training throughput as the cluster grows. The reasons for this are not obvious and require further investigation. The explanation we gave for overlap-only multi-node is not fully applicable here because message transmission length is minimised via gradient/delta dropping. We believe that the latency and contention for inter-node communication resources are greater as we scale the number of nodes, resulting in fewer synchronisations per client in the same time-frame and thus lower GPU-usage by the client communication thread, making the compute thread slightly faster.

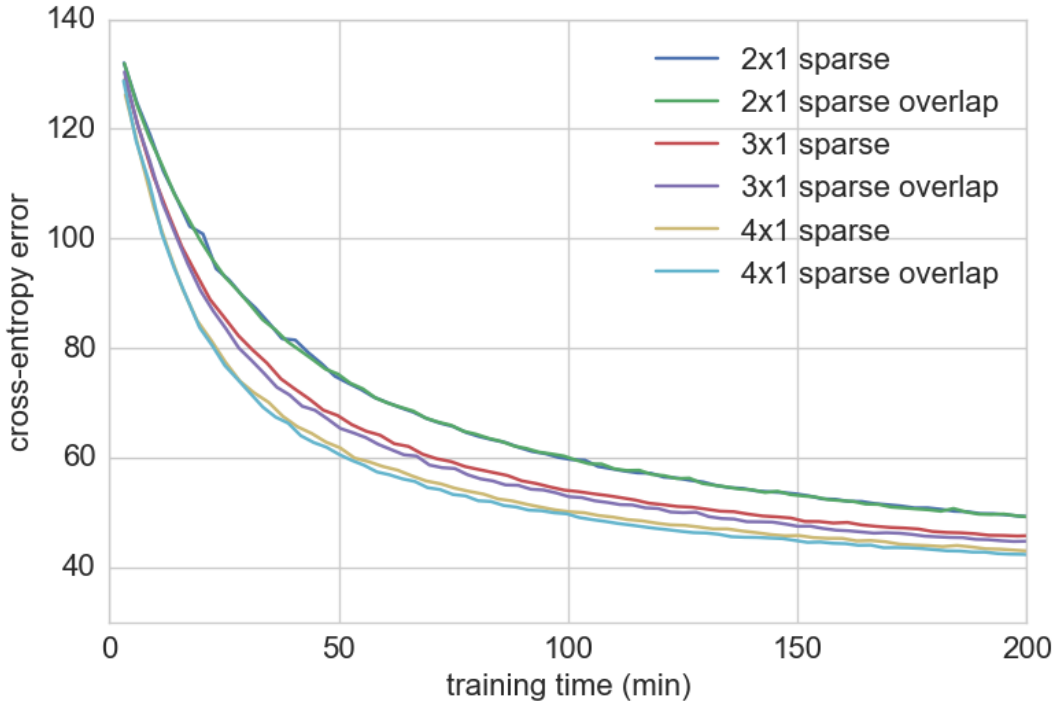


Figure 4.17: Convergence rate for sparse overlap multi-node (single-GPU nodes)

Figure 4.17 compares the convergence rate of sparse overlap with that of sparse-only multi-node for single-GPU nodes. While some configurations perform slightly better with overlap – e.g. *3x1 sparse* vs *3x1 sparse overlap* – the speed-ups are minimal. This is because communication in sparse multi-node is very fast, and since there are just as many clients as server shards in single-GPU multi-node, its benefits are minor and lessened by the increased complexity.

Figure 4.18 shows the same comparison for dual-GPU nodes. Surprisingly, the speed-ups from sparse overlap are much more significant here. In fact, in our 4 hour long experiments we measured a 30% increase in convergence rate when combining sparse communication and overlap for dual-GPU nodes. Observe that *3x2 overlap*

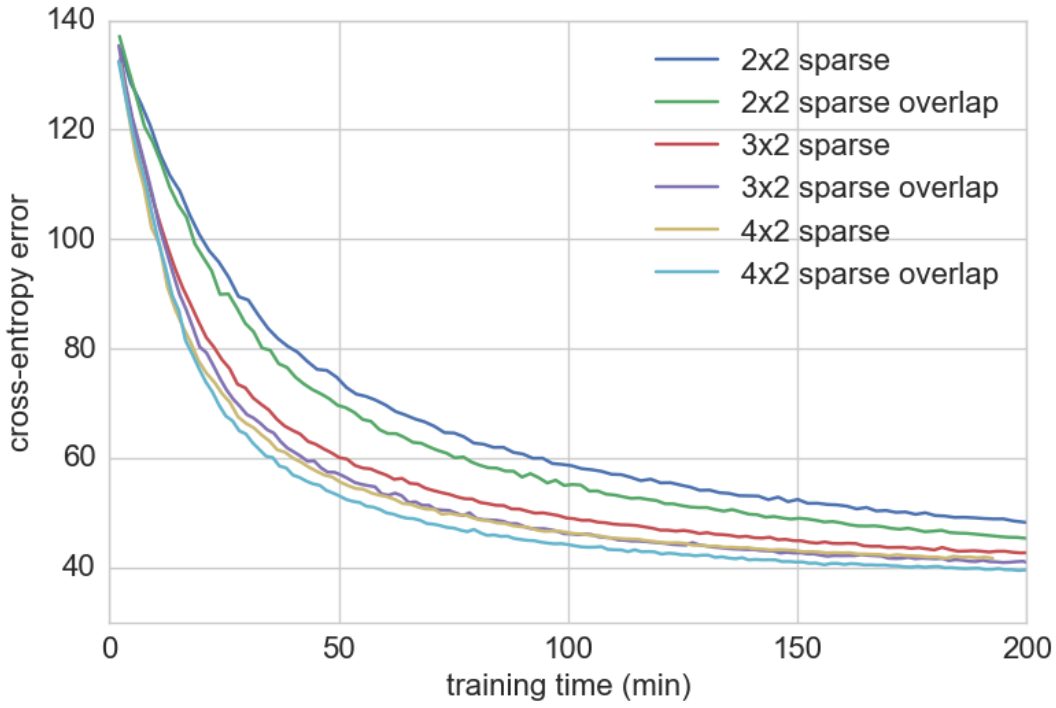


Figure 4.18: Convergence rate for sparse overlap multi-node (dual-GPU nodes)

performs just as well as *4x2 sparse overlap* in the diagram. This can be attributed to the contention for server shard access, since there are now double as many clients as there are server shards. Even though sending sparse data across nodes is cheap, a server shard's update takes a relatively long time due to the sparse-related computations.

4.3 Comparison to Related Works

The focus of our work is on how scaling across nodes affects convergence rate, while most multi-node papers focus on its effect on training speed in terms of data processed (e.g. words/sec). In training throughput, both our overlapping and sparse overlapping implementations achieve superlinear scaling, beating implementations such as those presented by Zinkevich et al. (2010) and Dean et al. (2012). However, as we saw for overlapping without sparse communication, training throughput does not necessarily result in faster training to convergence. For that metric, Zinkevich et al.'s SimuParallelSGD achieves a 5x speed-up when scaling from 1 to 10 machines, each running its own neural network. Our sparse overlap achieves the same speed-up using 7 machines. Furthermore, our approach generalises to any type of neural network, unlike Zinkevich et al.'s (Recht et al., 2011). Dean et al.'s (2012) Downpour SGD

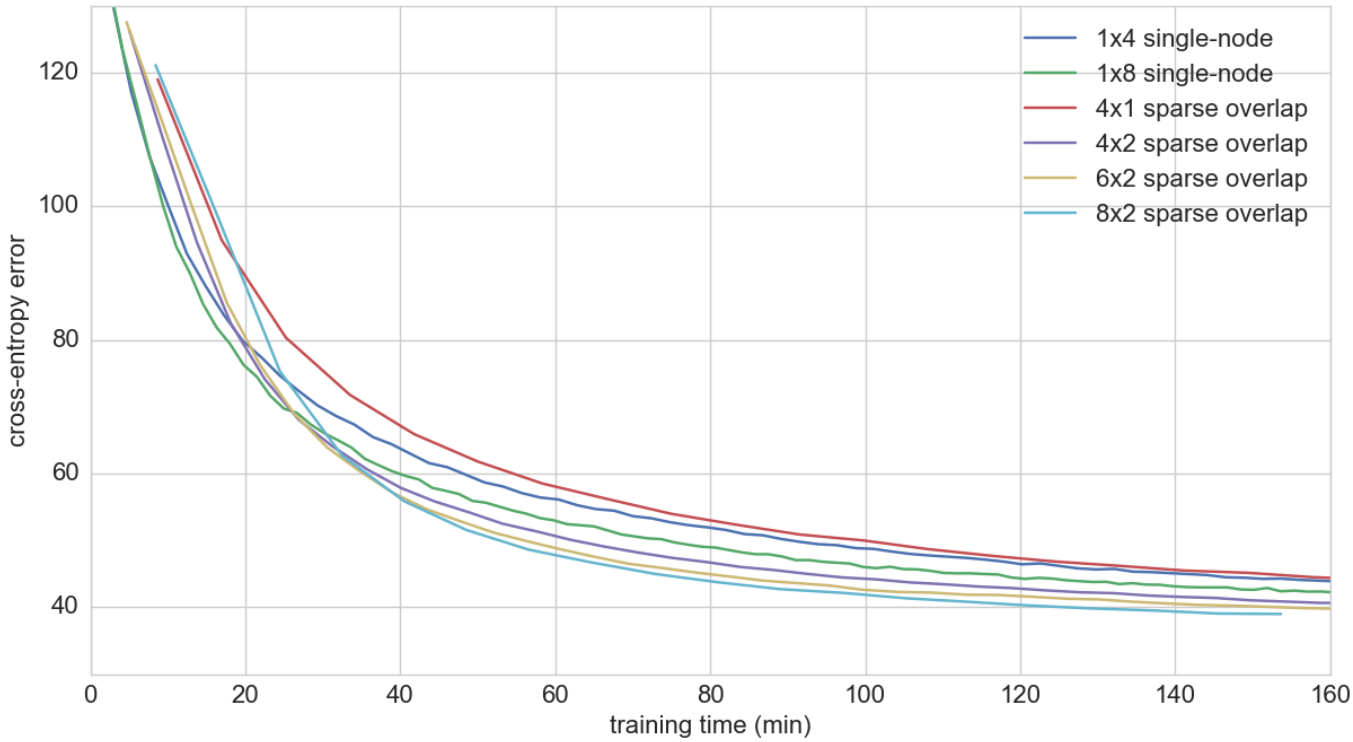


Figure 4.19: Convergence rate for sparse overlap multi-node compared to best performing single-node configurations

achieves a 2x speed-up on 600 CPU-based machines compared to a single-GPU. Our implementation achieves a greater speed-up on 3 GPU-based machines.

The main advantage of our implementation is its sparse communication capabilities. Therefore it is useful to compare our results to approaches implementing reduced communication methods. Seide et al.'s 1-bit SGD (2014) achieves a 4x convergence speed-up when training over 4 dual-GPU machines, compared to 1 single-GPU machine. We reached a similar speed-up for the same comparison in a 6-hour long test, but we believe that our implementation can reach convergence faster because the disparity in convergence rate grew towards the end of our relatively short test. Seide et al. also show an impressive 10x speed-up with 20 dual-GPU nodes. While we did not run an experiment at this scale, we expect our implementation to match – if not beat – this performance because our 99.9% sparsity provides greater compression than their 1-bit encoding. Alistarh et al.'s (2016) Quantized SGD paper gives promising results in terms of final accuracy at convergence while also heavily decreasing communication overhead. Unfortunately, they do not present results for convergence rate speed-ups

across machines. Conversely, our implementation focuses purely on convergence rate, neglecting final accuracy at convergence. However, we expect no accuracy loss as a result of sparse communication, based on Aji and Heafield's (2017) accuracy analyses and experiments (we use the same approach). We previously noted that the main limitation of Aji and Heafield's paper is the absence of multi-node experiments, where communication is a crucial bottleneck. This paper shows that sparse communication works very well in a multi-node environment and enables good scaling with respect to convergence rate. In fact, our implementation would scale disastrously if it were not for an effective communication reduction technique, as highlighted in our overlap-only multi-node analysis (section 4.2.3).

Chapter 5

Conclusion and Future Work

5.1 Overview

In this dissertation, we presented the implementation of a Stochastic Gradient Descent based multi-node training framework. We gave an overview of existing parallelisation strategies, introducing concepts such as data and model parallelism and describing recent approaches to reducing communication overhead. We then described how we adapted a single-node training implementation to enable training across machines. We explained various extensions that we incorporated into our framework to improve its scalability. Notably, we used server sharding to keep the amount of communication per node constant, communication overlap to maximise the utilisation of the GPUs, and sparse communication to reduce the fundamental cost of inter-node communication. Finally, we showed how various configurations of our implementation performed in the training of Neural Machine Translation networks. We found that multi-node in its simplest form is infeasible due to the excessive time spent waiting for synchronisations with remote server shards. Enabling communication overlap greatly increased the training throughput and provided somewhat better convergence rates. However, it still performed much worse than any single-node configuration due to the infrequent server synchronisations. Sparse communication greatly alleviated this problem and gave impressive scalability with respect to convergence rate. In fact, 4 machines with 2 GPUs each showed equally high convergence rate as 8 GPUs on the same machine, despite the increased complexity and overhead. Combined with communication overlap, we saw that our sparse implementation performs very favourably compared to related works, beating implementations like SimuParallelSGD (Zinkevich et al., 2010) and Google's Distbelief (Dean et al., 2012).

5.2 Limitations and Future Work

A major limitation of this work is that our experiments only provide a small glimpse of our multi-node framework's potential. Tuning its parameters and configuring which extensions to enable and disable to achieve optimal convergence rate is an entire dissertation project in itself. We introduced various settings in our methods, e.g. local optimisers inspired by Elastic Averaging SGD (Zhang et al., 2015a), that we implemented but did not present in our results due to the limited time and resources we had for running full experiments[1]. Furthermore, we focused more on the "multi-node" aspect than on the "multi-GPU" part of "multi-node multi-GPU". This is because GPU scaling on a single node has proven to work well, while scaling across machines shows mixed results in literature. Future work on this project includes optimising the multi-node framework for a large number of GPUs. In the current implementation, if every node has 8 GPUs, there are 8 clients for every server shard. This will most certainly decrease performance. A potential solution is to run multiple server shard threads per node, enabling simultaneous remote client synchronisations. We dismissed this idea in our methods because we considered inter-node data transfer the most important cost, however our experiments reveal that inter-node communication becomes nearly insignificant with 99.9% sparsity. With our multi-node framework in place, such modifications are trivial to implement. Further interesting extensions include Staleness-Aware SGD (Zhang et al., 2015b) and the update rule from Elastic Averaging SGD (Zhang et al., 2015a). In addition, removing locks and using unsafe communication protocols – inspired by Hogwild (Recht et al., 2011) and Dogwild (Noel and Osindero, 2014) – can further speed up training. Because this is an open-source project, we expect that our implementation will be used in practice and thus there will be an incentive to continuously improve its training speed.

[1] Gathering the results that were presented involved running over 100 different experiments for a total of more than 300 hours in the last three weeks of this dissertation alone.

Bibliography

Aji, A. F. and Heafield, K. (2017). Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*.

Alabi, T., Blanchard, J. D., Gordon, B., and Steinbach, R. (2012). Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2.

Alistarh, D., Li, J., Tomioka, R., and Vojnovic, M. (2016). QSGD: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*.

Arrow, K. J. and Hurwicz, L. (1958). *Decentralization and computation in resource allocation*. Stanford University, Department of Economics.

Baudet, G. M. (1978). Asynchronous iterative methods for multiprocessors. *Journal of the ACM (JACM)*, 25(2):226–244.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Chen, X., Eversole, A., Li, G., Yu, D., and Seide, F. (2012). Pipelined back-propagation for context-dependent deep neural networks. In *Thirteenth Annual Conference of the International Speech Communication Association*.

Chilimbi, T. M., Suzue, Y., Apacible, J., and Kalyanaraman, K. (2014). Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582.

Dahl, G. E., Sainath, T. N., and Hinton, G. E. (2013). Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE.

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

Deng, L.-Y. (2006). The cross-entropy method: a unified approach to combinatorial optimization, monte-carlo simulation, and machine learning.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Färber, P. (1997). Quicknet on multispert: fast parallel neural network training.

Hecht-Nielsen, R. et al. (1988). Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Langford, J., Smola, A. J., and Zinkevich, M. (2009). Slow learners are fast. *Advances in Neural Information Processing Systems*, 22:2331–2339.

Narendra, K. S. and Mukhopadhyay, S. (1997). Adaptive control using neural networks and approximate models. *IEEE Transactions on neural networks*, 8(3):475–485.

Noel, C. and Osindero, S. (2014). Dogwild!distributed hogwild for cpu & gpu. In *NIPS workshop on Distributed Machine Learning and Matrix Computations*.

Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM.

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. (2014). 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*.

Tsitsiklis, J., Bertsekas, D., and Athans, M. (1986). Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control*, 31(9):803–812.

Walton, J. (2016). How many gpus can you fit in a single system? http://www.pcgamer.com/how-many-gpus-can-you-fit-in-a-single-system/.

Wang, Z. and Bovik, A. C. (2009). Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117.

Williams, R. J. (1983). *Unit activation rules for cognitive network models*. Institute for Cognitive Science, University of California, San Diego.

Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, S., Choromanska, A. E., and LeCun, Y. (2015a). Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693.

Zhang, W., Gupta, S., Lian, X., and Liu, J. (2015b). Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950*.

Zinkevich, M., Weimer, M., Li, L., and Smola, A. J. (2010). Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603.