12-2022

Divide-and-Conquer Distributed Learning: Privacy-Preserving Offloading of Neural Network Computations

Lewis C.L. Brown
University of Arkansas, Fayetteville

Divide-and-Conquer Distributed Learning:
Privacy-Preserving Offloading of Neural Network Computations

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science

by

Lewis Brown
University of Arkansas
Bachelor of Science in Computer Science, 2019

December 2022
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

Qinghua Li, Ph.D.
Thesis Chair

Brajendra Panda, Ph.D.
Committee Member

Lu Zhang, Ph.D.
Committee Member

ABSTRACT

Machine learning has become a highly utilized technology to perform decision making on high dimensional data. As dataset sizes have become increasingly large so too have the neural networks to learn the complex patterns hidden within. This expansion has continued to the degree that it may be infeasible to train a model from a singular device due to computational or memory limitations of underlying hardware. Purpose built computing clusters for training large models are commonplace while access to networks of heterogeneous devices is still typically more accessible. In addition, with the rise of 5G networks, computation at the edge becoming more commonplace, and inspired by the successes of the folding@home project utilizing crowdsourced computation, we consider the scenario of the crowdsourcing the computation required for training of a neural network particularly appealing. Distributed learning promises to bridge the widening gap between singular device performance and large-scale model computational requirements, but unfortunately, current distributed learning techniques do not maintain privacy of both the model and input without an accuracy or computational tradeoff. In response, we present Divide and Conquer Learning (DCL), an innovative approach that enables quantifiable privacy guarantees while offloading the computational burden of training to a network of devices. A user can divide the training computation of its neural network into neuron-sized computation tasks and distribute them to devices based on their available resources. The results will be returned to the user and aggregated in an iterative process to obtain the final neural network model. To protect the privacy of the user's data and model, shuffling is done to both the data and the neural network model before the computation task is distributed to devices. Our strict adherence to the order of operations allows a user to verify the correctness of performed computations through assigning a task to multiple devices and cross-validating their results. This can protect against network churns and detect faulty or misbehaving devices.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Qinghua Li, for his invaluable input and guidance during the thesis process. My ability to pursue my graduate career was made possible through his support.

I'd also like to express my immense gratitude for my committee members, Dr. Brajendra Panda and Dr. Lu Zhang, for the time and knowledge they contributed to ensuring the quality of this thesis.

This work is supported in part by the National Science Foundation under award 1946391.

DEDICATION

To the shoulders of giants we all stand upon so that we may reach higher. I am able to produce this thesis because of the efforts of many. My family, my friends, my peers, my fellow human beings.

EPIGRAPH

Our need will be the real creator.

—*Plato*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PUBLICATIONS

Lewis Brown and Qinghua Li, "Privacy-Preserving and Secure Divide-and-Conquer Learning", *IEEE/ACM Symposium on Edge Computing (SEC), Workshop on Edge Computing and Communications (EdgeComm)*, 2022.

1 Introduction

It is known that Neural Networks (NN) have the capacity to become universal function approximators [1]. They have seen use in image processing [2], autonomous vehicles [3], content recommendation [4], gesture control [5], and more. Although there exist small NN models, complex problems and high dimensional data require deeper and wider models to have the capacity to represent the problem being learned [6]. In present day, the computation requirements presented by large scale models have surpassed current singular device capabilities. Because of this fact, the field of distributed machine learning has evolved out of necessity. These massive models are constrained to only distributed machine learning approaches where privacy often comes at an additional cost. Large scale NN models are usually trained in purpose built computing clusters, the cloud, or data centers since they need a huge amount of computation power. Without access to these computing environments, individual users do not have the computing resources to train, and sometimes even utilize, NN models of these sizes. On the other hand, there are many devices in the edge of the network that have idling CPU cycles. With the increasing rise of 5G networks, high bandwidth connections at the edge will allow distribution of large quantities of data to edge devices quickly. These factors, and more, shaped this thesis' research direction to create a privacy-preserving method of crowdsourced computation by offloading the computational burden of a neural network to a network of untrusted participants.

1.1 Motivation

Federated learning utilizes data parallelism to build a shared global model. Environments utilizing data parallelism inherently require additive solutions to add security and privacy into their approach. And because the complete input must be shared for model parallelism environments, the same additive requirement for security and privacy emerges. These observations lead to research into hybrid parallelism approaches, which ultimately

lead to the idea that data can be partitioned by the data dependencies for a given calculation. Building on this idea, data can be partitioned in such a way that the values can be rearranged (or shuffled) such that the difficulty in determining their original order scales with the width of the neural network. With the ubiquity of computing devices, we argue the crowdsourcing of computation power to untrusted participants is a highly applicable setting. Successes in crowdsourcing of computational problems like protein folding with the folding@home project [7] hint at the possibility of crowdsourcing neural network computation. In addition, a privacy-first solution was desired. Considering these factors, in this thesis, we perform a general investigation of privacy within distributed machine learning. While privacy is commonly considered in data parallelism environments like that of federated learning, we find that model and hybrid parallelism environments are considerably less researched. We additionally note that no existing literature, to our knowledge, notes the differing privacy concerns due to the choice of parallelism.

Lastly, the heavily debated replication crisis [8] is a notable issue for all forms of machine learning, centralized and distributed alike. Due to the heavy reliance on GPUs for improved computation speed and the indeterministic nature of their normal operation, exact replication of results is largely impossible [9, 10]. In a perfect world, replication of results would be a streamlined process. A researcher would be able to pull code from a remote repository, and given the same data, would have the capability to reproduce results precisely to validate findings. We explore CPU specific and deterministic computation in a distributed setting where operations follow strict order of operations. This introduces the ability to validate operations performed by untrusted nodes to serve as a defense against active attacks and detection of faulty or misconfigured nodes alike.

1.2 Research Questions

In this thesis, we explore the idea of spatially decoupling the underlying computations that make up a feed forward neural network and offloading them to distributed parties. With this idea, we try to answer the following research questions:

1. How to design a framework for decoupling neural network training into smaller tasks, offloading them to computation-limited devices, and aggregating the results to obtain a trained neural network model?

2. How might privacy be preserved and quantified with this approach?

3. What is the communication cost of this approach?

4. How might faulty nodes or active attacks be detected while using this approach?

1.3 Contributions

In this thesis, we answer our proposed research questions through the creation and evaluation of a fusion of techniques we coin "Divide-and-Conquer Learning" (DCL) [11]. Using DCL, a user can divide the training computation of NN into neuron-sized computation tasks and distribute them to distributed devices based on their available resources. The computation results are returned to the user and aggregated in an iterative process to obtain the final NN model. A method was designed for dividing the training computation including data and model into small tasks based on stochastic gradient descent (SGD). Weights are grouped by their associated input neuron and their positions shuffled, thus being "decoupled" from their previous spatial indexing. Likewise, the input data is grouped by features and their positions shuffled. While input and model parameters are known, the knowledge of how partitions are reordered, distributed, and aggregated is not disclosed to untrusted nodes/devices. Since the number of possible shuffling operations depends on the dimensionality of the input and layer size, it is computationally hard for an attacker to reverse the shuffling and recover the original data and model structure. The difficulty of such reversing attack is further increased due to the errors introduced by the non-associativity of floating-point operations. In this way, privacy of the data and model is protected against those distributed nodes/devices. Moreover, DCL explores determinism and redundancy to verify the computation results of edge nodes and detects faulty nodes and data manipulation

attacks. We also perform a robust evaluation of this approach.

The contributions of this thesis are summarized as follows.

1. We propose DCL, a framework of offloading neural network training to distributed parties that achieves privacy for input, model parameters, and model output in untrusted environments. Our work achieves this by spatially decoupling computations by performing a multi-dimensional shuffle that does not affect the value space of computation. To our knowledge, this is the first work to explore this novel shuffling approach and its application in a distributed machine learning setting.

2. We show that our approach enables exact replication of training convergence through the strict and repeatable adherence to the order of operations. We find that doing so eliminates the error in outputted calculations, which is a key to our computation replication and verification schemes. We demonstrate that this deterministic setting allows for the detection of adversarial nodes through redundancy-based verification.

3. We present three possible schemes for enforcing deterministic aggregation: "hash sourced entropy", "seed sourced entropy", and "static". Each having their own pros and cons, but all being methods that ensure repeatable results.

4. We implement the partitioning scheme used in DCL, show our results are reproducible in both the distributed and centralized settings, and create a simulator for calculating possible computational throughput and memory requirements. To source values for this simulator, a benchmarking script was created to evaluate performance across a range of memory sizes and operations types.

5. We quantitatively evaluate the communication overhead of DCL. In addition, we evaluate DCL in various adversarial settings and quantify the difficulty of reverse engineering the shuffling scheme and breaching privacy.

1.4 Thesis Organization

Chapter 2 "Background" overviews foundational knowledge applicable to this thesis by giving a short primer on machine learning, distributed machine learning, adversarial attacks on machine learning, and highlights related work to this thesis. Chapter 3 "Approach" further expands on the details the DCL approach more concretely, listing node interactions, how data propagates a distributed network of nodes, and some of the various use cases DCL can be applied to achieve privacy. Chapter 4 "Evaluation" presents our results and discusses adversarial attack vectors. Chapter 5 "Conclusions" wraps up the thesis with closing thoughts and discusses directions for future work.

2 Background

2.1 Machine Learning

Machine Learning (ML), a subfield of Artificial Intelligence (AI), has not only become an increasingly publicized discipline of research but also become increasingly integrated technology in our daily lives. The past three decades have witnessed a massive resurgence in ML. During this time, researchers have surpassed numerous milestones in speed, accuracy, network size, and helped push the creation of software, libraries, frameworks, and other tools to enable the easy integration of ML. These advancements have allowed ML to symbiotically evolve to meet the demands of big data. During this time, it was discovered that to model the complex patterns within these large-scale datasets, so too would the NNs be needed to grow in size and complexity [12].

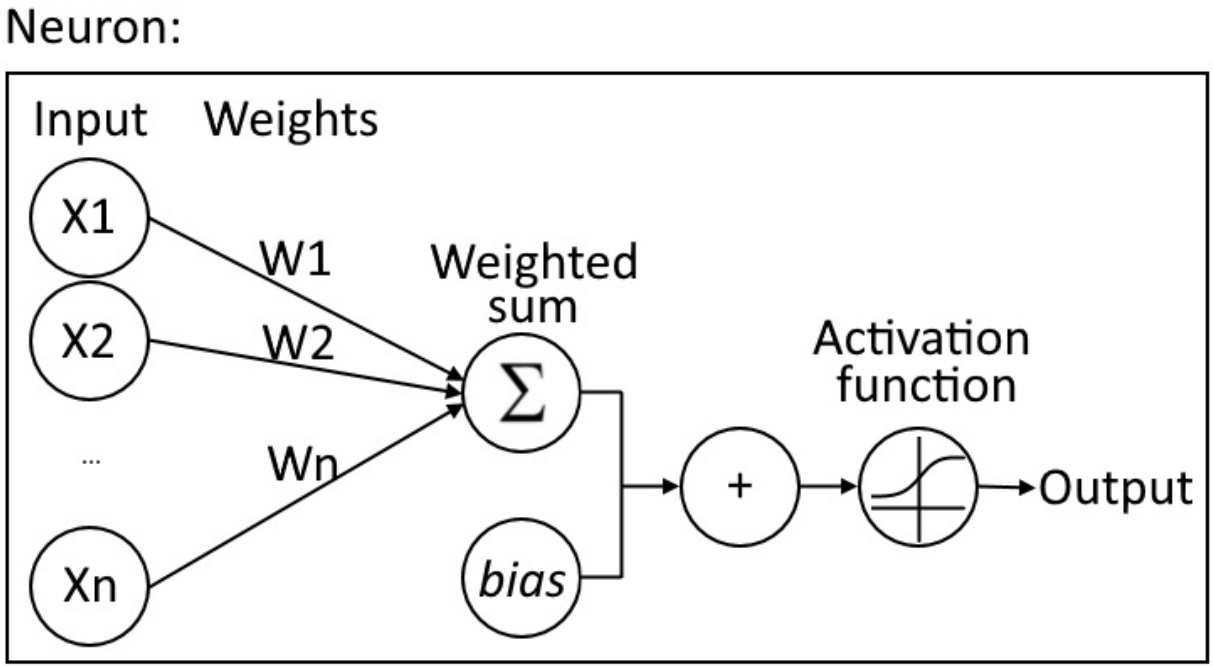

Figure 2.1: Operations for a singular neuron

NNs have not always been the high-performance computing challenges they are today.

All NNs trace their lineage back to the simple perceptron. First formally proposed by Frank Rosenblatt in 1957, the perceptron was created to learn probabilistic patterns from passed inputs and works in a manner that could be considered comparable to a biological neuron [13]. The perceptron, more commonly referred to as a neuron in modern literature, is considered the smallest division of a NN. To illustrate this division and its functionality, Fig. 2.1 visualizes the function of a singular neuron.

The perceptron was a great leap forward towards universal function approximation, but is unable to learn an embarrassingly simple logic gate, XOR (exclusive-or). This problem was often noted with the perceptron approach, weakened its adoption by researchers. It would remain unanswered for many years (this period of time is colloquially referred to as the first "AI winter" [14]) until later it was discovered that a Multi-Layer Perceptron (MLP), multiple layers of fully connected perceptrons, could resolve this [15]. Fig. 2.2 and Fig. 2.3 bridge the neuron visualization presented in Fig. 2.1 to the more commonly used and simplified directed graph.

This discovery led to many others in the importance of neural network architecture and lead to a general resurgence of the ML field. These advances would help pave the way for the creation of deep learning wherein neural networks with multiple hidden layers are studied as visualized in Fig. 2.4.

To be useful, neural networks often go through many stages of operations to become trained models. While these stages of operations are not rigidly standardized, Fig. 2.5 showcases a possible workflow which is detailed further.

First and foremost, a data collection phase is performed to amass a dataset that will be used to train and evaluate the model. Secondly, the collected data is preprocessed. This preprocessing step can consist of normalization, removal of anomalous items, and other operations to ensure only representative data is being trained on. Next, the model is initialized during which its weights are assigned randomly generated values (typically). With the model initialized, the training phase may begin. The collected input dataset is passed into the model, the required computations are performed, and output is generated. With

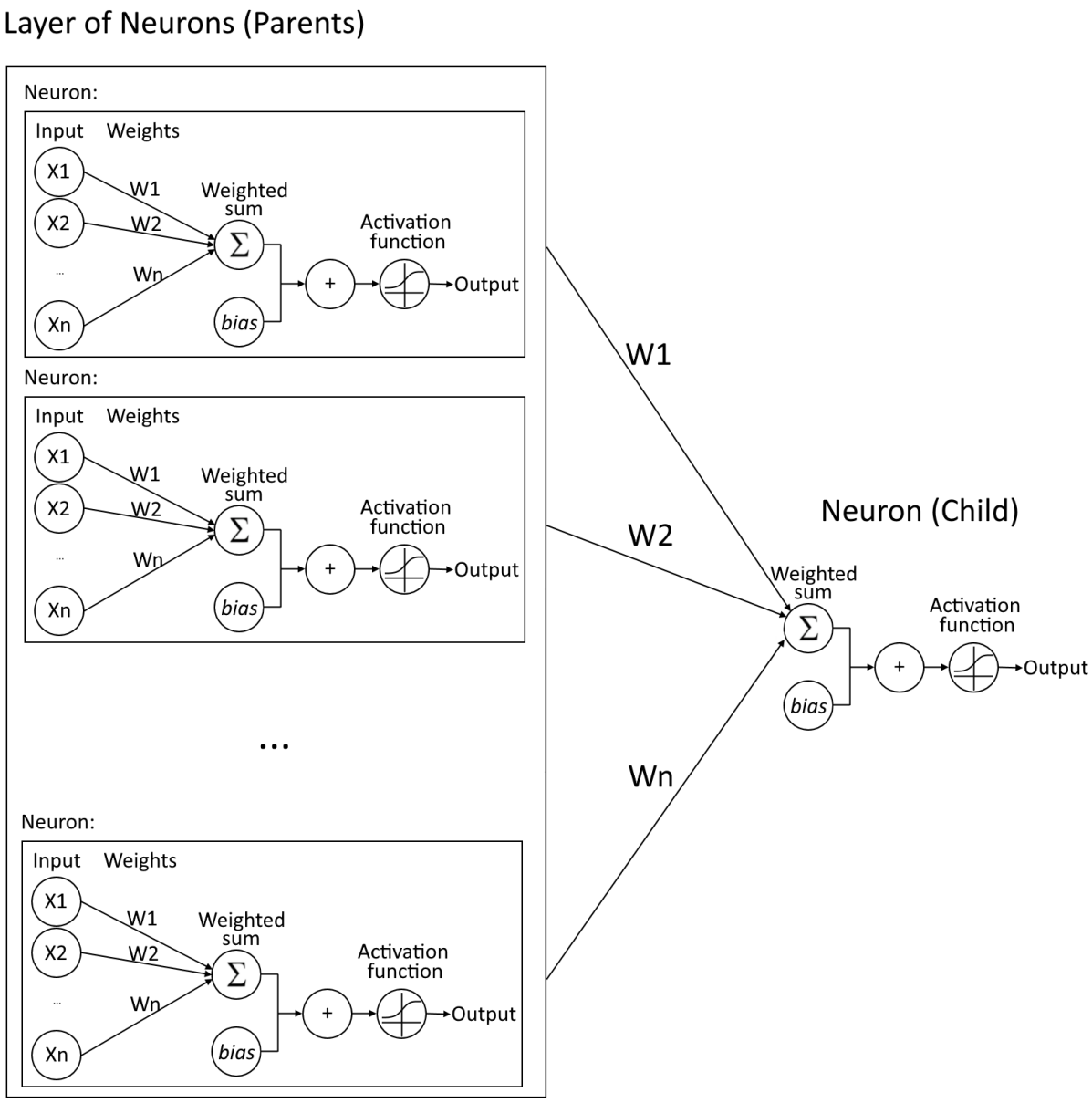

Figure 2.2: Context of a layer of neurons feeding into a neuron

the output created, a calculation of error is performed and used to update the weights in an effort to minimize error. This process continues for a defined number of iterations, epochs. Some researchers choose to evaluate their trained model every iteration while others defer this process to post convergence or intermediately over the course of training. After this process is complete, a trained model is yielded. With this trained model, validation can be

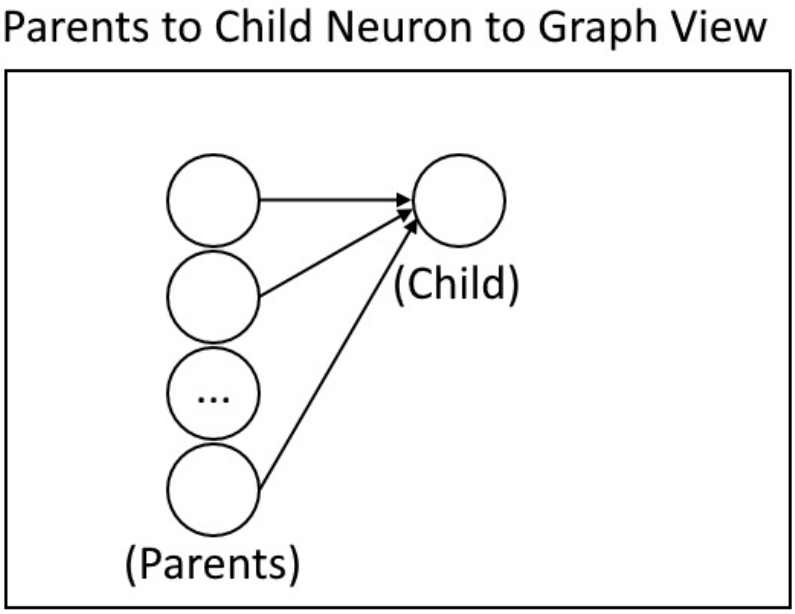

Figure 2.3: A layer of neurons feeding into a neuron in directed graph format

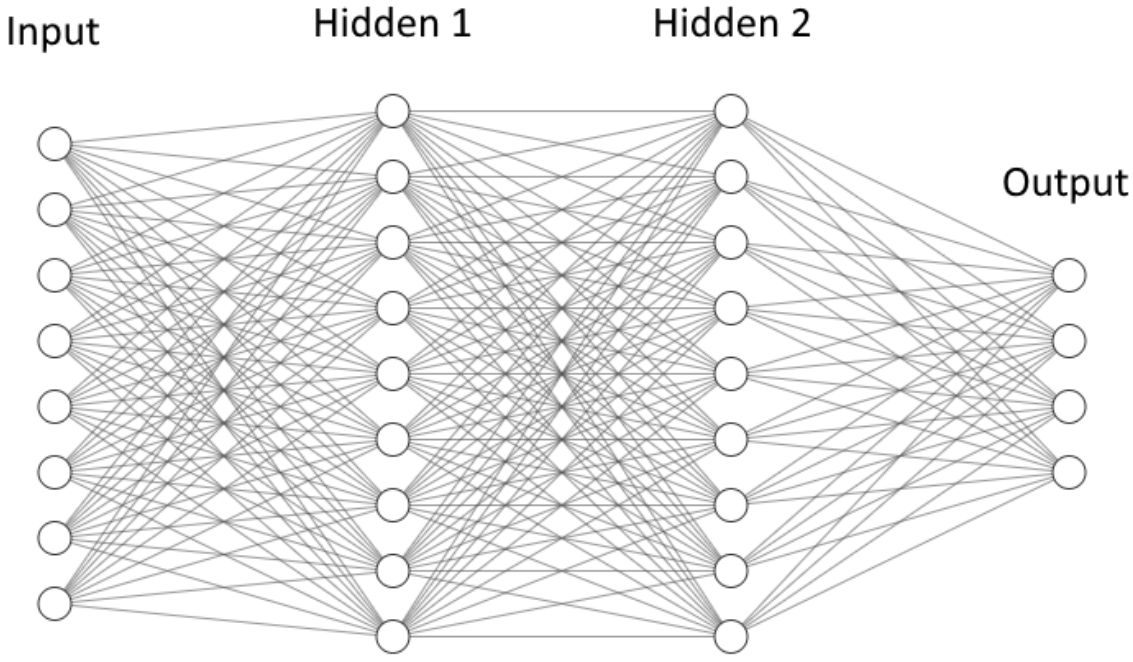

Figure 2.4: A fully connected 4 layer NN composed 8, 9, 9, 4 neurons respectively [16]

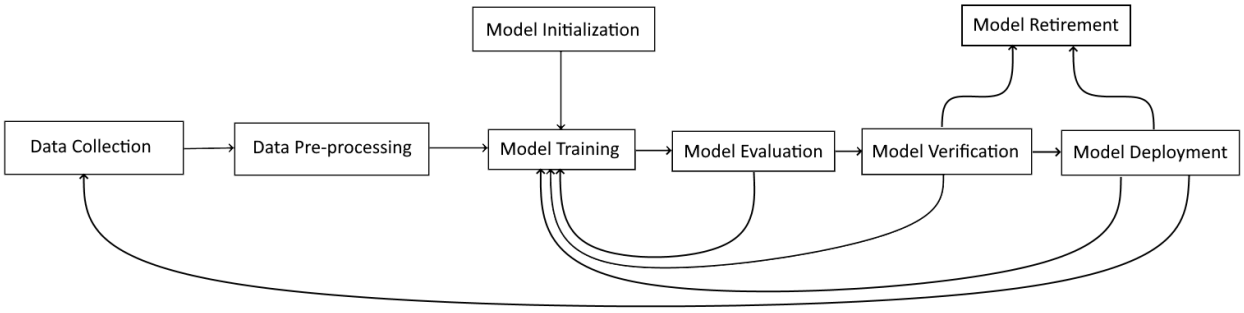

Figure 2.5: An example machine learning operational workflow

performed to ensure it is within compliance to be used within its deployment setting. The verification is important in catching models that have failed to converge or have converged into a local minima after the training process prior to deployment. Lastly, we consider model deployment, wherein the trained model is used for its created purpose. The model deployment phase is not the last stage for a model, but rather the last stage for a span of time. Models can reenter the training phase to be retrained with new data, used to create new input for future models, or retired in favor for a different one.

2.2 Distributed Machine Learning

Distributed machine learning looks to take the problem of centralized machine learning and efficiently distribute the workload across a network of machines. This can be conducted in many ways through the selection of the various techniques and options available to distributed machine learning. In this section, we briefly discuss some of the underlying configuration choices when considering distributed machine learning systems.

A common first consideration for accelerating machine learning computation is the choice of scaling up or scaling out. Scaling up, or vertical scaling, is the addition of more resources to a singular device, while scaling out, or horizontal scaling, is the addition of more devices to the network. This thesis focuses on the scaling out dynamic where privacy is most concerned. Another option to consider when selecting distributed machine learning is the choice of parallelism. How is data partitioned and parallelized across nodes to induce

speedup? Data parallelism opts to distribute a shared model and partitions the input dataset among nodes. Conversely, model parallelism chooses to distribute a shared input dataset, and partitions the model among nodes. Lastly, hybrid parallelism encompasses the dual usage of both data and model parallelism. Hybrid parallelism comes in many forms. Some examples include, but are not limited to, both model and input domains being parallelized, using model parallelism for a subset of the model and using data parallelism for another subset of the model, and so on. The distribution of data is also a factor to distributed machine learning approaches. In some cases, the distribution of data may be known (aware) or not (unaware). Some approaches like federated learning are especially sensitive to the distribution of data as computation power is often not equal across nodes. This mismatch in distribution and computing power can lead to class imbalance and is a highly researched topic. Additionally, the distribution of data can be further described by considering the disjointedness of subsets. Nodes who have disjoint subsets do not share elements and vice versa. Another consideration of distributed machine learning approaches is the scheduling of computations. There are many ways to accomplish this, but there are two distinct types of scheduling: synchronous or asynchronous. In this section, we will cover the big two: total asynchronous and bulk synchronous. Total asynchronous allows nodes to communicate without waiting for a response. The benefit of this approach is its speedup potential but comes at the expense of convergence and error being associated with the imbalance in peer latency and computation speed. Bulk synchronous contrasts this by enforces synchronization between each computing and communication phase similar to that utilized by the mapReduce paradigm [17]. While this approach comes with a reduction in speedup potential, it is guaranteed to produce correct output. We consider bulk synchronous scheduling in this thesis and synchronization must occur every layer. The last option we consider is the various additive approaches to achieving privacy. Homomorphic encryption (HE) seeks to enable computation on encrypted values, but significantly increases the computation cost of operations [18]. Differentially private updates seek to enforce differential privacy to gradient updates but doing so introduces noise that affects model convergence and accuracy [19].

Secure aggregation schemes seek to aggregate values in a privacy preserving manner while also providing a precise and accurate summation of inputs. Federated learning was improved upon with the addition of a secure aggregation protocol [20], but the obliviousness to the underlying data being summated leaves it open to a variety of active attacks.

2.3 Adversarial Attacks on Machine Learning

Machine learning has solidified its use in our modern-day information era. The utility of these models come with their possible attack surfaces. The last decade has seen many works exploring adversarial attacks on machine learning [21]. Both the centralized and distributed settings are vulnerable to attacks, but the distributed setting adds additional vectors for information leakage. In this section we explore the various classifications, settings, and types of attacks.

When evaluating adversarial attacks, we must first consider the adversarial agents; the subjects that actually perform these sophisticated attacks. Adversarial agents come in two classifications: insider or outsider. Insider agents are subjects that participate in normal operations of a given system whereas outsider agents are not. Outsider agents employ techniques like eavesdropping to gain information from the intercommunication of insiders. These agents can be further distinguished by their adversary model. We consider honest, semi-honest (or sometimes called honest-but-curious (HBC)), and malicious participants. Honest participants perform operations as intended, do not diverge from communication protocols, and do not seek to learn information from obtained messages. Semi-honest nodes are similar to honest participants in that they also perform operations as intended and do not diverge from communication protocols, but contrast in that they attempt to learn additional information from obtained messages. Semi-honest participants may also collaborate with other semi-honest and malicious adversaries in efforts to learn more information from obtained messages than they would otherwise. Lastly, malicious participants completely diverge from honest and semi-honest participants; they may choose to perform operations incorrectly, diverge from communication protocols, and employ these aforementioned capa-

bilities to learn additional information from obtained messages. The intentions of malicious adversaries may vary. They may intend to simply disrupt communications or silently accumulate data until a calculated strike can be performed. With a solid understanding of these adversarial agents and their possible adversary models, we move on to the various settings and attacks these agents perform.

Attacks on machine learning occur in different settings and are as diverse as the systems they seek to exploit. Attacks can be categorized by the setting in which the attack occurs. For instance, a white-box environments describes attacks with direct access to the model while black-box environments describe attacks with only the ability to supply input and computed output. In these environments and depending on their honesty rating, adversarial agents can perform active or passive attacks. Active attacks influence the system due to their direct interaction. Passive attacks are information gathering operations where communications are stored for later analysis.

To wrap up this section, there are several common attacks: data poisoning, model poisoning, and intelligence gathering. Data poisoning is performed via the active inclusion of malicious data. When this data is trained on, it can produce detrimental performance or functionality akin to a backdoor [22]. These carefully crafted data items are commonly referred to as adversarial examples [23]. In short, adversarial examples are created with the sole purpose of "duping" the neural network. These inputs are purposely created to negatively affect convergence, accuracy, and overall utility of the trained model. The next attack is model poisoning. Model poisoning and data poisoning achieve similar results, but rather than injecting malicious input data, the adversarial agent submits malicious gradient or network updates in a malicious manner. This could be to prevent convergence, corrupt the learning system, or subtly introduce unexpected functionality. The last attack is intelligence gathering. This attack is characterized by the collection of data for later analysis. This data can take many forms, can be collected through active or inactive (passive) methods, and may or may not be in a form that provides immediate utility without further analysis. Because of the existence of passive methods, honest-but-curious nodes can utilize this attack in an

undetectable manner.

2.4 Related work

2.4.1 Federated Machine Learning

Our work takes great inspiration from federated learning (FL) [24, 25, 26]. FL is a highly researched and commonly utilized distributed machine learning approach. FL differs from typical distributed machine learning methods in that no raw input data is transmitted between parties but instead transmit gradient updates created from locally stored data. FL leverages data parallelism which allows all parties to disjointly train a shared global model with very few synchronizations operations required. FL promises low communication overhead with a focus on high speedup in comparison to centralized training, but it natively suffers from leakage of data via gradients and the global model itself [19]. To remediate this, differentially private gradient updates can be used, but come at the cost of reduced model accuracy/performance due to the induction of noise [27]. Additionally, homomorphic encryption, or secure multiparty computing, can be used to protect both the gradients and global model but comes at the expense of much higher computation cost which results in a lower convergence speed [28]. A primary fixation of our DCL approach is the utilization of untrusted devices while still providing the data and neural network parameters in a manner that enables any computational device to participate in training. Federated learning falls short due to the requirement of the complete model context needing to be stored by any node participating in the training process and the possibility of a singular node preventing convergence [29] as well as data and model poisoning attacks [30].

2.4.2 DistBelief

DistBelief [33] is a framework for training large scale neural networks on distributed networks of commodity machines. It explores the use of Central Processing Unit (CPU) performed computation but does so with an asynchronous communication approach. It

uses both data and model parallelism (hybrid parallelism), and explores speedup gains from partitioning by layer and allows direct communication among nodes. In contrast to our approach, raw data is distributed from one node to another and knowledge of both input and model domains are known. Most other distributed training methods (e.g., [31]) in the traditional distributed computing context also suffer from this problem. Their work does not consider the privacy implications of training and communication of raw inputs, model parameters, outputs. However, it does show there is a place for CPU specific training in a field dominated by GPU powered training.

2.4.3 Distributed Learning in Edge Computing

In recent years, much work has been done to train NN models and run NN-based inferences in the edge computing context, by optimizing the use of resources from the cloud, edge, and devices [34, 35, 36, 37, 38, 39]. They include data parallelism, model parallelism, and hybrid strategies; however, they usually involve use of local data generated at edge devices, which is a key difference from our considered scenario where a user offloads the computation of their own data and model training to distributed devices. We are inspired by the research presented by Xing et al [31] wherein large scale compute clusters are leveraged in both data-parallel and model-parallel contexts. Their work highlights the important considerations for applying distributed machine learning, but differs from our work as it does not explore the privacy considerations of the data transmitted and assumes a controlled, clustered network of devices. Additionally, the work presented by Chen et al [40] demonstrates the use of backup workers is a viable strategy for synchronous optimization approaches, which is comparable to our rescheduling of tasks to many nodes. Lastly, the work shown by Kang et al [41] applies a layer resolution partitioning approach in the context of an edge computing environment by showcasing many different network and hardware variations and their effect on throughput. Their work focuses on conditionally offloading a fraction or majority of a model to a cloud for external processing while showing that there are cases where local computation is more efficient. In addition, energy consumption and speedup are the primary measurements for

their work and privacy of the transferred data or model parameters is not discussed.

3 Approach

3.1 System Model

DCL's system model uses a hierarchical form of roles to identify and organize nodes on a network. To better describe the responsibilities and capabilities of nodes on the network, the role nomenclature in Table 3.1 was created.

Table 3.1: All node types and a brief description of their operation.

Node types	Description
Analyst	Initiator of the NN offloading tasks responsible for task partitioning, shuffling, and aggregation.
Scheduler	Advertises and manages a executor pool to facilitate task assignment and result collection.
Executor	Executes offloaded computations and returns computed result.

An analyst is the initiator of a learning process and the owner of training data. It determines the NN model structure and parameters, partitions the model an data into small computation tasks, assigns the tasks to schedulers, and aggregates the results returned from schedulers. It can also handle some computation tasks based on its available resources. A DCL system could have many analysts that offload their NN training independently. Without loss of generality, we describe our solution considering only one analyst.

Schedulers function as an in-between for the analyst and executors. Each scheduler manages a pool of executors (including itself, since a scheduler has computing resources too and can perform some computation tasks). After receiving computation tasks from an analyst, a scheduler assigns the tasks to its executors, collects the results from the executors, and returns the results to the analyst. Each scheduler is responsible for evaluating and advertising its constituent executors' total computing power, managing said pool of executors to ensure that they leave the network gracefully or are handled in the event of dropout/failure, and being the primary point of contact between analysts and executors. Schedulers are typically high availability, high uptime devices that sometimes take the form of dedicated servers. Having a high-speed networking device for a scheduler allows for analysts to maintain fewer connections and allows executors to contribute from a comparatively slower network

speed, say a consumer home network. Schedulers are also responsible for estimating the combined computing capabilities like FLOP performance and latency of their constituent executors to better advertise their available resources to analysts. In the edge computing architecture, edge nodes are good candidates to serve as schedulers.

Executor nodes perform computations that are passed to them by schedulers. Executors perform operations for a singular scheduler. Executors nodes can take many forms, from limited capability edge devices to high performance devices. They are expected to have a network connection that enables them to receive computation tasks from schedulers and transmit the results in a timely manner.

With DCL's role nomencalture described, we provide a quick naming convention for describing networks by their node composition. Networks are described by the number of analysts, schedulers and executors per scheduler. For instance, Fig. 3.1 showcases a 1-2-3 network. This network is composed of one analyst, two schedulers, and 3 executors for each scheduler. In the case that the number of executors differ across schedulers, the number of executors is specified in array notation. For example, a network composed of a single analyst, three schedulers with each having four, five, and six executors respectively would be described as a 1-3-(4,5,6) network.

Nodes that cannot simultaneously load input and model context into RAM are unable to reasonably contribute to FL training. DCL comparatively relaxes memory requirements of training. A node only needs to be able to store the largest layer of a model and its input, a fraction of the complete model and input context that FL training would require each participate to store. This opens the door to the utilization of very limited performance devices. Networking communication speeds remain paramount to determining if a given node is performant.

Communication is a central to DCL achieving operational performance when comparing to other distributed learning techniques. The communication cost of synchronization every layer is high, and the amount of data that must be transferred to and from nodes can be burdensome. A high bandwidth network is required to achieve adequate performance, but

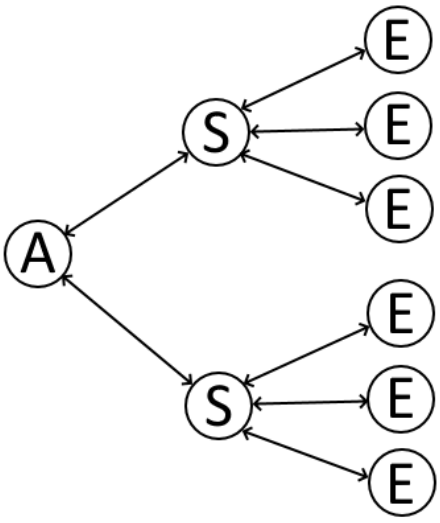

Figure 3.1: A visualization of a 1-2-3 network.

the computational requirements to distribute values is far less than performing the computations themselves. This requirement can be alleviated by the utilization of high-bandwidth links between particular nodes during training and using commodity bandwidth networks between others where the amount of data to transfer is comparably less. Fig. 3.2 visualizes the bandwidth requirements between nodes. As high-bandwidth 5G and even 6G networks emerge, the communication cost might not be a big issue.

3.2 Security Model

DCL was heavily influenced by and follows an antipattern approach to FL by seeking to provide a solution to the reverse trust environment assumed in FL which, we argue, is a more applicable setting. In the case of FL, seemingly honest parties jointly train a shared, global model. These parties are weary of an honest but curious (HBC) central server reverse engineering the provided gradient updates into the locally stored inputs used to create them. To prevent the central server from determining a specific node's updates, a secure aggregation scheme is used. DCL contrasts FL by presenting a trusted analyst that wishes to leverage the computation power of untrusted devices. The analyst is weary of untrusted distributed

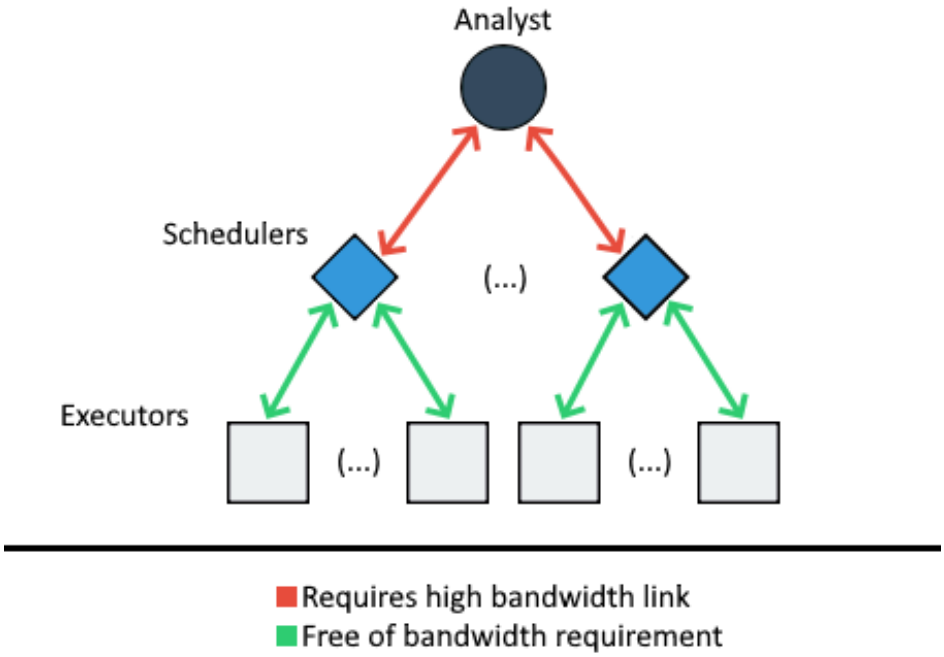

Figure 3.2: A general visualization of a network detailing network link requirements between nodes.

devices attempting to reverse engineer the provided computations to discover the input, model, and outputs.

FL suffers from many forms of active attacks simply due to its allowance of nodes using undisclosed local data for training; a perfect setting to quietly employ an adversarial example or other active attacks. DCL avoids this by placing the burden of providing data to the central facilitator of training (i.e., the analyst). Doing so allows the party whom is responsible for the correctness of the model to ensure that the model is being trained on legitimate data, avoid the data and computation distribution heterogeneity problems that are applicable to FL, and perform distribution-aware normalization and other pre-processing operations to inputs prior to training.

DCL offers semantic security in that only negligible information can be feasibly extracted from these shuffled computations. We consider that all pixel values are known in a

fully adversarial and collaborative environment. In this environment, the image's histogram can easily be inferred, but the parties lack the undisclosed knowledge as to "where each pixel goes". Consider Fig. 3.3 below which displays a possible reconstruction of a provided input image.

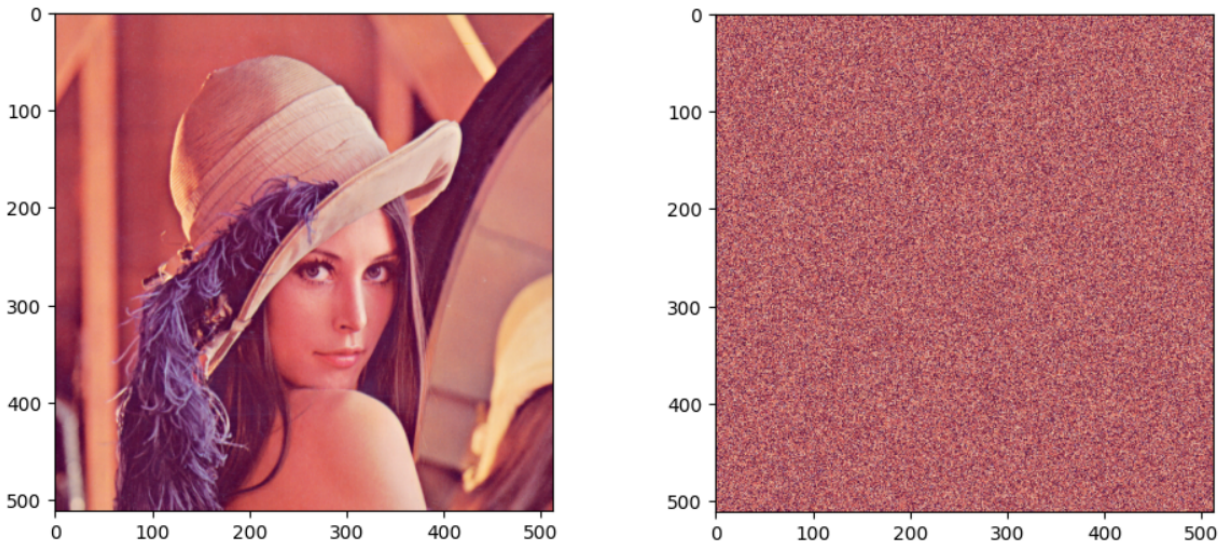

Figure 3.3: A source image and a possible reconstruction from shuffled values.

3.3 Process Overview

An analyst first formulates a training problem, including a training dataset, the NN structure, weights, hyperparameters like learning rate, and so on. Then it inquires a number of schedulers to see the amount of computing resources they and their pool of executors have. It might adjust the training problem based on the available resources. Next, it partitions the training into neuron-sized small tasks. In this process, random shuffling is done for each partition for data and model privacy protection. The shuffling on the input and parameter domains is done in a manner that does not affect the value space of computation results but only their spatial positions. Then it assigns the partitioned tasks to selected schedulers. Each scheduler assigns its portion of the tasks to its executors. Executors with more computation resources can take more tasks; vice versa.

After an executor completes the tasks assigned to it, it reports the results back to its scheduler, which will forward the results to the analyst. The confidentiality and integrity of the result report communication can be easily protected with standard secure communication methods, assuming the analyst's public key is known to the executors (e.g., the public key certificate can be included in the task and sent to executors). The analyst manipulates the results to the unshuffled ordering, and aggregates the results. These aggregated results form the input for the following NN layer and the process repeats for each layer. Then the backward calculation of the NN is done layer by layer similarly. That complets one epoch of training. This workflow continues until the defined number of epochs is met. After which, the desired model is produced.

An analyst can replicate a task multiple times and allocate them to multiple executors. It can then use majority vote to select the correct computation result and detect misbehaving executors that generate wrong results.

When an executor is recruited or joins the executor pool by connecting with the scheduler, the scheduler provides the executor with a benchmark to evaluate the node's available computational performance. The scheduler makes note of the connection speed and the demonstrated compute capability, and they may additionally perform this test over time, regularly or as needed, to maintain a good estimate of the available resources at the node. This may be desired if the executor node is not purely dedicated to training and handles other workloads in addition to handling and executing computation requests. With this information, the scheduler can better divide the computations composing a job request to its pool of executors and advertise this number to the analyst node.

Executor node dropout is a concern for uninterrupted operation. To combat node dropout, a staleness time is set for every computation task that, when met, allows for a configured fallback executor to be allocated and perform the computation. The scheduler handles these cases of node dropout and reissues the computation task to another executor, making notes and rebalancing the distribution of computation tasks for the next training job request.

3.4 Partitioning, Shuffling, and Assignment

Prior to partitioning, DCL requires knowledge of the input and neural network weights, which we refer to as the factor space. With this knowledge, the analyst partitions the input and weights into tasks. A task covers all computations for a given neuron. We consider the context of an image dataset and visualize the data partitioning scheme where values are grouped by image feature in Fig. 3.4. We extend it in Fig. 3.5 to give context to the original/source orderings prior to the shuffling operation. Next, we apply the shuffling operation to both the input and weight matrix in Fig. 3.6. Relative to their original source orderings, the input vectors are column-wise shuffled whereas the weight vectors are row-wise shuffled. This shuffling operation does not effect the value space but only the spatial position of these values and can be thought of as a permutation of elements that can be later reversed on the produced computation results. With these shuffled tasks, the analyst can assign them to nodes. Assigning of tasks to schedulers and executors can adopt different strategies. For example, the number of tasks assigned to a scheduler or executor can be proportional to its computation performance, network uptime, throughput, and so on. An executor receives a task in the format of Fig. 3.7 and produces the outer product shown in Fig. 3.8. Schedulers receive and re-transmit these produced outer products values back to the analyst where the results are unshuffled and then aggregated, which we detail in the next section.

Lastly, we discuss possible shuffling techniques devised for DCL. We coin "static layer shuffling" (SLS) the use of shuffled sequences for both input and model that is generated by layer and repeatedly used for future recomputations of that layer. And contrasting, we coin "dynamic layer shuffle" (DLS) the repeated generation of a random shuffle sequence every layer computation from a source of seeded/reproducible entropy. This can be accomplished by using a seeded random number generator or a hash chain to generate reproducible entropy. We note that for small neuron count layers benefit from the use of SLS where only one shuffling is generated, as DLS exposes a set of shufflings that are not the source ordering.

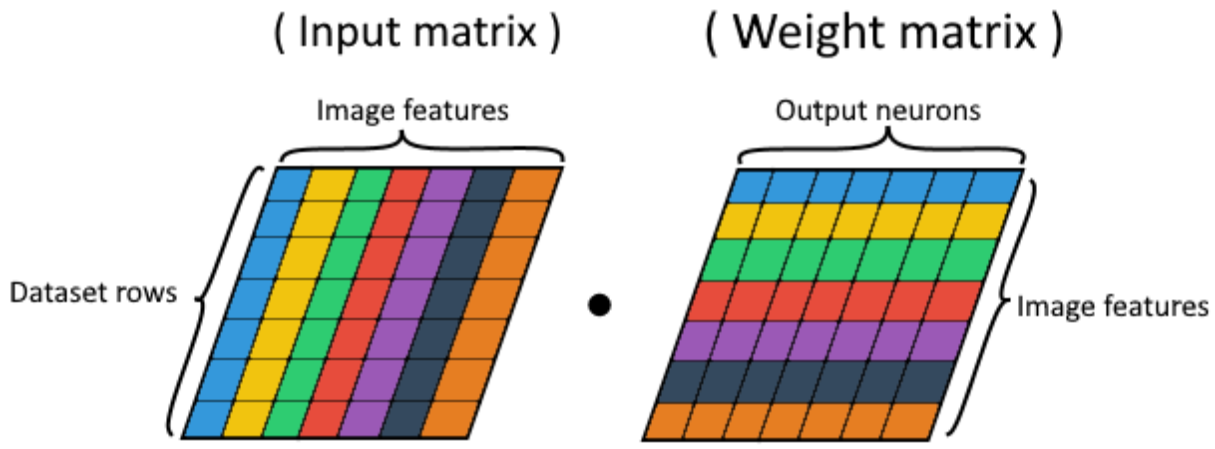

Figure 3.4: A visualization describing the element association between an input and weight matrix.

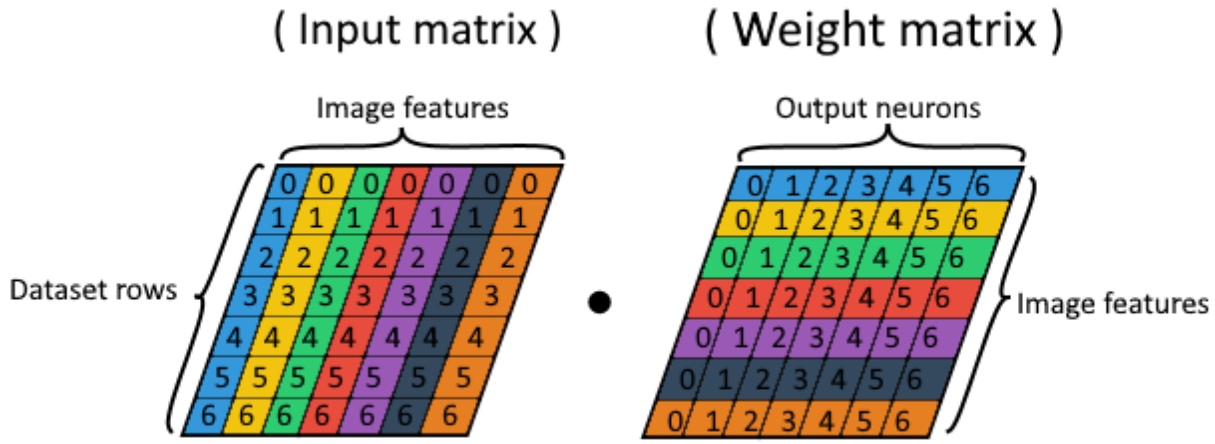

Figure 3.5: A visualization describing the the source ordering for a input and weight matrix prior to shuffling.

This effectively reduces the set of possible shuffling orders, making deploying attacks to discover the original source ordering easier. Conversely, while SLS allows the analyst to avoid the cost of generating new shuffling orders for each future layer of computation, DLS ensures that attackers who discover the shuffling pattern for all or a subset of calculations

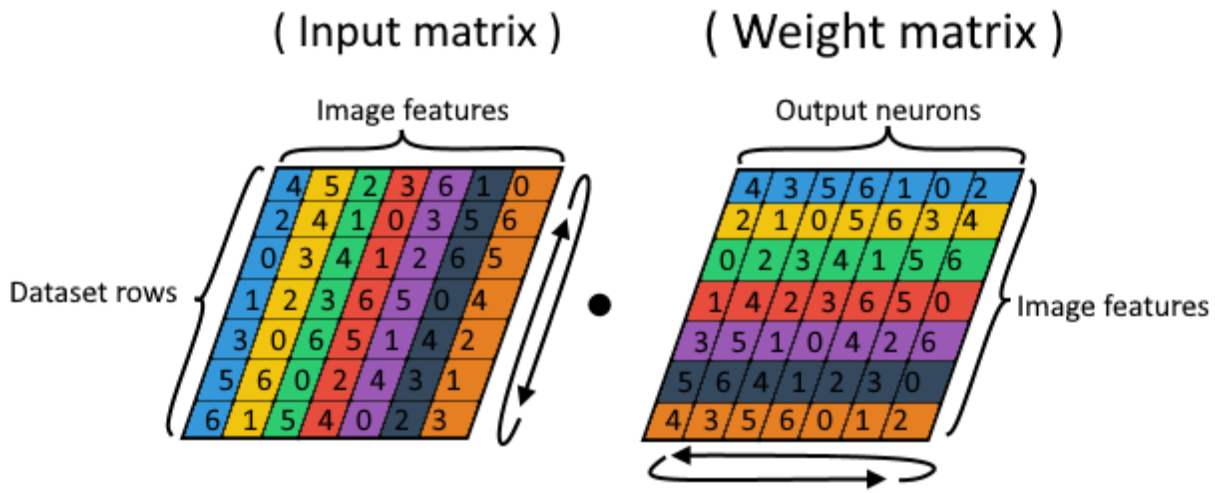

Figure 3.6: A visualization describing the the shuffling process for a input and weight matrix.

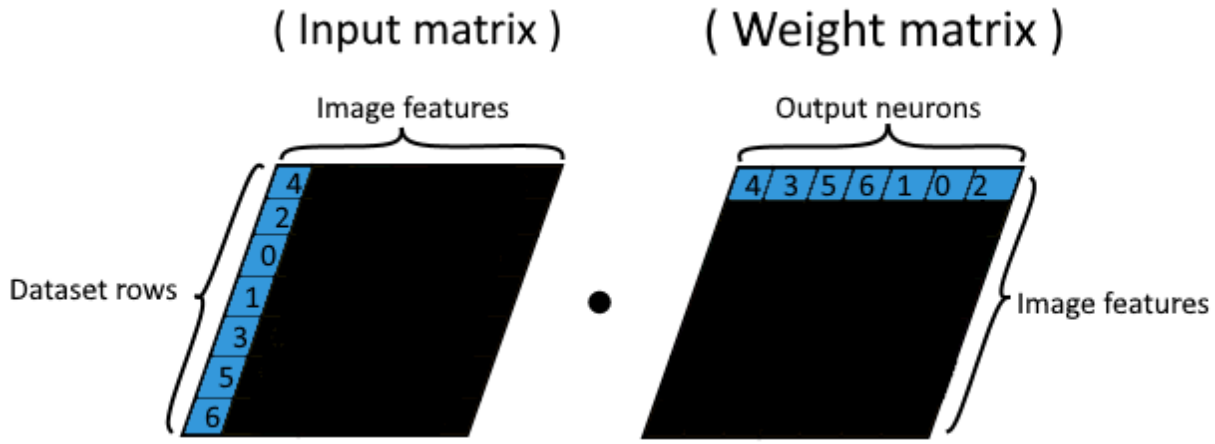

Figure 3.7: The context of a single shuffled input and weight vector.

for a given epoch will not be able to apply the learned knowledge to other epochs.

3.5 Aggregation

Determinism of operations for later recomputation is one major goal of this work. Through this, the analyst may discover faulty or adversarial nodes wishing to disrupt the

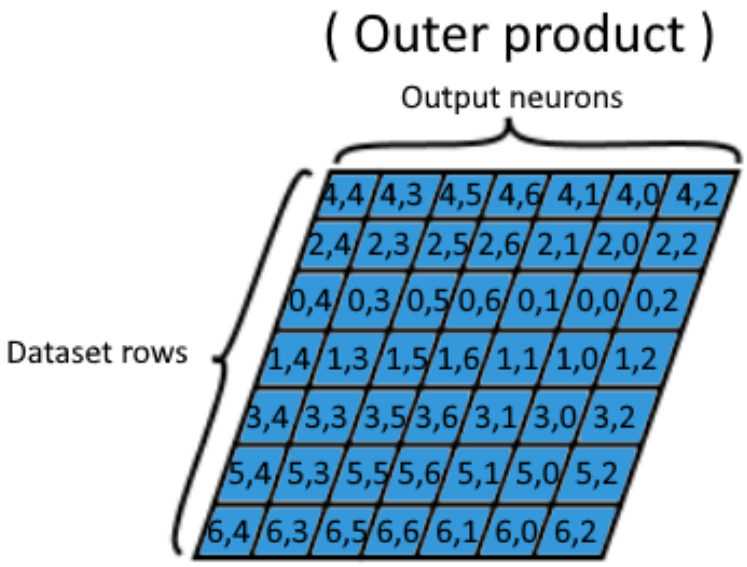

Figure 3.8: The computed outer product from the single shuffled input and weight vectors in Fig. 3.7 in shuffled order.

training process. We note that the order in which weighted sums are aggregated and the order in which input and weights are multiplied as crucial factors in achieving determinism. We visualize the unshuffling and aggregation of the received outer products in Fig. 3.9. While addition and multiplication are associative in the context of mathematics, floating point addition and multiplication are not associative. Performing operations out of sequence introduces error and in most contexts this error is negligible. In pursuit of determinism, however, this error is a source of entropy if the order of operations is not strict and reproducible. Luckily, strict order of operations can be enforced as the very minute amount of error introduced by differing orders is detectable. Similar to our shuffling approaches, static and dynamic approaches for aggregation were considered. We propose "static layer aggregation" (SLA) and "dynamic layer aggregation" (DLA) as possible aggregation schemes for DCL. SLA is the consistent use of an ordering for the aggregation of computations results, whereas DLA uses a source of reproducible entropy like the seeded random number generation or a hash chain to generate the aggregation orders. We note the use of DLA increases the difficulty of reversal engineering the global model and dataset, as the attacker cannot use

successful reverse engineering attempts of previous layers to discover the aggregation order for later layers and will be forced to recalculate possible aggregation orders. DLA additionally alleviates the problem of a less performant node holding up the aggregation process as the order of aggregation will differ across layer calculations thus enabling faster aggregation speeds.

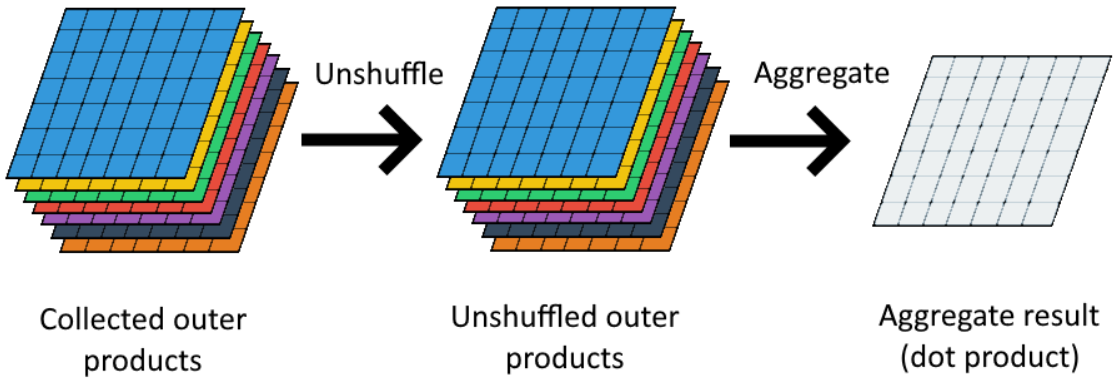

Figure 3.9: A visualization overviewing the unshuffling and aggregation to produce the desired dot product result.

4 Evaluation

4.1 Communication Cost Analysis

We analyze the communication cost during the issuing of tasks and the receiving of the results as a function of the dimension of the input matrix I with dimension (a, b) and weight matrix W with dimension (b, c), where a is the number of data samples, b is the number of input features, and c is the number of connections to the following layer. We define D to be the number of bytes per element. The selected data type to store the values of input and weights makes a considerable impact on the amount of bandwidth used transmitting tasks to executors and receiving their results. The bandwidth used by the analyst during the issuing of tasks, T is given by:

$$T = D * (a * c * b) \tag{4.1}$$

This value is further used to calculate the bandwidth used by each scheduler, U, by taking the result above and dividing it by the number of schedulers in this training, s:

$$U = O/s \tag{4.2}$$

And this process is repeated for calculating the bandwidth used by each executor, V, by dividing by the number of executors in the scheduler's pool, e:

$$V = U/e \tag{4.3}$$

To simplify calculations, we assume every scheduler has the same number of executors. Next, we calculate the bandwidth used by each executor, X, when transmitting the result to its scheduler:

$$X = D * (a * c) \tag{4.4}$$

X is the size of a single outer product task result. With this value, we may calculate the bandwidth used by each scheduler, Y, when transmitting results to the analyst where e is the number of executors for that given scheduler:

$$Y = X * e \tag{4.5}$$

And lastly, we calculate the total bandwidth used by the analyst, Z receiving all results, s:

$$Z = Y * s \tag{4.6}$$

We use the above equations to calculate the communication cost when passing the complete MNIST dataset and connecting weights to a 128 neuron layer using values $a = 60000$, $b = 784$, $c = 128$. Table 4.1 shows the results assuming use of floats $D = 4$.

Table 4.1: Bandwidth used per node by type during calculation of MNIST input layer to 128 neuron layer using floats to a 10 scheduler, 100 executor DCL network

	Issuing tasks	**Receiving results**
Analyst	0.1886 GB	24.58 GB
Scheduler	0.0189 GB	2.458 GB
Executor	0.0019 GB	0.2458 GB

4.2 Computation Cost of Attacks against Privacy

The privacy of our approach hinges on scheduler and executor nodes being unable to spatially reconstruct the shuffled data and model weights they receive back into their original orderings. Our first considered attack is the attempt to determine the output neuron weights based on the updates that are applied to them during backpropagation. Since the weights that connect to the same neuron are not equally updated, this attack fails.

Next, we analyze the possibility of inferring the full data and model via exhaustive search. Exhaustive search to discover the original orderings of data is possible only if all

computation results are obtained by the attacker. That means all involved executors need to collude for such attack to succeed. In practice, the chance of such strong collusion is very low so long as the number of executors for a training is not too small. As we analyze next, even if all executors collude, it is still computationally infeasible to do exhaustive search.

We analyze the computation cost of determining the correct reorderings of the input and weights provided in tasks. Specifically, we evaluate the worst-case-scenario for an analyst: a group of colluding executors have managed to obtain complete context of tasks from the preceding two rounds of communication. Through this, they have complete knowledge of the input and weight values that compose the computations but lack the knowledge of their source orderings to produce the inputs for the proceeding round of computation. Determining this information exposes the given layer's interconnectedness of neurons. Repeating this process for all layers allows for the inference of the ordering of the input dataset, thus violating the privacy offered by DCL for both the model and input dataset. To demonstrate the scale of difficulty to do this, we quantify the difficulty of breaking a singular layer's ordering by quantifying the space of possible orderings, use a max theoretical speeds at which this space could realistically be searched, and determine the amount of time required to perform this search in the worst case scenarios. We omit the addition of a bias vector, a common layer operation, to the computed computations to reduce the difficulty of the attack. With all these factors, this represents the best-case scenario for a malicious group of colluding executors and a conversely worst-case scenario for the responsible analyst.

Continuing, we define the variables and their dimension that are used to generate the partitioned tasks:

- input matrix I with dimension (a, b)
- weight matrix W with dimension (b, c)
- weighted input matrix Z with dimension (a, c)

Weighted input matrix Z is produced by:

$$Z = I \cdot W \tag{4.7}$$

The weighted input matrix is used to produce the activated matrix A with dimension (a, c) and where f is the differentiable activation function:

$$A = f(Z) \tag{4.8}$$

I and W are partitioned into tasks and distributed to e executors. These tasks are computed to produce the outer products that, when unshuffled and aggregated, produce Z. We partition I column-wise and W row-wise and these partitions are permuted from their original orderings. For simplicity, we consider that all executors are assigned an equal number of tasks, and assume that the dimension b is evenly divisible by e, the number of executors. The number of possible aggregation orderings, P, provided that each task is uniquely shuffled, is calculated by:

$$P = b! \tag{4.9}$$

The number of possible shufflings for a computed outer product, S, is given by:

$$S = (a! - 1) * (c! - 1) \tag{4.10}$$

We define d to be the non-zero minimum difference between values in the set of all outer product calculation results, and we compare this value to k, the accumulated error during the aggregation of the computed outer-product results. This value is produced by multiplying the number of aggregation operations, b, by the largest possible amount of error from unit roundoff, r which is defined to be half the smallest representable value. In the case of floats: $r = 2^{-24}$, and doubles: $r = 2^{-53}$. When the amount of accumulated error during aggregation, $k = b * r$, does not exceed d, exhaustive search will produce a singular result, and the first

attempted reordering of computations could yield the result. The time complexity of this search is $O(S)$. When k exceeds d, however, exhaustive search will produce a set of possible orderings instead of a single "correct" result, thus requiring complete traversal of the set of possible shufflings and aggregations. In this case, the time complexity is increased to $O(S * P)$.

We use the above equations and consider tasks that compose the complete MNIST dataset and connecting weights to a 128 neuron layer using values $a = 60000, b = 784, c = 128$. We compute the runtime using the above calculations with the largest grouping of computation power to date set by the Folding@Home project [7] in 2020 at 2.3 exaFLOPS, and we use this figure to calculate the expected computing time in Age of Universe (AU) units. We find that the computing time when $k > d$ is $4.6 * 10^{260980}$ AU and when $k < d$ is $4.6 * 10^{260832}$ AU. The computing requirement for breaking a permutation solution like ours with brute force search is currently impossible for even highly organized adversaries.

4.3 Correctness Validation

To verify our approach, we created a dynamically sizeable fully connected multi-layer neural network and employed the commonly used MNIST dataset as input. We chose a very simple neuron network with layer sizes 784, 128, and 10 neurons for the input layer, hidden layer, and output layer respectively. We verified the correctness of DCL by comparing its trained model with a centralized training scheme that use the same dataset and NN model configurations. The centralized training served as a control, where no partitioning or shuffling is involved. The other one, DCL training, performed computation on the partitioned and shuffled format. Each partition was executed sequentially and then the results were deterministically aggregated. We found that the two obtained models were identical, validating that the partitioning and shuffling approach of DCL does not affect the value space.

4.4 Exploring Possible Implementations

We created an implementation wherein a series of web servers would receive and transmit obtained data to other defined clients. These web servers were virtualized on Server A in container environments and communicated over an internal network bridge to evaluate the best case transmission times between processes. Many of our efforts were originally limited to the utilization of python due to its high utilization and hardware and software support. Many networking libraries were queried and our produced results for this experiment highlight the extreme need for a high-performance networking library to achieve reasonable speedup in comparison to centralized training. According to our experiments and the libraries we queried, the fastest transfer speed maxed out around 3.5gb/s, which prompted exploration into other languages like c++ and Go which demonstrate the ability to reach 16gb/s during our initial tests. We argue that an efficient implementation should spend a majority time computing rather than transmitting data and that throughput of floating-point operations should be maximized. In addition to overhead introduced from transmission, the serialization of data before and after transmission was a sizeable component for some implementations. We additionally explored the transmission of raw bytes between nodes, but note that, while there are performance gains in avoiding en/deserialization, there are major security concerns for this approach. Doing so allows for attacks ranging from simple ones, like buffer overflows and denial of service (DoS) attacks, to advanced and unexpected vulnerabilities arising from unintended functionality in how the data is processed.

4.5 Operation Benchmarks

To evaluate the performance of used devices, a python benchmarking script was created and ran from each device. These tests were conducted with ranging input sizes for the add, subtract, multiply, divide, and dot product operations. Each operation was performed one hundred times and the average of was taken and used as the data point. Figures 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 display these results in graph format. These results are specific to aliased

devices: laptop, desktop, and server A listed in table 4.2.

Table 4.2: Utilized devices for operation benchmark

Used Devices	**Alias**
PowerEdge R620 server, Dual Intel Xeon E5-2670 2.60GHz, 256GB RAM	Server A
Desktop, FX6300 4.1GHz, 16GB RAM	Desktop
Laptop, Intel i5 5200 2.20GHz, 8GB RAM	Laptop

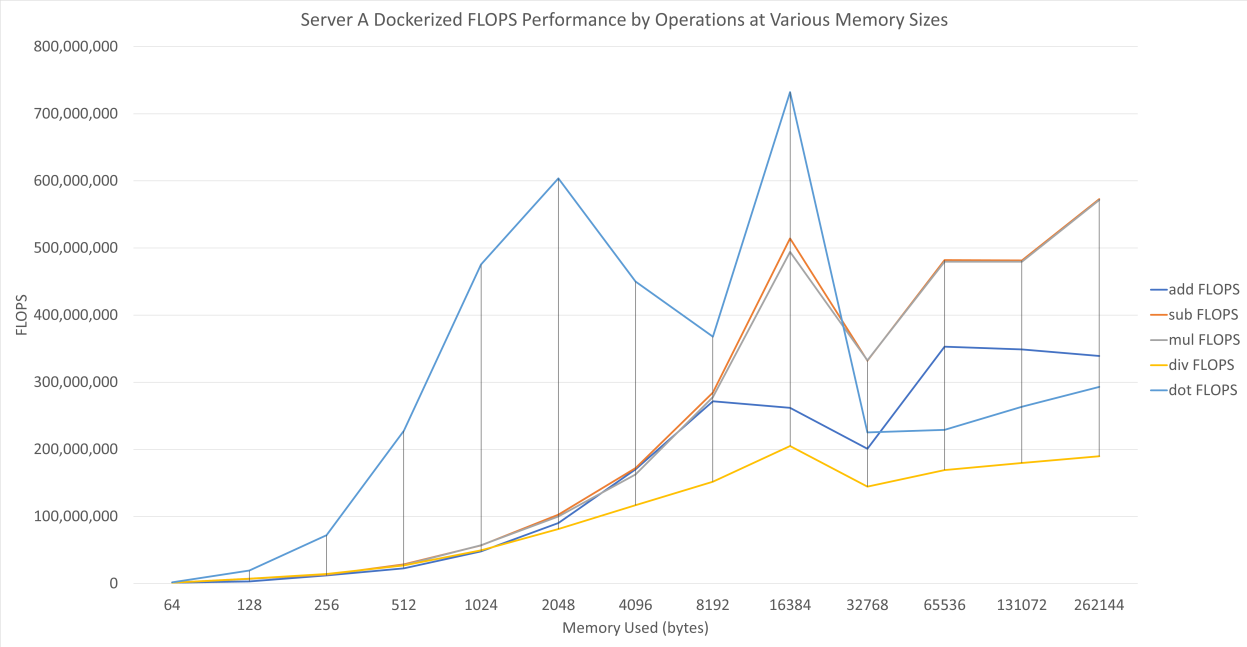

Figure 4.1: Server A Dockerized FLOPS Performance by Operation at Various Memory Sizes

Several observations can be made from these gathered results. Most notably, the overhead of containerization is nonnegliable when considering operational performance. Containerization universally reduced operational throughout across all tests and devices. We additionally note that many operations experience reduced performance when constrained by memory availability when comparing across the laptop, desktop, and server devices. These results showcase that operational throughput differs by the type of operation. We note that the tested CPUs used for these tests were highly optimized for quick calculation of the dot product. Next came the subtract and multiply operations with addition and division being

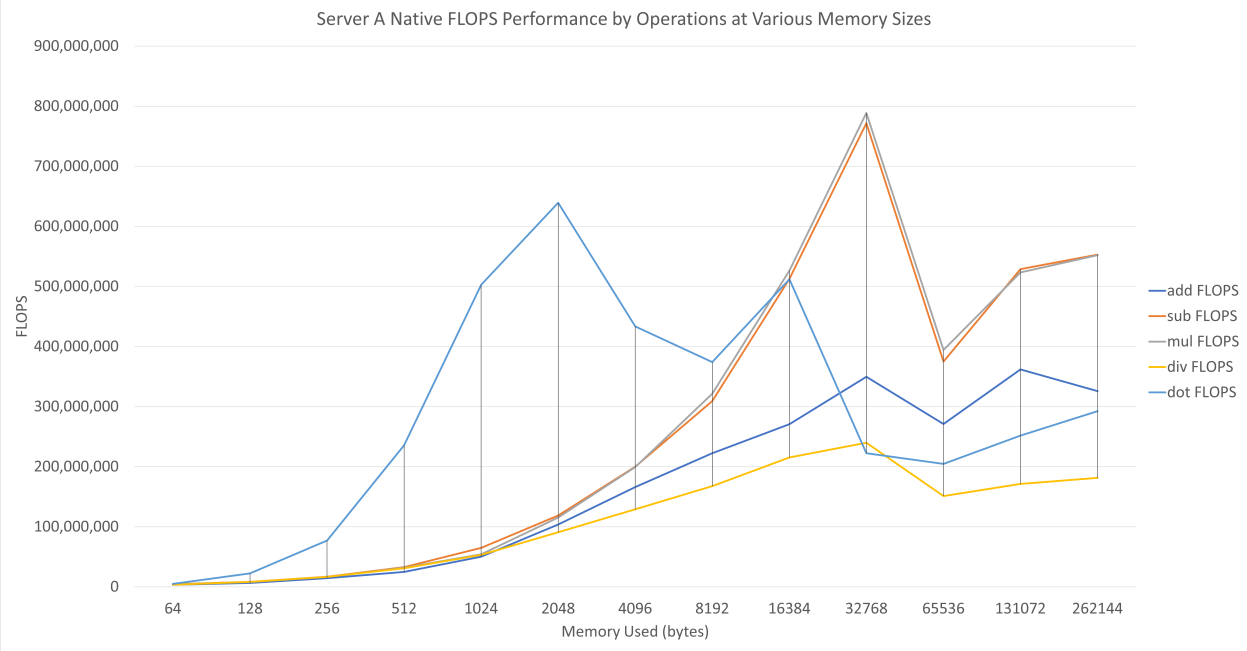

Figure 4.2: Server A Native FLOPS Performance by Operation at Various Memory Sizes

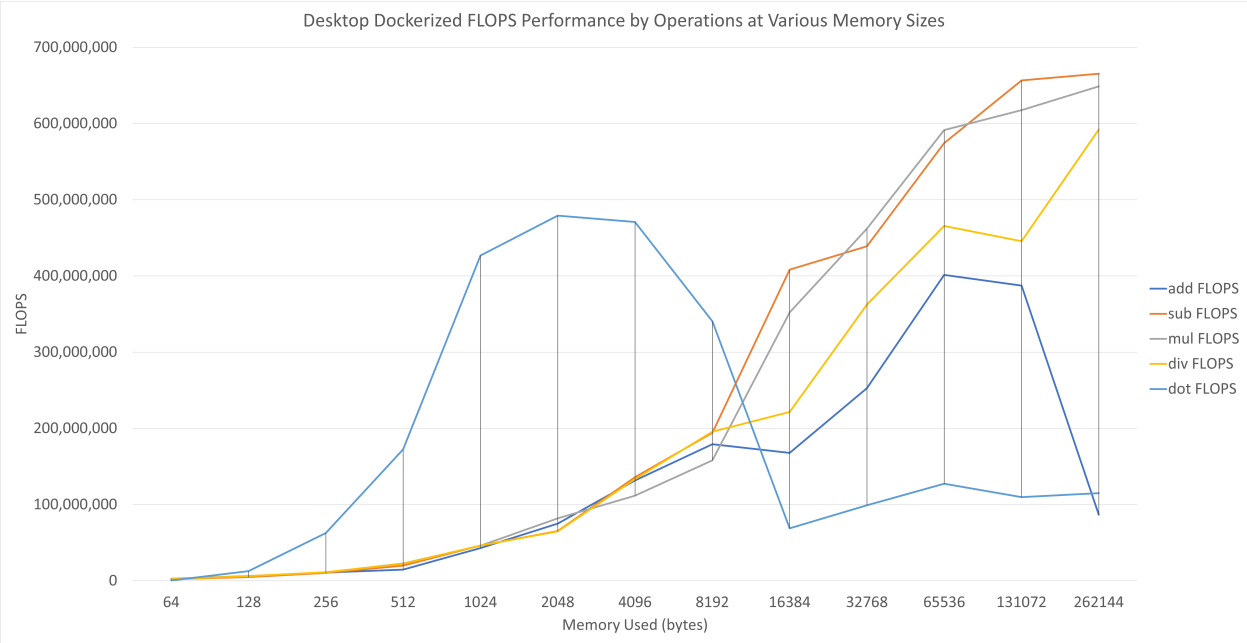

Figure 4.3: Desktop Dockerized FLOPS Performance by Operation at Various Memory Sizes

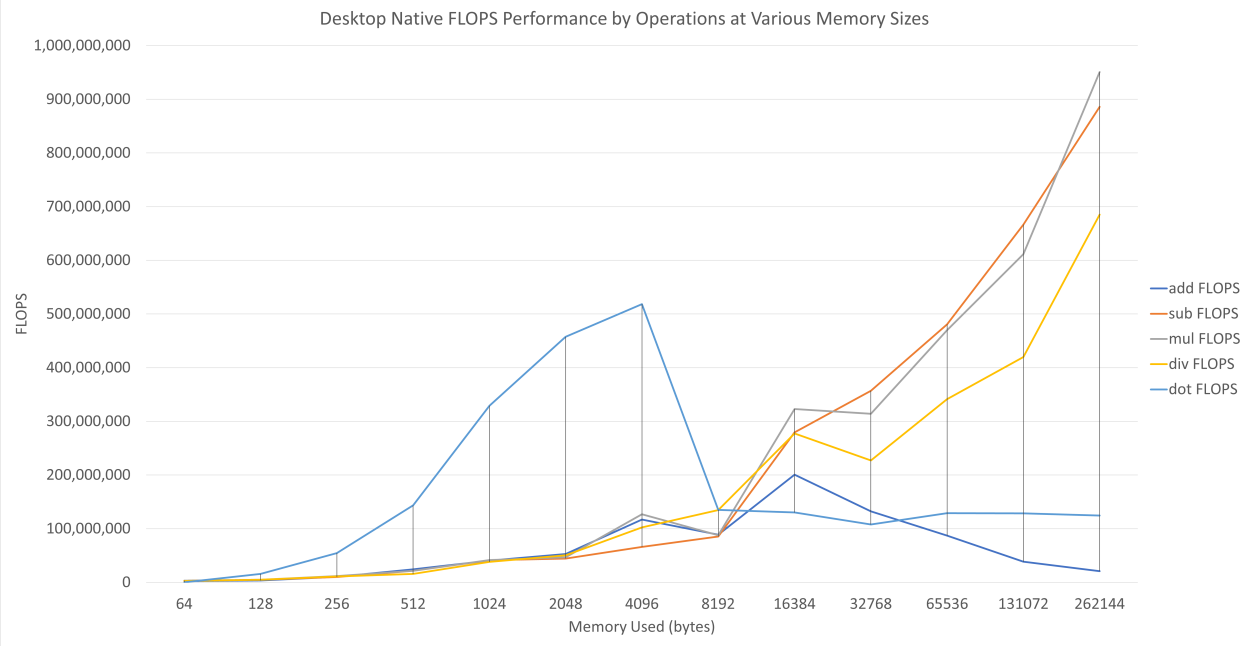

Figure 4.4: Desktop Native FLOPS Performance by Operation at Various Memory Sizes

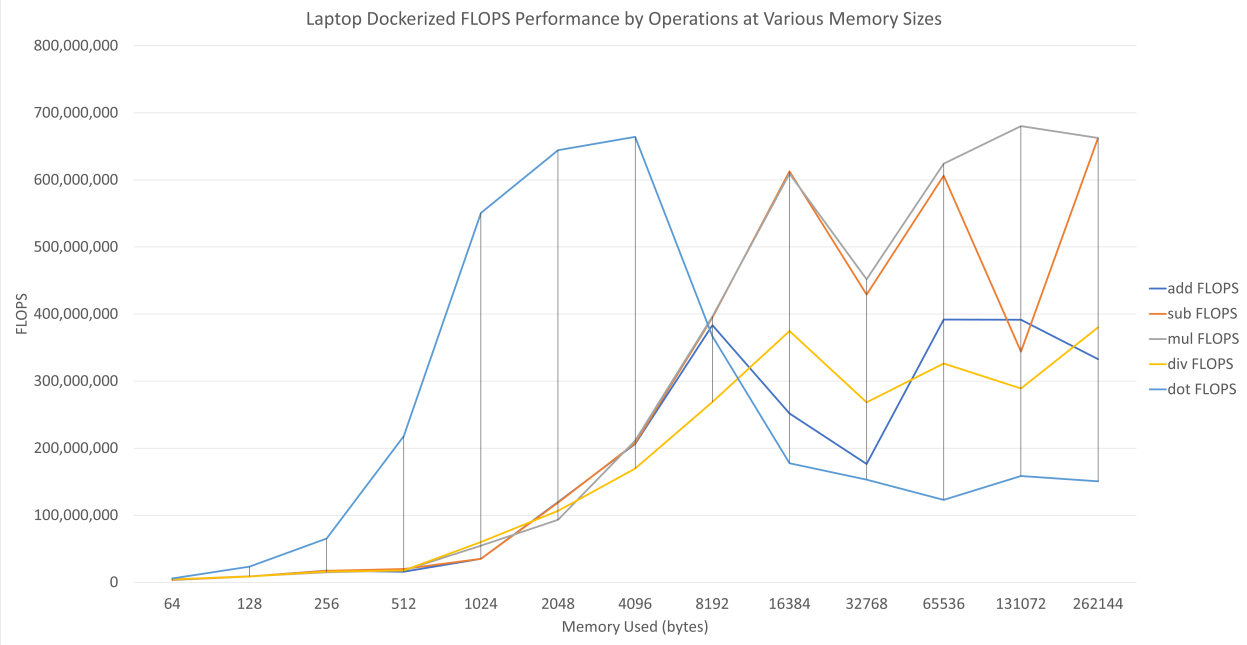

Figure 4.5: Laptop Dockerized FLOPS Performance by Operation at Various Memory Sizes

the most expensive. The benchmarking data can help schedulers to estimate the computation power of executors.

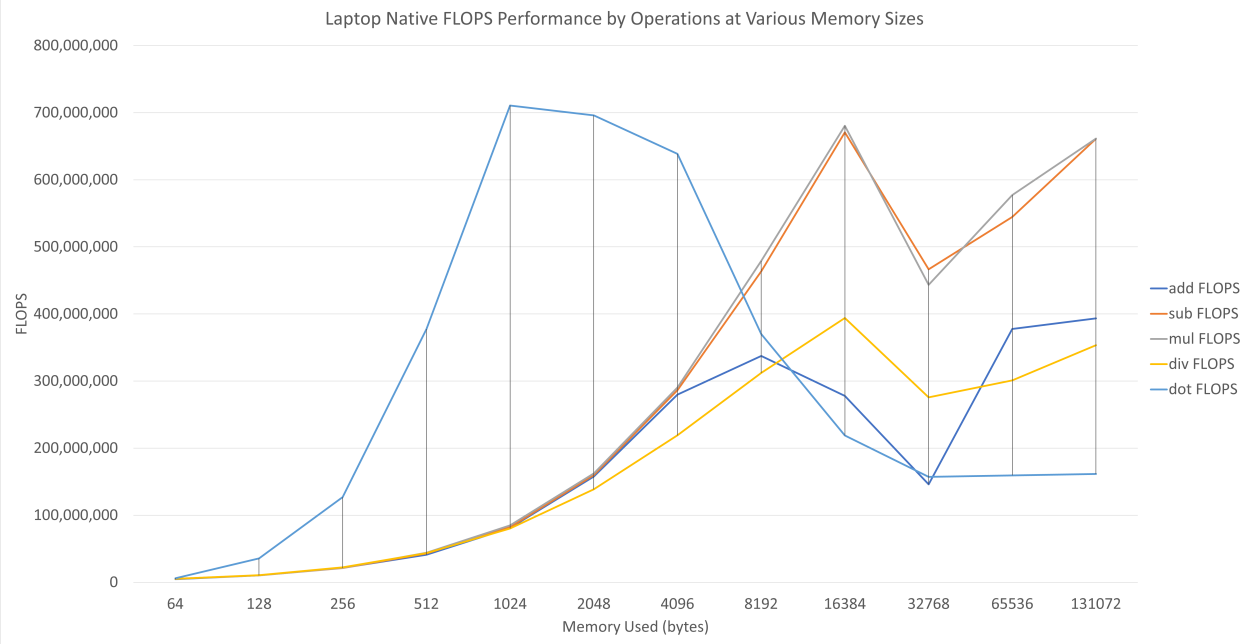

Figure 4.6: Laptop Native FLOPS Performance by Operation at Various Memory Sizes

Using these results, we justify our distribution of operations. SGD training is overwhelmingly composed of multiplication operations. This allows our analyst to be responsible for a small fraction of the operations. To verify this, we created a python script that calculates the expected number of computations for each node type. Utilizing this and a 128 neuron layer using the MNIST dataset as input, we find that the analyst, which only aggregates the returned outer products, is responsible for 0.02% of the total computations for the layer under our offloading approach.

4.6 Example Use Cases

4.6.1 Distrusting Party Using a Cloud Provider

In this scenario, we present the case of a distrusting party using a cloud provider. This distrusting party can take many forms. It could be a researcher, business, or organization that does not have the capacity to train a model. Additionally, they do not have any guarantees as to the trustworthiness or security of the hardware provided by the cloud provider.

While privacy of the data and model is an issue, the distrusting party still wishes to leverage the computation power of an external entity in a distrusting manner. This issue is further exacerbated by the additional computation coming at a cost in cloud computing environments which disincentivizes approaches like homomorphic encryption. An implementation of DCL for the above presented setting could look like the configuration illustrated in Fig. 4.7. A web server handles a request that provides data that will be passed to the model. The web server performs basic sanitization and checks the data prior to forwarding it to cloud provider. Within the cloud provider, the distrusting party has provisioned many devices. These devices make up the analyst, schedulers, and executor nodes of a DCL network. The most important device to consider is the analyst node. The analyst node takes the form of a hardened, software enclave, as only the analyst node requires security considerations. It stores the model which is combined with the received data to create computations. These computations are passed to a determined number of schedulers nodes whom provide task scheduling services to their constituent executor nodes. Executors receive the computations from their scheduler node, perform the operations, and return the response. Schedulers similarly gather the responses from their executors and issue a joint response to the analyst. Lastly, the analyst gathers all responses from all schedulers, unshuffles the responses, aggregates them, and utilizes the result as input to the next layer, repeating this process until a defined number of epochs and a trained model is yielded.

4.6.2 Edge Computing

In our recent work [11], we explore DCL applied to an edge computing environment. With the increasing rise of 5G networks, high bandwidth connections at the edge will allow distribution of large quantities of data to edge devices quickly. This bridges the gap to enable limited performance devices on the edge to convert idle cycles to meaningful work. In addition, the hierarchical organization of DCL networks lessens the link requirement between edge nodes and edge devices. Fig. 4.8 shows how DCL may be applied to an edge computing setting.

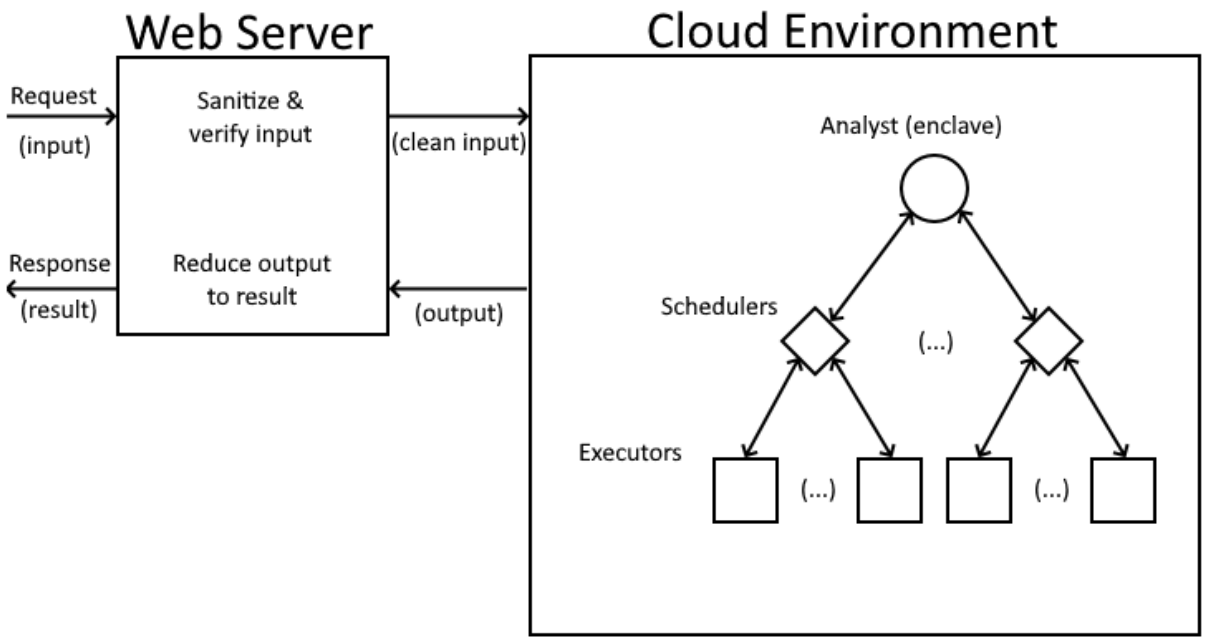

Figure 4.7: A possible web server to cloud computing configuration of DCL.

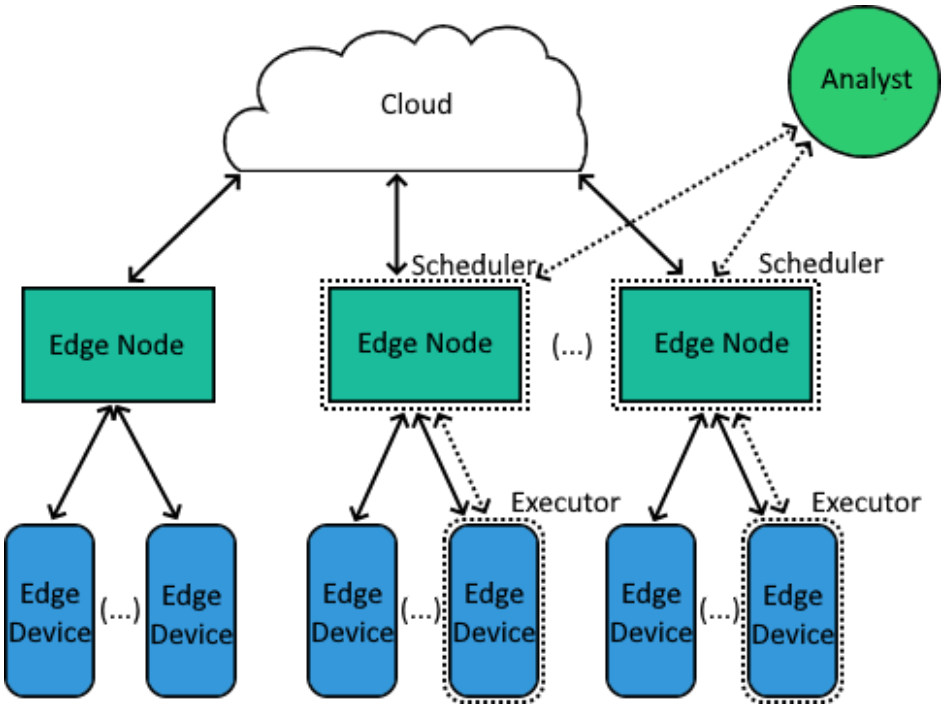

Figure 4.8: DCL applied to an existing edge computing service.

4.6.3 Public Crowdsourcing

In this scenario, we present a similar use case of a distrusting party wishing to leverage crowdsourced computing power. We consider the folding@home project a great source of inspiration of the possibilities presented by donated computation. When exposing data publicly, the privacy of the input and model is an issue and external computing power must be used in a distrusting way. The DCL approach is ideal for this scenario as computations are scaled down to neuron resolution, thus allowing even connected devices with limited system capabilities to contribute. Because load balancing is deferred to schedulers, users can chose to donate some percentage of their CPU time running an executor instance.

5 Conclusion and Future Work

5.1 Conclusion

This thesis presented a DCL framework for the offloading of neural network training in a privacy-preserving manner. The framework explores privacy of data in a novel value-aware manner and additionally explores determinism as a means to provide computation verification and detection of faulty and misbehaving nodes through the issuance of computations to multiple executors. The shuffling scheme provides a novel solution to privacy that is computationally hard to reverse. We proposed a partitioning scheme that acknowledges and groups data by the data dependencies of operations. We additionally presented three different methods for the tasks of shuffling and aggregating values. We showed that the privacy guarantees are quantifiable and are backed by the computational difficulty to reverse the multidimensional shuffling operation applied prior to offloading of tasks. Lastly, while the communication cost of this approach is high, the division of operations and networking structure of nodes enables devices with limited computational performance and network bandwidth to meaningfully to contribute to a DCL network where other approaches cannot.

5.2 Future Work

Our work makes us wonder if the nondeterministic aggregation scheme commonly present in GPU powered training leads to a significant source of entropy in calculations for sufficiently dense layers. It is well established that GPU calculations are indeterministic due to floating point addition and multiplication not being associative combined with GPU thread scheduling being indeterministic. Indeterministic aggregation leads to variance in results. Although admittedly small in magnitude, it is a source of entropy in calculation. Its magnitude directly scales with number of values to be aggregated, thus becoming more of a problem for layers with sufficiently high dimension. These differences in the produced neuron output cascade with each layer, where a small change in the weighted sum prior to activation

can create an avalanche effect across layers. Some activation functions are more sensitive to this noise than others. This presents the possibility of incorrect classification due to nonadherence of strict order of operation. In cases where a network is reaching convergence and error values are low, this entropy poses the most danger for a mostly if not fully trained model. Perhaps the use of a deterministic solution for the last 5-10% of training could lead to better convergence than the sole utilization of an indeterministic solution? This question is left as a possible direction for future work.

In its present state, DCL's scope is limited to feed forward models. The data dependencies of more complex neural network architectures should be considered in future work. Our work does not consider convolutional neural networks, which utilize a data dependency that would require an alternative partitioning scheme due to the kernel values being spread across all input features.

Bibliography

[1] Cybenko, George V.. "Approximation by superpositions of a sigmoidal function." Mathematics of Control, Signals and Systems 5 (1992): 455.

[2] Mehdy, M. M., et al. "Artificial neural networks in image processing for early detection of breast cancer." Computational and mathematical methods in medicine 2017 (2017).

[3] Kebria, Parham M., et al. "Deep imitation learning for autonomous vehicles based on convolutional neural networks." IEEE/CAA Journal of Automatica Sinica 7.1 (2019): 82-95.

[4] Afoudi, Yassine, Mohamed Lazaar, and Mohammed Al Achhab. "Hybrid recommendation system combined content-based filtering and collaborative prediction using artificial neural network." Simulation Modelling Practice and Theory 113 (2021): 102375.

[5] Qi, Wen, et al. "Multi-sensor guided hand gesture recognition for a teleoperated robot using a recurrent neural network." IEEE Robotics and Automation Letters 6.3 (2021): 6039-6045.

[6] Nguyen, Thao, Maithra Raghu, and Simon Kornblith. "Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth." arXiv preprint arXiv:2010.15327 (2020).

[7] Larson, Stefan M., et al. "Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology." arXiv preprint arXiv:0901.0866 (2009).

[8] Hutson, Matthew. "Artificial intelligence faces reproducibility crisis." (2018): 725-726.

[9] Faria, José M. "Non-determinism and failure modes in machine learning." 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2017.

[10] Cooper, A. Feder, Jonathan Frankle, and Christopher De Sa. "Non-Determinism and the Lawlessness of Machine Learning Code." Proceedings of the 2022 Symposium on Computer Science and Law. 2022.

[11] Lewis Brown and Qinghua Li. "Privacy-Preserving and Secure Divide-and-Conquer Learning" IEEE/ACM Symposium on Edge Computing (SEC), Workshop on Edge Computing and Communications (EdgeComm) (2022).

[12] Hunter, D. et al. "Selection of Proper Neural Network Sizes and Architectures—A Comparative Study," in IEEE Transactions on Industrial Informatics, vol. 8, no. 2, pp. 228-240, May 2012, doi: 10.1109/TII.2012.2187914.

[13] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[14] Mitchell, Melanie. "Why AI is harder than we think." arXiv preprint arXiv:2104.12871 (2021).

[15] Wang, Haohan, and Bhiksha Raj. "On the origin of deep learning." arXiv preprint arXiv:1702.07800 (2017).

[16] Lenail, Alexander. "NN-Svg." NN SVG, https://alexlenail.me/NN-SVG/.

[17] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004).

[18] Naehrig, Michael, Kristin Lauter, and Vinod Vaikuntanathan. "Can homomorphic encryption be practical?." Proceedings of the 3rd ACM workshop on Cloud computing security workshop. 2011.

[19] Geyer, Robin C., Tassilo Klein, and Moin Nabi. "Differentially private federated learning: A client level perspective." arXiv preprint arXiv:1712.07557 (2017).

[20] Bonawitz, Keith, et al. "Practical secure aggregation for federated learning on user-held data." arXiv preprint arXiv:1611.04482 (2016).

[21] Akhtar, Naveed, and Ajmal Mian. "Threat of adversarial attacks on deep learning in computer vision: A survey." Ieee Access 6 (2018): 14410-14430.

[22] Chen, Xinyun, et al. "Targeted backdoor attacks on deep learning systems using data poisoning." arXiv preprint arXiv:1712.05526 (2017).

[23] Kurakin, Alexey, Ian J. Goodfellow, and Samy Bengio. "Adversarial examples in the physical world." Artificial intelligence safety and security. Chapman and Hall/CRC, 2018. 99-112.

[24] Li, Li, et al. "A review of applications in federated learning." Computers & Industrial Engineering 149 (2020): 106854.

[25] Kairouz, Peter, et al. "Advances and open problems in federated learning." Foundations and Trends® in Machine Learning 14.1–2 (2021): 1-210.

[26] Zhao, Yue, et al. "Federated learning with non-iid data." arXiv preprint arXiv:1806.00582 (2018).

[27] Geiping, Jonas, et al. "Inverting gradients-how easy is it to break privacy in federated learning?." Advances in Neural Information Processing Systems 33 (2020): 16937-16947.

[28] Ma, Chuan, et al. "On safeguarding privacy and security in the framework of federated learning." IEEE network 34.4 (2020): 242-248.

[29] Bagdasaryan, Eugene, et al. "How to backdoor federated learning." International Conference on Artificial Intelligence and Statistics. PMLR, 2020.

[30] Tolpegin, Vale, et al. "Data poisoning attacks against federated learning systems." European Symposium on Research in Computer Security. Springer, Cham, 2020.

[31] E. P. Xing et al., "Petuum: A New Platform for Distributed Machine Learning on Big Data," in IEEE Transactions on Big Data, vol. 1, no. 2, pp. 49-67, 1 June 2015, doi: 10.1109/TBDATA.2015.2472014.

[32] Quiring, Erwin, et al. "Adversarial preprocessing: Understanding and preventing Image-Scaling attacks in machine learning." 29th USENIX Security Symposium (USENIX Security 20). 2020.

[33] Dean, Jeffrey, et al. "Large scale distributed deep networks." Advances in neural information processing systems 25 (2012).

[34] Kang, Yiping, et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge." ACM SIGARCH Computer Architecture News 45.1 (2017): 615-629.

[35] Hao, Pengzhan, and Yifan Zhang. "EDDL: A Distributed Deep Learning System for Resource-limited Edge Computing Environment." 2021 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 2021.

[36] Chakrabarti, Ayan, et al. "Real-time edge classification: Optimal offloading under token bucket constraints." 2021 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 2021.

[37] Qian, Jia, and Mohammadreza Barzegaran. "A Decomposed Deep Training Solution for Fog Computing Platforms." 2021 IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 2021.

[38] Teerapittayanon, Surat, Bradley McDanel, and Hsiang-Tsung Kung. "Distributed deep neural networks over the cloud, the edge and end devices." 2017 IEEE 37th international conference on distributed computing systems (ICDCS). IEEE, 2017.

[39] Zhou, Li, et al. "Distributing deep neural networks with containerized partitions at the edge." 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19). 2019.

[40] Chen, Jianmin, et al. "Revisiting distributed synchronous SGD." arXiv preprint arXiv:1604.00981 (2016).

[41] Kang, Yiping, et al. "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge." ACM SIGARCH Computer Architecture News 45.1 (2017): 615-629.

VITAE

2016-2018	Full Stack Developer CTTP - University of Arkansas Research Center Fayetteville, AR USA
2016-2018	Full Stack Developer Thrive GmbH Bietigheim-Bissingen, Germany
2018	Software Developer Intern Walmart Bentonville, AR USA
2018-2020	Full Stack Developer CTTP - University of Arkansas Research Center Fayetteville, AR USA
2019	B. S. in Computer Science University of Arkansas Fayetteville, AR USA
2020-2022	Graduate Research Assistant University of Arkansas Fayetteville, AR USA
2022	Graduate Teaching Assistant University of Arkansas Fayetteville, AR USA
2022	Graduate Certificate in Cybersecurity University of Arkansas Fayetteville, AR USA
2022	M. S. in Computer Science University of Arkansas Fayetteville, AR USA