DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

Distributed Robust Learning

Akhil Yerrapragada

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Akhil Yerrapragada, `akhily@kth.se`
Master in Software Engineering of Distributed Systems
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
KTH Royal Institute of Technology

Examiner

Amir H. Payberah, `payberah@kth.se`
KTH Royal Institute of Technology

Supervisor

Nicolae Paladi, `nicolae.paladi@ri.se`
Research Institutes of Sweden

Abstract

Accuracy obtained when training deep learning models with large amounts of data is high, however, training a model with such huge amounts of data on a single node is not feasible due to various reasons. For example, it might not be possible to fit the entire data set in the memory of a single node, training times can significantly increase since the data-set is huge. To avoid these problems decentralized training is devised. In decentralized training, using the technique of Data parallelism, multiple nodes/workers make a local copy of the model and train it using a partition of the data-set. Each of these locally trained models are aggregated at some point to obtain the final trained model. Architectures such as Parameter server, All-reduce and Gossip use their own network topology to implement decentralized training. However, there is a vulnerability in this decentralized setting, any of the worker nodes may behave arbitrarily and fail. This type of failure is called byzantine failure. Here, arbitrary means any of the worker nodes may send incorrect parameters to others, which may lead to inaccuracy of the global model or failure of the entire system sometimes. To tolerate such arbitrary failures, aggregation rules were devised and are tested using Parameter server architecture. In this thesis we analyse the fault tolerance of Ring all-reduce architecture to byzantine gradients using various aggregation rules such as Krum, Brute and Bulyan. We will also inject adversaries during the model training to observe, which of the aforementioned aggregation rules provide better resilience to byzantine gradients.

Keywords

Byzantine resilient decentralized training, Gradient aggregation rules, (α, f)-Byzantine resilience, Fault tolerance, Ring all-reduce.

Abstract

Noggrannheten som erhålls vid träning av djupinlärningsmodeller med stora datamängder är hög, men det är inte möjligt att utbilda en modell med så stora mängder data på en enda nod av olika skäl. Det kan till exempel inte vara möjligt att passa hela datamängden i minnet på en enda nod, träningstiderna kan öka avsevärt eftersom datamängden är enorm. För att undvika dessa problem utformas decentraliserad träning. I decentraliserad utbildning, med hjälp av tekniken för dataparallellism, gör flera noder / arbetare en lokal kopia av modellen och tränar den med hjälp av en partition av datamängden. Var och en av dessa lokalt utbildade modeller samlas vid någon tidpunkt för att få den slutliga utbildade modellen. Arkitekturer som Parameterserver, All-reduce och Gossip använder sin egen nätverkstopologi för att implementera decentraliserad utbildning. Det finns dock en sårbarhet i denna decentraliserade inställning, någon av arbetarnoderna kan uppträda godtyckligt och misslyckas. Denna typ av fel kallas bysantinskt fel. Här betyder godtyckligt att någon av arbetarnoderna kan skicka felaktiga parametrar till andra, vilket ibland kan leda till felaktighet i den globala modellen eller att hela systemet misslyckas. För att tolerera sådana godtyckliga fel utformades aggregeringsregler som testas med hjälp av parametern serverarkitektur. I den här avhandlingen analyserar vi feltoleransen för Ring all-reduceringsarkitektur till bysantinska gradienter med hjälp av olika aggregeringsregler som Krum, Brute och Bulyan. Vi kommer också att injicera motståndare under modellutbildningen för att observera vilka av de ovannämnda aggregeringsreglerna som ger bättre motståndskraft mot bysantinska lutningar.

Nyckelord

Byzantinsk motståndskraftig decentraliserad träning, Gradientaggregeringsregler, (α, f)-Byzantinsk motståndskraft, Feltolerans, Ring all-reducera.

Acknowledgements

Firstly I would like to thank my parents, sisters and other family members for their incredible support throughout the period.

Special thanks to Amir H. Payberah, a benevolent professor from whom I received a lot of support over the thesis period and my supervisor Nicolae Paladi, for trusting in my ability and providing an opportunity to work under their leadership. Your support and feedback over the course of my thesis greatly assisted my overall understanding on the concept and motivated to explore further.

I would also like to thank Bank of India for sponsoring my education from past two years. It has been a fascinating journey with loads of learning that will assist throughout my carrier.

Contents

Chapter 1

Introduction

Decentralized deep learning is becoming widely important considering the capability to process huge information faster [1] and generate accurate models [2]. It can do so using techniques such as *Model parallelism* and *Data parallelism* [3]. In Model parallelism, we split the model's layers on every worker and and train the layers with the entire data-set. Such approach has the capacity to train large models in a decentralized way, however, it is also important to note that the Model parallelism can create imbalances when split across workers. On the other hand, in Data parallelism we replicate the entire model on every physical worker and train them using a mini-batch of the data-set in each iteration. This approach is widely adopted considering the ease of computing gradients at each iteration with reference to the mini-batch the worker has and sharing them with other workers.

Two important factors to consider here are the optimization algorithm, and architectural design that makes the communication between workers less latent. Among the optimization algorithms available [4], *Stochastic Gradient Descent* (SGD) is adopted considering the sub linear rate of convergence in a large scale learning perspective. Some of the well known architectures such as *Parameter server* [5] [6], *All-reduce* [7] and *Gossip* [8] can be used to implement decentralized model training. Such architectures have their own limitations on different factors such as latency and model convergence.

However, when operating under such decentralized conditions it is important to make sure that the chosen decentralized architecture is robust to failures. A failure can be anything that tries to make the architecture digress from the actual behaviour. One of such failure is known as *Byzantine failure* [9] where a worker behaves arbitrarily and fail. More specifically, a worker may send incorrect gradients to other workers and cause the model to not converge to an acceptable level. The reason is because the aggregation rule used by the workers cannot tolerate even a single byzantine gradient. To address this problem, *Gradient Aggregation Rules* (GAR's) are devised, namely, *Brute* [10], *Krum* [11], *DRACO* [12] and *Bulyan* [10]. These algorithms are known to provide *(α, f)-Byzantine Resilience* [11], meaning, the gradient that deviates the most from actual gradients at most by an angle will be considered as a byzantine gradient and will be excluded from the aggregation. The goal of each of these algorithms is to make sure that the model training remains unaffected despite receiving byzantine gradients from a byzantine worker.

Despite using a GAR if the model training still remains effected by byzantine gradients, then the aggregation rule offered weak/no byzantine resilience and vice versa. While Brute and Krum are known to offer weak byzantine resilience, Bulyan and DRACO offers strong byzantine resilience. *Multi-Bulyan* [13] was proposed proving that a GAR can also be fast along with being strong byzantine resilient. Other algorithms such as *Guan Yu* [14] and *LiuBei* [15] are known to offer byzantine resilience to both Parameter servers and workers.

In order to simulate the behaviour of any of the above on a single computing device, it is important to have a distributed message passing platform that helps to test the resilience of aforementioned architectures to byzantine workers. *Kompics* [16] a peer-to-peer component framework for implementing distributed algorithms greatly assists in such a scenario, which we will be using for analyzing the the architecture's behaviour.

1.1 Problem

The byzantine resilient GAR's stated above are analysed for resilience using Parameter server approach but not on all-reduce architecture.

The goal of this thesis is to analyze the fault tolerance of Ring all-reduce architecture to byzantine gradients by using Krum/M-Krum, Brute and Bulyan GAR's.

1.2 Methodology

The objective of this thesis is to observe the effects of byzantine gradients on Ring all-reduce architecture and understanding the architecture's resilience to such in presence of a byzantine resilient GAR. The approach of this project is empirical as it involves experimenting with various GAR by injecting byzantine attacks. Below are the methods required fro completion of this thesis:

- Implementation of a communication efficient Ring all-reduce architecture, which will be made robust to byzantine failures by GAR's.
- Integrating *Multi Layered Perceptron* (MLP) [17] to the byzantine resilient Ring all-reduce.
- Injecting adversaries when training MLP using *MNIST* data-set [18].
- Observing resilience of the architecture to adversaries with increase in worker count.

1.3 Limitations

The scope of this thesis is limited to analyzing the byzantine resilience of Ring all-reduce architecture using Krum, Brute and Bulyan GAR's. *Deeplearning4j* [19], a library to build artificial neural networks for java based implementations will be used for building deep learning models and the experiments will only be carried out using MNIST data-set. The implementation will be carried out using *Scala* programming language [20], therefore, when building a deep learning model, the number of neurons per layer are limited to a specific number considering the Java thread limitation of 65535 bytes per thread. This may lead to slightly less accurate model. Analysis using other distributed architectures, programming languages, message passing libraries and data-sets will be left for future work.

1.4 Ethics and Sustainability

The code base and analysis performed on the All-reduce architecture by injecting adversaries is implemented independently by the student. Therefore, it will not lead to any ethical issues. Regarding sustainability, the overall project is carried out using open source software's that are free of cost and using a single computing source, which makes it both economical and sustainable.

1.5 Structure

The structure of thesis is as follows:

Chapter 2 covers all the relevant background information including system architectures and aggregation rules required to get an in-depth understanding of this thesis.

Chapter 3 explains the Ring all-reduce architecture and integration of it with various GAR's.

Chapter 4 describes the adversary and the effect it has on aggregation rules.

Chapter 5 depicts the robustness of Ring all-reduce architecture to byzantine gradients when using various GAR's under varying worker counts.

Chapter 6 discusses the possible future work in building distributed systems robust to byzantine failures.

Chapter 2

Background

2.1 Kompics

Kompics is a message passing framework used to build distributed abstractions. It helps simulating the real world behaviour of peer-to-peer systems. It does so by building protocols using components. Components are the building blocks of the Kompics framework and are considered to have the facility to be reusable. These components execute concurrently and will respond to the incoming events. An event is considered as a message that is being transferred between components or services within a component. This is called Event-based communication.

One component can communicate with other using ports and the component that is being communicated to can be in a different node. These ports are connected by channels and can pass information in First In First Out (FIFO) order. A component can internally have many services such as a broadcast service that can transmit events to all nodes. Handlers in a component run the algorithm being implemented and are subscribed to a port that constantly listens to the incoming events. Components in Kompics are stateful, meaning, we can achieve synchrony between multiple components in different nodes or in the same node using the state.

In our implementation, we use *Kompics Scala* [21]. Kompics Scala is an extension of Kompics, which enables us with better utilization of pattern matching features and in-return eases implementation of distributed algorithms such as distributed SGD. Also we use two components, one for generating the network topology and the other for running services. The former component is responsible for generating the required communication structure for each node, meaning, each node in the network will receive the information via this component on the topology in which they should be communicating. Here it is important to note that only one node from the list of subscribed nodes is responsible for generating the network topology for all nodes. The latter component can also be called as the *Main component* since it hosts our services namely, *Byzantine resilience* and *Distributed model training*. While former service is responsible for making the architecture byzantine resilient, the latter is responsible for training the deep learning models.

Every node taking part in the architecture has this component with the aforementioned services internally and will execute it in parallel with other nodes. The execution will be triggered once the handler subscribed to a port receives an event. This incoming event will be pattern matched with reference to the implementation in the handler to trigger a specific function. This way nodes in the network communicate with each other. Each of the nodes will have three states, namely,*Waiting*, *Collecting* and *Seeding*. These states will be used in accordance with the role a node plays in the implemented architectures. The timer is responsible to listen to the incoming events periodically.

2.2 Data Parallelism

Data parallelism is one of ways that assist training the deep learning models quickly [22]. As explained earlier the idea behind Data parallelism is simple, we replicate the model to multiple nodes and train the model with a mini-batch of the data-set in each node. The gradients computed by each node are shared with other nodes to update their weights and biases with respect to received gradients. The main reason for adopting this approach is considering that it reduces the training time significantly. To ensure this, various communication mechanisms have originated, namely, synchronous, asynchronous, stale-synchronous and local stochastic gradient descent. While synchronous mechanism being the traditional approach where the nodes participating in communication process will wait for others to complete their computation before proceeding to next step, asynchronous ensures vise versa. The problem with synchronous approach is that even a single node can cause significant delay. To eliminate this, we have stale-synchronous approach where we just ignore the worker that is slow and proceed with the computation. Finally, local SGD ensures that the nodes perform computation for multiple iterations before performing the model aggregation. Another key aspect in Data parallelism is the communication congestion. When sharing the parameters with other nodes, it is important for the entire topology to make sure that the network traffic is optimal since the whole point of Data parallelism is to ensure computation is fast. Techniques such as Quantization and Sparsification have evolved to ensure minimum congestion. While the former considers only certain bits of a dimension in a vector, the latter zeros out an unimportant dimension while transmitting the vector in a network. Though these approaches appear to have provided a solution for communication congestion, they have significant effect on model consistency as the truncated vector can only contribute to the delay in convergence.

Overall, the performance of the above discussed approaches strictly depend on the architecture they are in. We have three important architectures overall to implement data parallelization, namely, Parameter server, All-reduce and Gossip. In Parameter server approach we can achieve an accurate model using synchronous communication but it comes with significantly high network traffic. One the other hand, using asynchronous communication can lead to less accurate model but reduces network traffic greatly. Using other approaches such as local SGD and stale synchronous can only lead to moderate accuracy and congestion levels. In All-reduce approach can only have synchronous and local SGD approaches considering that the architecture's topology is not suitable for asynchronous and stale synchronous communication framework. The synchronous approach is capable in achieving better accurate models with low traffic congestion though the communication frequency among nodes is quite high, on the other hand, using local SGD can lead to moderate accuracy maintaining the traffic congestion similar to the synchronous approach. Finally, the gossip architecture can be operated with synchronous, asynchronous and local SGD communication frameworks. This architecture is known to provide low model accuracy and network congestion considering that a node can only communicate with peers. In this thesis we will be using All-reduce with synchronous approach [3].

2.3 Multi-Layer Perceptron (MLP)

Before looking into Multi-Layer Perceptron [23], let us quickly understand Perceptron [24]. A perceptron or a neuron is a binary classifier that is known to solve linearly separable problems. It takes weighted inputs and passes them through an activation function to generate output. For example, AND and OR logic gates can be considered as linearly separable, meaning, if we assume a cartesian plane, we will be able to clearly separate the data points into two where one side of it referring to data belonging to one type and the other side belonging to another. However, imagine if we have a non linearly separable problem like XOR as shown in figure 2.3.1 or ability to classify a number between ten different classes.

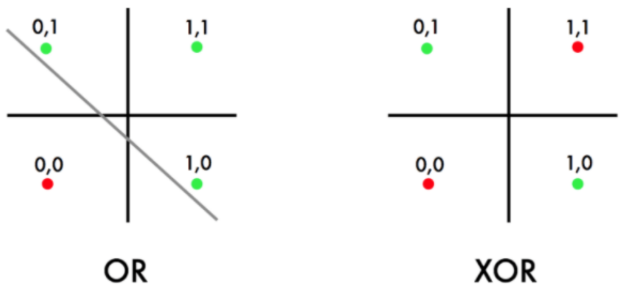


Figure 2.3.1: The XOR Problem [25]

In such a case perceptron cannot provide a solution, which leads to multi layer perceptrons that has not just one but multiple perceptrons spread across multiple layers. Generally, we have input layer, hidden layer and an output layer. The weighted input will now pass from input layer to output layer via hidden layer and this is called as a fully connected feed-forward neural network. It uses backpropagation to train and update the gradients of it.

2.4 The Optimization Algorithm

In this thesis we will using stochastic gradient descent as our optimization algorithm. We parallelize the implementation of it among multiple nodes [26], meaning, each worker will compute a mini-batch gradient descent [27] of the available training samples and the equation for it can be represented as mentioned in 2.1. Here G represents the gradient of the loss function ∇l computed locally on a node, x is the mini-batch chosen for training, w is the computed weight.

$$G_i\left(w, \beta_i\right)=\frac{1}{\left|\beta_i\right|} \sum_{x \in \beta_i} \nabla l(w, x) \tag{2.1}$$

At each iteration on the mini-batch we have two phases, feed forward and back propagation. While the feed-forward phase computes the loss value of chosen mini-batch w.r.to weights and biases during an iteration, the back propagation computes the gradients w.r.to the loss value computed in the earlier phase. These gradients on a node are then shared with other nodes and an aggregation is then performed at some point depending on the architecture the node is in. The aggregation looks as mentioned in 2.2. Here F represents the function in a node that performs the aggregation of all received gradients. k represents the total number of workers.

$$F\left(G_1, \cdots, G_k\right) = \frac{1}{k} \sum_{i=1}^{k} G_i\left(w, \beta_i\right) \tag{2.2}$$

These aggregated gradients are then used to update the weights locally on a node as mentioned in 2.3 and we proceed with next iteration. The number of iterations η are custom specified and we choose it depending on obtained model accuracy at each iteration.

$$w := w - \eta F\left(G_1, \cdots, G_k\right) \tag{2.3}$$

2.5 System Architectures

Among the above mentioned three system architectures, we will be discussing more on the Parameter server and All-reduce approaches. The Gossip architecture will not be discussed in this thesis and will be considered out of scope.

2.5.1 Parameter Server

In Parameter server architecture, a node can act in two roles. One as a Parameter server and the other as a Worker. During an iteration, a worker is responsible to compute the gradient of a mini-batch as mentioned in equation 2.1 and shares it with the Parameter server. The parameters server is responsible to receive the gradients from workers and perform the gradient aggregation mentioned in equation 2.2 and updates the weights as mentioned in equation 2.3. These updated weights are then shared with the workers to update their local weights before continuing with the next iteration. Though this architecture is widely known and adopted, it is known to cause network congestion with linear increase of Parameter servers or workers and is prone to single point of failure [28]. Figure 2.5.1 depicts workers sharing gradients to Parameter server to perform the aggregation.

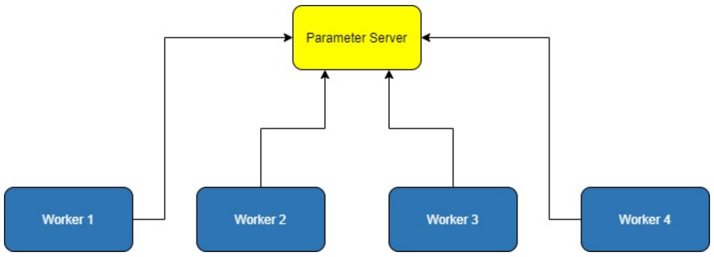


Figure 2.5.1: Parameter server architecture

2.5.2 All-Reduce

Before looking into All-reduce architecture and variants of it, let us quickly understand the Reduce operation. A Reduce operation simply converts a set of numbers into smaller set. Let us look into an example, imagine our Reduce operation is multiplication and we have four workers where three of them send an input to fourth worker. The role of fourth worker is to reduce the inputs received and the output will now be the multiplied (reduced) value. It is important to observe here that the final output is in fourth worker but not in the one's that shared their inputs. In All-reduce, all workers including the one's shared inputs will have the final output. Here a node can play only one role, i.e. worker. We have two phases in All-reduce, one is the share phase and the other is the reduce phase. The final goal is to ensure that all workers participating in the communication process has the reduced results. In our implementation, during share phase, a worker is responsible to compute gradients and share them with other workers. During the reduce phase, a worker is responsible to receive the gradients from all other workers and reduce them to a single vector before updating the weights w.r.to them, meaning, all workers will have identical parameters at the end of each round. We will discuss further on this in detail in the upcoming sections.

Considering linear increase in data and necessity for computational power, this architecture can provide better support compared to the Parameter server architecture. This is also known to use optimal network bandwidth [28]. Here it is important to note that all workers in the network will execute the aggregation rule where as in the Parameter server approach only the Parameter server is responsible to perform the aggregation.

The topology of All-reduce can be built in a way that it is communication efficient.

Some of the known types are, All-to-all all-reduce, Master-worker all-reduce [29], Tree all-reduce, Round-robin all-reduce, Butterfly all-reduce [30] and Ring all-reduce [31]. Figure 2.5.2 is an example of All-to-all all-reduce topology.

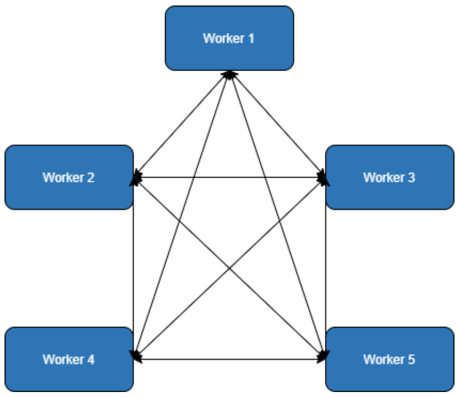


Figure 2.5.2: All-to-all all-reduce architecture

In All-to all-reduce topology, every worker sends the data to every other worker. Post receiving the data from all other workers, a worker will perform reduce operation. Though this approach looks straightforward, there is a lot of network traffic that might cause delay in transmitting messages and linear increase in workers can only contribute to additional latency. In Master-worker all-reduce, one of the workers act as a master and receives all the data from other workers. Post receiving it reduces the data and re-shares with all workers. Though this approach appear to have solved the network traffic problem, it is still vulnerable to single point of failure and is not scalable. We have other approaches such as Tree all-reduce, Round-robin all-reduce and Butterfly all-reduce that tries to reduce the bandwidth and latency, however, Ring all-reduce is known to provide scalability with less traffic congestion. In Ring all-reduce we have two phases, the first is Share-reduce and the second is All-share phase; for each round we have both the phases executed. During Share-reduce phase, each worker w will split the data into n partitions and send the data at ith position to the worker at $(w + 1)\%n$th position. Here n is total number of workers and i refers to the current worker index in the topology. The worker receiving the data at index i will perform a reduce operation on it with its own data on same index and forwards it to successor in the ring. Finally, at the end of $n - 1$ Share-reduce rounds, each worker in the topology will hold a completely reduced data for a specific index. During share-only phase a worker will share the reduced data it has with all other worker in the same ring fashion.

Let us draw a quick comparison between Master-worker all-reduce and Ring all-reduce in terms of network latency. For master-worker variant, the communication cost to send data from the master to all the workers is $N(n-1)$ and for all workers to send their data to the master is $N(n-1)$. Here N is the data length and n is the total number of workers participating in communication. The total communication cost for master-worker is $2(N*(n-1))$. In Ring all-reduce, during the Share-reduce phase, each worker sends $N/n*(n-1)$ data to other workers. During All-share phase the communication cost remains the same, which makes it $2(N/n*(n-1))$ in total. This clearly shows that the Ring all-reduce is communication efficient when compared to master-worker variant. In this thesis implementation we will be using Ring all-reduce variant considering the advantages like less communication traffic and high scalability.

2.6 Byzantine Resilience

From equation 2.2, we can understand that each node will perform an aggregation of the gradients. This function can be referred to as the aggregation rule that takes all gradients computed from nodes as an input and outputs a single gradient vector that will be shared with all the nodes. Now let us imagine if our aggregation rule is Average and one of the inputs received is byzantine, meaning, one of the node that shared the gradients is behaving arbitrarily and sharing incorrect gradients. In such case, the average aggregation rule will simply include the byzantine gradient sent by byzantine worker and share the aggregated vector with all other nodes, meaning, all the nodes involved in communication are effected by this byzantine worker.

Now the solution to this problem is to make sure that the aggregation rule somehow filters out the incorrect gradients received from byzantine workers to make the entire architecture robust to byzantine failures. To address this we have distance and median based aggregation rules. The distance based rules being Krum/M-Krum and Brute and the median based being Bulyan. The idea behind the distance based is simple, if a computed N-dimensional gradient received from a node is far by distance from other N-dimensional gradients received from remaining nodes, we can consider it as a byzantine. Consider the figure 2.6.1, where we can see the the byzantine gradient (red arrow) is far by distance from non byzantine gradients (dashed black arrow) while the optima is pointing towards minimum (blue arrow). Here, it is important to know how far a byzantine gradient is from actual gradient before terming it as a byzantine gradient by a node. To address this problem the aggregation rules employed various strategies, however, they sometimes fail to differentiate between byzantine and actual gradients. Therefore, they were known to offer weak byzantine resilience.

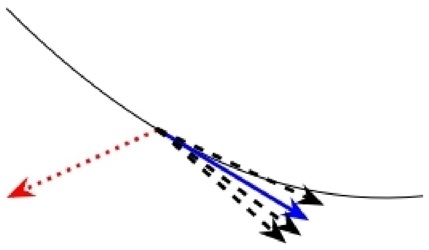


Figure 2.6.1: A byzantine gradient digressing from reaching optima during gradient descent [11, Figure 1]

On the other hand we have median based aggregation rules such as Bulyan that are known to offer strong byzantine resilience. The idea is that each co-ordinate of the N-dimensional vector is agreed by majority of the vectors instead of a specific number [10] as in distance based aggregation rules.

2.7 Gradient Aggregation Rules (GAR's)

In this section we will discuss more on GAR's and their byzantine resilience. Let us first look into the problem with Average aggregation rule. The Average aggregation rule cannot tolerate even a single byzantine gradient. To prove this let us look into the equation 2.2, which simply averages all the gradients. Now let us assume that one worker G_k is byzantine and proposed $kU - \sum_{i=1}^{k-1} G_i$. If we include the value proposed by this worker in averaging i.e. in equation 2.2, it will then be $F = U$, which clearly shows that the average GAR doesn't tolerate a byzantine worker and can lead to a model convergence that is unacceptable. In the result section, we will look into an example to see if this holds true in real-time analysis and also to prove correctness of the implementation [11, Lemma 1].

2.7.1 (α, f) - Byzantine Resilience

(α, f) - Byzantine Resilience is a condition that a GAR must satisfy in order to be proven to provide either weak or strong byzantine resilience. For a GAR to be considered

(α, f) - byzantine resilient, it must satisfy two conditions. As shown in figure 2.7.1, first the vector F must not be far from the estimated actual gradient (non byzantine) $\mathbb{E}[\mathrm{g}] = \mathcal{J} = \nabla J(w)$. "How far?" is measured by the angle α, which is of range $[0°, 90°]$. Overall, $\|\mathbb{E}[F] - \mathcal{J}\|$ should be $\leq r$. Second, moments of F must only be controlled by $\mathcal{J}$ i.e. the correct gradient estimator.

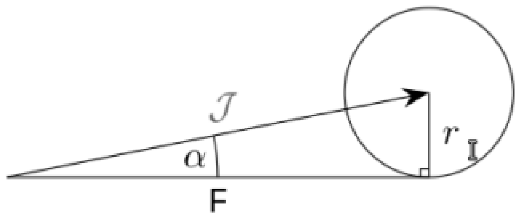


Figure 2.7.1: (α, f) - Byzantine Resilience [11, Figure 2]

In the upcoming sections we will discuss further in detail on some GAR's that satisfy the (α, f) - Byzantine Resilience criteria.

2.7.2 Krum

Krum is one of the most recognized algorithms, which is being referred to as a base implementation either for comparison or for the advancement of analysis in the field of federated learning, building robust performance efficient distributed systems and building byzantine resilient GAR's [32] [33] [34] [35] [36]. The idea behind Krum GAR is simple, for $n \geq 2f + 3$ where n is the total worker count and f are the byzantine workers, we exclude the gradients that are too far away from the current gradient. In detail, for every gradient received from a worker we pick $n - f - 2$ closest vectors and compute the score of it. The equation for computing the score is determined as mentioned in 2.4. Here G_i refers to as the current gradient and G_j refers to the gradient in $n - f - 2$ closest vectors.

$$s(i) = \sum_{i \to j} \|G_i - G_j\|^2 \tag{2.4}$$

Finally, we pick a gradient G_{i_*} from a worker i that has the least computed score as mentioned in 2.5. We determine this G_{i_*} and $n - f - 2$ closest vectors of it to be in a tightly packed cluster when compared to other gradients and their closest vectors.

$$F\left(G_1, \cdots, G_n\right) = G_{i_*} \tag{2.5}$$

In Multi-Krum (M-Krum) a variant of Krum, we select m gradients among the proposed G_n gradients whose score is the lowest. Here m is in range $1 \leq m \leq n - f - 2$. Finally we compute the average of these selected m gradients as shown in the equation 2.6.

$$\frac{1}{m} \sum_{i} G_{i_*} \tag{2.6}$$

Though this approach appear to be a stronger solution to provide byzantine resilience in shorter dimensions, it suffers from *Data Concentration* problem in *The Curse of Dimensionality* [37] for higher dimensions. To mitigate this problem BRIDGE [38] is devised. Krum or M-Krum is known to provide weak byzantine resilience since it is distance a based function that agrees on only some gradients.

2.7.3 Brute

Brute is an (α, f) - byzantine resilient GAR that offers weak byzantine resilience. For it to tolerate f byzantine workers, it must satisfy the condition $n \geq 2f + 1$ where n is the total worker count. We first obtain subsets R each of length $n - f$ from overall submitted gradients. As mentioned in equation 2.7, we pick a subset whose maximum computed distance between the gradients in that subset is minimum when compared with other subsets. Here, G_{i_*} and G_{j_*} represents gradients belonging to a subset and X represents the set of gradients.

$$\arg\min_{X \in R} \left(\max_{(G_i, G_j) \in \mathcal{X}^2} \left(\|G_i - G_j\| \right) \right) \tag{2.7}$$

We finally take that subset and average all the gradients as mentioned in equation 2.8. Though this approach offers good resilience to byzantine gradients, it is computationally expensive and is not feasible approach when having a large worker count.

$$F\left(G_1, \cdots, G_{\mathrm{n}}\right)=\frac{1}{n-f} \sum_{G \in \mathcal{S}} G \tag{2.8}$$

2.7.4 Bulyan

Finally we have Bulyan, which is also an $(\alpha,\ f)$ - byzantine resilient GAR but it offers strong byzantine resilience. This algorithm acts as a solution for the high dimensionality problem we discussed earlier since it is median based and requires each coordinate to agree on majority of the gradients. Also, Bulyan is currently being used to analyse the fault tolerance to byzantine gradients in the field federated learning [39] [40]. For it to tolerate f byzantine workers, it must satisfy the condition $n \geq 4f+3$ where n is the number of workers and f are the byzantine workers. Bulyan has two stages, during stage one we recursively use a weak byzantine resilience GAR until θ iterations to obtain a set of gradients S that are mostly non-byzantine. Here, $\theta=n-2f$. During stage two, we first obtain the median of θ selected gradients as mentioned in equation 2.9. Here $i \in [1..d]$ where d is the total dimension count.

$$\text{median}[i]=\arg \min_{m=Y[i], Y \in S}\left(\sum_{z \in S}|Z[i]-m|\right) \tag{2.9}$$

Post obtaining the median, we find β closest coordinates to the median coordinate as mentioned in equation 2.10. Here $\beta=\theta-2f \geq 3$.

$$M[i]=\arg \min_{R \subset S,|R|=\beta}\left(\sum_{x \in R}|X[i]-\text{median}[i]|\right) \tag{2.10}$$

Finally, we compute the average of the β selected gradients as mentioned in equation 2.11.

$$F[i]=\frac{1}{\beta} \sum_{X \in M[i]} X[i] \tag{2.11}$$

Now this approach guarantees model convergence even in presence of byzantine gradients. The two stage approach of it appears as if the model is being updated only with non-byzantine gradients. We will discuss more on this in Results chapter.

Chapter 3

Ring All-Reduce Robust to Byzantine Failures

In this section we will be discussing in depth on how the ring topology is obtained using Kompics and how the presence of a byzantine worker effects the Ring all-reduce architecture. We will also delve into the implementation of deep learning model and provide a thorough analysis on how the chosen GAR's provide resilience to such byzantine workers when training the deep learning model.

3.1 Topology

In previous sections, we have discussed on various services hosted on components in Kompics and how the All-reduce topology works. As a beginning step we spawn multiple workers that will trigger their components internally. The ports of these components are open to receive any incoming events from other workers. Here, it is important for one of the workers to take the responsibility of knowing the addresses of all the workers to generate the topology. For the convenience of differentiating with other workers we call this worker as *Bootstrap Server* and the rest as *Bootstrap Client's*. The bootstrap server will first be initiated, and it waits for the bootstrap clients to connect to it as shown in 3.1.1. Once bootstrap server reaches the threshold (n), i.e., minimum number of connecting workers (including itself), it initiates ring topology by assigning predecessors and successors to each worker.

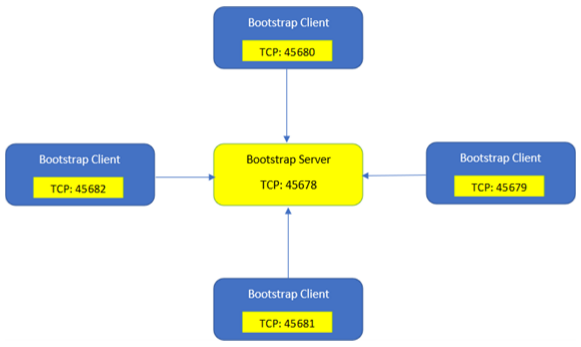


Figure 3.1.1: Bootstrap server waiting for clients to connect

The bootstrap server will then share the assigned predecessor and successor information to all the connected clients including itself as shown in 3.1.2.

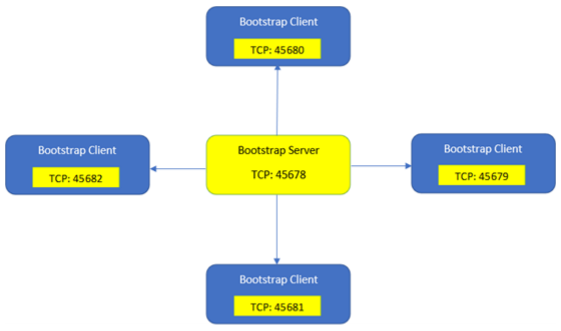


Figure 3.1.2: Bootstrap server sending predecessor and successor information to clients

Once the information is sent to all workers, the bootstrapping is complete. We can assume the ring-topology as shown in 3.1.3 and each node in the ring can only communicate via *Transmission Control Protocol* (TCP) ports with its successor.

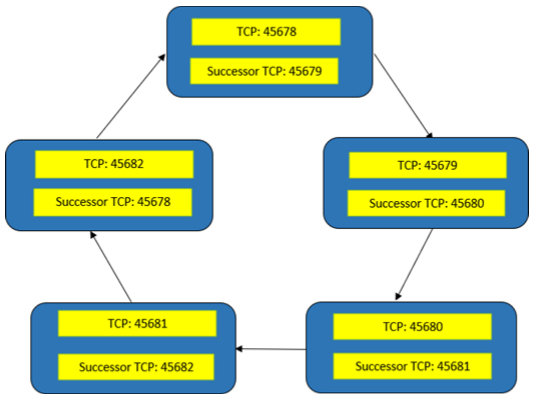
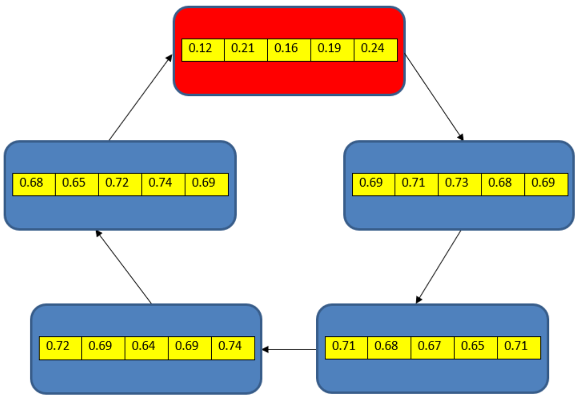


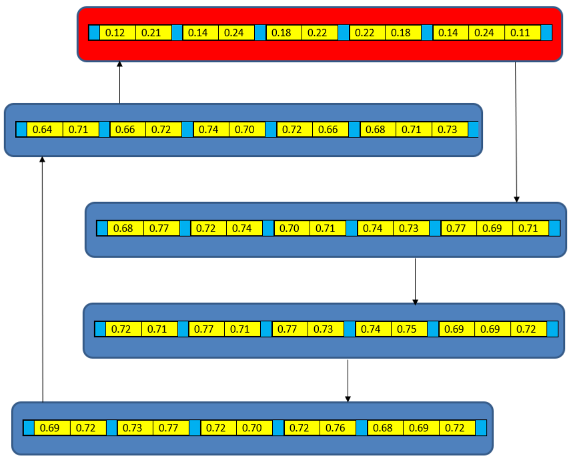
Figure 3.1.3: Ring topology

3.2 Share-Reduce

This is the first phase of Ring all-reduce. Before beginning Share-reduce, each worker will trigger the Distributed model training service available in the main component. We will discuss more about this service in the next section. The output of this service is a generated list of gradients. For example, let us consider the following figure 3.2.1, which depicts each worker with the computed gradients. Here, it is important to note that the worker in red (Node one) is byzantine and the gradients of it are clearly digressing from rest of the workers. We will use this example to delve deeper into the Ring all-reduce working mechanisms.



(a) Gradient vector length proportional to number of workers

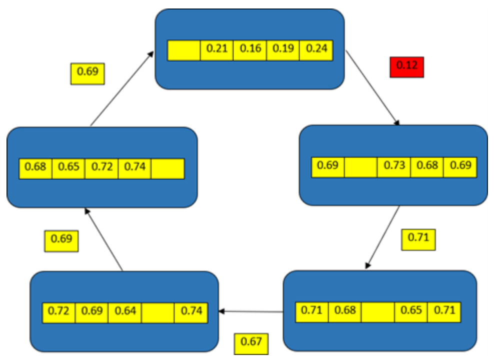


(b) Gradient vector with each index containing partitions of gradients

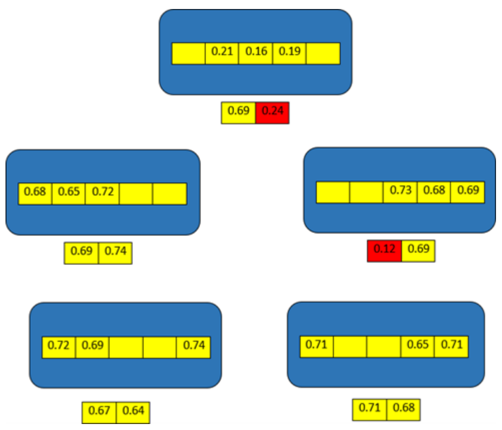
Figure 3.2.1: Workers with computed gradients

For a total of five workers, we have four Share-reduce and All-share phases respectively. During Share one as shown in 3.2.2a, each worker i will send the gradient at index i to the successor. In our example, we have chosen the length of gradient vector to be proportional to the number of workers for the ease of understanding. Generally, the number of gradients are significantly larger. In such a case we assume the total number of gradients to be split into k partitions where k is the total number of workers and each worker will then share the partition at index i instead of a gradient. This is depicted in 3.2.1b. The worker receiving the gradient of index i will select from its own data the gradient matching the incoming index. In case if the worker received a partition of gradients, it will select its own partition matching the incoming index and each gradient at index j in partition will be matched with the gradient j of incoming partition.

During Reduce one as shown in 3.2.2b, the incoming gradient and the one selected by the current worker will then be appended together. This will then be shared again (Share two) to the successor of current worker as shown in 3.2.3a and so on.

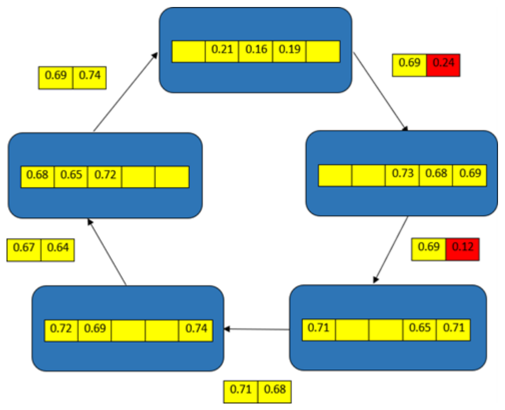


(a) Share one

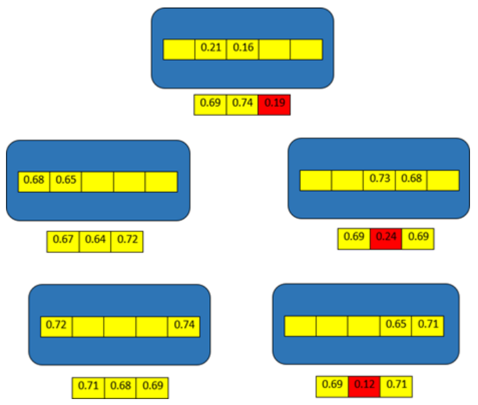


(b) Reduce one

Figure 3.2.2: Share-Reduce phase one

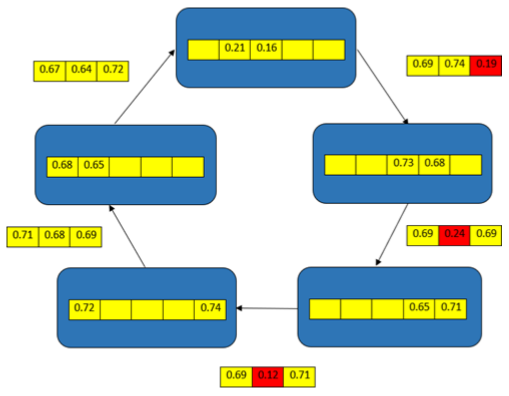


(a) Share two

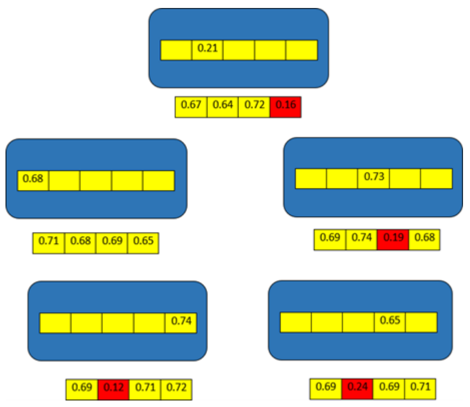


(b) Reduce two

Figure 3.2.3: Share-Reduce phase two

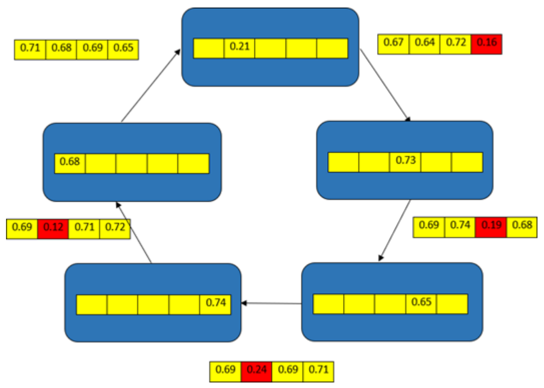


(a) Share three

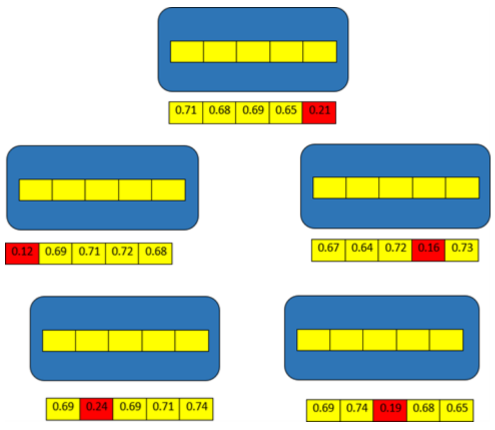


(b) Reduce three

Figure 3.2.4: Share-Reduce phase three



(a) Share four



(b) Reduce four

Figure 3.2.5: Share-Reduce phase four

At the end of Share-reduce phase, each worker will hold a list of gradients computed by all workers for a specific index. For example, in Reduce four, worker five will hold all the information belonging to index zero of each worker. Here, it is important to note how the gradients from a byzantine worker got spread across all workers in the ring.

3.3 Byzantine Resilience using Krum/M-Krum

Once Share-reduce phase is completed, we trigger the GAR function in each worker by giving the received gradients as input. Let us consider we give below as input in worker five and our GAR to be M-Krum.

0.12	0.69	0.71	0.72	0.68

Figure 3.3.1: Gradient vector with one byzantine gradient

Here, we obtain two closest gradients for each gradient. For example, the closest gradients for 0.12 is 0.69 and 0.68 in our example. Post obtaining the closest for all other gradients in the list, we start computing scores for each of them. The score can be computed using the following:

$$s(i) = \sum_{i \to j} \|G_i - G_j\|^2 \tag{3.1}$$

Below are the scores,

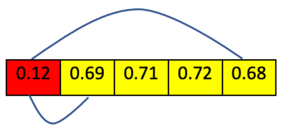


Figure 3.3.2: Two closest vectors from first gradient

$s(0) = (0.69–0.12)^2 + (0.68–0.12)^2 => 0.3249 + 0.3136 => 0.6385$

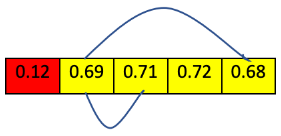


Figure 3.3.3: Two closest vectors from second gradient

$s(1) = (0.69–0.68)^2 + (0.71–0.69)^2 => 0.0001 + 0.0004 => 0.0005$

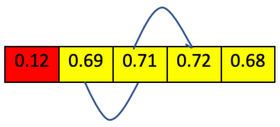


Figure 3.3.4: Two closest vectors from third gradient

$s(2) = (0.71–0.69)^2 + (0.72–0.71)^2 => 0.0004 + 0.0001 => 0.0005$

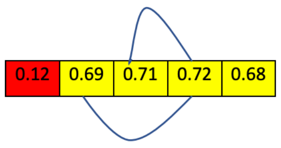


Figure 3.3.5: Two closest vectors from fourth gradient

$s(3) = (0.72–0.71)^2 + (0.72 - 0.69)^2 = 0.0001 + 0.0009 => 0.001$

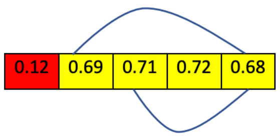


Figure 3.3.6: Two closest vectors from fifth gradient

$s(4) = (0.69–0.68)^2 + (0.71–0.68)^2 = 0.0001 + 0.0009 = 0.001$

After computing the scores, we can sort the values obtained in the ascending order to obtain the least two (m) scores. The order will be $s(1) < s(2) < s(3) < s(4) < s(0)$, therefore we can pick $s(1)$ and $s(2)$. Hence, gradients one and two will be selected for being averaged ignoring the byzantine gradient.

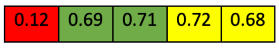


Figure 3.3.7: Selected gradients as non-byzantine by M-Krum

The average of both would be 0.70, which is our value for $G(0)$. Similarly, every other worker will compute for $G(1)$, $G(2)$, $G(3)$, $G(4)$ respectively and will share with other workers during All-share phase, which we will be discussing in the upcoming sections.

3.4 Byzantine Resilience using Brute

Let us assume that the GAR in each worker is Brute using the same input as depicted in figure 3.3.1. Though Brute is capable of tolerating more byzantine workers in this case, say, for five workers it can tolerate gradients from two byzantine workers, we will proceed in our example using one byzantine worker considering ease of comparison between Brute and other GAR's. Firstly, we obtain subsets of length $n - f$ from the given input as show in figure 3.4.1.

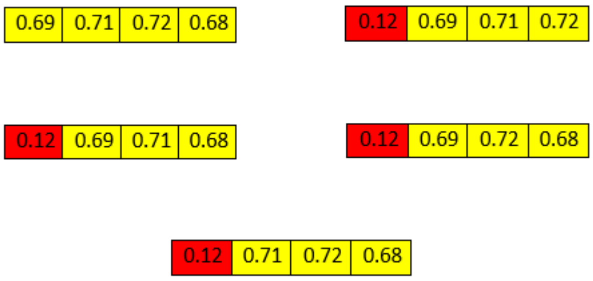


Figure 3.4.1: Generated subsets S of input figure 3.3.1

Post obtaining all the subsets, for each subset we compute maximum distance between the gradients. Let us take for instance the subset $S(0)$ shown in figure 3.4.2

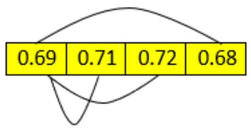


Figure 3.4.2: Distance from gradient one to every other

$s(0) = (0.69)^2 - (0.71)^2 + (0.69)^2 - (0.72)^2 + (0.69)^2 - (0.68)^2 \Rightarrow -0.028 - 0.0423 + 0.0137 \Rightarrow 0.0566$

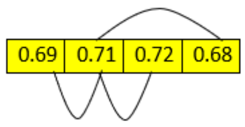


Figure 3.4.3: Distance from gradient two to every other

$s(1) = (0.71)^2 - (0.69)^2 + (0.71)^2 - (0.72)^2 + (0.71)^2 - (0.68)^2 \Rightarrow 0.028 - 0.0143 + 0.0417 \Rightarrow 0.0554$

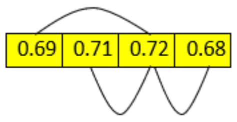


Figure 3.4.4: Distance from gradient three to every other

$s(2) = (0.72)^2 - (0.69)^2 + (0.72)^2 - (0.71)^2 + (0.72)^2 - (0.68)^2 \Rightarrow 0.0423 + 0.0143 + 0.056 \Rightarrow 0.1126$

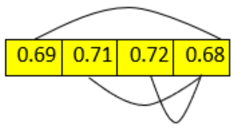
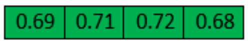


Figure 3.4.5: Distance from gradient four to every other

$s(3) = (0.68)^2 - (0.69)^2 + (0.68)^2 - (0.71)^2 + (0.68)^2 - (0.72)^2 \Rightarrow -0.0137 - 0.0417 - 0.056 \Rightarrow 0.1114$

Here, $s(1) < s(0) < s(3) < s(2)$, meaning, the maximum value obtained in this subset $s(2)$. Similarly, we compute for other subsets to obtain the maximum value in their subset. It can be clearly understood that the one's with byzantine gradients will surely have larger maximum distances compared to one's that have non byzantine gradients. This way we can conclude that $S(0)$ will have small maximum distance when compared to other subsets since it doesn't have any byzantine gradients.



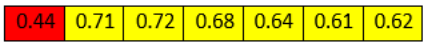
0.69	0.71	0.72	0.68

Figure 3.4.6: Chosen non-byzantine gradients to perform averaging

Finally, with reference to equation 2.7 we compute the average of the chosen gradients and share it with other worker during the All-share phase.

3.5 Byzantine Resilience using Bulyan

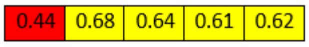
Let us now assume that the GAR in each worker is Bulyan. We cannot assume the same input as shown in figure 3.3.1 since Bulyan requires at-least seven workers to tolerate one byzantine worker. Let us consider the following input as shown in figure 3.5.1



0.44	0.71	0.72	0.68	0.64	0.61	0.62

Figure 3.5.1: Gradient vector with one byzantine gradient

During phase one, we give the aforementioned input to one of the weak byzantine resilience GAR's to obtain θ gradients. Let us assume we choose Brute as this GAR. From equation $\theta = n - 2f$, we can say that our θ value in this case will be 5. Therefore, we use Brute until θ iterations and obtain five gradients which will mostly be the non-byzantine gradients since the vector chosen as example is not high dimensional. However, this might not be the case in high dimensional scenarios where there can be byzantine gradients that escape Brute since it offers weak byzantine resilience. Let us now assume that Brute failed to filter out the byzantine gradient and after five iterations the output at the end of phase one looks as depicted in figure 3.5.2.

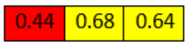


0.44	0.68	0.64	0.61	0.62

Figure 3.5.2: Gradients chosen by Brute after θ iterations

At the beginning of phase two, we first obtain the median of θ selected gradients. The median co-ordinate with reference to equation 2.9 in this case is fourth co-ordinate i.e. 0.62. Let us now obtain subsets of length β, where $\beta = \theta - 2f \geq 3$, therefore, in our case $\beta = 3$.

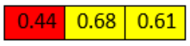
Since we now have subsets and the median value, let us compute for each subset



0.44	0.68	0.64

Figure 3.5.3: Subset 1 of 10

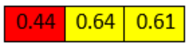
$s(0) = (0.44 - 0.62) + (0.68 - 0.62) + (0.64 - 0.62) => -0.18 + 0.06 + 0.02 => |-0.1| => 0.1$



0.44	0.68	0.61

Figure 3.5.4: Subset 2 of 10

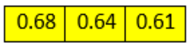
$s(1) = (0.44 - 0.62) + (0.68 - 0.62) + (0.61 - 0.62) => -0.18 + 0.06 - 0.01 => |-0.13| => 0.13$



0.44	0.64	0.61

Figure 3.5.5: Subset 3 of 10

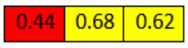
$s(2) = (0.44 - 0.62) + (0.64 - 0.62) + (0.61 - 0.62) => -0.18 + 0.02 - 0.01 => |-0.17| => 0.17$



0.68	0.64	0.61

Figure 3.5.6: Subset 4 of 10

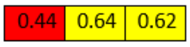
$s(3) = (0.68 - 0.62) + (0.64 - 0.62) + (0.61 - 0.62) => 0.06 + 0.02 - 0.01 => |0.07| => 0.07$



0.44	0.68	0.62

Figure 3.5.7: Subset 5 of 10

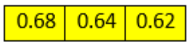
$s(4) = (0.44 - 0.62) + (0.68 - 0.62) + (0.62 - 0.62) => -0.18 + 0.06 + 0 => |-0.12| => 0.12$



0.44	0.64	0.62

Figure 3.5.8: Subset 6 of 10

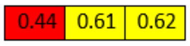
$s(5) = (0.44 - 0.62) + (0.64 - 0.62) + (0.62 - 0.62) => -0.18 + 0.02 + 0 => |-0.16| => 0.16$



0.68	0.64	0.62

Figure 3.5.9: Subset 7 of 10

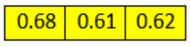
$s(6) = (0.68 - 0.62) + (0.64 - 0.62) + (0.62 - 0.62) => 0.06 + 0.02 + 0 => |0.08| => 0.08$



0.44	0.61	0.62

Figure 3.5.10: Subset 8 of 10

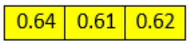
$s(7) = (0.44 - 0.62) + (0.61 - 0.62) + (0.62 - 0.62) => -0.18 - 0.01 + 0 => |-0.19| => 0.19$



0.68	0.61	0.62

Figure 3.5.11: Subset 9 of 10

$s(8) = (0.68 - 0.62) + (0.61 - 0.62) + (0.62 - 0.62) => 0.06 - 0.01 + 0 => |0.05| => 0.05$

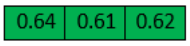


0.64	0.61	0.62

Figure 3.5.12: Subset 10 of 10

$s(9) = (0.64 - 0.62) + (0.61 - 0.62) + (0.62 - 0.62) => 0.02 - 0.01 + 0 => |0.01| => 0.01$

With reference to equation 2.10 If we sort the above in ascending order, we have $s(9) < s(8) < s(3) < s(6) < s(0) < s(4) < s(4) < s(11) < s(5) < s(2) < s(7)$, therefore our minimum is $s(9)$ and the gradients are as depicted in 3.5.13.



0.64	0.61	0.62

Figure 3.5.13: Generated subsets from θ selected gradients

With reference to equation 2.11, we will now compute the average of all the selected gradients and use them to share it with other workers during the All-share phase.

3.6 All-Share

During All-share phase each worker will share the final computed gradient with other workers as shown in 3.6.1.

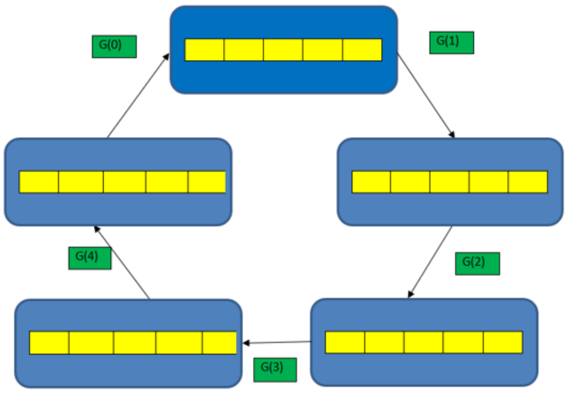


Figure 3.6.1: All-share Phase

At the end of All-share phase, each worker will hold a copy of final gradients as depicted in 3.6.2

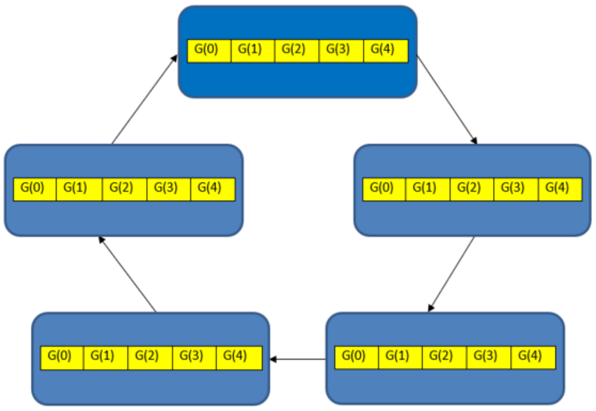


Figure 3.6.2: Workers with non-byzantine byzantine gradients

3.7 Training MLP with MNIST

In this section we will discuss further on Distributed model training service. This service receives the input from Byzantine resilience service i.e. gradients that are non-byzantine (mostly) and outputs new set of gradients. It hosts our deep learning model, which is a multi layer network provided by deeplearning4j that has SGD as the optimisation algorithm and contains one layer that accepts inputs of size 28*28 and outputs 10 classes. This makes a total of 7840 weights and 10 biases that will be generated by the model per iteration. In our above example we used a gradient vector of length five for ease of understanding. In real time, we share a total of 7850 gradients among the workers. Each worker in topology will build a model with the same structure as mentioned earlier. The MNIST data-set, which is being used to train the model has 60,000 images each of it being a 28*28 pixel grey scale image. Here, it is important to note that the data-set is split into mini batches of size 128 and each worker will only train on specific number of mini batches and not on entire data-set, which is because we want to achieve Data parallelism. We also use a test data-set of 10,000 samples from MNIST to measure accuracy of the model. To inject byzantine gradients, we use a *random* function provided by deeplearning4j to replace the model's gradients with these custom generated and update the weights and biases of it. We share these gradients with other worker during Share-reduce phase instead of actual generated gradients. In the next section, we will discuss further on how these injected byzantine gradients are used to formulate an attack.

3.8 Attack on GAR's

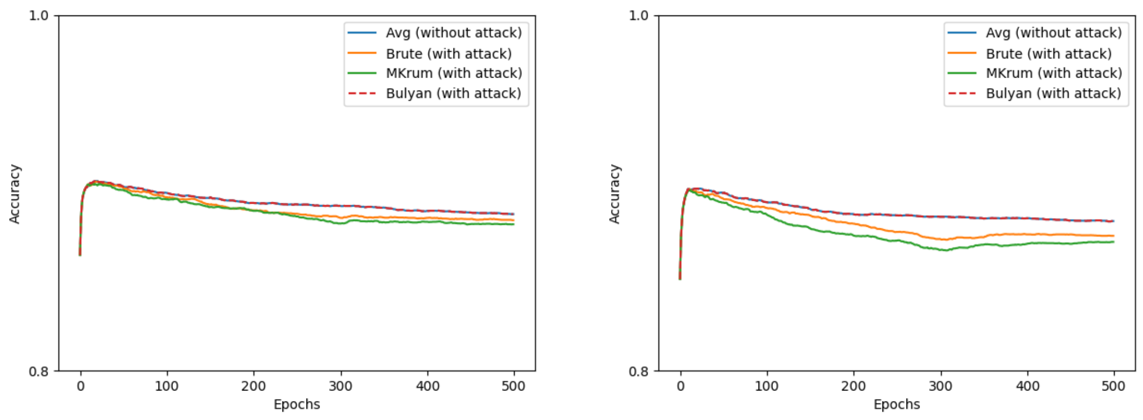
With reference to [41] [10] we formulate an attack and observe the effect it has on M-Krum, Brute and Bulyan in Ring all-reduce architecture. The attack is simple, when the GAR receives a set of n gradients among which f are byzantine, it picks f gradients which are assumed to be byzantine and use these set of gradients for further computation according to the algorithm.

Let us now observe the effect of such attack on GAR's starting with M-Krum Brute and Bulyan. M-Krum is a distance based GAR that suffers from the curse of dimensionality and is known to offer weak byzantine resilience. Since it is an l_p norm based GAR that filters out byzantine gradients based on distance minimization scheme, it does allow considerable margin to accommodate byzantine gradients as gradients from correct workers. Such GAR can only contribute to prevent convergence of a model to an acceptable level. Brute on the other hand also filters out byzantine gradients based on distance minimization scheme, which made it to provide weaker resilience to byzantine gradients just like M-Krum. We take inspiration of our attack from this drawback to prove stronger resilience of Bulyan. When attacked M-Krum and Brute, they pick f gradients instead of $n - f$, meaning, they accommodate byzantine gradients as real gradients when under attack. We also attack Bulyan and ensure that the weak byzantine resilience algorithm picks always f gradients at the end of phase one. These gradients are then given as an input for phase two execution of Bulyan. The median based approach in phase two of Bulyan reduces the probability of picking the byzantine gradients chosen in phase one as much as by $\mathcal{O}(1/\sqrt{d})$ [10]. This phase two median approach ensures as an extra protection from byzantine gradients missed by the weak byzantine resilience GAR and ensures Bulyan to provide strong byzantine resilience.

Chapter 4

Result

In this section, we will be discussing the results obtained in this thesis implementation. We observe the resilience of Ring all-reduce to byzantine gradients in the presence of M-Krum, Brute and Bulyan GAR's by injecting the adversary mentioned in chapter 3.8. In our experiments, when training the deeplearning model with MNIST data-set, we attack until 300 epochs to see which of the aforementioned GAR's offered better resilience to byzantine gradients by extrapolating a graph with X axis representing number of epochs and Y axis representing the accuracy. The learning rate of the model in this case is 0.5 which is considerably high. The reason for choosing such high learning rate is because it can significantly increases the effectiveness of the attack and vice versa.



(a) Resilience offered by Ring all-reduce with two byzantine's (f) among 11 workers (n)

(b) Resilience offered by Ring all-reduce with three byzantine's (f) among 15 workers (n)

Figure 4.0.1: Accuracy obtained on testing until 500 epochs using MNIST data-set

From figures 4.0.1a and 4.0.1b, we can clearly observe that the Bulyan GAR showed significant resistance to the attack when compared to M-Krum and Brute. This also shows that the byzantine gradients missed by the weak byzantine resilience GAR during phase one of Bulyan are successfully filtered out during the median based phase two. Phase two of Bulyan is capable of obtaining a subset of gradients that are closest to the median and not on specific coordinates like M-Krum or Brute. On the other hand, M-Krum suffered more than Brute when attacked. This is because when the attack lets GAR's to choose f among the total submitted gradients, M-Krum averages with m i.e. in our case $n - f - 2$; the weight added by correct gradients is lower compared to Brute which averages with $n - f$. Therefore, with increase in number of byzantine gradients M-Krum suffered to the attack more than Brute. Overall, it can be concluded that Bulyan offers strong byzantine resilience in ring all-reduce approach with good model convergence compared to M-Krum and Brute. Since Parameter server server approach is known to perform poorly with linear increase in number of workers, ring all-reduce can be adopted in such a case which is capable of achieving similar performance in the presence of GAR's which can be observed from the above result.

Chapter 5

Conclusion

In this section we provide a brief summary of the results and potential future works that can extend the current implementation.

5.1 Summary

So far we have discussed M-Krum, Brute and Bulyan GAR's, and Ring all-reduce architecture in depth. First we generate the required ring topology with workers and build necessary components and services using Kompics. We use one of the service to train the deeplearning model using MNIST data-set and another to run a GAR that provides byzantine resilience during the training process. We have also provided an example for each GAR used in this implementation to have a clear understanding of their capabilities to filter out byzantine gradients. Post setup, we have conducted experiments using the aforementioned with varying worker counts and injecting adversaries. Overall, Bulyan performed well when compared to M-Krum and Brute, and showed better resilience to byzantine gradients under adversary in Ring all-reduce architecture. The adversary enables workers to choose byzantine gradients until certain number of epochs to observe which of the aforementioned GAR's provide better resilience. This approach indeed proves that Bulyan is capable of achieving performance similar to the Average GAR not only in Parameter server approach as mentioned in [10] but also in All-reduce architecture.

5.2 Future Work

Below are some ideas for possible future implementations:

Analyzing M-Krum, Brute and Bulyan GAR's performance in Gossip architecture: considering Gossip architecture's capabilities such as less communication frequency and traffic, it will be interesting to draw a comparison between the training times overall between All-reduce and Gossip architecture's. Slow to moderate model convergence can also be taken into consideration in expansion to the above.

Observing the effects of byzantine gradients on GAR's with larger worker counts: current experiments are conducted with the number of workers being less than 30. This is due to physical machine limitations. Considering Bulyan's requirement of having high worker counts to tolerate moderate to low byzantine workers, the experiments are conducted with a smaller number of byzantine workers, say, for 30 workers we can only have six byzantines. To ensure clear understanding on performance of GAR's, all other experiments were conducted with the same worker and byzantine count. Conducting experiments on larger counts may assist with better understanding on effects of byzantine gradients on GAR's.

More experiments using All-reduce variants: current experiments are conducted using Ring all-reduce. As discussed earlier we have variants of All-reduce, which might assist with reducing training times and it will be interesting to see if there is a trade off between model convergence and training times.

Analyzing Multi-Bulyan in both All-reduce and Gossip architectures: Multi-Bulyan is a GAR that has been proposed at the time of this implementation and has been analyzed for byzantine resilience in Parameter server architecture. Further analysis on it can be implemented using both All-reduce and Gossip architectures.

Experiments using other data-sets and larger models: In our current implementation we used MNIST data-set for model training and GAR experimentation purposes. It will be interesting to see if the aforementioned GAR's can achieve performance similar to MNIST when using CIFAR and other data-sets. Also, as mentioned in the limitations section, the Java thread limitation of 65535 bytes has caused us to build shorter models by means of the number of free parameters. It will be greatly beneficial to compare if the GAR's, especially distance based functions perform the same with reference to our result.

Bibliography

[1] You, Yang, Li, Jing, Reddi, Sashank, Hseu, Jonathan, Kumar, Sanjiv, Bhojanapalli, Srinadh, Song, Xiaodan, Demmel, James, Keutzer, Kurt, and Hsieh, Cho-Jui. *Large Batch Optimization for Deep Learning: Training BERT in 76 minutes*. 2020. arXiv: 1904.00962 [cs.LG].

[2] Sun, Chen, Shrivastava, Abhinav, Singh, Saurabh, and Gupta, Abhinav. *Revisiting Unreasonable Effectiveness of Data in Deep Learning Era*. 2017. arXiv: 1707.02968 [cs.CV].

[3] Tang, Zhenheng, Shi, Shaohuai, Chu, Xiaowen, Wang, Wei, and Li, Bo. *Communication-Efficient Distributed Deep Learning: A Comprehensive Survey*. 2020. arXiv: 2003.06307 [cs.DC].

[4] Bottou, Léon, Curtis, Frank E., and Nocedal, Jorge. *Optimization Methods for Large-Scale Machine Learning*. 2018. arXiv: 1606.04838 [stat.ML].

[5] Li, Mu, Andersen, David G., Park, Jun Woo, Smola, Alexander J., Ahmed, Amr, Josifovski, Vanja, Long, James, Shekita, Eugene J., and Su, Bor-Yiing. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[6] Chilimbi, Trishul, Suzue, Yutaka, Apacible, Johnson, and Kalyanaraman, Karthik. "Project Adam: Building an Efficient and Scalable Deep Learning Training System". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 571–582. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

[7] Kendall, Wes. *MPI Reduce and Allreduce*. 2014. URL: `https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/` (visited on 06/01/2021).

[8] Yates, Roy D. *The Age of Gossip in Networks*. 2021. arXiv: `2102.02893 [cs.IT]`.

[9] Lamport, Leslie, Shostak, Robert, and Pease, Marshall. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: `10.1145/357172.357176`. URL: `https://doi.org/10.1145/357172.357176`.

[10] Mhamdi, El Mahdi El, Guerraoui, Rachid, and Rouault, Sébastien. *The Hidden Vulnerability of Distributed Learning in Byzantium*. 2018. arXiv: `1802.07927 [stat.ML]`.

[11] Blanchard, Peva, El Mhamdi, El Mahdi, Guerraoui, Rachid, and Stainer, Julien. "Machine Learning with Adversaries: Byzantine Tolerant Gradient Descent". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 118–128. ISBN: 9781510860964.

[12] Chen, Lingjiao, Wang, Hongyi, Charles, Zachary, and Papailiopoulos, Dimitris. *DRACO: Byzantine-resilient Distributed Training via Redundant Gradients*. 2018. arXiv: `1803.09877 [stat.ML]`.

[13] El-Mhamdi, El-Mahdi, Guerraoui, Rachid, and Rouault, Sébastien. "Fast and Robust Distributed Learning in High Dimension". In: *2020 International Symposium on Reliable Distributed Systems (SRDS)*. 2020, pp. 71–80. DOI: `10.1109/SRDS51746.2020.00015`.

[14] Mhamdi, El Mahdi El, Guerraoui, R., Guirguis, A., and Rouault, Sébastien. "SGD: Decentralized Byzantine Resilience". In: *ArXiv* abs/1905.03853 (2019).

[15] Mhamdi, El Mahdi El, Guerraoui, Rachid, and Guirguis, Arsany. *Fast Machine Learning with Byzantine Workers and Servers*. 2020. arXiv: `1911.07537 [cs.LG]`.

[16] Arad, Cosmin, Dowling, Jim, and Haridi, Seif. "Developing, Simulating, and Deploying Peer-to-Peer Systems Using the Kompics Component Model". In: *Proceedings of the Fourth International ICST Conference on COMmunication System SoftWAre and MiddlewaRE*. COMSWARE '09. Dublin, Ireland: Association for Computing Machinery, 2009. ISBN: 9781605583532. DOI: `10.1145/1621890.1621911`. URL: `https://doi.org/10.1145/1621890.1621911`.

[17] Peng, Zhao. *Multilayer Perceptron Algebra*. 2017. arXiv: 1701 . 04968 [stat.ML].

[18] LeCun, Yann and Cortes, Corinna. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[19] Team, Eclipse Deeplearning4j Development. *Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0*. 2016. URL: https://deeplearning4j.org/ (visited on 06/02/2021).

[20] Odersky, Martin, Spoon, Lex, and Venners, Bill. *Programming in Scala: Updated for Scala 2.12*. 3rd. Sunnyvale, CA, USA: Artima Incorporation, 2016. ISBN: 0981531687.

[21] Kroll, Lars, Carbone, Paris, and Haridi, Seif. "Kompics Scala: Narrowing the Gap between Algorithmic Specification and Executable Code (Short Paper)". In: *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. SCALA 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 73–77. ISBN: 9781450355292. DOI: 10.1145/3136000.3136009. URL: https://doi.org/10.1145/3136000.3136009.

[22] Shallue, Christopher J., Lee, Jaehoon, Antognini, Joseph, Sohl-Dickstein, Jascha, Frostig, Roy, and Dahl, George E. *Measuring the Effects of Data Parallelism on Neural Network Training*. 2019. arXiv: 1811.03600 [cs.LG].

[23] Brownlee, Jason. *Crash Course On Multi-Layer Perceptron Neural Networks*. 2016. URL: https://machinelearningmastery.com/neural-networks-crash-course/ (visited on 06/06/2021).

[24] Minsky, Marvin and Papert, Seymour. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[25] Ahire, Jayesh Bapu. *Demystifying the XOR problem*. 2020. URL: https://dev.to/jbahire/demystifying-the-xor-problem-1blk (visited on 06/06/2021).

[26] Sharma, Anuraganand. "Guided parallelized stochastic gradient descent for delay compensation". In: *Applied Soft Computing* 102 (Apr. 2021), p. 107084. ISSN: 1568-4946. DOI: 10.1016/j.asoc.2021.107084. URL: http://dx.doi.org/10.1016/j.asoc.2021.107084.

[27] Ruder, Sebastian. *An overview of gradient descent optimization algorithms*. 2017. arXiv: 1609.04747 [cs.LG].

[28] Alqahtani, Salem and Demirbas, Murat. *Performance Analysis and Comparison of Distributed Machine Learning Systems*. 2019. arXiv: `1909.02061 [cs.DC]`.

[29] Garcia, Edir. *Visual intuition on ring-Allreduce for distributed Deep Learning*. 2017. URL: `https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da` (visited on 05/30/2021).

[30] Zhao, Huasha and Canny, John. *Sparse Allreduce: Efficient Scalable Communication for Power-Law Data*. 2013. arXiv: `1312.3020 [cs.DC]`.

[31] Gibiansky, Andrew. *Bringing HPC techniques to deep learning*. 2017. URL: `https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/` (visited on 05/29/2021).

[32] Baruch, Moran, Baruch, Gilad, and Goldberg, Yoav. *A Little Is Enough: Circumventing Defenses For Distributed Learning*. 2019. arXiv: `1902.06156 [cs.LG]`.

[33] Muñoz-González, Luis, Co, Kenneth T., and Lupu, Emil C. *Byzantine-Robust Federated Machine Learning through Adaptive Model Averaging*. 2019. arXiv: `1909.05125 [stat.ML]`.

[34] Rajput, Shashank, Wang, Hongyi, Charles, Zachary, and Papailiopoulos, Dimitris. *DETOX: A Redundancy-based Framework for Faster and More Robust Gradient Aggregation*. 2020. arXiv: `1907.12205 [cs.LG]`.

[35] Yang, Zhixiong, Gang, Arpita, and Bajwa, Waheed U. "Adversary-Resilient Distributed and Decentralized Statistical Inference and Machine Learning: An Overview of Recent Advances Under the Byzantine Threat Model". In: *IEEE Signal Processing Magazine* 37.3 (May 2020), pp. 146–159. ISSN: 1558-0792. DOI: `10.1109/msp.2020.2973345`. URL: `http://dx.doi.org/10.1109/MSP.2020.2973345`.

[36] Xie, Cong, Koyejo, Oluwasanmi, and Gupta, Indranil. *Phocas: dimensional Byzantine-resilient stochastic gradient descent*. 2018. arXiv: `1805.09682 [cs.DC]`.

[37] Team, Great Learning. *Understanding Curse of Dimensionality*. 2020. URL: `https://www.mygreatlearning.com/blog/understanding-curse-of-dimensionality/` (visited on 05/31/2021).

[38] Yang, Zhixiong and Bajwa, Waheed U. *BRIDGE: Byzantine-resilient Decentralized Gradient Descent*. 2019. arXiv: `1908.08098 [stat.ML]`.

[39] Prakash, Saurav and Avestimehr, Salman. *Mitigating Byzantine Attacks in Federated Learning*. 2021. arXiv: `2010.07541 [cs.DC]`.

[40] Zizzo, Giulio, Rawat, Ambrish, Sinn, Mathieu, and Buesser, Beat. *FAT: Federated Adversarial Training*. 2020. arXiv: `2012.01791 [cs.LG]`.

[41] Lyu, Lingjuan, Yu, Han, Ma, Xingjun, Sun, Lichao, Zhao, Jun, Yang, Qiang, and Yu, Philip S. *Privacy and Robustness in Federated Learning: Attacks and Defenses*. 2020. arXiv: `2012.06337 [cs.CR]`.

Appendix - Contents

Appendix A

Ring All-Reduce Snippet

A.1 Topology Generator

```
import com.larskroll.common.collections._;
import java.util.Collection;
import se.kth.rise.networking.NetAddress;

@SerialVersionUID(6322485231428233902L)
// Node
class Node(self: NetAddress, pred: NetAddress, succ: NetAddress, index: Int,
    succIndex : Int) extends Serializable
{
    def get_current_address(): NetAddress = {
      return self
    }
    def get_pred_address(): NetAddress = {
      return pred
    }
    def get_succ_address(): NetAddress = {
      return succ
    }
    def get_index(): Int = {
      return index
    }
    def get_succ_index(): Int = {
      return succIndex
    }
    override def toString(): String =
    {
        val sb = new StringBuilder();
        sb.append("Predecessor: " + pred);
        sb.append("Current node: " + self);
        sb.append("Successor:" , succ);
        sb.append("My index in ring: ", index)
        sb.append("My succesor index in ring: ", succIndex)
        return sb.toString();
    }
}

object RingTopology {
    var successorN : NetAddress = _ ;
    var predecessorN : NetAddress =_ ;
    var currentN : NetAddress =_ ;
    var succI : Int = 0;

  def generate(nodes: Set[NetAddress]): scala.collection.immutable.Set[Node]
      = {
    val sortedAddress = nodes.toList sortBy (_.getPort())
    val leng: Int = sortedAddress.length - 1
    var allNodes = scala.collection.immutable.Set[Node]()

    sortedAddress.zipWithIndex.foreach{ case (item, index) =>
    currentN = sortedAddress(index);
    index match {
      case index if index == 0 => {
      succI = index + 1;
      successorN = sortedAddress(index + 1);
      predecessorN = sortedAddress(leng);
      }
      case index if index == leng => {
      succI = 0;
      successorN = sortedAddress(0);
      predecessorN = sortedAddress(index - 1);
      }
      case _ => {
      succI = index + 1;
      successorN = sortedAddress(index + 1);
      predecessorN = sortedAddress(index - 1);
      }
    }
    allNodes += new Node(currentN, predecessorN, successorN, index, succI);
  }
  // All nodes with predecessors and successors assigned
  allNodes
  }
}
```

Listing A.1.1: Topology generator component generating ring topology for all connected nodes

A.2 Gradient Sharing

```
  boot uponEvent {
    // Receive all assignments from Topology generator component
    case InitialAssignments(assignment) => {
      // Set BootstrapServer's predecessor and successor
      assignment foreach { node =>
          if(self == node.get_current_address()){
            currentNI = node.get_index()
            succNI = node.get_succ_index()
            predecessorN = node.get_pred_address()
            successorN = node.get_succ_address()
          }
      }
      // Send the generated topology to all workers
      active foreach { node =>
        trigger(NetMessage(self, node, Boot(assignment)) -> net);
      }
      // BootstrapServer open to receive gradients
      ready += self;
      // Create a model and generate gradinets
      generateGradients(1, bootThreshold, finalGradients);
      var converter = transporter(currentNI).map(Array(_))
      // Trigger Share-Reduce phase
      trigger(NetMessage(self, successorN, Msg(converter, currentNI)) -> net);
    }
  }
```

Listing A.2.1: BootstrapServer sending ring topology and gradients to all other workers

A.3 Handler

```
net uponEvent {
    case NetMessage(header, Msg(incGradient, index)) => {
      var preProcessor = transporter(index).map(Array(_))
      // Process and add the received gradinets from workers
      receivedGradients = allVals(incGradient, index, preProcessor);
      index match {
      // Trigger Share - Reduce
      case index if index != succNI => trigger(NetMessage(self, successorN,
          Msg(receivedGradients(index), index)) -> net);
      //  Trigger All share phase
      case _ =>
      finalGradients += (index -> ListBuffer());
      receivedGradients(succNI) foreach { eachList =>
        // Byzantine resilience GAR
        avg = MultiKrum.MultiKrumInit(eachList.toList, closestVectors,
            bruteAvg, epochCount);
        // Update final gradients of an index
        finalGradients.update(index, finalGradients(index) :++
            ListBuffer(Array(avg)));
      }
      println("Computed final gradient " + finalGradients(index) + " for index
          " + index);
      // Share with all other workers
      trigger(NetMessage(self, successorN, SharePhase(finalGradients(index),
          index)) -> net);
      }
    }
}
```

Listing A.3.1: Handler in workers sending and receiving events (gradients)

A.4 Handler Triggering All-Share Phase

```
net uponEvent {
    case NetMessage(header, SharePhase(incGradient, index)) => {
      index match {
      // Update received gradinets and forward the index to next worker
      case index if index != succNI => println("Final gradient for index ",
          index, incGradient);
      finalGradients += (index -> ListBuffer());
      finalGradients.update(index, incGradient);
      trigger(NetMessage(self, successorN, SharePhase(incGradient, index)) ->
        net);
      // Trigger model training once Share-reduce is completed
      case index if index == succNI =>
      // Begin model training with received gradients
      // Train the model until 500 epochs
      if(epochCount <= epochs){
        // Final gradients given as an input to model
        var trainedGrads = generateGradients(2, bootThreshold, finalGradients);
        // Reset variables
        receivedGradients = scala.collection.mutable.Map()
        finalGradients = scala.collection.mutable.Map()
        processGradients = scala.collection.mutable.Map()
        // Increment epochs
        epochCount += 1;
        var preProcessor = trainedGrads(currentNI).map(Array(_))
        // Trigger Share-reduce again
        trigger(NetMessage(self, successorN, Msg(preProcessor, currentNI)) ->
          net);
      }
      case _ => // Do Nothing
      }
    }
}
```

Listing A.4.1: Handler receiving gradients in all-share phase and triggering model training

A.5 Pre Processor

```
def allVals(incGradient: ListBuffer[Array[Float]], index : Int, currGradient:
    ListBuffer[Array[Float]]):
    scala.collection.mutable.Map[Int,ListBuffer[Array[Float]]] = {
    processGradients += (index -> ListBuffer())
    incGradient.zipWithIndex.foreach{ case(x,i) =>
      processGradients.update(index, processGradients(index) :++
        ListBuffer(currGradient(i) ++ x))
    }
    processGradients
}
```

Listing A.5.1: Pre-processor including gradients received from workers

A.6 Model generation

```
def generateGradients(incPhase : Int, threshold: Int, sharedGrads:
    scala.collection.mutable.Map[Int,ListBuffer[Array[Float]]]):
    ListBuffer[ListBuffer[Float]] = {
      // Initiate model and generate gradinets
      if(incPhase == 1) {
        MLPMnist.modelInit(currentNI);
        gradient = MLPMnist.triggerInitialGradinets(currentNI);
        transporter = round(gradient.toList, threshold)
      }
    // Begin training with the updated gradinets
    if(incPhase == 2) {
        val sortProcess =
            scala.collection.mutable.Map(sharedGrads.toSeq.sortBy(_._1):_*)
        val buffer = sortProcess.map{case(i, x) => x};
        gradient = MLPMnist.triggerTraining(buffer.flatten.flatten.toArray,
            epochCount, currentNI, epochs);
        transporter = round(gradient.toList, threshold)
    }
    transporter
}
```

Listing A.6.1: Workers generating model and beginning training

Appendix B

Byzantine Resilient GAR's Snippet

B.1 M-Krum

```
import Ordering.Float.IeeeOrdering;
import scala.collection.mutable.ListBuffer;

object MultiKrum {
  // Search index to set searchable points
  def findCrossOver(arr: List[Float], num: Float, start: Int, end: Int): Int
      = {
    if (arr(start) <= num) {
      return start
    }
    if (arr(end) > num) {
      return end
    }
    val center: Int = (end + start) / 2
    if (arr(center) <= num && arr(center + 1) > num) {
      return center
    }
    if (arr(center) < num) {
      return findCrossOver(arr, num, center + 1, start)
    }
    return findCrossOver(arr, num, end, center - 1)
  }
  // Obtain n-f-2 closest vectors
  def closestVectorsToEach(arr: List[Float], eachNum: Float, k: Int, len:
      Int): Float = {
    var count: Int = 0
    var summation: Float = 0.0f
    var left: Int = findCrossOver(arr, eachNum, 0, len - 1)
    var right: Int = left + 1
    if (arr(left) == eachNum) {
      left -= 1; left + 1
    }
    while (left >= 0 && right < len && count < k) {
      if (eachNum - arr(left) < arr(right) - eachNum){
        var num = arr({ left -= 1; left + 1 })
        summation += MKrum(num, eachNum)
      }
      else{
        var num = arr({ right += 1; right - 1 })
        summation += MKrum(num, eachNum)
      }
      count += 1; count - 1
    }
    while (count < k && left >= 0) {
      var num = arr({ left -= 1; left + 1 })
      summation += MKrum(num, eachNum)
      count += 1; count - 1
    }
    while (count < k && right < len) {
      var num = arr({ right += 1; right - 1 });
      summation += MKrum(num, eachNum)
      count += 1; count - 1
    }
    summation
  }
  // Begin M-Krum
  def MultiKrumInit(arr: List[Float], closestVectors: Int, mKrumAvg: Int,
      epochC: Int): Float = {
    var length = arr.length;
    var sortedList = arr.sorted
    var squared = scala.collection.mutable.ListBuffer.empty[Float]
    sortedList foreach { each =>
    var eachGen = closestVectorsToEach(sortedList, each, closestVectors,
        length)
    squared += eachGen
    }
    MKrumAvg(squared, sortedList, mKrumAvg, epochC)
  }
  // Compute squared distance
  def MKrum(inp : Float, grad: Float): Float = {
    var sqrdDist: Float = 0;
    sqrdDist = Math.abs(Math.pow((inp - grad),2).toFloat);
    sqrdDist
  }
  // Compute average with m = n-f-2
  def MKrumAvg(lb: ListBuffer[Float], sl: List[Float], mKrumAvg: Int, epochC:
      Int): Float = {
    val clb = lb.toList;
    val map = clb.zipWithIndex.map{ case (v,i) => (i,v) }.toMap
    var count: Int = 0
    var sum: Float = 0.0f;
    var avg: Float = 0.0f;
    // Attack until 300 epochs picking f gradients
    if(epochC <= 300){
      val smap = map.toList.sortBy(_._2).reverse
      while(count < mKrumAvg){
      val (key, vals) = smap(count)
      sum += sl(key)
      count += 1;
      }
    }
    if(epochC > 300){
      val smap = map.toList.sortBy(_._2)
      while(count < mKrumAvg){
      val (key, vals) = smap(count)
      sum += sl(key)
      count += 1;
      }
    }
    avg = sum/mKrumAvg;
    println(avg)
    avg
  }
}
```

Listing B.1.1: M-Krum GAR

B.2 Brute

```
import Ordering.Float.IeeeOrdering;
import scala.collection.mutable.ListBuffer;


object Brute {
  def BruteInit(input: List[Float], subSets: Int, bruteAvg: Int, epochCount:
      Int): Float = {
    var allCombinations = input.combinations(subSets).toList;
    var maxVal: Float = 0.0f;
    var maxed : List[(Float, List[Float])] = List()
    // Subsets of length n-f
    allCombinations foreach { each =>
    var squaredList = each.map(x => x*x);
    squaredList.zipWithIndex.foreach{ case(x,i) =>
        squaredList.zipWithIndex.foreach{ case(y,j) =>
            if(i!=j){
                var diff = Math.abs(x-y)
                if(diff > maxVal){
                    maxVal = diff;
                }
            }
        }
    }
        // Maximum value in a subset
        maxed = maxed:+((maxVal, each));
        maxVal = 0.0f;
    }
    var sorted : List[(Float, List[Float])] = List()
    // Attack until 300 epochs picking f gradients
    if(epochCount <= 300) {
    sorted = maxed.sortBy(r => (r._1)).reverse
    }
    // No attack
    if(epochCount > 300) {
    sorted = maxed.sortBy(r => (r._1))
    }
    // Minimum of maximum
    var finlGrads = sorted(0)._2
    var summed = finlGrads.reduceLeft(_ + _) / bruteAvg;
    summed
  }
}
```

Listing B.2.1: Brute GAR

B.3 Bulyan

```
import Ordering.Float.IeeeOrdering;
import scala.collection.mutable.ListBuffer;

object Bulyan {
  def closestNumber(inputLst: List[Float], target: Float): Float = {
    var index: Int = 0
    var dist: Float = Math.abs(inputLst(0) - target)
    for (i <- 1 until inputLst.length) {
      val closestDist: Float = Math.abs(inputLst(i) - target)
      if (closestDist < dist) {
        index = i
        dist = closestDist
      }
    }
    inputLst(index)
  }
  def BulyanInit(input: List[Float], slectionSetLen: Int, byzantineCount:
      Int, epochC: Int): Float = {
    println("Bulyan Initiated")
    var chosen: Float = 0.0f;
    var receivedSet: ListBuffer[Float] = ListBuffer();
    receivedSet ++= input
    var selectionSet: ListBuffer[Float] = ListBuffer();
    var maxVal: Float = 0.0f;
    var maxed : List[(Float, Float)] = List();
    var minBulVal: Float = 0.0f;
    var minedBul : List[(Float, List[Float])] = List();
    // Step 1
    // theta = n - 2f
    while(selectionSet.length < slectionSetLen){
        println("ReceivedSet "+ receivedSet)
        chosen = bruteInit(receivedSet.toList, receivedSet.length -
            byzantineCount, receivedSet.length - byzantineCount, epochC);
        var closestVal = closestNumber(receivedSet.toList, chosen);
        receivedSet -= closestVal;
        selectionSet += closestVal;
        println("Selection set ", selectionSet)
    }
    // Step 2
    // Median calculation
    selectionSet.zipWithIndex.foreach{ case(x,i) =>
        selectionSet.zipWithIndex.foreach{ case(y,j) =>
            if(i!=j){
                var diff = Math.abs(x-y);
                maxVal = maxVal + diff;
            }
        }
        maxed = maxed:+ ((x, maxVal));
        println("All median values ",maxed)
        maxVal = 0.0f;
    }
    // beta = theta - 2f
    var allCombinations = selectionSet.toList.combinations(3).toList;
    var incSort = maxed.sortBy(r => (r._2))
    var medianGrad = incSort(0)._1
    println("Min of Max ", medianGrad);
    // Subset verification
    allCombinations foreach { each =>
```

```
      each.zipWithIndex.foreach{ case(y,j) =>
             var diff = Math.abs(y - medianGrad);
             minBulVal = minBulVal + diff;
      }
    minedBul = minedBul :+ ((minBulVal, each));
    minBulVal = 0.0f;
  }
  println("Min bulyan vals ", minedBul)
  var sorted : List[(Float, List[Float])] = List()
  sorted = minedBul.sortBy(r => (r._1))
  var finlGrads = sorted(0)._2
  println("Min of Max ", finlGrads);
  var summed = finlGrads.reduceLeft(_ + _) / 3;
  summed
 }
 // Brute GAR
 def bruteInit(input: List[Float], closestVectors: Int, bruteAvg: Int,
     epochC: Int): Float = {
  println("Brute Initiated")
  var allCombinations = input.combinations(closestVectors).toList;
  var maxVal: Float = 0.0f;
  var maxed : List[(Float, List[Float])] = List()
  allCombinations foreach { each =>
  var squaredList = each.map(x => x*x);
  squaredList.zipWithIndex.foreach{ case(x,i) =>
      squaredList.zipWithIndex.foreach{ case(y,j) =>
          if(i!=j){
              var diff = Math.abs(x-y)
              if(diff > maxVal){
                  maxVal = diff;
              }
          }
      }
  }
      maxed = maxed:+((maxVal, each));
      maxVal = 0.0f;
  }
  var sorted : List[(Float, List[Float])] = List()
  // Attack until 300 epochs picking f gradients
  if(epochC <= 300) {
  sorted = maxed.sortBy(r => (r._1)).reverse
  println("Sorted descending ",sorted);
  }
  if(epochC > 300) {
  sorted = maxed.sortBy(r => (r._1))
  println("Sorted acsending ",sorted);
  }
  var finalGrads = sorted(0)._2
  println("Min of Max ", finalGrads);
  var summed = finalGrads.reduceLeft(_ + _) / bruteAvg;
  summed
 }
}
```

Listing B.3.1: Bulyan GAR

Appendix C

MLP Snippet

C.1 Model

```
val conf = new NeuralNetConfiguration.Builder()
      .seed(rngSeed)
      .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
      .learningRate(0.5)
      .list
      .layer(0, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD) //
         0_W (weights - 784 * 10) // 0_b (bias - 10)
        .nIn(numRows * numColumns)
        .nOut(10)
        .activation(Activation.SOFTMAX)
        .weightInit(WeightInit.XAVIER)
        .build)
      .pretrain(false).backprop(true)
      .build
    var model = new MultiLayerNetwork(conf)
    model.init()
    model.setListeners(new ScoreIterationListener(5))
```

Listing C.1.1: MLP implemented using deeplearning4j and Scala

C.2 Byzantine Gradients

```
// Random gradients for weights
var weightShape = Array(784, 10)
var b_1d = Nd4j.rand(weightShape, Nd4j.getDistributions().createUniform(-1,
    999))
// Random gradients for biases
var biasShape = Array(10)
var b_2d = Nd4j.rand(biasShape, Nd4j.getDistributions().createUniform(-100,
    900))
// Update the model with the above generated byzantine gradients
model.gradient().setGradientFor("0_W", b_1d)
model.gradient().setGradientFor("0_b", b_2d)
// Update weights
model.update(model.gradient())
```

Listing C.2.1: Random byzantine gradients being injected to the above discussed model