Scaling a convolutional neural network based Flower counting application in a distributed GPU cluster

A Thesis Submitted to the
College of Graduate and Postdoctoral Studies
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Mohammed Rashid Chowdhury

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Or

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9
Canada

Abstract

Taking advantage of modern data acquisition techniques, the researchers of P^{2}IRC located at the University of Saskatchewan developed an application to monitor the status of the flower growth during different phases of the blooming period and the yield prediction of canola crops. Though the application could predict the near accurate number of flowers in a few scenarios, its inability to function under challenging situations such as misinterpreting sun reflection or dust along the roadside as flowers have motivated the researchers to find an alternative approach of counting flowers. In addition to being a more accurate version, another goal is for the new application to be faster to infer the number of flowers and scalable in distributed environments.

Putting these goals in mind, in this thesis, a Convolutional neural network (CNN) based flower counting application is developed and evaluated taking inspiration from two other previous works where CNN was used for counting heads in dense crowds and predicting the number of bacterial cells from medical imagery. In addition to that, the application addresses the performance and the accuracy goals previously mentioned. Two challenges of using the neural network are (a) the training needs a large volume of data to converge to a low error and (b) the training is computationally expensive and it takes longer time to complete.

To address the first challenge, experiments were run with both "ground truth" estimated using a modified version of the previous flower counter, and ground truth from manual annotation. To address the problem of long training time, two distributed versions of the proposed application were created based on two different distributed architectures called Parameter Server and Ring-AllReduce. Moreover, a detailed explanation of the proposed CNN's architecture along with its memory footprints and GPU utilization is also organized as an in-depth case study to help trace the model's memory consumption during training.

From different sets of experiments, the new flower counter application is observed more accurate than its previous version and both implementations of its distributed versions successfully reduced the total completion time as a result of being linearly scalable when more workers are added to run the training. The Ring-AllReduce version performed slightly better than the Parameter Server, but the differences were not substantial.

Acknowledgements

I want to express my genuine gratitude to my supervisors, Dr. Dwight Makaroff and Dr. Derek Eager. This thesis would not have made possible unless they showed me the right direction and advised me in tough situations that were impossible for me to overcome. Their patience and lenience of tolerating my mistakes is something that I can never forget. Besides my supervisors, I would like to thank my other committee members: Dr. Michael Horsch and Dr. Matthew Links for their valuable suggestions and feedback.

I am also grateful to my parents and my lovely wife who have always taken my side whenever I needed any kind of support. Lastly, I would like to thank God for making everything so easy for me.

This thesis is dedicated to my mother Roushan Akhter Jahan, my father Md. Salahuddin Chowdhury and my dear wife Dil Humyra Sultana Borna. I can never repay what you have done for me. Thank you very much for your love and support.

- Rashid.

Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CONV	Convolutional
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DoH	Determinant of Hessian
FC	Fully Connected
FCRN	Fully Convolutional Regression Networks
GPU	Graphics Processing Unit
gRPC	gRPC Remote Procedure Calls
HDFS	Hadoop Distributed File System
KDE	Kernel Density Estimation
MAE	Mean Absolute Error
MCNN	Multi-column Convolutional Neural Network
MKL	Math Kernel Library
MPI	Message Passing Interface
MSE	Mean Squared Error
NCCL	NVIDIA Collective Communications Library
PS	Parameter Server
POOL	Pooling
P^2IRC	Plant Phenotyping and Imaging Research Centre
RDMA	Remote Direct Memory Access
SIMD	Single Instruction Multiple Data
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
WFBP	Wait Free Back-Propagation
VGG	Visual Geometry Group

1 Introduction

Extracting useful features from image datasets has been considered a complex, yet essential, task in the field of computer vision [22]. Over the past few years, the resurgence of deep neural network (DNN) research has enabled machine learning researchers to demonstrate unprecedented improvement in complex feature extraction from large image datasets [3], [14], [18], [37]. A Convolutional neural network (CNN) is a specialized neural network architecture that is mostly applied to analyze visual data. With the growing size of high dimensional images and the complexity of neural network architectures, the overall training duration has become an important concern in the application development life cycle [17]. In general, training a convolutional neural network requires processing large number of floating point parameters, for which hardware accelerators such as GPUs (Graphics Processing Units) are considered well suited, because of their SIMD based (Single Instruction Multiple Data) architecture. GPUs are now widely used in both industry and the research community for training neural network applications and chip manufacturing companies such as NVIDIA and Intel are releasing improved and powerful versions of GPUs each year to keep up with the growing demand [8]. In addition to GPUs, recently, other hardware accelerators such as customized Application Specific Integrated Circuits (ASIC) (e.g, TPU (Tensor Processing Units)) have gained popularity among machine learning practitioners for improving the overall training completion time.

Though GPUs can boost performance for training CNN applications, there are certain cases when the training process becomes so computationally intensive that it takes days, sometimes even weeks to converge on a single machine with multiple GPUs [4]. In such cases, the training needs to become distributed. Distributed training in multi-GPU clusters doesn't always guarantee superior performance since achieving a similar convergence rate as a single node often requires substantial engineering efforts. Scaling a CNN model across multiple nodes involves proper synchronization among parallel workers and careful handling of communication complexities associated with sharing model parameters.

In general, for data parallel CNN models, two methods of parameter sharing are to use a centralized Parameter Server architecture, or to use MPI (Message Passing Interface)-based collective communication methods [35]. Initially, the Parameter Server-based architecture was considered as the standard for sharing parameters across nodes in distributed clusters and it is widely supported by almost all deep learning frameworks. However, a recent study [29] showed that a ring-based communication pattern called Ring-AllReduce (from the family of collective communication operations) is capable of efficient gradient aggregation and gradient synchronization among multiple GPUs distributed across multiple nodes. Furthermore, it claimed to offer better scalability and resource usage compared to the Parameter Server architecture in distributed

settings.

Alternatives to the Parameter Server architecture have been explored in few works done in the past [3], [15], [33]. Motivated by all these studies, one of the key objectives of this project is set to conduct a case study of performance comparison between Parameter Server and Ring-AllReduce. The results can inform the researchers of P^{2}IRC (Plant Phenotyping and Imaging Research Centre)[1] with respect to how to implement CNN training so as to achieve near linear scaling in distributed GPU clusters.

1.1 Thesis Motivation

The Plant Phenotyping and Imaging Research Centre (P^{2}IRC) at the University of Saskatchewan runs innovative research and training programs to provide solutions to national and global food security. Using advanced image acquisition technologies, socioeconomic analyses and high-performance computing, P^{2}IRC combines the sciences of plant bioinformatics and genomics with crop phenotyping to design innovative crop breeding solutions.

The University of Saskatchewan's laboratory for Performance Studies of Distributed Computing Systems (DISCUS lab) is associated with the high-performance computing section of the P^{2}IRC project. As a part of the project, our lab chose to optimize compute-intensive image processing applications. One application of interest in P2IRC is a Flower Counter application that estimates the number of canola flowers in field images. In addition to being an indicator of the yield potential, flower counts can be used for monitoring the timing of the blooming period and the status of flower growth within those periods.

The existing version of the application [11] approximates the total number of canola flowers using a Determinant of Hessian (DoH) blob detection algorithm. The images used for that work were taken mostly during the summer season of 2016, from cameras deployed in different canola test plots. Each day, the cameras periodically (once per minute) took pictures of the field from dawn till dusk. The large number of images collected from the fields comprise the dataset for the application. The complex image processing algorithm together with the large dataset that the application needs to process results in a high computation time. The application is sequential in nature and runs entirely on the CPU. Image processing libraries such as openCV and PIL are used to build some of the core parts of the model and converting those parts to run on a GPU requires extra engineering effort. In terms of accuracy, the application seldom predicts accurate numbers. So, an alternative approach is considered that uses a convolutional neural network to regress the number of canola flowers present in images of the fields. Considering the workload and the complexities associated with a neural network architecture, the design of a distributed version of the Flower Counter application needs to be outlined so that the researchers can have results in a more timely fashion.

[1] `https://p2irc.usask.ca/` (accessed September 13, 2019)

1.1.1 Research Questions

This thesis aims at completing two primary objectives. First, building an alternative version of Flower Counter application which is more accurate and faster in inferring the right number of flowers. Second, crafting an optimal version of the newly developed Flower Counter application that is scalable in a distributed GPU cluster. Top priority is triaged to the later objective than the formal.

Scalability can't be achieved by mere addition of computational resources. In addition to proper resource utilization, the communication cost among the workers needs to be minimized so that the speedup achieved with each added computational resource remains close to linear. Keeping that in mind, to make the Flower Counter application distributed, two training architectures are selected. The first one is the standard Parameter Server architecture and the second one is the MPI based Ring-AllReduce algorithm. While developing the distributed version, the performance of each architecture will be analyzed. Though the primary objective of this thesis is to scale the application in a distributed environment, a short study over the intra-machine scalability (multiple GPUs inside a single machine) is also planned. For evaluating the performance of the Flower Counter application, several metrics are used which will be described in detail in Chapter 4. So in a nutshell, this thesis aims to find the answer to the following research questions.

1. Can a Flower Counter using a CNN be developed that is more accurate upon the current Flower Counter?

2. Is the inference time of the CNN based Flower Counter is faster than the current Flower Counter?

3. Can the training of the Flower Counter be sped up effectively by using a distributed training architecture, making use of multiple machines each with a GPU?

    - How does the Flower Counter application scale and how much improvement is gained with each added worker when the Parameter Server is used?
    - How the results with the Parameter Server architecture compares to those when Ring-AllReduce is used?
    - Which one is better and more convenient to work with?

TensorFlow [2] is chosen as the neural network framework for the implementation of the new CNN-based flower counting application. TensorFlow uses a data flow based Graph programming model and it offers both high and low level abstraction which is needed for distributed experiments. The framework is flexible in multi-platforms and it supports the deployment of the trained model in a production environment. Standard Distributed TensorFlow includes an implementation of the Parameter Server architecture, while the Ring-AllReduce architecture is implemented by a recently released framework called Horovod [29]. Horovod depends on OpenMPI and the NVIDIA Collective Communications Library (NCCL) to execute the ring algorithm. More details about these frameworks will be given in Chapter 2.

1.2 Thesis Statement

This thesis determines if an accurate, high-performance deep learning based canola flower-counting application can be developed and deployed on a distributed cluster-based platform and/or with multiple GPUs configured to enhance the speed of operation.

I intend to construct an alternative, accurate version of the Flower Counter application which will be using a convolutional neural network as its backend to predict the number of flowers in a canola field. Moreover, a scalabale version of the application will be developed through comparing the optimal gradient accumulation, synchronization and the communication pattern between two distributed training architectures: Parameter Server and Ring-AllReduce. Different sets of experiments will be conducted in the DISCUS lab's distributed GPU cluster to investigate whether the newly developed Flower Counter application is compatible to the way the distributed architectures handles cluster resources.

The findings from this thesis are expected to help the researchers of P^2IRC to establish scalable, distributed version of the convolutional neural network applications which are too time consuming if run on a single CPU.

1.3 Thesis Organization

The rest of this document is structured as follows. Chapter 2 starts with a brief discussion about previous works related to counting small objects, some of which inspired the CNN model of the new Flower Counter application. In addition to that, the necessary background for understanding the distributed neural network training and the concepts of both Parameter Server and Ring-AllReduce architectures are described in the same chapter. Chapter 3 contains the development details along with the memory footprints of the convolutional neural network used for creating the Flower Counter application. Beside talking about the architecture of the CNN model, a brief statistical review about the dataset is also given. The preparation of the input dataset from the collected raw images and the procedure of creating ground truth from both manually and automatically annotated image data will be described in that chapter too. Followed by that, in Chapter 4, detailed explanations of the experimental design for both distributed architectures are given with the hardware and software configurations of the testbeds. Different metrics used for evaluating the accuracy and the scalability of the application are also reviwed in that chapter. Chapter 5 is about the final results produced from different set of experiments done to measure how accurate the new application is and how does it scales when both of the distributed architectures are used. Based on the resource utilization, the throughput and the completion time of the experiments, a conclusion will be drawn regarding the performance comparison between Parameter Server and Ring-AllReduce. Finally, in Chapter 6, concluding remarks about the contribution of this work towards the P^2IRC project are presented along with the indication of possible future directions.

2 Background and Related Work

In this chapter, the first few sections will introduce the basic concepts of convolutional neural network (CNN) and parallelization techniques for large scale distributed training. Optimal parallelization often requires gradient aggregation and efficient communication between worker nodes. The Parameter Server architecture is widely used and can satisfy both of these requirements. However, recently, a few other gradient aggregation and communication mechanisms have been developed as a substitute for Parameter Server architecture. In this thesis, one of these mechanisms, called Ring-AllReduce will be compared with Parameter Server to determine if the architecture influences the application's scalability. So, a brief introduction along with the related works about Parameter Server and the Ring-AllReduce will be described after the basic introduction of CNN.

The implementation of the Flower Counter application depends heavily on a neural network framework called TensorFlow. To create the Parameter Server version, Standard Distributed TensorFlow was used, whereas the Ring-AllReduce was executed using a framework called Horovod, which was extended on top of Distributed TensorFlow. A brief discussion about the working mechanism and the basic concepts of both of these frameworks will be found in the subsequent section of this chapter. Finally, this chapter concludes with a detail discussion about previous works where convolutional neural networks were used for counting small objects in challenging scenarios.

2.1 Distributed Convolutional Neural Network Training

2.1.1 Convolutional Neural Network

An artificial neural network is a computing system that was inspired from biological neurons that constitute human brain.[1] A regular neural network can be seen as a collection of layers where each layer is comprised of tweak-able connected units that can pass signals to units of other layers. Generally these units are referred to as neurons. When used for classification, a standard regular neural network receives input at its first layer, then transforms the input through a series of hidden layers to produce a prediction of class scores. In this kind of setup, each neuron in a hidden layer is fully connected to all neurons of the previous layer which makes them very computational expensive. A depiction of a regular neural network can be seen at Figure 2.1.

[1] `https://en.wikipedia.org/wiki/Artificial_neural_network` (accessed September 13, 2019)

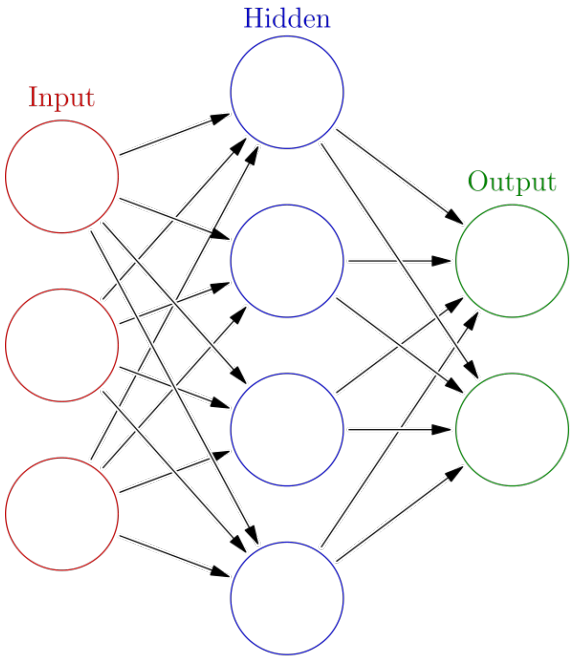


Figure 2.1: Artificial neural network architecture.

For image recognition related tasks, the regular neural network may fit well when the image's resolution is small (such as images from MNIST[2] dataset, where each image has 28 X 28 pixels) but may not be suitable for larger images [9]. For example, when an 100 X 100 sized image (10,000 pixels) is passed to an artificial neural network which has 1000 neurons in its input layer, then there will be 10 million connections just at the first layer. This makes regular neural network processing very compute intensive. In modern architectures, the number of hidden layers and the number of neurons used in each layer are many; hence the name deep neural network.

Convolutional neural network (CNN) is a specialized artificial neural network that was inspired from the study of brain's visual cortex [9]. The most essential building block of a CNN is the Convolution layer (CONV). In this layer, a set of N dimensional arrays also known as filters are set and each filter slides across the width and height of the incoming input of the convolution layer.[3] While sliding, dot products are calculated between the values of the kernel and the values of the input. As a result, a 2-dimensional activation map is created which is basically a map that shows what the filters have responded at every spatial position of the input. Over the training time, the CNN learns filters that gets activated when there are presence of visual features such as patterns, shapes or edges of certain orientations.[4]

Pooling (POOL) is another layer that is used in many cases in conjuction of CONV layers. The main objective of pooling layers is to reduce the dimensionality of each feature map, retaining the most important

[2] `https://en.wikipedia.org/wiki/MNIST_database` (accessed September 13, 2019)
[3] `http://cs231n.github.io/convolutional-networks/#conv` (accessed September 13, 2019)
[4] `http://cs231n.github.io/convolutional-networks/#conv` (accessed September 13, 2019)

features. It can be viewed as a way of sub-sampling the inputs (feature maps if the POOL layer is placed after CONV layer) in order to reduce the number of trainable parameters. Using POOL layers helps reduce the computation load and memory usage. Similar to CONV layers, there are fixed sized filters that are slid across the height and width of inputs. While sliding, instead of calculating the dot product, the kernel now takes the max value of the input region upon which the kernel is presently positioned. The POOL kernels compared to CONV layer's kernel do not have trainable parameters as there are no values to be set in the kernel to take the max. That means if a POOL layer is positioned after a CONV layer, then each neuron in the POOL layer is connected to a limited number of neurons of the feature map that was produced from the CONV layer. POOL layer is usually destructive in nature, but it saves a lot of computations [9]. Instead of taking the max value, POOL filters can also be set to take the average value. For this thesis, MAX POOL was used.

Terminology

Throughout the rest of this thesis document, there are many important terms that will be used repetitively. A short introduction of those Terminology is given below.

Epoch: Traversing the whole dataset once is referred to as one epoch. Usually, training with the same dataset is repeated several times so that the neural network model can learn from its mistakes in previous epochs and rectify the errors in classification or regression next time when it sees similar data.

Loss function: The loss or cost function can be interpreted as a method that evaluates how good or bad the neural network model is. For example, in linear regression problems, the loss function measures the distance between the predicted value and the original ground truth. If the differences are high, then that means the regression model is not trained well compared to the other models that produced lower errors.

Gradient descent (GD): An iterative method of optimization that gradually tweaks the model parameters in an attempt to minimize the cost function. Successful implementation of gradient descent ensures model convergence to a state where the model errors are minimum and there are no further improvement achievable in terms of accuracy.

Mini-batch stochastic gradient descent (mini-batch SGD): There are few variants of gradient descent that are used while training a neural network model. In this thesis, Mini-batch stochastic gradient descent (mini-batch SGD) is used. Due to the large size of the real life datasets, it becomes very impractical to load the whole dataset in to memory. To tackle this problem, in mini-batch SGD, instead of loading the whole dataset, only a mini-batch of data is randomly chosen and fetched from the dataset during each training step. The gradients are computed for the selected mini-batch and this process is continued so that every datum of the dataset is used atleast once. For example, if a dataset is comprised of N samples, then if five samples are chosen to be randomly picked for each batch, then there will be N/5 training steps to complete an epoch. If the number of epochs is set to X then NX/5 times data will be fetched randomly from the training dataset.

2.1.2 Parallelization Techniques

Distributed training is essential when the model is too big to fit inside the computer memory due to the size of the dataset required to be processed. This is particularly important with respect to limited GPU memory systems. Two parallelization strategies that are widely used for resolving these problem are model parallelism and data parallelism [3], [15], [17].

In model parallelism, different parts of a single neural network model are trained across multiple workers. This procedure requires the model to be split into small separate chunks that are trained by different workers [17]. This method of parallelization is considered challenging as there are always be some kind of sequential dependencies among layers inside the neural network. For example, in a CNN model, if there is more than one FC layer and the layers are placed in different workers, then the worker which holds the second FC layer needs to wait until the worker training the preceding FC layer finishes and sends the output. Vertically splitting the model can resolve the issue in some degree, but it introduces additional cross-device communication overhead that can cancel out the parallelization benefits [9]. Besides that, this parallelization technique is highly dependent on the architecture of the neural network, so in future it will require more engineering effort if the entire model is needed to be changed [9].

In contrast to model parallelism, data parallelism partitions and distributes the data among all the available worker machines. Each worker contains a replica of the same model, but uses a unique partition of the dataset. During training, in each iteration, every worker fetches a different mini batch of data from its allocated data partition and computes the error between the prediction and labeled output for that mini batch. As each worker trains on a different sample, its contribution to error (also known as its gradient) is also different. After each iteration, gradients computed from each worker are used for updating model parameters that are shared by all workers.

Gradient aggregation and model parameter update are done in two ways: synchronous and asynchronous update [9]. In synchronous update, an aggregator waits until every worker finishes computing gradients. Once all the gradients are available, they are aggregated and averaged before updating the shared model parameters. So, before proceeding to the next mini batch, each worker must wait until the aggregator updates the model parameters. Depending upon the computing capabilities or the network capacity, some workers may perform faster than others. In that case, all workers who have finished computing gradients have to wait for slower workers (stragglers). Another downside is that immediately after the parameters are updated, they are broadcast to every worker almost at the same time which may cause bandwidth saturation for the aggregator. Despite these drawbacks, faster convergence is ensured by synchronous updates compared to asynchronous updates as more accurate gradient estimations are achieved at each iteration [6], [9],[41].

In asynchronous update, after receiving each gradient from a worker, the aggregator updates the parameters immediately without averaging or waiting for other workers to finish. There is no synchronization during training and when a long-running worker is done processing a mini batch, the model parameters may have been updated by other workers several times. The simplicity of this approach and the absence of synchro-

nization results in good overall throughput at a cost of poorer gradient estimation. The convergence of the neural network usually takes more steps and consequently, more time, as the workers share different versions of model parameters and there are no guarantees that the computed gradients from individual workers will be converging to the right direction. Severely out-of-date gradients are called *stale* gradients that are suspected as one of the main reasons the training algorithm occasionally diverges on some dataset when asynchronous update is used [9].

2.2 Parameter Server

When scaling a neural network application in a distributed cluster, a Parameter Server is assigned the responsibility of gradient aggregation. A Parameter Server (PS) stores the global model parameters as specialized key-value pairs and lets each worker access them via network communication in a client-server based scheme [6],[41]. In trivial data parallel distributed training, each node in the cluster can take one of two potential roles: worker or a Parameter Server. Workers are responsible to process the data, compute gradients and then send the gradients to Parameter Servers, whereas Parameter Servers as mentioned above, average the gradients, update global model parameters and ensure the paramters are accessible by all the workers. In addition to these responsibilities, a PS also supports replication after aggregation, fault tolerance and dynamic scaling when more nodes are added or removed from the cluster [20]. The design of an ideal Parameter Server is given in Figure 2.2 (a).

In distributed neural network training, Parameter Server design is considered as an active field of research. Over the last few years, Parameter Server designs have become mature through improvements made in areas such as communication mechanisms for parameter synchronization [41], usage of distributed GPUs [6] and combination of specialized PS hardware & software that balances IO, memory and network bandwidth [21].

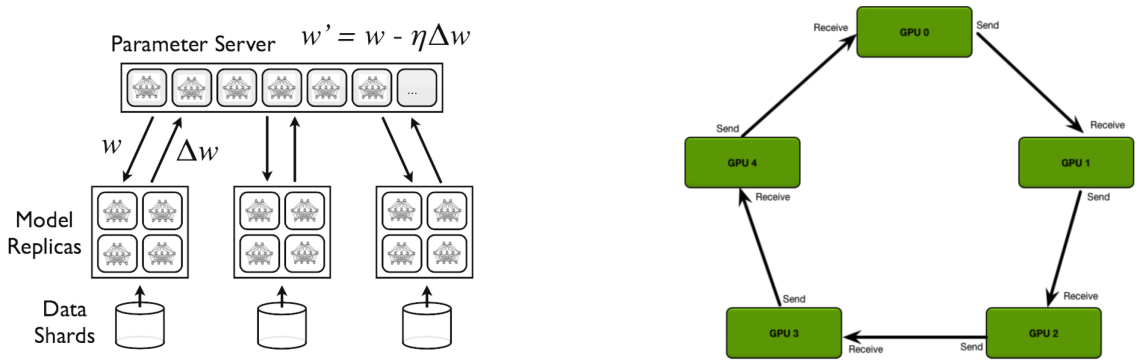


Figure 2.2: Distributed training Architectures (Synchronous SGD) (a) Parameter Server (Li *et al.* [7]). (b) Ring-AllReduce (Zhang *et al.* [42]).

Scaling deep learning application across GPUs distributed among multiple machines was a goal of Cui *et al.* in their project called GeePS [6]. GeePS is a GPU specialized Parameter Server architecture that provides a possible solution of training deep learning application in limited GPU memory. It also handles all

the complexities associated with synchronization and communication among workers that only CPU-based Parameter Servers were considered capable of solving. In GeePS, GPU memory is explicitly managed as a cache for the intermediate state of network layers and model parameters. The Parameter Server library was integrated as an additional module with a modified version of Caffe (a neural network framework) [16]. Each worker process created by Caffe was linked to an instance of the GeePS library upon creation and GeePS shards the parameters across all instances.

Each GeePS instance has a parameter shard and a parameter cache. The parameter caches are used for storing snapshots of local parameters and they read from and write to the parameter shards. GeePS does not support replication of parameter shards; they use frequent checkpointing to offer fault tolerance. There are two parts in each parameter cache, one pinned to the GPU memory and the other to the CPU. When the model is too big to fit inside the GPU VRAM, the least recently used data is swapped to CPU memory. This helped GeePS to demonstrate its scalability by training a 20 GB neural network (5.6 billion connections) in GPUs with 5 GB of VRAM, with CPU memory containing the model parameters and intermediate steps most of the time. Though GeePS demonstrated impressive performance gain using GPU with limited memory, the usefulness of the framework began to dwindle with the advancement of recent GPU architectures that come with more VRAM and more computing units [3].

In the Parameter Server architecture, frequent synchronization of the model parameters takes place over the network which shifts the performance bottleneck from computation to communication [21]. Being motivated by this problem, Zhang *et al.* created Poseidon [41], an efficient communication architecture for training deep neural network applications on Distributed GPUs. Poseidon is built upon two techniques: Wait free back-propagation (WFBP) and Hybrid Communication (HybComm). In general, during back-propagation, at each iteration, gradients computed for each layer are sent back to the Parameter Server after gradients from all layers are computed. Gradients calculated from upper layers do not affect the gradients of lower layers. Wait free back-propagation takes advantage of this design by concurrently scheduling the computation of the lower layers with communication of upper layers during back propagation.

Despite overlapping computation with communication, in situations where commodity Ethernet is shared among communication heavy applications, WFBP could not reduce network overhead. The second technique, hybrid communication, offered a solution of this problem by proposing a structure-aware message passing protocol which utilized the best of both client-server PS scheme and peer-to-peer (P2P) broadcasting architecture. For P2P scheme, sufficient factor broadcasting (SFB) was used. In a CNN, the model parameters as well as the gradients are represented as a set of matrices. For layers where the gradients were suspected to exceed the bandwidth, SFB was used to reduce the number of parameters by decoupling the gradient matrix into two separate vectors (called sufficient factors (SF)). Upon receiving the sufficient factors, the gradients could be reconstructed by a simple multiplication among the received factors.

As neural network models are usually predefined, the size and number of parameters can be measured based on the input data. Poseidon initially finds out potential transferable parameters using wait-free back-

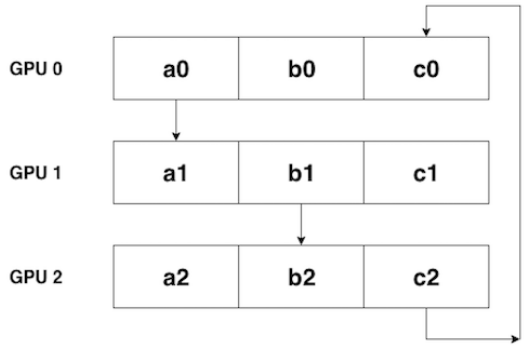
propagation. Then, it makes an approximation of communication overhead by measuring the size of the model and the cluster. Finally, based on the approximation, Poseidon uses hybrid communication to choose the optimal method of transferring the parameters, before the communication happens. The authors mentioned that Poseidon can be combined with GeePS to enable better speedup and can cover a wide range of setups in distributed clusters of GPUs. The experiments revealed Poseidon's effectiveness on achieving near-linear scaling of various neural network models on multiple GPUs in environment with limited bandwidth.

Though the Parameter Server architecture is still considered as a standard approach for aggregating and averaging gradients, it requires ML practitioners to tackle challenges such as identifying the right ratio of workers to Parameter Servers and taking on the complexity of code restructuring for a distributed cluster [29]. Finding the optimal ratio requires careful observation of network and computational usage. If multiple Parameter Servers are used, this configuration is most likely to over-saturate the network interconnect; on the other hand, if only one Parameter Server is used, it may become a computational or network bottleneck. In addition to this problem, transforming existing neural network application to use distributed training may require a steep learning curves for developers. This may lead to spending more time in code restructuring than original application development.

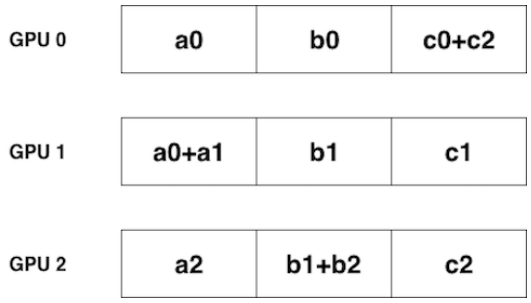
2.3 MPI based Aggregation - Ring-all-reduce

As an alternative to the centralized client-server-based PS approach, a different MPI based method called Ring-AllReduce has recently demonstrated better efficiency in scaling neural network applications in distributed GPU clusters [29], [10], [28]. The Ring-AllReduce algorithm was proposed by Patarasuk and Yuan [25] to provide a contention-free communication mechanism on clusters with different networking technologies and nodal architectures for processing big datasets. The idea was later adopted and extended by the researchers at Baidu to work in a distributed cluster [10] where GPUs were placed in a logical ring. The idea can be better understood by the visual representation found from Zhang [42] which is included in Figure 2.2(b). It can be seen from the figure that no central server is responsible for aggregating the gradients. Each worker or GPU only receives gradients from predecessor neighbour and sends the locally computed gradient to its successor GPU on the ring. If the number of GPUs is assumed to be N, then the mini-batch data is divided into N chunks. In each training step, every GPU gets its own split of mini batch data, computes the gradients, sends the gradients to the next GPU and receives the gradients from previous neighbour. During the first N-1 iterations, which constitutes the scatter-reduce phase, the received gradients are added to the local gradient stored in the GPU buffer. Different stages of scatter-reduce are portrayed in Figure 2.3 (a-c). The figures are reproduced based on the understanding of the examples given in Zhang [42]. After the scatter-reduce phase is finished, the next phase called AllGather runs for the next N-1 iterations. During this phase, each GPU exchanges its stored values with other GPUs. In contrast to scatter-reduce, during AllGather, instead of summation, simple data copy takes place (see Figure 2.3 (d-f)). So, in total, 2(N-1)

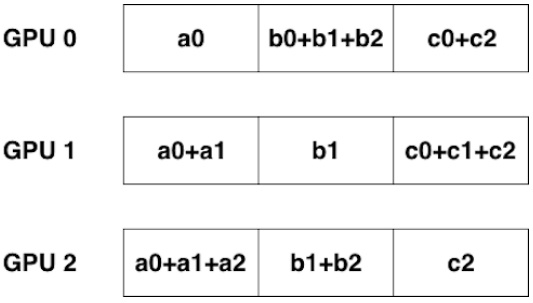
iterations are required to complete updating the gradients.



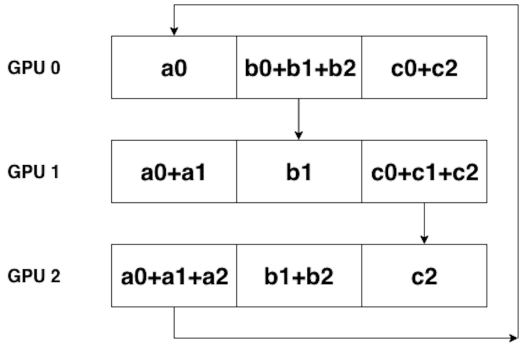
(a) Scatter reduce: Dataflow for First iteration.



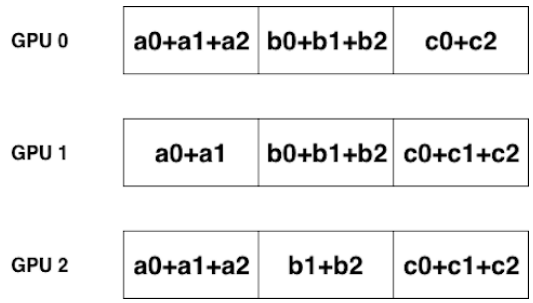
(b) Scatter reduce: Dataflow after First iteration.



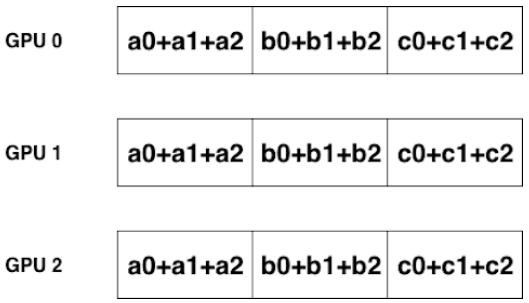
(c) Scatter reduce: Dataflow after N-1 iterations.



(d) AllGather: Dataflow at the beginning of All-Gather (at Nth iteration).



(e) AllGather: Dataflow after Nth iteration.



(f) AllGather: Dataflow after 2(N-1) iterations.

Figure 2.3: Dataflow of different stages in Ring-AllReduce.

At the time of writing, from a review of the related literature, two implementations of Ring-reduce-based solutions are found: baidu-allreduce[5] and Horovod [29]. Horovod was influenced by baidu-allreduce and in fact, in the initial development phase, Horovod adopted baidu-allreduce as the draft implementation of the Ring-AllReduce algorithm for TensorFlow [29]. However, later, Horovod replaced the baidu-allreduce implementation with NCCL (NVIDIA Collective Communications Library) which provided a highly optimized

[5] `https://github.com/baidu-research/baidu-allreduce` (accessed September 13, 2019)

version of Ring-AllReduce for neural network training in distributed GPU clusters. More information and detail about Horovod is given in Section 2.7.

Ring-AllReduce is not the only solution used as substitute for the Parameter Server architecture in distributed training. In previous work [3],[12] and [15], binomial-reduction-tree-based communication patterns were used as an alternative of Parameter Server. Binomial-reduction-tree is a symmetric parallelization approach where all worker machines communicate using a binary tree-like pattern. In a recent work developed at Facebook research division [12], the reduction-tree-based communication pattern was used to achieve 90% scaling efficiency while training ResNet-50 model [14] within an hour. The main objective of that work was to propose an empirical solution for the optimization difficulties associated with training large minibatches. The authors successfully demonstrated that while training the ResNet-50 model, increasing the batch size from 256 images to 8192 can maintains its accuracy if the learning rate is adjusted as a function of the mini-batch size. The ResNet-50 model was scaled from 8 to 256 GPUs, which was made possible by using an optimized version of within and across-machine collective communication library.

Three phases of communication were mentioned in the work. In the first phase, the buffers from the GPUs within a worker machine were summed into a single buffer before sending them to other worker machines. In their case, each machine was equipped with eight GPUs, so the buffers from these GPUs were summed and then represented as a single buffer. In the second phase, the resultant buffers from each worker were shared and then summed across the cluster. Finally in the last phase, the results were broadcast back onto each GPU of each worker machine.

The local reduction and broadcast mentioned in phase one and three were implemented using NVIDIA's NCCL (NVIDIA Collective Communications Library) library. For phase two (inter-server communication), the authors mentioned using recursive halving and doubling algorithm [26], [35] which consists of a reduce-scatter collective followed by an all-gather. In the first step of reduce-scatter collective, the workers communicate in pairs (worker 0 with 1, worker 2 with 3 and so on). The workers send and receive different halves of their input buffer. For example, worker 0 sends the second half of its input buffer and receives the first half of the buffer from worker 1. Upon receiving, before going to the next step, a reduction over the received data is performed. In the next step, the worker 0 receives first half from worker 2 which now contains the partial sum of the first halves of both worker 2 and 3. Now after performing the sum, worker 0 contains the final sum of the first halves of the buffers received from all workers. If this procedure is followed at each step, the number of machines doing work is cut in half but the distance to the destination is doubled.

After reduce-scatter, all-gather collective starts. This stage is about broadcasting the final sum to all other workers. Broadcasting is done by retracing the communication pattern from the reduce-scatter phase in reverse. In this way, after broadcasting finishes, every GPU inside every worker machine contains the updated model parameters. The recursive halving and doubling algorithm consists of $2\log_2(N)$ communication steps as opposed to $2(N-1)$ of Ring-AllReduce [12]. In a bandwidth limited scenario, the authors compared both of the communication patterns and revealed that using recursive halving and doubling led them to gain 3X

speedup over the ring algorithm for communication among workers.

The reduction-tree-based apporach was used in another work called S-Caffe [3]. In that work, the authors claimed to fully exploit the available resources of modern GPU clusters by using three of their proposed co-designs which were used in conjunction to achieve the optimal scalability. In their first proposed method, the authors added support of a new CUDA Aware MPI based parallel reduction design with the Caffe framework by combining GPUDirect RDMA[6] and GPU kernels to support large-sized GPU-based reduction operations. GPUDirect RDMA is a technique that utilizes standard features of PCI Express to enable a direct path for data exchange between the GPU and a third-party peer device. Examples of third-party devices are storage adapters, network interfaces, video acquisition devices, etc. The proposed CUDA-Aware MPI Design was referred to as S-Caffe Basic (SC-B). The second co-design was about utilizing the opportunities to overlap the computation and communication phases to maximize efficiency. The procedure was made possible by taking advantage of MPI-3 semantics especially the non-blocking collective (NBC) operations which helped to broadcast data for the next forward pass (before the data was even requested) while processing current forward-backward pass. This co-design was referred to as S-Caffe Optimized Broadcast (SC-OB). Finally, while describing the motivation of the last co-design, the authors expressed that the flat, single Binomial-Tree-based algorithm falls short in the context of GPU based communication and large-scale reductions. As a solution to that problem, a new hierarchical mechanism called Efficient DL-Aware Hierarchical Reduce (HR) for MPI communicators was proposed, which was a slight modification of the basic tree-based reduction technique. In conjunction with HR, an overlapped gradient aggregation technique called SC-OBR was also designed, together named as SC-OBR+HR. Three of these proposed methods were compared seperatly. SC-OB produced good result compared to SC-B but combined with SC-OBR, HR turned out to be best among the rest of the co-designs for GoogLeNet-based training on 160 GPUs [3].

The binomial-reduction-tree seems promising as another alternative of Parameter Server. However, due to the simplicity of using Ring-AllReduce (thanks to the high level Horovod API) and time constraint, the option of using binomial reduction tree is left as a future work for this project.

2.4 CNN based Counting Models

Using CNN as a method for counting instances has been explored in areas such as counting microscopic cells [38], bacteria [5], estimating number of vehicles in traffic jams [23], parking lots and counting people in huge concerts or in community parks [27], [36], [40], [43]. An efficient CNN-based counting application needs to be flexible in variable situations, such as different perspective view of the objects that users are interested in, type of the objects to be counted, distribution of the objects, overlapping or occlusion in dense areas and finally training, testing speed. The counting techniques may not be general in all cases, as the approach to solve problems such as counting a dense crowd in a concert or at a stadium can turn out to be completely

[6] `https://docs.nvidia.com/cuda/gpudirect-rdma/index.html` (accessed September 13, 2019)

different compared to counting the number of people in a family photo [13]. Accurately counting objects is a challeging task when individual localization and detection is necessary for instances which are very small in size and when the small instances overlap. As an alternative to this hard task, the same counting problem was cast as predicting the density map where the integral over a region of the map gives the count of objects within the selected region [19]. An integral over the entire density map gives the total object counts for the given map. This approach of counting objects requires training images with dot annotation (a process of placing dot at the center of the objects of interest) instead of merely providing the overall object counts in training images.

This methodology inspired Xie *et al.* [38] to build an automated CNN-based cell counting application for microscopic images where the authors showcased the applicability of predicting a density map in place of traditional segmentation based methods. According to the authors, solving the cell counting problem can be approached from two different directions: one is detection-based approach where prior detection or segmentation of the object instances are required; the other is based on density-based estimation which doesn't require any prior detection or segmentation at all. Out of these two different methods, the authors preferred the density-estimation-based approach over single-cell segmentation method as the former had advantages in certain situations such as cell clumping or overlapping of cells where the segmentation based approach doesn't work satisfactorily.

The authors demonstrated two different Fully Convolutional Regression Networks called FCRN-A and FCRN-B which, for a given cell image, infer the density map for the input image. The model of both networks were inspired by the VGG-net [30] architecture. After the predicted density map was produced from the final layer of the CNN, the loss was calculated based on the mean square error (MSE) between the output heat map and the ground truth density map. Based on the loss function, back-propagation and stochastic gradient descent were used for minimizing the error between predicted and ground truth density maps. The procedure of creating a ground truth density map was described as putting a dot in the center of each cell in all the training images and then running a Gaussian kernel at each dot (the center of each cell) with radius set to two. Then superposition of these Gaussians forms a density surface which is considered the density map. During training, the authors preferred cutting the large images into small patches so as to increase the amount of data for training.

Crowd counting is another application where a CNN can be applied for density estimation. According to a recent survey conducted by Sindagia and Patel [31], CNN-based crowd counting models can be classified into several categories such as scale-aware, context-aware, patch-based and whole image-based models. Scale-aware models are more robust to variation of scale and work well with arbitrary crowd density as well as arbitrary perspective view from capturing cameras. To achieve this robustness, different techniques such as multi-column or multi-resolution architectures are used. For context- aware models, local and global contextual information such as geometric shape or size of the objects, exclusion of certain background noises, inclusion of additional regions around the subject of interest, etc. are incorporated during training. This

kind of model sometimes requires a perspective map of the input image to predict the crowd count. For patch-based models, the CNN is trained using small patches cropped from the original input images. The patch can be of variable size or fixed size. During the inference stage, a sliding window is moved over the whole test image and the predictions are calculated for each window. Later, the predictions for each window are aggregated to obtain the total count of the objects in the image. Lastly, for whole-image-based models, the inference is conducted on the whole image instead of computationally expensive sliding windows. The patch-based and whole-image-based models can also be coupled with scale and context-aware models. Among these various classifications of CNNs, a scale-aware model was found to be more appropriate for the research in this thesis, because the images of the canola fields were captured from different cameras and the perspective view of the canola field was not the same.

As the perspective view related problem was addressed and resolved with scale-aware multicolumn-based CNN, a similarly architected CNN was used as the main structure of the flower counter application in this thesis research. Within the last few years, a handful of scale aware multicolumn-based CNNs were developed, some of which significantly improved the accuracy of counting crowds. One such model is Multi-column Convolutional Neural Network (MCNN) proposed by Zhang *et al.* [43]. This simple, but effective CNN model is comprised of three parallel shallow CNNs, each of which utilizes filters with different sized receptive fields. Each comprising CNN was represented as a column and the resulting feature maps from each CNN were merged together to predict the final density map.

The reason behind using filters with different-sized receptive fields was explained as a solution to perspective distortion; in a crowded scene, it is usual that different-sized heads will be captured and a single filter with fixed size receptive field will be unlikely to extract the characteristics of crowd density at different scales. Except for the filter size and number of filters, the rest of the network structure for all columns was kept the same. The loss function was defined as the Euclidian distance between the estimated density map and the ground truth. The authors also explained how the quality of the density map plays a significant role and they proposed a geometry adaptive kernel to produce the ground truth density map.

The traditional method of generating a density map does not work well with the crowd density, because the head size becomes smaller in far distances and becomes bigger in near distances from cameras. Simply marking a dot into the center of the head as annotation and running a gaussian kernel with a fixed spread parameter can work well with sparse bigger heads but may not work well with small heads as the fixed spread parameter of gaussian kernel will cause overlapping of areas. To mitigate that problem, the spread parameter of the gaussian kernel needed to be set as a variable. The difficulty of accurately measuring the size of each head was addressed with the proposed geometric adaptive kernel, which solved the problem by relating the head size as the distance between the centers of two neighboring persons in crowded scenes. So, for each head, the distances of its k-nearest neighbors were calculated and the mean of the distances was set as the spread parameter of the gaussian kernel. For evaluating the MCNN model, both the mean absolute error (MAE) and the mean squared error (MSE) were used.

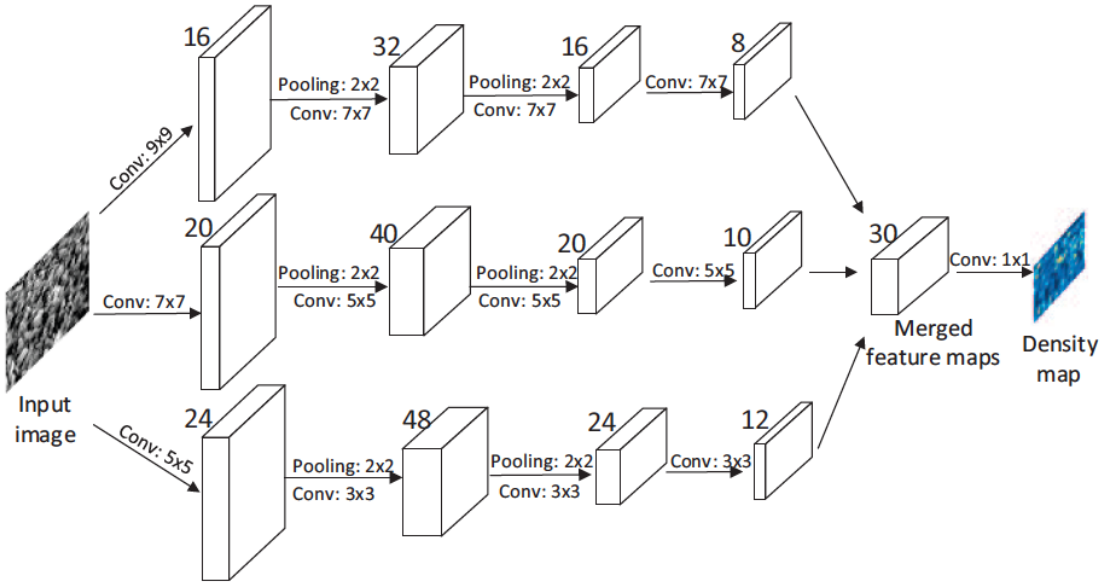


Figure 2.4: MultiColum Convolutional neural network architecture. ([43]

2.5 Overview of TensorFlow

TensorFlow is an open source machine learning framework that was built by the researchers from the Google Brain team[7] for the purpose of implementing and deploying large scale machine learning model in wide variety of heterogeneous environments [1]. The framework provides a dataflow-based programming abstraction which has made the process of creating neural network applications more flexible by offering both local and distributed training options for the developers and researchers who don't want to get involved into complex theories and behind-the-scenes working mechanisms associated with local or distributed training. The construction of the dataflow graph is wrapped by a high level scripting interface which lets users experiment with user-defined optimization algorithms and neural network architectures without modifying the core system. Distributed TensorFlow is robust in the sense that it can efficiently handle hundreds of GPU-enabled servers for faster training and allows the trained model to be deployed for inference on various platforms ranging from small mobile and embedded devices such as smart-phones, tablets and IoT up to large-scale distributed clusters in a datacenter [1], [2].

2.5.1 Core Concepts

To represent computations and the operating states of an algorithm, TensorFlow uses a unified directed dataflow graph [2]. In general, the graph is comprised of set of nodes and edges where each node or vertex

[7] `https://ai.google/research/teams/brain` (accessed September 13, 2019)

represents a unit of computation and the edges represents the flow of data (whether consumed or produced by the computation). The computations at the nodes are termed *operations* and the data that flows through the edges are termed *tensors*.

operation: An *operation* can be seen as an abstraction of an computation (e.g., matrix multiplication, or addition). The behavior of an *operation* is determined by a named "type"(such as Const or MatMul) and zero or more compile-time attributes. The attributes are used to make the *operations* polymorphic - it helps determining both the expected data types and arity of input and outputs associated with corresponding *operation*.

tensor: A *tensor* represents data that flows in or out of *operations* in a data-flow graph. It is a multi-dimensional array of various primitive data types (e.g., int32, float32, float 16, or string). A device-specific allocator manages the appropriate size of the backing store buffers for the tensors and uses reference counters which are later used for deallocating memory when the number of references become zero.

variables: A *variable* holds a mutable buffer that can be used to contain shared parameters of the model during training. Shared parameters are made of *tensors*. In most of the cases, neural network applications need their defining dataflow-graph to be rerun multiple times. Unfortunately, *tensors* can't live past the point when a single execution of the graph is finished. In order to persist the *tensors*, this special *operation* called *variables* is used. A *variable* doesn't have any inputs but it returns a *reference handle* to read and write a persistent mutable tensor. The handles can be passed to other *operations* such as *AssignAdd* (equivalent to +=) that updates the referenced tensor after calculation [1].

Session: A TensorFlow application can be thought to be consisted of two discrete sections: defining the computational graph (`tf.graph`) and executing the defined graph (using a `tf.Session`).[8] When the graph is defined, there are no computations that take place. The graph doesn't contain any values; it is just a prototype that shows how the operations are connected to each other.[9] In order to execute the graph, the TensorFlow system needs to be initiated through the creation of a *Session*. The *Session* interface has a primary function called *Run* which evaluates the *operations* that the client program invokes calls to execute. Before evaluating the requested operation, *Run* gets involved into the process of computing the transitive closure of all nodes and finds the order of execution based on the dependencies of the nodes. After the dependencies are found, *Run* feeds the *tensors* to computational nodes (*operations*) that are requested to be evaluated. In most cases, a Session is set up once with a graph and the *Run* calls are executed thousands or millions of times to execute the full or few distinct subgraphs [1].

2.5.2 Execution Model

The dataflow graph of a typical TensorFlow application can consist of multiple subgraphs which can be executed concurrently and can interact amongst themselves via shared variables and queues. The schematic

[8] `https://www.TensorFlow.org/guide/low_level_intro` (accessed September 13, 2019)
[9] `https://danijar.com/what-is-a-TensorFlow-session` (accessed September 13, 2019)

of the dataflow graph of a TensorFlow application is given in Figure 2.5.

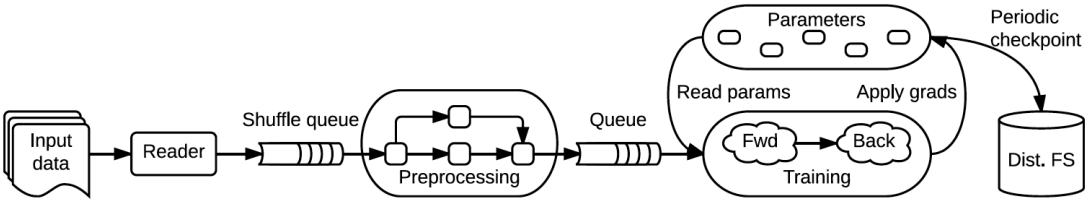


Figure 2.5: The schematic of the dataflow graph of a TensorFlow application [2].

The application starts with subgraphs that are involved in reading inputs and preprocessing the input data which is already fetched from the filesystem. The core training subgraph depends on the processed input batches, and during a training session, a separate checkpointing subgrpah periodically runs to save the trained model parameter in a distributed file system. Checkpointing ensures fault tolerance and the subgraph associated with checkpointing can be run concurrently with the training subgraph. Similar concurrency can be achieved in the beginning subgraphs of the application. When the Reader fills the input queue, the preprocessing subgraph can concurrently start decoding inputs from queues and map callable functions to the sequence of input data.

The communications among the subcomputations are explicitly defined in the dataflow graph, which simplifies TensorFlows distributed execution. Each individual *operation* is assigned to a particular device such as CPU or GPU and each device can execute specific *kernels* on its assigned operation. The term *kernel* means a particular implementation of an *operation* and different devices can have their own implementation for the same *operation*. This is made possible through TensorFlow's kernel registration mechanism where multiple kernels with specialized implementation for a single operation can be registered for particular data types or devices.

2.5.3 Input Pipeline

One of many key factors that plays a crucial role in achieving peak performance is the efficiency of the input pipeline. Due to the faster processing power of the GPUs, the execution time of a single training step has been radically improved. The faster rate of completing the training steps makes the CPU processing more prone to become the bottleneck. To mitigate the problem, TensorFlow offers a feature-rich dataset API called `tf.data` which effectively utilizes the CPU to extract, transform and load the image data from local or remote persistent storage to the accelerators like GPUs or TPUs. To take advantages of these optimization, the dataset API was used for designing the input pipeline for all of the experiments of this thesis. An in-brief introduction of the most prominent features used from the dataset API are given in the following description:

- `tf.data.Dataset.map`: Before getting fed into the CNN model, the input data needs to be extracted

and then transformed into tensors. Extraction and transformation of the input images often involves applying a user defined function to every element of the input dataset. The input images are often independent from one another. The `tf.data.Dataset.map` function takes this advantage to parallelize the pre-proessing step across multiple CPU cores. To specify the level of parallelism, `tf.data.Dataset.map` uses a argument called *num_parallel_calls*. Setting the optimal value of *num_parallel_calls* depends on characteristics of the training data, the cost of the map function, and most importantly, the hardware architecture. According to the performance guide of `tf.data.Dataset.map`,[10] *num_parallel_calls* should be set to the number of available CPU cores. Setting a value much greater than the number of available CPUs can cause inefficient scheduling which can result in visible slowdown.

- `tf.data.Dataset.prefetch`: This feature helps overlapping the computations of producer and consumer. Usually the data pre-procssing steps are taken care by the CPU cores, while the GPU cores are preserved for running the compute-intensive convolutional operations. In a naive synchronous implementation, while the CPUs are kept busy pre-processing the data, the GPUs sit idle. In the same way, once the GPUs get the processed tensors and start training, the CPU sits idle. So, the time required to process a training step is measured as the sum of CPU's data pre-processing time and GPU's training time. Use of `tf.data.Dataset.prefetch` overlaps the pre-processing and model execution of a training step. When the GPU performs a training step, the CPU starts to process the data required for the next step. This is enabled by `tf.data.Dataset.prefetch` as it allocates an internal buffer and spawns a background thread that prefetches elements from the image dataset ahead of the time they are requested. In this way, the time required to perform a step reduces to the maximum of the training time and the data transformation time.

2.6 Implementation of Parameter Server using TensorFlow

On a cluster, a TensorFlow application is deployed as a set of *tasks*, each of which has the same graph execution API. The *tasks* are named processes that can communicate with each other over the network. Each task can associate itself with a single or multiple devices. Cross-device communication usually happens through RDMA (Remote direct memory access) or direct GPU-to-GPU transfer [2].

A subset of the deployed *tasks* can serve a common purpose. With the Parameter Server implementation in Standard Distributed TensorFlow, some *tasks* together assume the role of Parameter Server. Those tasks are referred as *ps tasks* and they typically host nodes that store and update variables. Beside the *ps task* group, there is another type of task called *worker task* which hosts stateless nodes that perform compute-intensive calculations. Among the worker *tasks*, there is a *chief task* which (a) coordinates model training with its fellow tasks, (b) counts the number of training steps that have been completed, (c) signals to stop

[10] `https://www.TensorFlow.org/guide/performance/datasets` (accessed September 13, 2019)

running *Session* when the stop condition is met and (d) writes the checkpoint files to the filesystem.[11]

2.6.1 Node Placement

TensorFlow optimizes the overall performance by automatically determining placements of the *operations* (nodes) across multiple tasks and the devices that are contained by those tasks [2]. The node placement algorithm takes a cost model as its input. The cost model contains information such as the estimated size (in bytes) of the tensor that flows in and out of each graph node, and estimated completion time that the node can take, given its input tensors. The estimation is calculated based on heuristics associated with the operation types and from placement decisions that were taken earlier in the previous executions of the graph. Starting from the source of the dataflow graph, the node placement algorithm tries to find a set of feasible devices for each node that is found during the graph exploration. Each device from the available device list is selected one after another and is examined to measure an overall cost value, assuming the operation will be placed in the selected device. The cost model helps in this step to pass information about the estimated cost of execution time of the node (operation). If the node has dependencies, then communication cost, such as transmitting inputs to the current node from other devices to the considered device is also measured. The device whose overall cost score is the lowest is selected as optimal device to host the operation. These steps are repeated for all the remaining nodes (operations) of the graph to find their right placement.

2.6.2 Cross Device Communication

To communicate across devices (CPUs and GPUs), at first the node placement decision needs to get finalized. After each node is allocated to its suitable device, the main graph is partitioned into a set of subgraphs. If there are any cross-device edges from one node to another, then two new nodes called *Send* and *Receive* node are added to the subgraphs. The cross-device edge is then removed and replaced by two edges: one connecting the source node to the *Send* node and the other connecting the *Receive* node to the destination node. This transformation helps TensorFlow to use *Send* and *Receive* nodes to coordinate among themselves to pass information across devices. The transformation is shown in Figure 2.6.

This same concept is applied to cross-machine communication. For distributed execution, *Send* and *Receive* nodes communicate across worker processes that run on different machines. TensorFlow supports protocols such as gRPC over TCP and RDMA over Converged Ethernet for transferring the data between tasks.

[11] `https://www.oreilly.com/ideas/distributed-TensorFlow` (accessed September 13, 2019)

2.7 Overview of Horovod

Horovod is a high level API that sits on top of TensorFlow [29]. It offers efficient inter-GPU communication via MPI-based ring reduction technique which optimally utilizes the available network to take full advantage of hardware resources. Horovod utilizes message passing interface (MPI) to launch TensorFlow programs among the worker machines in a distributed cluster.

2.7.1 Core Concepts

The core principles of Horovod revolve around the MPI concepts *size, rank, local rank, allreduce, allgather* and *broadcast*.[12] A brief introduction to these concepts is given below.

- *size*: *size* represents the total number of processes that Horovod is set to launch. Suppose, for example, the training script is to be run on two machines each equipped with two GPUs. In this case, the total number of processes will be four and that is the value of *size*.
- *rank*: represents the unique process ID given to each launched process. For the above example, the rank starts from 0 and ends at 3.
- *local rank*: represents the unique process ID within the worker machine. The *local ranks* of two TensorFlow proesses running on two GPUs from the first worker machine are 0 and 1.
- *allreduce*: an operation that aggregates gradients among multiple processes and then sends the updated gradient back to them.
- *allgather*: every process runs this operation to collects data from all other processes.
- *broadcast*: an operation that broadcasts data from one process to every process. Usually this operation is run from the process with rank 0.

Visual representations of *allreduce*, *allgather* and *broadcast* are given in Figure **??** (a), (b), (c) respectively.

[12] `https://github.com/uber/horovod/blob/master/docs/concepts.md` (accessed September 13, 2019)

2.7.2 Tensor Fusion

One of the unique features of Horovod is a Tensor Fusion algorithm[13] which fuses a large number of small-sized tensors into a single big tensor before invoking the Ring-AllReduce operation. While analyzing a few CNN models in distributed cluster, the authors found that the ResNet [14] model contained large tensors that tended to have many tiny AllReduce operations. As the AllReduce was reported to work optimally with large sized tensors, the authors proposed the idea of Tensor Fusion to take the full advantage of AllReduce.

In the first step, the algorithm determines the tensors which are ready to be reduced. Then a buffer is allocated for storing the first few tensors from the ready list. The buffer is named `fusion buffer` and by default the size of the buffer is set to 64 MB. The buffer size is referred as `HOROVOD_FUSION_THRESHOLD` in Horovod's API. After the buffer allocation, the tensors from the ready list are kept copying into the fusion buffer until `HOROVOD_FUSION_THRESHOLD` is reached. Once the buffer is filled, the AllReduce operation is executed into the `fusion buffer`. After the reduction is done, the data are sent to the output tensors. These steps are kept repeating until there are no more tensors left to be reduced.

Horovod was compared with Standard Distributed TensorFlow to study the scalability of two CNN models over 25 GbE TCP in a distributed GPU cluster. Horovod demonstrated 88% better performance compared to distributed TensorFlow for training Inception V3 and ResNet-101 model using 128 GPUs [29].

Horovod also has support for Remote Direct Memory Access (RDMA) capable networking. In an additional set of benchmarking tests, Horovod with RDMA was compared with Horovod with TCP. Three CNN models; Inception V3 [34], ResNet-101 [14] and VGG-16 [30] were used in the benchmarking tests. While scaling Inception V3 and ResNet-101 models, Horovod with RDMA was reported to be faster than Horovod with TCP. However, the improvement was not significantly higher; using RDMA lets Horovod achieve a 3-4% increase in throughput over TCP networking. Although a small improvement was observed for Inception V3 and ResNet-101 models, Horovod with RDMA demonstrated 30% speedup compared to its TCP variant for scaling VGG-16 model. The improvement was explained due to VGG-16's high number of model parameters compared to other models.

2.8 Implementation of Ring-AllReduce using Horovod

Horovod depends on OpenMPI to spawn remote processes across the worker machines through ssh. The MPI process starts all the other processes and assigns a global and local rank to each process as described in previous section 2.7. Each process pins a single GPU with its local rank which makes sure that no two processes are assigned to the same GPU. The information about the ranks and their corresponding GPU is then added to TensorFlow's device list. After each process is pinned to a GPU, Horovod uses the global ranks of the processes to create a communication link between the neighboring worker machines. This step

[13] `https://github.com/uber/horovod/blob/master/docs/tensor-fusion.md` (accessed September 13, 2019)

is done via NCCL which implements a highly optimized Ring-AllReduce collective communication algorithm for NVIDIA GPUs. The communication link is formed as a ring where each worker is only allowed to pass data to one of its neighbours. Horovod has an implementation of a broadcast operation that is used for consistent initialization of the CNN model on all workers. The broadcast operation usually starts from the process with global rank 0.

2.9 Summary

In this chapter, the concepts of distributed architectures, the neural network frameworks and previous works related to CNN based counting applications have been presented. The sections containing the description of the chosen frameworks have also introduced terminology which will be referred frequently in the upcoming chapters.

This chapter mentions two parallelization techniques: model parallelism and data parallelism. Both the Parameter Server and Ring-AllReduce uses synchronous data parallel model to optimize the distributed training. Standard Distributed TensorFlow utilizes the Parameter Server architecture, while Horovod implements Ring-AllReduce. The neural network structure of the flower counter application was influenced by the CNN model used in Zhang *et al.* [43] and was merged with the idea presented in Xie *et al.* [38].

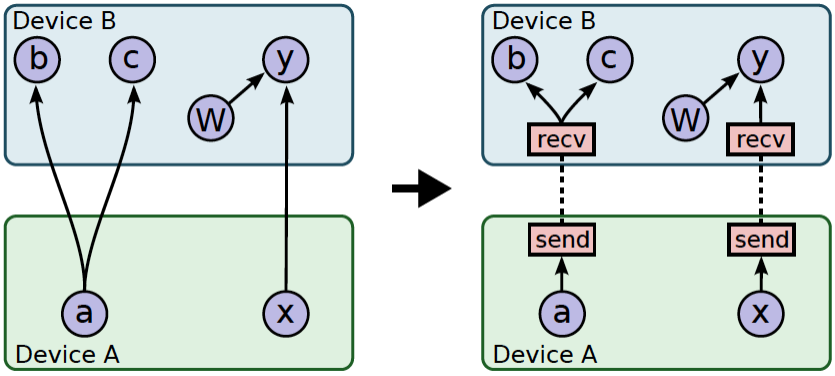


Figure 2.6: Cross Device Communication : Before and after scenario of inserting *Send* and *Receive* nodes [1].

3 Development and characterization of the alternative flower counting Application

This chapter describes the development stages and the overall specification of the proposed flower counting application. At first, a brief structural detail along with the overall memory allocation of the CNN model will be discussed. Followed by that, a brief characterization of the datasets being used for training the application will be shown. The datasets are comprised of images of canola flowers captured from various test fields. In order to use these images an initial filtering was done and then the filtered images were tiled into small sub-images. In order to generate the ground truth, these tiles needed to be annotated. Manually annotating images is a lengthy, time consuming task. This process of annotating the available image datasets provided a reasonable dataset for the scaling experiments on single machine-multiple GPUs but was still insufficient for distributed experiments. As a temporary solution to this problem, it was decided that the previous version of the flower counting application would be used as an automatic flower annotator to generate the "ground truth" for the distributed training. A detailed comparison of the manual annotation and the automatic annotation on the same sampled dataset is portrayed at the final section of this chapter.

3.1 Design

The structure of the CNN model of the Flower Counter application is inspired by the MCNN model proposed by Zhang *et al.* [43]. The purpose of the MCNN model was to estimate crowd count by predicting the density map of a given crowded scene. The model was comprised of three columns, each with different filter sizes. Each column represented an individual CNN itself. The network structure of the individual columns were kept identical except their filter size. The structure of the MCNN model is shown in Figure 2.4. It can be seen from Figure 2.4 that each column follows an identical structure (CONV-POOL-CONV-POOL). The feature maps from the last layer of the columns are stacked and forwarded to a final CONV layer which produces the density map. The loss function is defined by measuring pixel-wise Euclidean distance between the predicted density map and ground truth.

Though the MCNN model is believed to be efficient, there was one certain remark from the authors that was noteworthy. The authors clarified that due to usage of POOL layers twice, the spatial resolution of each image was reduced by 1/4, causing the output resolution of the predicted density map to be 1/4 times the resolution of the input image. This process led them to down-sample each training image by 1/4 before

generating their corresponding ground-truth density map. So, each ground-truth density map was 1/4 times the resolution of the original input image and each was later compared with the predicted density map of the same size (due to the usage of POOL layers) to calculate the loss. One last thing to be noted here, the input images were down-sampled only for creating the ground-truth density map; during the training phase, the input images with original resolution were used.

However, this down-sampling can destroy useful information such as finding the right location of the objects in the density map, as it introduces distortion while re-sizing the image. A similiar problem regarding the usage of max pool layer was raised in another CNN-based cell counting application [38]. As a solution, the authors suggested performing upsampling followed by a convolution layer which undoes the spatial reduction caused by the max pool layer. For upsampling they used bilinear interpolation. Following this advice, the MCNN model was modified by inserting two "transpose convolutional layers" also known as Deconvolutional layers [9] to upsample the reduction made through the max pool layers. Except for the insertion of two transpose convolutional layers, the rest of the structure was kept identical to MCNN. The design of the modified structure is given in Figure 3.1.

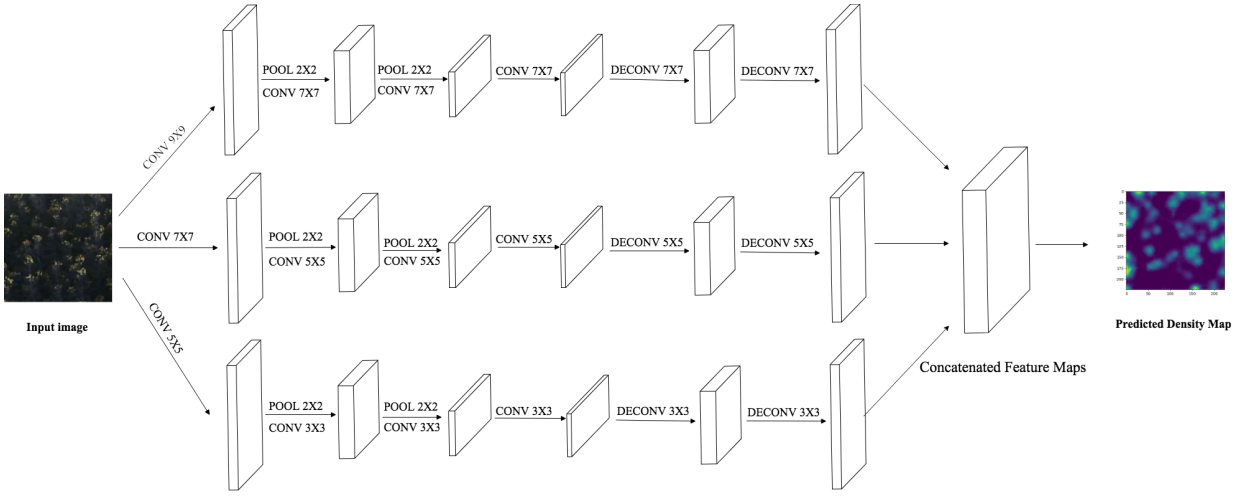


Figure 3.1: Modified Version of the MultiColumn Convolutional neural network architecture

3.2 Memory Allocation

As the dataflow graph for the CNN model is fixed, it is possible to trace the maximum memory allocation for each trainable model variable. For example, the first kernel used in the first convolution layer on the first column has a size of 9 x 9. Assuming the first convolution layer takes one image of size 224 x 224 x 3 as input and produces 16 feature maps[1] as output, the number of trainable paramters on that layer will be (9 x 9 x 3 + 1) x 16 = 3904 (here the +1 corresponds to the bias terms). Now, assuming the data type of

[1] map that highlights the areas of an image that are most similar to the filter [9]

each parameter is *float32*, the memory allocated by the trainable parameters from the first layer of the first column will be 3904 x 4 bytes = 15616 bytes.

The memory used by the model parameters is not the total amount of memory consumed during training the CNN. For example, assume that the first layer is a convolution layer with kernel size of 9 x 9, stride[2] of 2 and produces 16 feature maps. The input image for the convolution layer has three channels (R,G,B) and its size is 224 x 224. The total memory needed in the layer is calculated as follows. At the very beginning, the input image is considered as the incoming feature map for the first layer. So, the number of feature maps in first layer = 3 (equal to the number of color channels of the input image) and the number of feature maps it produces = 16 (defined by the user). As the stride is 2, the size of each output feature map will be 112 x 112 (the stride divides the input feature maps by the value it is set to). Assuming the datatype is set to *float32*, the first layer takes 112 x 112 x 16 x 4 = 802816 bytes for the output feature maps that will be used in the next layer. However, this is not the final value, because the size of the model parameters also needs to be added. To find the size of the trainable parameters, we can use the first example given above. It can be seen that the trainable parameters take 15616 bytes. So the total consumed memory = 802816 + 15616 = 818432 bytes for processing the single convolution layer that is used in this example.

Depending on the size of the input image, the size of the feature maps and the value of the stride, the total memory consumed during training varies; note also that the total memory consumed during the forward pass is not freed until the gradient calculation is finished. This is because the gradient calculation during the back-propagation phase requires preserving all the intermediate values computed during the forward pass [9].

Keeping track of the size of the model parameters is crucial for distributed training, because whatever distributed architecture is used, the model parameters are kept synchronized among workers during training. In each training step, the model parameters are updated and they are sent back and forth among workers.

The sizes of the trainable model parameters from all layers of the three columns are given in Table 3.1 (this should not be mistaken for the total memory consumed during training the CNN). Table 3.2 contains the trainable parameters from the last layer which concatenates the output from the three columns. All the information in Table 3.1 and 3.2 is generated using a lightweight TensorFlow framework called tf.slim.

By summing up the sizes of all the trainable parameters, it was found that, for the Flower Counter application, 138505 parameters in total will be trained and to accommodate the parameters, 1279140 bytes will be needed. This is important information for distributed architectures, as these parameters will be shared back and forth between the worker machines and the Parameter Server utilizing gRPC over TCP.

3.3 Dataset

The dataset is comprised of the images collected from different cameras positioned in different locations of various canola fields during the summer period of the year 2016. In total, pictures taken from four cameras

[2] the amount the kernel slides. It can also be represented as the distance between two receptive fields [9]

Table 3.1: Memory required by the trainable variables of the three columns of the CNN Model.

Column Name	Column 1		Column 2		Column 3	
Variable Name	Variable Description	Variable Size (in bytes)	Variable Description	Variable Size (in bytes)	Variable Description	Variable Size (in bytes)
conv1/kernel:0	9x9x3x16	15552	7x7x3x20	11760	5x5x3x24	7200
conv1/bias:0	16x1	64	20x1	80	24x1	96
conv2/kernel:0	7x7x16x32	100352	5x5x20x40	80000	3x3x24x48	41472
conv2/bias:0	32x1	128	40x1	160	48x1	192
conv3/kernel:0	7x7x32x16	100352	5x5x40x20	80000	3x3x48x24	41472
conv3/bias:0	16x1	64	20x1	80	24x1	96
conv4/kernel:0	7x7x16x8	25088	5x5x20x10	20000	3x3x24x12	10368
conv4/bias:0	8x1	32	10x1	40	12x1	48
deconv1/kernel:0	7x7x8x8	12544	5x5x10x10	10000	3x3x12x12	5184
deconv1/bias:0	8x1	32	10x1	40	12x1	48
deconv2/kernel:0	7x7x16x8	25088	5x5x20x10	20000	3x3x24x12	10368
deconv2/bias:0	16x1	64	20x1	80	24x1	96

Table 3.2: Memory required by the trainable variables from the last layer of the CNN model which concatenates the output from the three columns.

Final Convolution Layer		
Variable Name	Variable Description	Variable Size
final_conv/kernel:0	1x1x60x1	240
final_conv/bias:0	1x1	4

were collected throughout the aforementioned period of time. The cameras continuously captured images once per minute from dawn till dusk. The images from all the cameras were collected on five separate days. On each collection day, the captured pictures were transferred from the memory card and then the cameras were positioned again in the same spots. Manually positioning the cameras caused slight changes of perspective views. Therefore, images taken from the same camera are not guaranteed to have the same perspective view after each collection day. Not all cameras were included in the experimental dataset as there were many days that were captured before or after the flowering season. Those days were filtered immediately.

Throughout the rest of this document, the days captured from any camera will be referred as the format: "camera-day" where camera represents the camera id and the day presents the date of data collection. The date is formatted as: mmdd (where the mm stands for month and the dd stands for day). So the images captured on the second of August by camera 1109 will be represented as 1109-0802.

3.3.1 Image Filtering and Preparation of Ground Truth

After collecting the data, an initial filtering was conducted to exclude the images that were captured in darkness or at the days when there was fog at the beginning of the day. Each filtered image was split into 224x224 sized small tiles. The decision of splitting the images was influenced from the previous methodologies followed by Zhang *et al.* [43] and Xie *et al.* [38]. Splitting the images into small tiles significantly increased the total number of samples and also made the manual annotating process easier.

It is important to note that the whole image was not split into tiles. Each captured image contained portions from neighboring fields and adjacent roads. Our region of interest was only the portion of the field where the capturing camera was placed. For each image, a single coordinate was selected from which a variable sized window was cropped. The height and width of the window were selected to be a multiple of 224. Using this information, the code was written in such a way that given the single coordinate for each image, a variable sized window will automatically be cropped and then split into 224x224 sized tiles. As a large number of images needed to be split, a python script was developed and deployed in Apache Spark to split the images in parallel. The python script for both the sequential and the distributed version can be found in the Appendix section of this thesis. The overall characteristics of the dataset are given in Table 3.3. The table does not include information about images captured from Cameras 1122 and 1225 as they were not split into tiles.

Table 3.3: Characteristics of the Dataset

Camera ID	**Before Split**			**After Split**		
	Total Days	Number Of Images	Resolution	Total Days	Number Of Images	Resolution
1108	34	33113	1280x720	34	301102	224x224
1109	20	19896	1280x720	20	287980	224x224
1207	29	28338	1280x720	29	316545	224x224
1237	27	25826	1280x720	26	281560	224x224
Totals	159	153739	-	109	1187187	-

3.3.2 Manual Annotation

Though the datasets were ready, to start training the Flower Counting application, the ground truth density maps were needed to be prepared as well. The process of creating density map required knowing the coordinates of the flowers. So, the idea was to first locate and annotate all the visible flowers from each image, and then to develop software that would extract the coordinates of each flower. Annotating the flowers was done by simply applying a red cross symbol of size 2 into the centre of each visible flower using GIMP.[3] The

[3] `https://www.gimp.org/` (accessed September 13, 2019)

Red color (RGB: 255, 0, 0) was used as there is no red in any of the images.

Four camera-days have been fully annotated (July 4, August 2, 3, 5 and August 6). A sampling of eight other days has also been completed (July 5 - 12). After the manually annotated tiles were saved in a directory, the directory address was given as input to a python script that detects the cross shapes across the images and returns the coordinates of the centers of the crosses.

As described previously, one of the main objectives of this thesis was outlined as developing a scalable version of the application by evaluating two distributed architectures in distributed setup. To meet the time constraint, the scalable versions were decided to be developed in parallel and for testing those versions, more data was needed at that time.

After the development phase, the manually annotated images completed at that time were considered sufficient for scalability related experiments on a single machine but they were inadequate for running distributed experiments. Considering the long time it takes for manual annotation, it was decided to use an automated method for estimating flower locations for the scalability related experiments.

3.3.3 Automatic Annotation

The automated method for estimating flower locations was based on a previously developed software application [11]. This application was developed earlier by the researchers of P2IRC and several image processing techniques were used on that application to detect and count canola flowers. The previous version was comprised of two significant parts: one is related to filtering the images and grouping them into clusters and the other is counting flowers by running a blob detecting algorithm for each example of the clusters. The purpose of the clustering phase is to identify the images that are most likely to produce better results. This is done by computing histogram shifts for each image with respect to the average histograms of Lab color space.[4] This process helps flagging images as either good or bad. After that, all the good images are sorted and grouped into two clusters based on the percentage of yellow pixels. Both of the clusters are then forwarded to the counting phase of the application which involves (a) converting the images from BGR to CIELab colorspace, (b) sigmoid mapping, and finally (c) running Determinant of Hessian (DoH) blob detection algorithm. The final output of the application is a text file which contains image names and their corresponding estimation of flower counts.

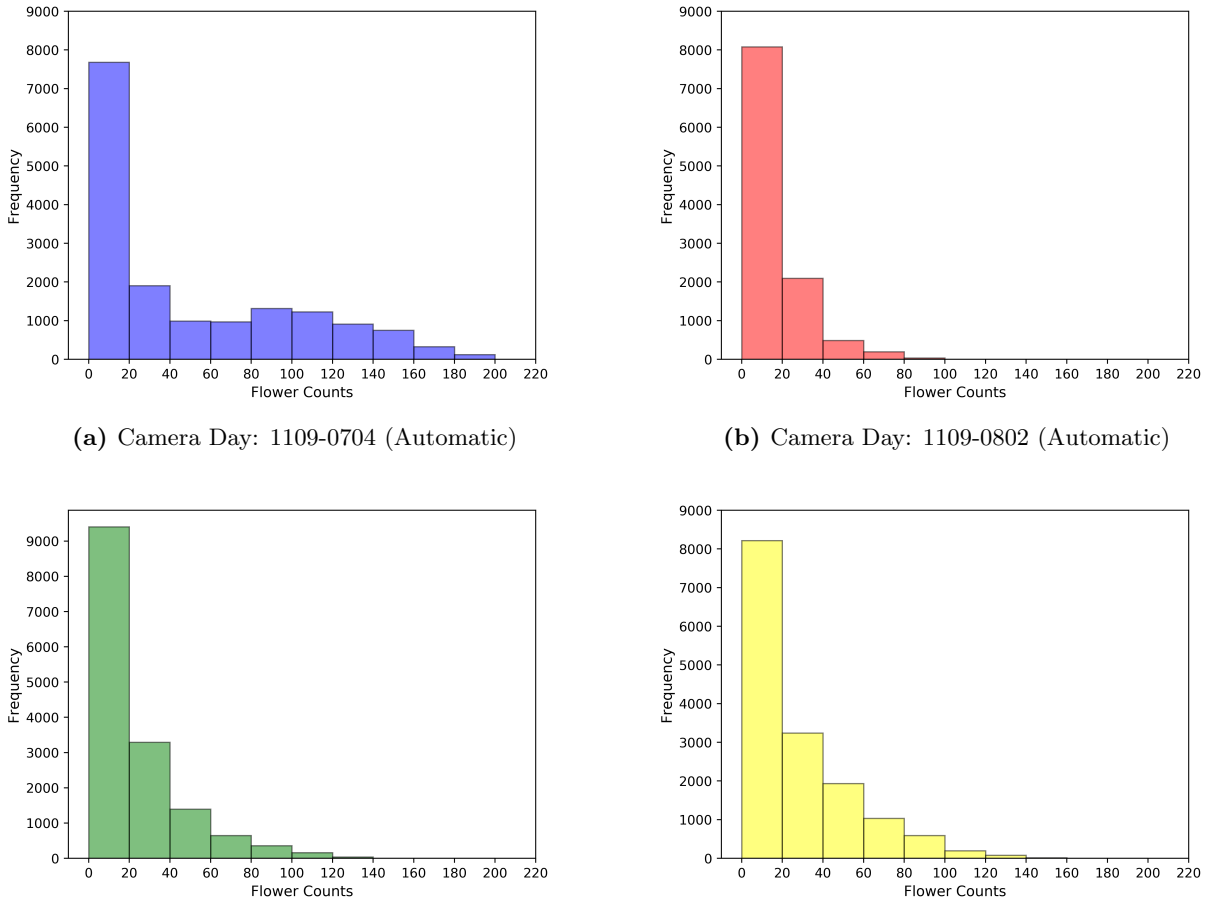
This version of the flower counting application was reported to work well only under particular conditions. In the experiments performed in this thesis research, the application was found to return satisfactory results only when there are few flowers in the field. It is no longer under active development. Though the application often does not give accurate flower counts, the application was modified and applied to automatically annotate the images. However, using the application as a ground truth generator had one caveat. As the prediction of the application is not correct, the ground truth generated from the application could affect the learning process of the CNN model.

[4] `https://en.wikipedia.org/wiki/CIELAB_color_space` (accessed September 13, 2019)

That being said, it is an interesting study to observe how the CNN application adapted to situations when the data are incorrectly labeled and to what extent it learns to produce similar results as the previous flower counting application. To adapt the previous flower counting application for ground truth estimation, it was modified to return the location of detected flowers as well as the estimation of flower counts. This was necessary, since the ground truth is needed in the form of a density map.

3.3.4 Comparison between Manually and Automatically Annotated Datasets

In order to understand the difference between the number of flowers counted by both manual and automatic annotation, the previous version of the Flower Counter application was run on four days that were manually annotated. Histograms representing distributions of counts of automatically annotated flowers are given in Figure 3.2.



(a) Camera Day: 1109-0704 (Automatic)

(b) Camera Day: 1109-0802 (Automatic)

(c) Camera Day: 1109-0805 (Automatic)

(d) Camera Day: 1109-0806 (Automatic)

Figure 3.2: Histograms of flower counts (Automatic Annotation)

Manual annotation reveals that with the three days during August, most of the captured images have very few marked flowers; the vast majority have less than five. Compared to these days, the images captured

on the 4th of July have a wide range of flowers. The skewness in the frequency of flower counts can be better portrayed through the bar graphs given in Figure 3.3.

Figure 3.4 gives a side-by-side comparison of the results from both annotating methods, using box plots. The side-by-side comparison from the box plots provides insights about the quality of automatic annotation. In Figure 3.4, for all the box plots the horizontal line inside the box represents the median of the whole distribution. The box is the inter-quartile range (IQR) and the lines are the quartiles $+-1.5*IQR$. Outliers are beyond these limitations.

It is clearly visible that the automatic annotation counted more flowers than the manual annotation. Usually in images where the flowers are hardly distinguishable from leaves, stems or pods due to the higher intensity of sun reflection, the application recognizes leaves as flowers. If the automatically annotated datasets are used for generating ground truth density maps, then the CNN model will be trained considering sun reflections as flowers; this is wrong, but this is the only viable solution to find ground truth estimates for large numbers of images, to be able to evaluate the scalability performance of the Deep Learning architectures.

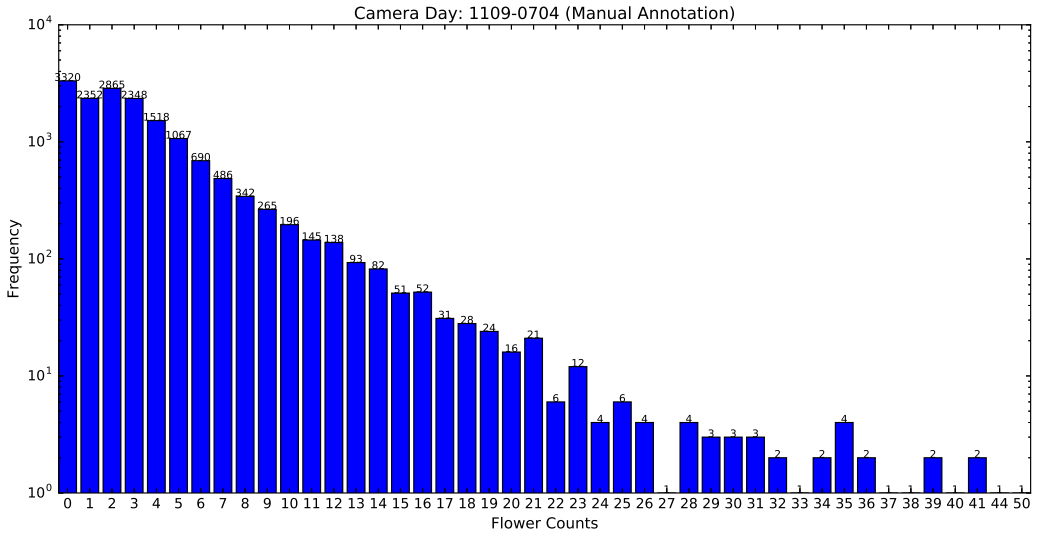
3.3.5 Generating Ground Truth

The final step is generating the ground truth density maps. For each tile, a two dimensional empty array with the same height and width of input image tile is created. Based on the coordinates of the flowers (resulting from either manual or automatic annotation), the value 255 is put in the exact same locations inside the two dimensional array. So, the array contains the value 255 only on the center locations of flowers with the rest of the values set to zero. The ground truth is finally generated by blurring non-zero elements of the array with a Gaussian kernel normalized to sum to one. An example of three image tiles with their corresponding density maps generated using the coordinates produced from both manual and automatic annotation are given in Figure 3.5.

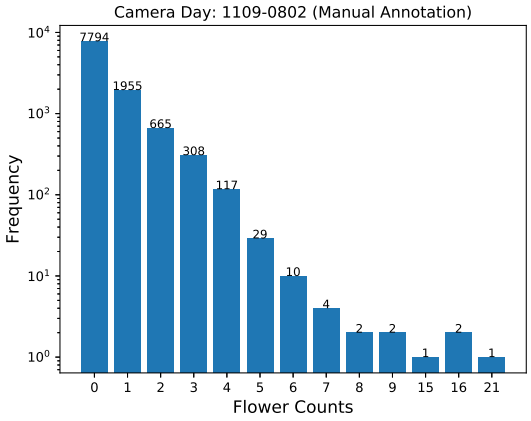
It can be noticed from figure 3.5(b) that how the sun reflection affects the previous Flower Counter. While manually annotating, 39 flowers were visible whereas the previous application predicted 216 flowers for that same image. The application also undercounted in scenarios where the flowers were sparse; which can be seen in figure 3.5(a) and (c).

3.4 Summary

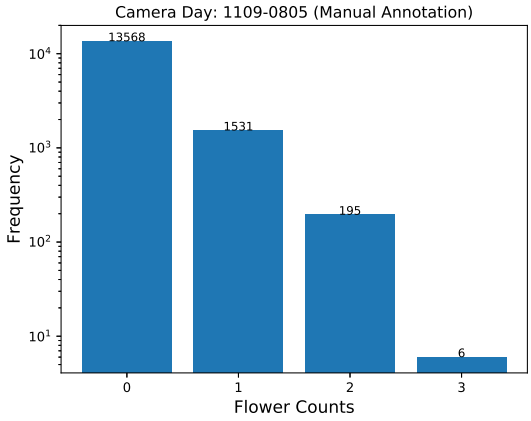
In this chapter, an overall overview of the proposed CNN-based Flower Counter is discussed in detail. Besides providing the architectural structure of the used CNN-model, an in depth analysis of the required memories for each individual layer is given in table 3.1 and 3.2. Information regarding the dataset and the procedures followed for creating the ground truth density maps are also explained in the concluding sections.



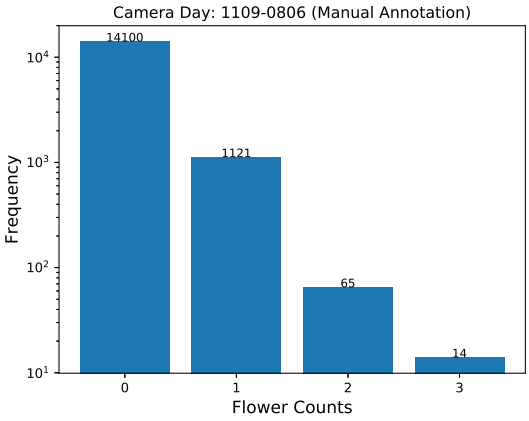
(a) Camera Day: 1109-0704 (manual)



(b) Camera Day: 1109-0802 (manual)

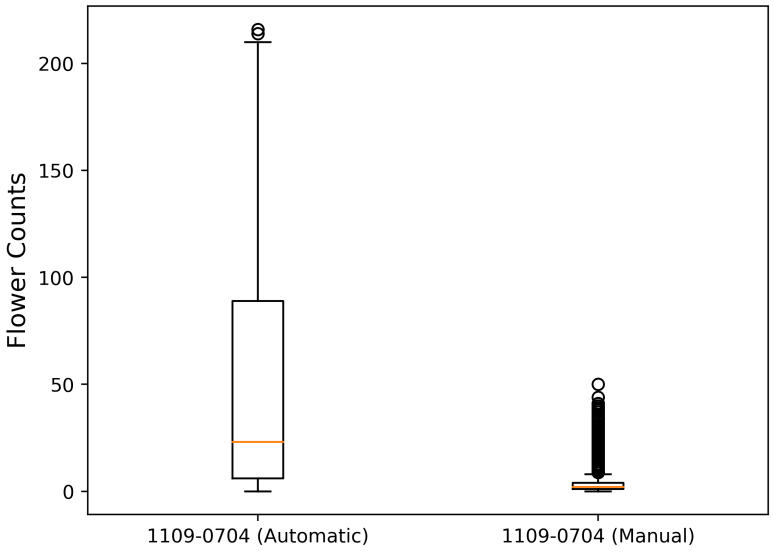


(c) Camera Day: 1109-0805 (manual)

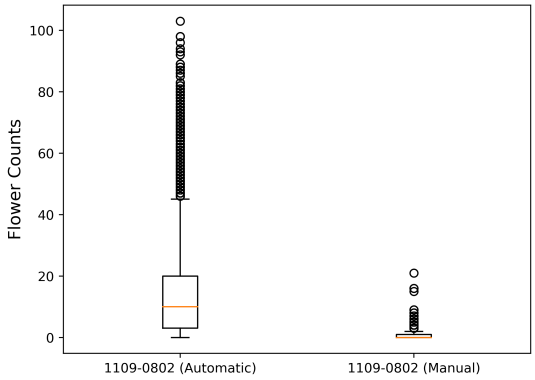


(d) Camera Day: 1109-0806 (manual)

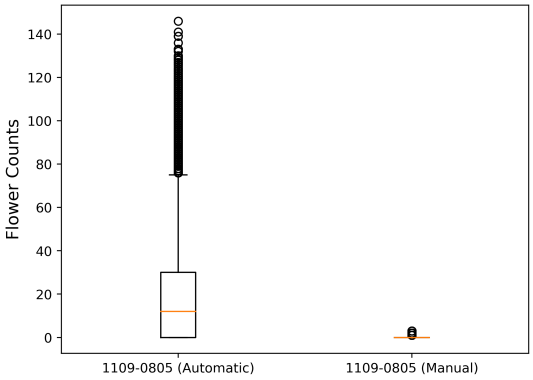
Figure 3.3: Bar Graphs of flower counts (Manual Annotation)



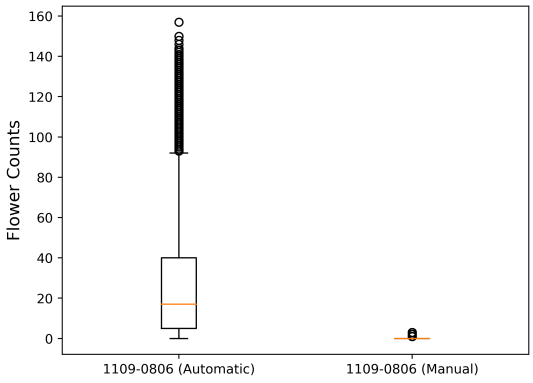
(a) Camera Day: 1109-0704



(b) Camera Day: 1109-0802



(c) Camera Day: 1109-0805



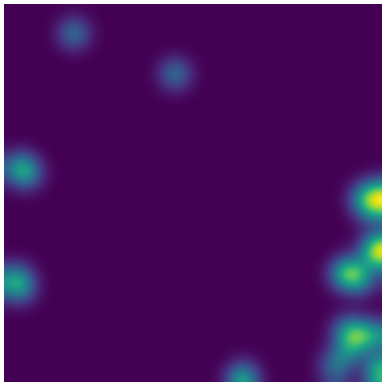
(d) Camera Day: 1109-0806

Figure 3.4: Box plots of flower counts

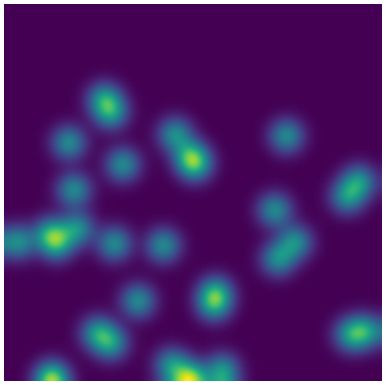
(a) Image Tile: frame000231_1_3 from Camera Day: 1109-0802

(b) Image Tile: frame001201_0_0 from Camera Day: 1109-0704

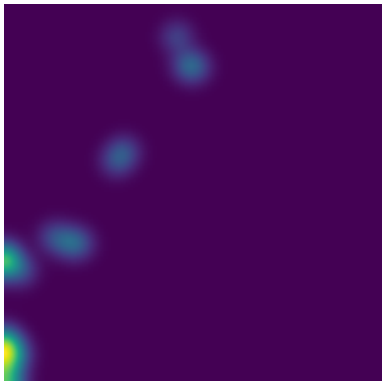
(c) Image Tile: frame000154_1_4 from Camera Day: 1109-0802



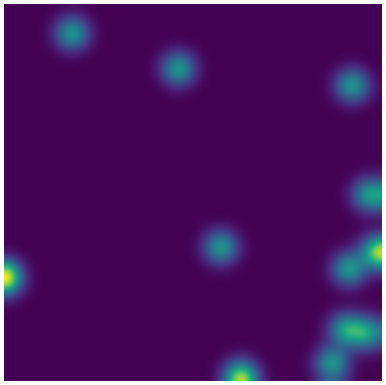
(d) Density Map for tile: frame000231_1_3 from Camera Day: 1109-0802, Number of Flowers annotated : 21 (Manual Annotation)



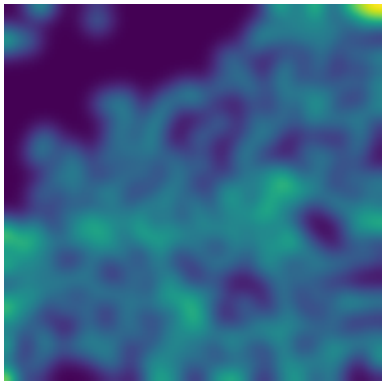
(e) Density Map for tile: frame001201_0_0 from Camera Day: 1109-0704, Number of Flowers: 39 (Manual Annotation)



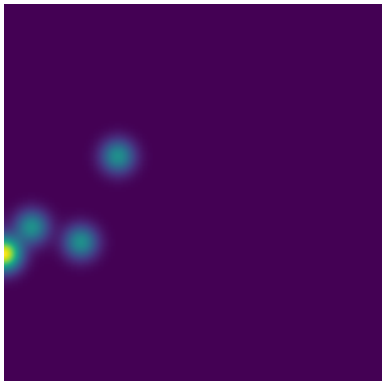
(f) Density Map for tile: frame000154_1_4 from Camera Day: 1109-0802, Number of Flowers: 16 (Manual Annotation)



(g) Density Map for tile: frame000231_1_3 from Camera Day: 1109-0802, Number of Flowers annotated : 12 (Automatic Annotation)



(h) Density Map for tile: frame001201_0_0 from Camera Day: 1109-0704, Number of Flowers: 216 (Automatic Annotation)



(i) Density Map for tile: frame000154_1_4 from Camera Day: 1109-0802, Number of Flowers: 4 (Automatic Annotation)

Figure 3.5: Image tiles with corresponding Density Maps.

4 Experimental Design and Configuration

This chapter presents the overall methodology of running the experiments in different testbeds. The chapter is structured into four different parts. The first part is about the methodology and the outline of the experiments conducted to analyze the accuracy and the scalability of the CNN model in both local and distributed environments. This part presents every step by step procedure of how the image dataset goes through the preprocessing stages in the input pipeline to the training stages hold by the CNN model. The application's synchronization of the model parameters in distributed training stages for both parameter server and the Ring-AllReduce architecture is also discussed in detail.

The second part presents various evaluation metrics used for measuring the performance of the distributed architectures and the accuracy of the application. The performance metrics were used alongside proper profiling tools to better explain the resource utilization, throughput and possible bottlenecks in local and distributed training. Besides performance, a different set of evaluation metrics were used to demonstrate the correctness and generality of the CNN model on both manually and automatically annotated datasets. The successful measurement of these metrics required careful setup and configuration of the testbeds. The setup process along with the hardware and software configurations to run the experiments are included in the third part of this chapter. Followed by that, an in-detail explanation of the configuration setting of the Flower Counter application is presented in the fourth part. Finally, the chapter ends by summarizing important details and information discovered.

4.1 Methodology and Outline of Experimental Design

4.1.1 Accuracy

To measure the accuracy of the Flower Counter application, the ground truths generated from both the manual and the automatic annotation were used separately. One concern about using the annotated ground truth was the skewness of the distribution of flower counts. Recall from the previous chapter that most of the images that were either manually or automatically annotated had zero flowers. Among all the manually annotated camera-days, the images from 1109-0802 were annotated at first. These images were used in the initial experiments where the whole dataset was shuffled and the first 70% of the data was selected as the training dataset and the rests as testing dataset. For the first few experiments, it was observed that the training dataset contained images that have mostly zero or fewer than four flowers. This imbalanced distribution of flower counts was believed to affect the accuracy of the application; to mitigate the problem

a reasonable solution was needed.

As a solution, it was decided that the dataset will be divided into multiple bins. Among these bins, the one bin which has minimum number of images will be selected and the number of images that the bin contains will be set as the number of samples to be randomly taken from each bin. This process of over-sampling the minority group and under-sampling the majority group decreased the total number of training images but it was a fair choice to prevent the model from becoming biased for specific groups of images.

For the scalability related distributed experiments, the same camerday (1109-0802) was selected. The automatically annotated ground truths were used. Based on the frequency of the flower counts, the whole dataset was decided to be divided into three bins. The first bin was set to contain the images that have fewer than twenty flowers, the second bin to hold images with more than twenty but fewer than forty flowers and the third bin to contain more than forty but fewer than two hundred flowers. By doing so, the bins ended up containing 7688, 1976, 686 images respectively. So, the number of samples to be collected from each bin was set to 686 and altogether 2058 images were selected to comprise the dataset. 70% of the images from these 2058 were selected as the training set and rest of the images were selected as testing set, so the training set was comprised of 1440 images.

The accuracy of the CNN model was tested using dataset with manually annotated ground truths. For the experiment, the earlier camera-days of summer period were planned to be used. For the initial accuracy related experiment, camera-day 1109-0704 was selected. The binning technique was applied but this time the dataset was divided into seven bins where the bins contained images as mentioned in Table 4.1.

Table 4.1 shows that the first bin contained images which have only zero flowers, the second bin was filled with images containing only one flower. Likewise, the seventh bin is assigned to images which have eight to fifty flowers. This assignment led the bins to contain 2656, 2109, 2712, 2363, 1763, 2585 and 1911 images respectively. Following the strategy mentioned above, 1763 images were sampled from each bin; so, in total 1763 * 7 = 12341 images were selected as the whole dataset out of which 8638 images (70% of the dataset) were selected for the training set and 3703 images for the testset.

Table 4.1: Assigning images into multiple bins (camera-day 1109-0704). To be trained for 3000 epochs.

Bin index	**Flower count range**
1	0
2	1
3	2
4	3
5	4
6	5 - 7
7	8 - 50

The sampled dataset that follows the binning strategy mentioned in table 4.1 is set to be trained by the proposed CNN model for 3000 epochs. In an attempt to find how the model will perform when the training epoch is reduced to less than half, an additional experiment used 1000 epochs.

The binning selection was tweaked too. The number of bins was now reduced to six from seven. The rationale behind this reduction is that setting a bin which contains images with a specific flower count can still imbalance the dataset as the number of bins which have few flowers are many compared to bins which have more than eight flowers. Though the binning used for the experiment is not perfect, it is believed that the problem will start to get mitigated when images with more flowers will be added to dataset from more camera-days collected later in the season. The binning procedure must be adjusted according to the ground truth of the input dataset. The ranges of flower counts for the first three bins is kept similar as the previous binning technique, but the fourth, fifth and sixth bins are now changed to contain images with 3-4, 5-7 and 8 and above flowers respectively. Table 4.2 provides the binning information.

Table 4.2: Assigning images into multiple bins (camera-day 1109-0704). To be trained for 1000 epochs.

Bin index	**Flower count range**
1	0
2	1
3	2
4	3 - 4
5	5 - 7
6	8 and above

Rearranging the bins led to assign 2656, 2109, 2712, 4126, 2585 and 1911 images for six bins respectively. As a result, 1911 images are sampled from each bins and in total, 11466 images are selected to comprise the dataset. Out of these 11466 images, 8062 images are used as training set and remainder as testset. For the rest of document, the models trained for 3000 and 1000 epochs will be referred as *model-m* and *model-l* respectively.

In order to check how general the Flower Counter application is, both of the CNN models (*model-l* and *model-m*) were tested with another cameraday 1109-0705. Note that, not all the images from 1109-0705 were finished with manual annotation. Between 40 and 100 images were sampled from each tiling position of camera-day 1109-0705 to construct the generalized test dataset.

4.1.2 Scalability

Creating a distributed version of a CNN application starts with choosing the right method to speed up SGD calculation. One simple technique that is used for such purpose is to parallelize the computation of the gradient calculation [1]. Two common parallelization approaches that are mostly popular among CNN

practitioners are Data parallelism and Model parallelism. For this work, the data-parallel approach was selected due to its wide-spread practice and simplicity of execution. TensorFlow offers two different ways to execute data parallel structure: one is *In-graph replication* and other is *Between-graph replication*.[1] In order to better understand all techniques of data parallel structure, both of these replication methods were evaluated in this project. Two different versions of the CNN model were developed for this purpose. The first version implements *In-graph replication* which was built to be experimented in local environment (one single node containing eight GPUs), whereas the second version was tested in distributed cluster and it uses *Between-graph replication*. Throughout the rest of this thesis, the first version will be referred to as the "local" version of the Flower Counter application.

Parameter Server

One common aspect of both the local and distributed version is that both of them mimic the parameter server architecture. In the local version of the application, the GPUs inside the single machine were considered as workers and the CPU was used as controller for gradient aggregation and parameter sharing. For the distributed version, separate machines were used to host the *ps tasks* and the *worker tasks*. The machines hosting the *ps tasks* were not involved in running compute intensive convolutional operations, but they were responsible for preserving the updated model parameters. The only calculation involved in these machines was to aggregate the gradients from the worker machines and then replace the old model parameters with the updated values. Immediately after updating, the model parameters were sent to each worker machine so that they can start processing the next mini-batch of data.

Local Experiment (Single machine with multiple GPUs): Though both of the versions followed centralized architecture for gradient accumulation and gradient sharing, the characteristic that distinguished them was the way of parallelising the convolutional computations. As stated earlier, *In-graph replication* was used to build the local version. Generally in *In-graph replication*, a client process builds a single data flow graph with multiple copies of compute intensive parts. These copies of the parts are pinned to different devices of the same machine.[2] This is typically done via a loop which creates the same graph structure for each working device. After each iteration of the loop, when the compute intensive parts are allocated to a device, a replica of the subgraph (compute intensive part) is added to the main graph specifying which device this subgraph will run on. This method of duplication continues increasing the overall size of the main dataflow graph. For example, if a machine has N GPUs then the main graph will typically contain N copies of the compute-intensive parts. That means during training, when each device processes the entire graph, it needs to accommodate the copies of subgraphs that might only be relevant for other devices. *In-graph replication* is the initial parallelization approach tried by the TensorFlow developer's team.[3] This approach is

[1] `https://www.TensorFlow.org/deploy/distributed#replicated_training` (accessed September 13, 2019)

[2] `https://www.TensorFlow.org/deploy/distributed#replicated_training` (accessed September 13, 2019)

[3] `https://stackoverflow.com/questions/41600321/distributed-TensorFlow-the-difference-between-in-graph-replication-and-between` `noredirect=1&lq=1` (accessed September 13, 2019)

usually practiced in a single machine with multiple GPUs rather than distributed cluster due to it limitations of handling memory inefficiently.[4]

Distributed Experiment (Multiple machines equipped with one GPU each): In contrast to *In-graph replication*, *Between-graph replication* lets multiple client processes to build separate individual data flow graphs with single copy of compute intensive parts. Typically, each client process is associated with a *worker task* and each *worker task* runs in a separate machine. The graph residing in one worker does not contain any nodes or parts from the graph of other workers. The model parameters are shared through *ps tasks* which can be configured to run in separate machines or in any of the machines where the *worker tasks* are running on.

Though *worker tasks* are generally deployed in separate machines, it is possible that multiple *worker tasks* can be configured to run in the same machine with explicit definition of which accelerator (GPU or TPU) will be pinned to which *worker task*. In the case of allocating multiple *worker tasks* in a machine with a single GPU, TensorFlow allows sharing the same device among all the *worker tasks* by allocating a subset of the available GPU memory per task. For this thesis, both the *ps tasks* and *worker tasks* were set to run on separate machines.

Ring-AllReduce

In addition to the parameter server, another alternative solution was crafted for distributed training using a framework called Horovod to study the performance of Ring-AllReduce algorithm. Horovod implements ring-reduce by depending on OpenMPI and NCCL (NVIDIA Collective Communications Library). One of the strengths of Horovod is that it makes the process of converting existing TensorFlow application to Ring-AllReduce based architecture much easier by tweaking few changes in the TensorFlow code. However, the initial environment setup process requires substantial efforts as it is hard for TensorFlow, Horovod, MPI, and the hardware interfaces to sync well together.[5]

To implement Ring-AllReduce in Horovod, a worker machine with N GPUs is assigned to run N Tenosrflow processes. Each process is pinned to exactly one GPU of the machine. Similar to Standard Distributed TensorFlow, each worker runs a separate replica of the CNN model on different batches of data. After the model is run on all the batches, the loss function is calculated and depending on the contribution to the loss function, the adjustment of the model parameters is computed.

Compared to Standard Distributed TensorFlow, there are no *ps tasks* in Horovod. The gradients are calculated separately on each worker, but are averaged across the worker machines using the constructed ring. So the model parameters are updated without any help or intervention from a dedicated machine that stores or averages the gradients [29].

[4] `https://www.youtube.com/watch?v=la_M6bCV91M` (accessed September 13, 2019)
[5] `https://ai.intel.com/horovod-distributed-training-on-kubernetes-using-mlt/` (accessed September 13, 2019)

4.2 Evaluation Metrics

In addition to building the distributed structures of the Flower Counter application, an assessment was needed to determine how well the outcome from the architectures were going to satisfy the intent of getting better accuracy. Though there were many viable metrics that could have been used, the choices were kept simple to prevent ambiguity and repetition. In some certain cases it appreared to that, it was not enough for one metric to solely demonstrate the quality of the results. On those cases, other metrics were used collectively to quantify the practicality of the solution. The metrics used in this project are divided into two categories: Metrics for demonstrating Performance (Scalability) and Metrics for demonstrating Accuracy.

Metrics for demonstrating Performance (Scalability)

In order to asses the scalability of the local and the distributed version of the Flower Counter application three different metrics were chosen. The individual purpose and description of each of these metrics are given in the following bullet points.

- Total Completion Time: The primary objective of scaling a CNN application in distributed environment is to minimize the long training time. To measure the capability of the distributed solutions, the total elapsed time from the beginning of the application till the end needs to be counted. The time it takes to complete training the whole model can be a good indicator of how fast the distributed version is. To keep track of the total completion time, python's `time.time()` function was used at the application's starting and ending point to record the start and end time. Then the duration was calculated by simply subtracting the starting time from the ending time. The `time.time()` module usually returns a floating point number which represents the time in seconds since the epoch. Here epoch means the point where the time starts. For Unix the epoch is set to 1970.[6] For every distributed experiment, the total completion time was measured as the time taken by the slowest worker to finish training.

- Throughput: The scalability of a CNN application has been measured by throughput in much recent literature [3],[6],[21],[28],[29]. The throughput was measured in form of samples been processed per second. For the Flower Counter application, the throughput was calculated in every training step by dividing the number of images been processed on that step by the duration of the step. Keeping track of throughput from all the training steps is helpful in the sense that it can help us to detect potential bottleneck by isolating any specific training step that suspiciously takes a long time. For each of our performance related experiments, an average throughput was calculated at the very end of the training to show the scalability of the Flower Counter application.

- GPU utilization: Scalability of a distributed model depends mostly on how the resources are being utilized during training. Underutilization of resources often leads to poor scalability. In TensorFlow,

[6] `https://docs.python.org/2/library/time.html` (accessed September 13, 2019)

the compute-intensive parts are preferred to be scheduled on GPUs.[7] So, during training, it is crucial to keep track of the GPU resource usage. Profiling GPU resources helps us to pinpoint the performance issues. For example, during a long running training step, if the GPU utilization is not approaching 80-100%, then the bottleneck may be caused by the input pipeline[8] or synchronization among workers. In addition to that, profiling the GPU also helps in understanding important insights of the data. For long training steps, if the GPU utilization reaches above 80%, then that means the images been processed during those steps require more computation compared to other images.

The GPU utilization does not represent the amount of resources being used; it reports what percentage of time one or more GPU kernel(s) is active.[9]

The choice of the aforementioned metrics is believed to be sufficient to evaluate the scalability of the local and distributed version of the Flower Counter application. During the preliminary stages of building the distributed version, GPU memory utilization was also considered as a performance metric, but TensorFlow by default maps all the GPU memory of the selected GPU for the running process to prevent memory fragmentation. So the memory utilization remained constant over the period of training. One thing to be noted here is that, though TensorFlow by default allocates all the GPU memory for the running process, it offers the option of allocating a subset of the available GPU memory to each process. This configuration needs to be manually set before training. Once activated, if more memory is needed by the running process then TensorFlow allows memory growth as per the need of the application.[10]

Metrics for demonstrating Accuracy

Although the primary objective of this project is to construct a scalable version of the Flower Counter application, it is mandatory to examine how accurate the Flower Counter application is. Given an image of a canola field, the application is expected to produce an estimation or prediction of number of flowers present in the image. To find the accuracy of the prediction, three different metrics were chosen. The descriptions of each of these metric are given below:

- Mean Absolute Error (MAE): Mean Absolute Error is the average absolute distance between the predicted value and the true value. This simple metric is a good indicator of the average magnitude of the errors and it has been seen used alongside other metrics to evaluate the accuracy of CNN based-applications in previous studies [23],[38],[43].

- Root Mean Squared Error (RMSE): Root Mean Squared Error is the square root of the average of squared difference between the predicted value and the true value. In contrast to MAE, RMSE gets

[7] `https://www.TensorFlow.org/guide/using_gpu` (accessed September 13, 2019)
[8] `https://www.TensorFlow.org/guide/performance/overview` (accessed September 13, 2019)
[9] `https://stackoverflow.com/questions/40937894/nvidia-smi-volatile-gpu-utilization-explanation/40938696#40938696` (accessed September 13, 2019)
[10] `https://www.TensorFlow.org/guide/using_gpu` (accessed September 13, 2019)

affected by the variance associated with the frequency distribution of error magnitudes. When the variance increases RMSE also increases. Both of the MAE and RMSE are negatively-oriented: lower values mean better results.

- Relative Error (RE): The Relative Error is the absolute error between predicted value and the true value, divided by the magnitude of the original value. RE becomes helpful in bringing error into perspective by comparing approximation of values of widely differing range. For example, an absolute error of three is significant if the original number of flowers is five, but the error is acceptable for us if the true number of flowers is more than two hundred.

Periodically testing the accuracy on the validation set during training is a common practice in Neural network applications. This helps to monitor how well the application is learning on data which is not exposed to the model before. Instead of periodically checking the accuracy on a testset during training, it was decided that for the proposed application, the accuracy of the validation set will be measured after the model has finished training. The reason behind this decision is to discard any sort of computation which does not contribute to the overall completion time of training. As the completion time was used as a metric to judge the scalability of the distributed solution, it was ensured that no other computation except training the model is taking place during the run-time of the application. Once the application finished running (after getting trained for specified epochs), the trained model was loaded from the last checkpoint to run the validation set. However, the progression of accuracy on training dataset was observed using MAE in each training step. At very last, the inference time of the CNN models was compared with the previous Flower Counter using the same methodology of calculating the Total completion time as described in 4.2.

4.3 Hardware and Software Configuration

This section describes the hardware and software configurations of two different testbeds used for experimenting with the local and distributed version of the Flower Counter application. General information about the hardware specifications such as CPU's speed, cache memory, RAM, GPU clock speed etc. are provided alongside the choice of software being used to run the experiments. All the testbeds run Ubuntu 16.04.3 LTS as the Operating System. Python 3.5 is the main programming language chosen for this entire project.

4.3.1 Hardware Specification

Single machine with multiple GPUs

The local version was trained and tested in a server machine called *octopus.usask.ca* which is equipped with Intel(R) Xeon(R) CPU E5-2643 v4 (3.40GHz) CPU with 24 cores, 126 GB of RAM, 50TB of disk space, 32 KB of L1d and L1i caches, 256 KB of L2 cache and 20480 KB of L3 cache. Eight GeForce GTX 1080 Ti

GPUs are attached to machine, each of which has 3584 NVIDIA CUDA(R) Cores (Clocked at 1582 MHz), 11 GB of GDDR5X memory, 384-bit Memory Interface Width and 484 GB/sec Memory Bandwidth.

Exclusive excess was granted for limited time to run the experiment on *octopus.usask.ca* to observe the application's scalability across multiple GPUs in a single server machine.

Distributed GPU cluster

The DISCUS Lab at the University of Saskatchewan has a distributed GPU cluster comprised of ten physical machines interconnected with each other using a 1 Gigabit Ethernet switch. The machines have an homogeneous configuration: Intel(R) Core(TM) i7-2600 (3.40GHz) CPU with 8 cores, 16 GB RAM, 32 KB of L1d and L1i caches, 256 KB of L2 cache and 8192 KB of L3 cache. Each machine is equipped with one NVIDIA GT 1030 GPU which has 384 NVIDIA CUDA(R) Cores (Clocked at 1468 MHz), 2 GB of GDDR5 memory, 64-bit Memory Interface Width and 48 GB/sec Memory Bandwidth.

4.3.2 Software Setup

TensorFlow is the neural network framework chosen for this project. To implement the Ring-AllReduce architecture, Horovod was used alongside TensorFlow. Successfully launching TensorFlow application depends on the right CUDA driver, CUDA Toolkit and cuDNN versions. Furthermore, Horovod uses NCCL and openMPI to form the Ring-AllReduce based architecture. Table 4.3 that contains the name and version of the software used in running the local and distributed experiments.

Table 4.3: Software Versions

Experiment Type	Parallelization Scheme	TensorFlow	Horovod	Nvidia Driver	CUDA Toolkit	CUDNN	OpenMPI	NCCL
Distributed	Parameter Server	1.11.0	-	390.87	9.0	7.0	-	-
	Ring All Reduce	1.11.0	0.15.1	390.87	9.0	7.0	3.1.2	2.0
Local	Parameter Server	1.5.0	-	384.130	9.0	7.0	-	-

Monitoring tools and profiler

To monitor the GPU usage, a bash script was written which periodically runs the `nvidia-smi` command to capture a snapshot of GPU utilization of the running machine. The periodical output was appended to a `.json` file until the application stopped training. A custom parser was developed to extract information from the `.json` file. In addition to the script, TensorFlow's `timeline` module was used as a visual profiler to scrutiny the GPU events took place during training. The `timeline` module actually takes a snapshot of the user-specified training step. So, the code was configured to run `timeline` profiler periodically after every

500 training steps. The profiled data are stored in Chrome trace format and it can be viewed by loading the file in `chrome://tracing` from Google Chrome browser.

4.4 Application Settings

For experimenting with the local version, the input dataset was stored into the machine's local file system whereas for distributed experiments the dataset was stored into HDFS (Hadoop Distributed File System).

Below are the chosen parameters which were controlled during the experiments to observe the efficiency of the proposed scalable solutions.

4.4.1 *num_parallel_calls* in map function

After the training dataset was settled, the input images were forwarded through the input pipeline. A data parsing function responsible for loading the input images from filesystem and then decoding them from jpeg format was mapped to each element of the dataset. The corresponding ground truth values were also loaded from their numpy files in the following step. After the mapping process was done, both of the input images and their associated ground truths were converted into tensors. As stated earlier, these preprocesing steps can be parallelized across multiple CPU cores by specifying the *num_parallel_calls* argument in map function. For both distributed parameter-server and Ring-AllReduce architecture, *num_parallel_calls* was set to four for each worker machine which participated into training.

4.4.2 Number of epochs

In order to learn from the data, the full dataset needs to be passed several times to the same neural network model. When the entire dataset is processed once (through both forward-pass and back-propagation), we refer it as one epoch. The right choice of number of epoch depends on the diversity of the dataset and the convergence of the model. In general, the dataset is repeated several times until the model reaches an acceptable solution. As the main priority of this project is to find the right distributed architecture, the number of epochs was not set indefinite but initially set to constant at 3000. So, regardless of batchsize or the number of workers, the whole dataset was used 3000 times for all the distributed experiments. However, for the initial experiment conducted on local machine, the value was set to 6000 because the GPUs inside that machine had six times more GPU memory than the GPU that is in each machine of the distributed cluster. Due to the time limitation for executing the training, the value was dropped to 3000 for evaluating the distributed architectures.

4.4.3 Number of workers/machines

The distributed experiments started with two workers for both parameter server and ring reduce architectures. The throughput and the completion time collected from that experiment were set as baseline to compare the

scalability. The number of workers was increased in power of 2 up until 8 workers. While experimenting in the local version, the GPUs inside the single machine were defined as workers. The number of workers were set in a similar fashion as the distributed experiments.

4.4.4 Batch size

Setting the optimal batch size is an active area of research and recent literature has shown a tendency to use large mini-batches in an attempt to minimize the number of parameter updates necessary for training the neural network model [32], [12],[39], [17]. Large mini-batches are implemented by increasing the number of parallel workers, where each worker process a fraction of the large batch. As the number of worker grows, so does the batch size. While using data-parallel synchronous Stochastic Gradient Descent (SGD), the effective batch size is set as per worker batch size times the number of workers. Following the procedures of Goyal *et al.*[12], for all of the scalability experiments, the per worker batch size was kept constant when the number of workers were changed to measure the scalability.

The scheme of using large batch can be made more effective if the per worker batch size is set to a large value [12]. Due to the memory limitation of the GPU attached to each machine of our cluster, the batch size per worker was set to 8 images. The distributed experiments were conducted by differing the effective batch size from 16 to 64. For the local experiment, the effective batch size was varied from 128 to 512.

4.4.5 Prefetch size

In all of the experiments, to overlap the work of producer with the work of consumer, three batches of data were prefetched from the dataset ahead of the time of their request. One important thing to be mentioned here is that for distributed experiments, in order to make sure that each worker gets a unique set of data, the whole dataset was divided into N shards, where N represents the number of workers. So, each worker worked with its allocated shard and three batches of data were prefetched and saved into the worker's individual prefetch buffer.

4.4.6 Learning rate

There are many hyper-parameters that can be tweaked to improve the accuracy of the neural network application. It appeared that modifying these parameters can distract from the main objective of this project. So any parameter which can affect the accuracy of the model (except learning rate and batch size) were set to the default values preset by TensorFlow. For all of the experiments, no adaptive learning rate methods were used. Learning rate schedulers or adaptive learning rate are used in purpose of adjusting the pace of updating gradients with respect to the loss function. In general, at the beginning of training, the learning rate is set to a high value and then the value gradually decays as the training progresses. However, for our experiments, the learning rate was kept constant at 0.00001.

4.5 Summary

This chapter illustrates the detailed description of the experiments conducted to study the performance of the Flower Counter application in both local and distributed environment. Beside providing the software and hardware specification of the testbeds, an in-dept review of the application settings and the metrics used to evaluate the results are discussed in a manner so that the whole experimentation process becomes easy to follow. The list of all experiments containing the corresponding details of application settings for both distributed and local environment is given in table 4.4 and table 4.5 respectively.

Table 4.4: Application Settings For Distributed Experiments

Distributed architecture	***num_parallel_calls* in map function**	**Workers**	**Batchsize / per worker**	**Effective batchsize**	**Prefetch buffer size / worker**	**Epochs**	**Learning rate**
Parameter Server	4	2	8	16	24	3000	0.00001
	4	4	8	32	24	3000	0.00001
	4	8	8	64	24	3000	0.00001
Ring-AllReduce	4	2	8	16	24	3000	0.00001
	4	4	8	32	24	3000	0.00001
	4	8	8	64	24	3000	0.00001

Table 4.5: Application Settings For Local experiment

***num_parallel_calls* in map function**	**GPUs**	**Batchsize / GPU**	**Effective batchsize**	**Prefetch buffer size / GPU**	**Epochs**	**Learning rate**
8	2	64	128	192	6000	0.00001
	4	64	256	192	6000	0.00001
	8	64	512	192	6000	0.00001

5 Analysis and Results

This chapter contains the results of the experiments which were outlined in chapter 4. There are two parts to this chapter: (1) accuracy and (2) scalability. Exclusive access to compute resources is crucial for performance related experiments. However, the accuracy doesn't get affected if the CPU is being shared among other users. Accuracy experiments were conducted in a shared environment, and the performance experiments were conducted either on a) a single machine with exclusive access, or b) in a distributed cluster on a private network.

The first part concerns the results that shows the accuracy of the proposed CNN model. This part is subdivided into sections which reflect (a) the evaluation of the trained model on various testsets, (b) a visual representation of the predicted density maps and (c) the progress of loss functions during training.

The second part is about the results that demonstrates the scalability of the distributed architectures in DISCUS lab's homogeneous GPU cluster. A comparison of both architectures (Parameter Server and Ring-AllReduce) is manifested along with the possible reasoning of why one architecture performed better than other. In addition to that, the results from local experiment (Single machine multiple GPUs) are also added to demonstrate the scalability among multiple GPUs within one machine. This parts ends with a discussion about the performance related insights observed from all the local and distributed experiments.

5.1 Accuracy

In the first of the accuracy related experiments, images from camera-day 1109-0704 were chosen to construct the main training dataset. Two different versions of the proposed Flower Counter application were planned to be executed. The first version represents the CNN model which will be trained for 3000 epochs and the second version will be trained for 1000 epochs. The two versions were named *model-m* and *model-l* respectively. Both of the versions have their respective binning strategies as mentioned in section 4.1.1.

The binning technique was applied as described in Section 4.1.1. The whole dataset was divided into seven bins for *model-m*. The bins were assigned 2656, 2109, 2712, 2363, 1763, 2585 and 1911 images respectively. So 1763 images were sampled from each bin which resulted the dataset to contain 12341 images. Out of theses 12341 images, 8638 images were selected for training and 3703 images were chosen for testing.

For *model-l*, the six bins were assigned 2656, 2109, 2712, 4126, 2585 and 1911 images respectively. Sampling 1911 images from each bins filtered 11466 images, from which 8062 images were selected for training set and 3404 for testset. The trained parameters for both of the models were saved in the local file system.

For evaluating the testset, at first, the trained *model-m* was loaded from the its last checkpoint and the testset was executed through the CNN model with the back-propagation being disabled. The images from the testset were again assigned to bins using the same binning technique to see the results of each individual bin. MAE, RMSE and MRE calculated from the result are given in Table 5.1 (left).

As apparent from the results, the CNN model seemed to predict acceptable number of flower counts for the first few bins. A gradual increase of MAE can be noticed, with MRE decreasing over bins containing images of more flowers. This showcases the positive usage of relative error; even though the MAE is maximum in the last bin, the MRE shows when the number of flowers are considered, then the difference of prediction and original value is relatively lower than the first five bins. One disadvantage of using relative error is its sensitiveness to zero flowers in the ground truth which is apparent from the result of the first bin. This led to exclude the results of all images which have zero flowers only for the calculation of MRE.

Table 5.1: Accuracy of Testset 1109-0704

CNN Model trained for 3000 epochs (*model-m*)			
Bin with flower count	**MAE**	**RMSE**	**MRE**
0	0.04	0.23	nan
1	0.70	0.90	0.70
2	1.00	1.23	0.50
3	1.34	1.62	0.45
4	1.53	1.89	0.38
5 - 7	2.38	2.81	0.41
$>= 8$	5.60	7.70	0.39

CNN Model trained for 1000 epochs (*model-l*)			
Bin with flower count	**MAE**	**RMSE**	**MRE**
0	0.50	0.30	nan
1	0.66	0.88	0.66
2	0.87	1.14	0.50
3 - 4	1.26	1.61	0.360
5 - 7	2.08	2.53	0.35
$>= 8$	4.72	6.53	0.44
-	-	-	-

Next, the *model-l* was evaluated and the result from the experiment is given in Table 5.1 (right). Compared to the *model-m*, the metrics didn't differ too much for the first three bins where each bin contained flower images with same count. The rest of the bins were rearranged and if the last bin is compared which contains images of more than eight flowers in both cases, *model-l* achieves 15% better accuracy than *model-m*. Also, considering the time taken to finish training, *model-l* has managed to achieve satisfactory accuracy despite being trained for only one third of the overall training time of *model-m*.

The training of both setups were executed on a shared environment in *octopus.usask.ca*. Out of eight GPUs, two were used for the training. In the shared environment, the training of *model-m* took 81431 seconds to complete 3000 epochs whereas the *model-l* took 27727 seconds for completing 1000 epochs. This indicates that the performance is reasonably linearly scalable, even in a shared environment.

The accuracy of both *model-m* and *model-l* was tested with another camera-day 1109-0705. The accuracy wasn't expected to be very accurate as the images of 1109-0705 weren't used during training. However, experimenting with a dataset which was totally unseen from the trained model can provide insightful observations. One such potential observation could help determining the necessity of including each and every

camera-day into the training set. If the model can infer accurate flower counts for images of a cameraday which wasn't used during training, then it may be practical to skip that camera-day from the training set due to the expensive training time and annotating efforts.

As mentioned earlier in the last part of section 4.1.1, not all images from 1109-0705 were manually annotated. So far 708 images have been manually annotated and these images were used during testing. To demonstrate the accuracy, Kernel density Estimation (KDE) plot is used. The procedure of forming KDE involves drawing Gaussian (normal) curves (centered at each observation) which are summed to compute the value of the density at each point.[1] Due to the nature of the Gaussian KDE process, the curve can extend past the extreme (largest and smallest values) values from both end of the dataset. To control how far past the extreme values of the curve will be drawn, the cut parameter from seaborn python module was used so that any value outside the dataset is discarded. The prediction made from both trained models (3000 and

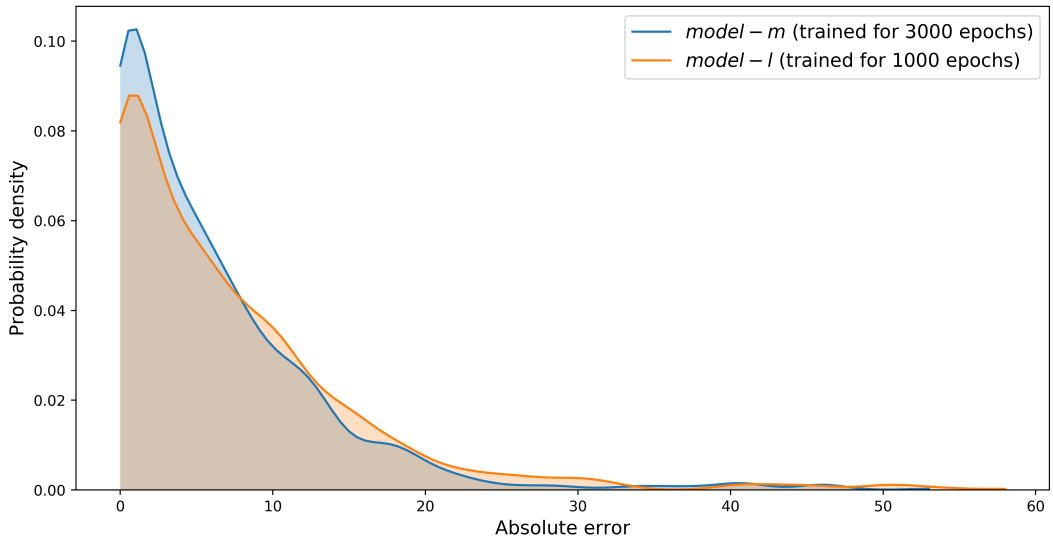


Figure 5.1: Kernel density estimation of the absolute error (Camera-day: 1109-0705).

The absolute error is more dense around 2 when the testset was evaluated by both *model-l* and *model-m*. The probability density gradually decreased for larger values. For *model-m*, the probability of absolute error to land between 0 to 8 was higher compared to *model-l*. When the absolute error was within the range of 0 to 8, the *model-l* performed better than *model-m* in a sense that, the area under the curve was bigger for *model-m* than *model-l*. So for lower values of absolute error, *model-l* performed better than *model-m*. On the other hand, *model-m* performed better than *model-l* when the range of absolute error was within 9 - 32. The difference was almost similar but the area under the curve within the aforementioned range was bigger for *model-l* compared to *model-m*. However, both of the models performed almost same when the absolute errors were within the range 32 - 50. The results from this experiment are satisfactory but not perfect. However, it

[1] `https://seaborn.pydata.org/tutorial/distributions.html` (accessed September 13, 2019)

is encouraging to observe that the models inspired from crowd counting have the capability to learn counting flower as well.

5.1.1 Comparison with older Flower Counter

To compare the proposed Flower Counter with the older version, the manually annotated images of flowers were assumed as ground truths for both applications. The same testset produced for *model-m* which contained 3188 images from camera-day 1109-0704 were used for the comparison. Same as before, Kernel density estimation (KDE) plot of absolute error is calculated. The result is plotted in Figure 5.2.

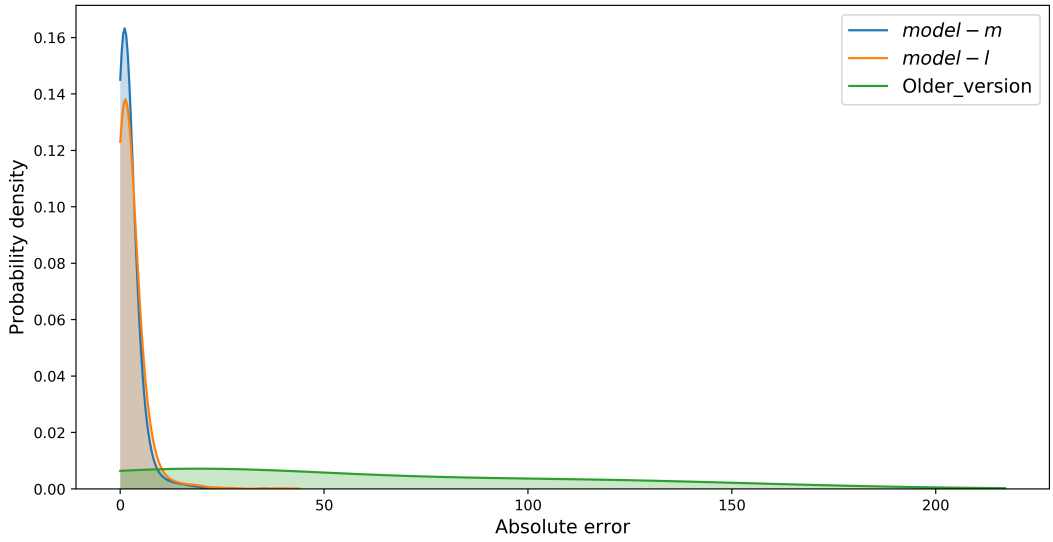


Figure 5.2: Comparison of absolute error using Kernel density estimation (Camera-day: 1109-0704)

For both *model-l* and *model-m*, the probability of absolute error to occur between 0 to 50 was 1 (the area under the curve within the min and max range represents the total probability). Compared to that for the older version, the probability of absolute error to be within the range 0 to more than 200 is one. This long range shows how unstable the older Flower Counter is, whereas for both *model-l* and *model-m* the absolute error was confined within 50. This proves the proposed CNN model performed significantly better in counting the right number of flowers than the previous Flower Counter when compared with the manually annotated ground truth.

5.1.2 Predicted density maps

Depending on the distribution of the flower counts for camera-day 1109-0705 (given in Figure 5.3), the result from Figure 5.1 may not look compelling, but what was surprising was the predicted density map made from the trained models. A sample of prediction from the testset 1109-0704 and 1109-0705 are given respectively

in Figure 5.4 and 5.5. The flower counts predicted by the models were float values which were rounded to nearest integer in these figures.

It can be seen under different scenarios, both of the trained CNN models were able to detect the right location of the flowers. Even during sunny weather, when the sun reflection could have made impact on the flower detection or in situations when there were dirt on the adjacent roadside of the canola field, the CNN models were able to correctly distinguish flowers from dirt (Figure 5.4 (b,e,h)) and sun reflection (Figure 5.5 (b,e,h)). Both the sun reflection and the dirt were not erroneously detected as flowers this time and the quality of the flower counts were satisfactory with respect to the previous Flower Counter. Similarly, for cases when the flowers were spread across different location of the image (left and rightmost columns from Figure 5.4 and 5.5), the CNN models were able to detect them as well. The models have not yet reached to perfection as there are still few undetected flowers in cases of excess sun reflection and dirt roads but in future work this will be the top priority to make the present Flower Counter even better.

Now, if the CNN models are compared between each other using the predicted density maps seen in Figure 5.4 and 5.5, then it becomes clear that *model-l* was accurate in detecting the right location of flowers (Figure 5.4 (d,g) and Figure 5.5 (d,g)). Despite these differences, both of the models demonstrate the learning capability of the modified version of the MultiColumn Convolutional neural network that was chosen to construct the Flower Counter application.

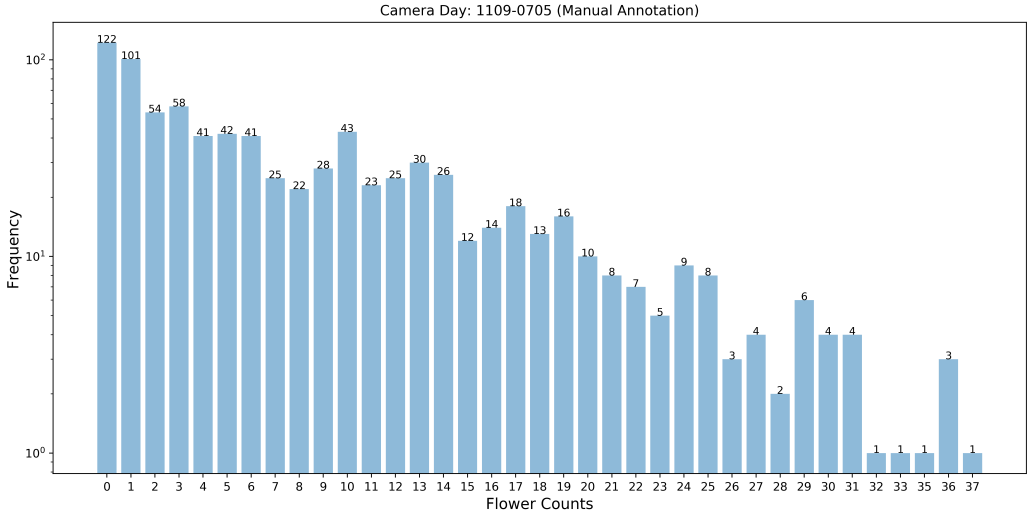


Figure 5.3: Bar Graphs of flower counts (Manual Annotation).

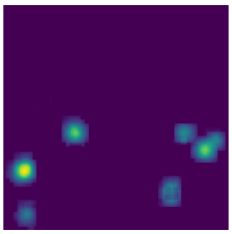
5.1.3 Inference time

In order to compare how fast the proposed CNN-model infers the flower counts, the older version of the Flower Counter application was rerun with the same 3188 images (from camera-day 1109-0704). The older

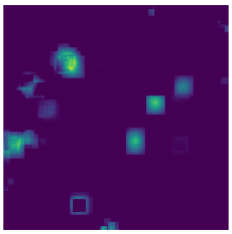
(a) Image Tile: frame000555_0_1 from Camera Day: 1109-0704 (manual count : 12)

(b) Image Tile: frame000983_0_0 from Camera Day: 1109-0704 (manual count : 9)

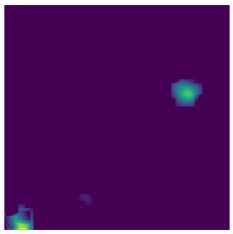
(c) Image Tile: frame000350_2_4 from Camera Day: 1109-0704 (manual count : 3)



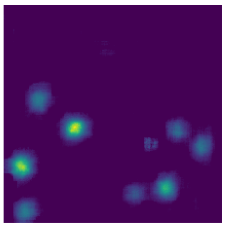
(d) Predicted Density Map from *model-m* : frame000555_0_1 (predicted count : 7)



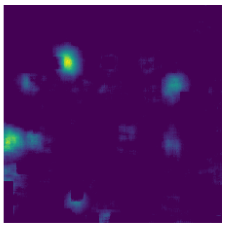
(e) Predicted Density Map from *model-m* : frame000983_0_1 (predicted count : 7)



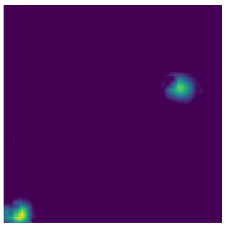
(f) Predicted Density Map from *model-m* : frame000350_2_4 (predicted count : 3)



(g) Predicted Density Map from *model-l* : frame000555_0_1 (predicted count : 9)

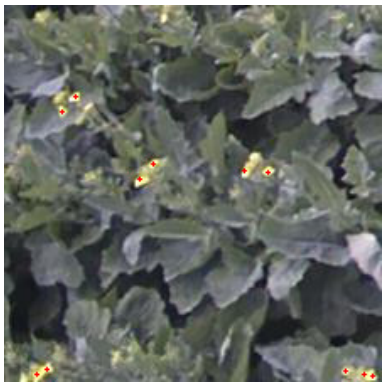


(h) Predicted Density Map from *model-l* : frame000983_0_1 (predicted count : 6)



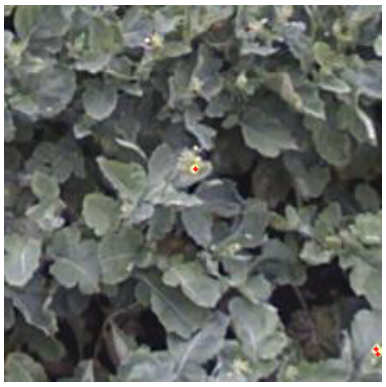
(i) Predicted Density Map from *model-l* : frame000350_2_4 (predicted count : 3)

Figure 5.4: Predicted Density Maps from the trained CNN models

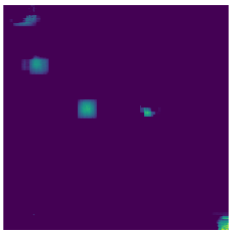


(a) Image Tile: frame000719_2_0 from Camera Day: 1109-0705 (manual count : 11)

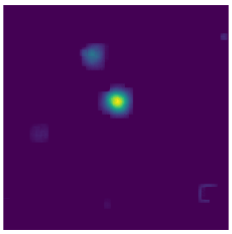
(b) Image Tile: frame000882_1_3 from Camera Day: 1109-0705 (manual count : 14)



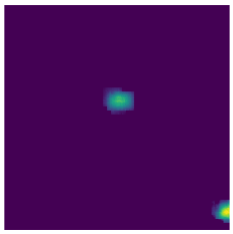
(c) Image Tile: frame000599_2_3 from Camera Day: 1109-0705 (manual count : 3)



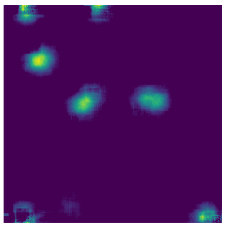
(d) Predicted Density Map from *model-m* : frame000719_2_0 (predicted count : 4)



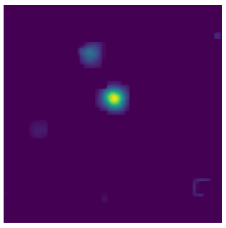
(e) Predicted Density Map from *model-m* : frame000882_1_3 (predicted count : 4)



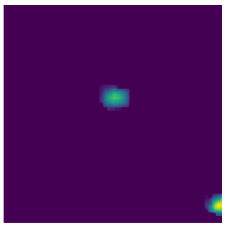
(f) Predicted Density Map from *model-m* : frame000599_2_3 (predicted count : 2)



(g) Predicted Density Map from *model-l* : frame000719_2_0 (predicted count : 8)



(h) Predicted Density Map from *model-l* : frame000882_1_3 (predicted count : 4)



(i) Predicted Density Map from *model-l* : frame000599_2_3 (predicted count : 2)

Figure 5.5: Predicted Density Maps from the trained CNN models

version took 220 seconds to predict the flowers, whereas the CNN-based Flower Counter application took 29 seconds to infer flower counts. That means once the CNN model is trained, it can process the same number of images 7.5X faster than the previous Flower Counter.

5.1.4 Progress of loss function

Before concluding the accuracy part, one more thing which needs to be discussed is the how the CNN models progressed minimizing loss during training. Figure 5.6 portrays the progress of loss that the CNN model made on each epoch while being trained for *model-m*. It is apparent from the Figure that there was an instantaneous drop of loss within the first two epochs. In each epoch, there were 58 training steps, which means 58 batches of images were processed during one epoch. The instantaneous drop visible in the first two epochs was actually a result of gradual decrease over first 116 training steps. However, the rate of decrease of loss was not the same in the subsequent training steps. The loss value of 4 was consistent from 2nd until 250th epochs then gradually reached approximately around 1 from 1000 to 3000 epochs. The progress of the loss function during distributed training is given in Figure 5.7a and 5.7b, representing the results from Parameter Server and Ring-AllReduce respectively.

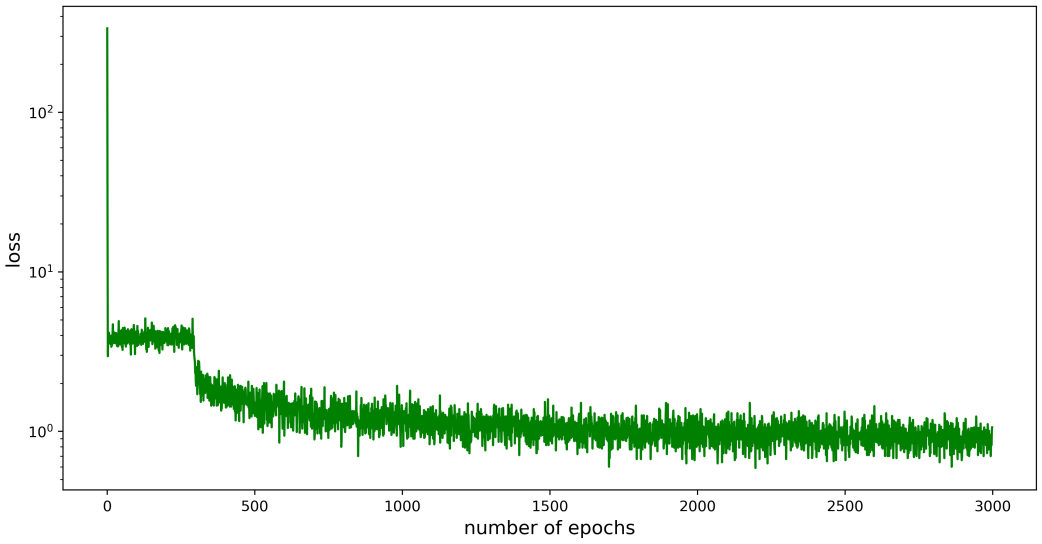
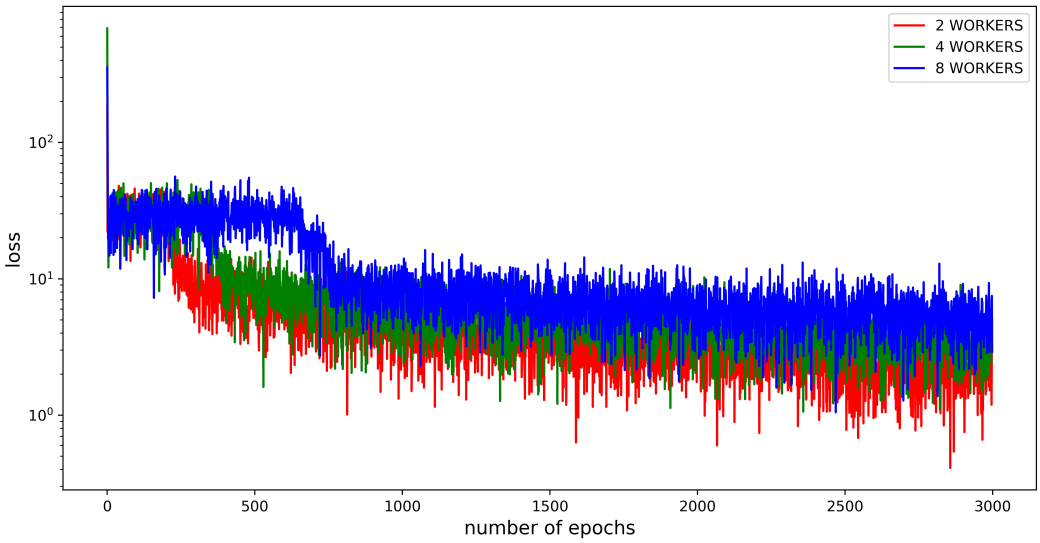


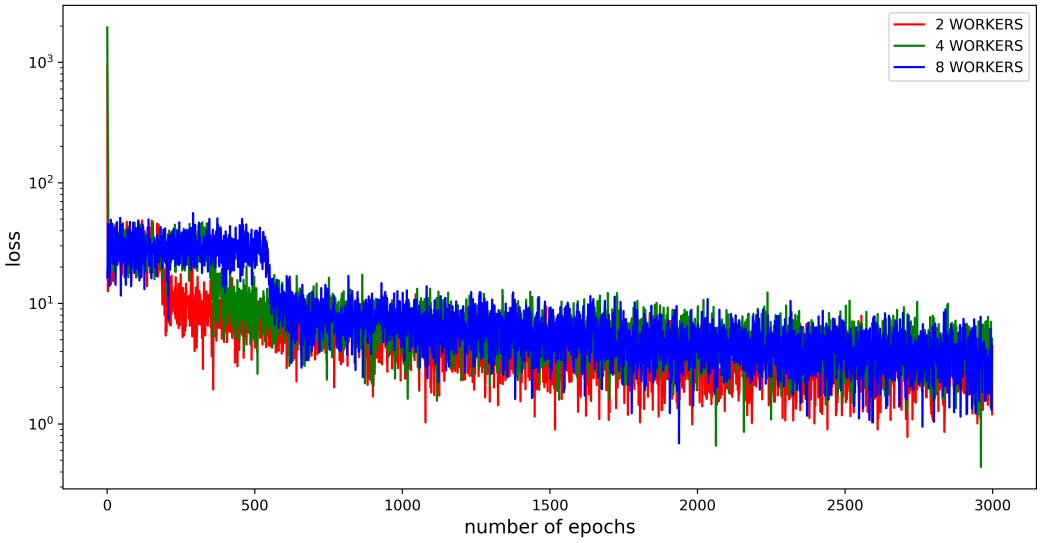
Figure 5.6: Progress of loss function during training.

For both of the distributed architectures, two workers took fewer epochs compared to four and eight workers to reduce the loss values. For example, it can be seen from figure 5.7a that two workers took around 225 epochs to reduce the loss below 20 whereas four and eight workers took around 350 and 600 epochs respectively to reach below 20.

After 3000 epochs, two workers managed to reach to the lowest loss value compared to four and eight workers. However, the difference was not that significant.



(a) Parameter Server



(b) Ring-AllReduce

Figure 5.7: Progress of loss function during distributed training

Another fact that can be noticed is that for different set of workers, Ring-AllReduce performed slightly better than parameter-server. For example, during Ring-AllReduce, eight workers took around 600 epochs to minimize the loss under 20, whereas the same number of workers took 750 epochs to reach the same loss value when Parameter Server was used. More investigation is needed to understand the reasons behind it.

5.1.5 Discussion

After evaluating results from all the aforementioned experiments, it was clear that the proposed Flower Counter is more accurate than the older Flower Counter. In terms of the time it takes to infer the number of flowers, the proposed CNN-model is faster than the previous Flower Counter on one dataset. When both of the CNN-models (*model-m* and *model-l*) were trained with dataset comprised of images from only camera-day 1107-0704, the *model-m* achieved better accuracy. On the other hand, for camera-day 1109-0705 which was not used during training at all, the *model-l* performed better than *model-m*. It is believed that the lack of diversity in the input images and the large number of epochs the *model-m* was trained for, have made *model-m* to perform better than *model-l* for camera-day 1109-0704.

5.2 Scalability

5.2.1 Total Completion Time

The first set of distributed experiments used the Parameter Server architecture. The experiments began with one Parameter Server and two workers. Based on the number of workers, the dataset along with the corresponding ground truths was sharded into two chunks so that each worker received a chance to work with a unique subset of data. Before training, in the pre-processing stage, the *num_parallel_calls* in map function was set to 4. Following the same manner, for the next consecutive experiments, the number of Parameter Server was kept static at 1, but the number of workers were increased by factors of 2. In this way, the scalability was measured on 2 to 8 workers. The number of epochs was set constant at 3000. The whole distributed experiments were replicated twice. After completing the second run of the experiments, the difference between the results from the first run was was observed to be less than 1%. Due to the similarities, only the results from the first run of the distributed experiments will be described in this section.

The total completion times measured from the first run of the experiments are given in Figure 5.8. The baseline for the distributed experiments is set to the time it takes for one Parameter Server and two workers to complete training 3000 epochs. For Parameter Server experiments, two workers finished training in 92585 seconds. Using four worker machines dropped the completion time to 55712 seconds, which is 1.67 times faster than the baseline. Increasing the number of worker machines to eight decreased the total training time to 31117 seconds, which achieved 3x speedup compared to the baseline.

The next set of distributed experiments used the Ring-AllReduce architecture. The same procedures were

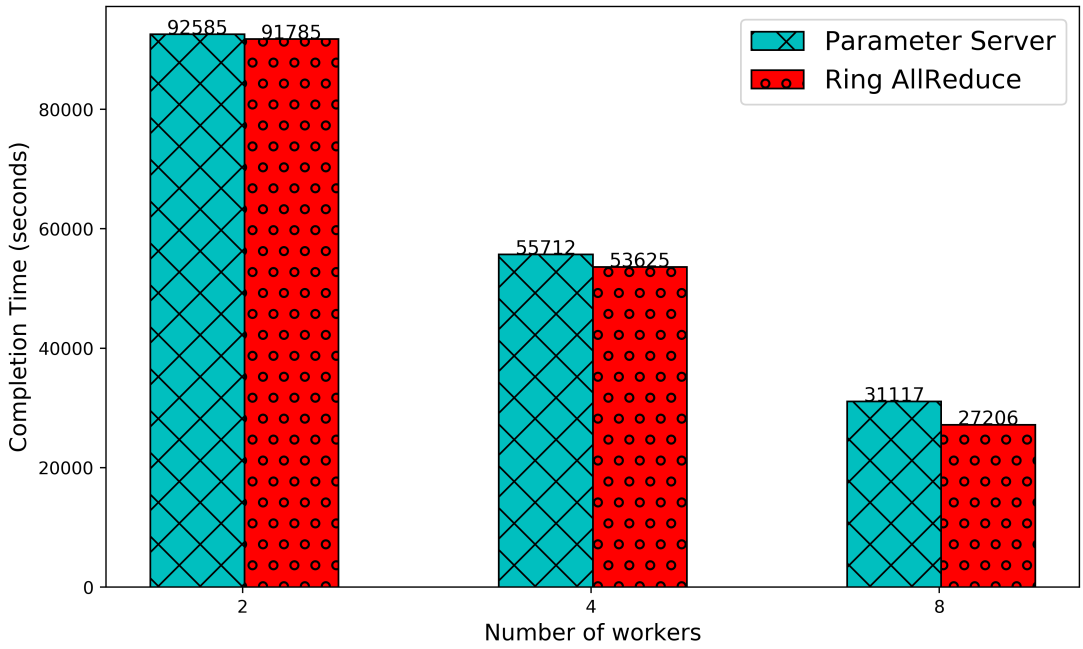


Figure 5.8: Comparison of Parameter Server and Ring-AllReduce - Total Completion Time (First run)

followed as the Parameter Server: the number of workers was varied from two to four to eight and the number of epochs were set to 3000.

For Ring-AllReduce, two worker machines completed the training in 91785 seconds. Compared to two workers from Parameter Server, no significant improvement was observed. However, a slight improvement started to be noticed when the number of workers were increased to four and eight. Compared to the Parameter Server architecture, using four and eight workers helped to achieve 3.7% and 12.6% improvement respectively.

For the Parameter Server architecture, when the number of worker machines were increased from two to four to eight, a near linear speedup was achieved. For the Ring-AllReduce experiments, if the result from two worker machines is set to baseline then after increasing the machines to four and eight increased the speedup to 1.71x and 3.37x respectively.

5.2.2 Average throughput (images per second)

The average throughput is presented in Figure 5.9. For Parameter Server, increasing the worker machines from two to four increased the throughput by 1.7 times. While being increased from two to eight workers, the throughput increased by 3.22 times.

While comparing Ring-AllReduce with Parameter Server, the average throughput of Ring-AllReduce when

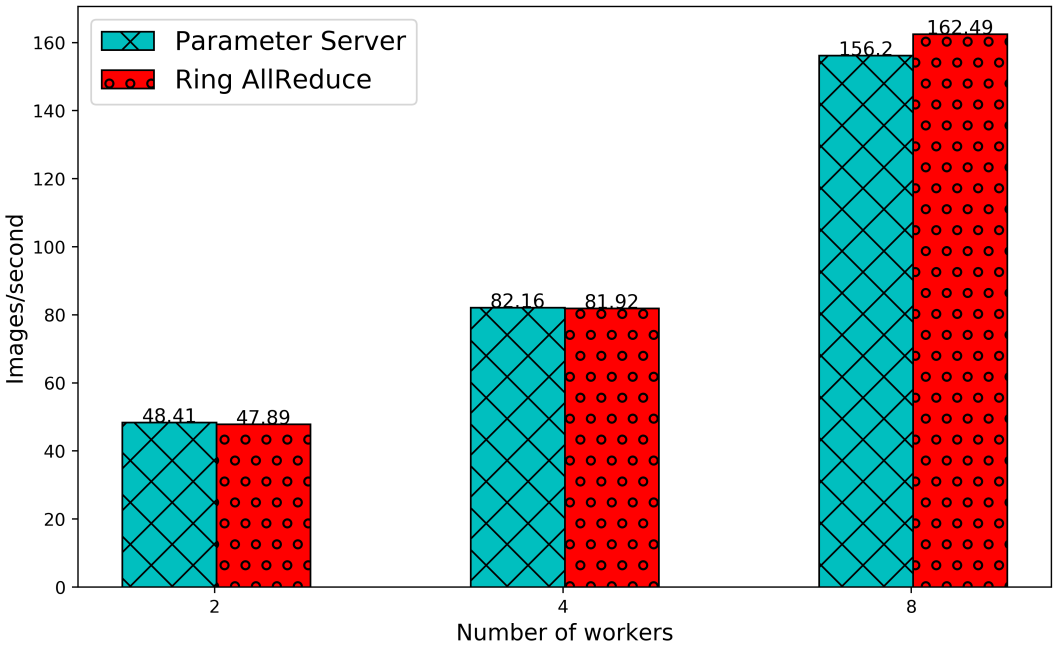


Figure 5.9: Comparison of Parameter Server and Ring-AllReduce - Average Throughput (First run)

two and four workers were used was not better than the Parameter Server. However, when the number of workers were increased to eight, Ring-AllReduce managed to achieve higher throughput. The comparison seen in Figure 5.9 did not reflect what was observed in Figure 5.8. Figure 5.8 displayed that Ring-AllReduce is better compared to Parameter Server in all cases when the number of workers were increased. This contradicts the almost identical results of two and four workers in Figure 5.9, as it shows Parameter Server provided higher average throughput than Ring-AllReduce, which means Parameter Server should be faster in completing the epochs as more images were trained per second than Ring-AllReduce.

This can question how meaningful the average throughput is. In order to get more insight, the distribution of the throughput was investigated for both of the distributed architectures. The mean, standard deviation and variance of the throughputs were calculated to see the spread. The result is portrayed in Table 5.2. For Parameter Server architecture, when the number of workers were increased, the standard deviation of throughput increased as well. The same scenario was observed for Ring-AllReduce, but compared to Parameter Server, the spread was not noticeably smaller. It means the throughput from the Ring-AllReduce experiments were more stable whereas for the Parameter Server experiments, the throughput was fluctuating.

5.2.3 GPU Utilization

As previously mentioned in Section 4.3.2, a bash script was written to profile GPU utilization that periodically captures the snapshot of the status of the GPU in the worker machine. When the application starts in each

Table 5.2: Comparison of Parameter Server and Ring-AllReduce - Statistics of the throughput (images/sec)

Parameter Server			
Number of Workers	**Mean**	**Standard Deviation**	**Variance**
2	48	4	17
4	82	9	73
8	156	27	705

Ring-AllReduce			
Number of Workers	**Mean**	**Standard Deviation**	**Variance**
2	48	3	8
4	82	5	25
8	162	10	99

worker, the bash script also starts and captures the status every five seconds for each GPU and saves the information to a .json file. For distributed synchronous training, in case of events when a worker performs poorly, it affects the performance of other workers. For a long training step, if one worker takes a longer time to compute the gradient then the other workers have to sit idle, and thus can cause poor GPU utilization. So, the GPU utilization from one worker can provide a high level abstraction of possible bottlenecks of the whole distributed training.

For all distributed experiments, the profiled data only from the chief worker is provided. The chief worker is the worker which periodically saves the checkpoint files, counts global training steps and coordinate the graph initialization with other workers. For Parameter Server and Ring-AllReduce the worker with task index 0 is automatically selected as the chief worker.

Table 5.3: Parameter Server vs Ring-AllReduce - GPU Utilization (Chief worker)

Parameter Server			
Number of Workers	**Mean**	**Standard Deviation**	**Variance**
2	93	20	389
4	64	38	1446
8	69	33	1089

Ring-AllReduce			
Number of Workers	**Mean**	**Standard Deviation**	**Variance**
2	96	14	191
4	97	12	150
8	96	15	215

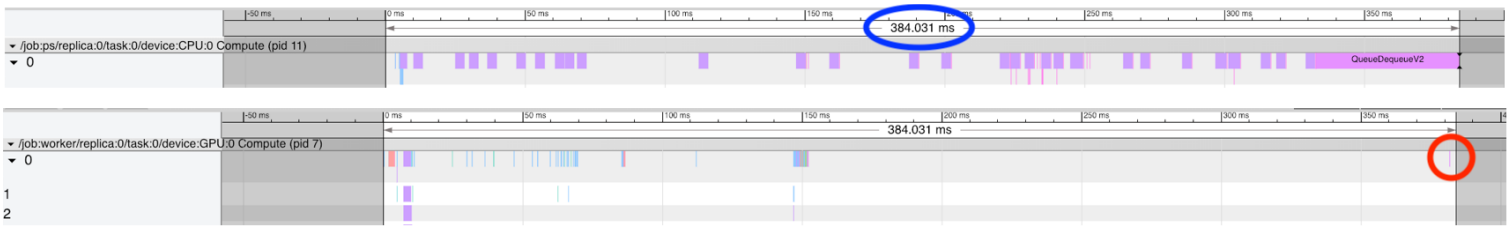
The results are given in Table 5.3. As apparent from Table 5.3, the Ring-AllReduce is consistent in maintaining same GPU utilization even when the number of workers were increased. As opposed to that, in Parameter Server, the mean GPU utilization started to decrease when the number of workers were kept increasing. The Ring-AllReduce was also stable in achieving higher GPU utilization for each individual experiment as the standard deviation shows the spread was not significant compared to the Parameter Server.

Though the GPU utilization is a good metric to get the high level abstraction of how much time the GPU is occupied for, it has some shortcomings too. Depending on the small batch size that every GPU had to process, on average, the training time of each batch did not take more than 500 ms for every experiment. As the training interval was set to five seconds, the snapshot taken after every five seconds may have captured

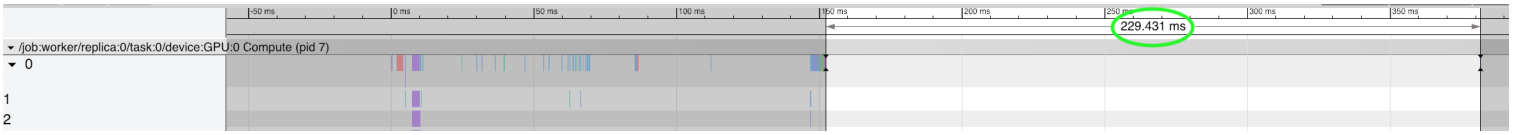
the exact time the input tensors were being transferred to GPU or the time when the GPU just finished processing a training step. However, it is highly unlikely that the GPU utilization captured after the fixed period of time will always be in the middle of situations when the GPU is idle.

To capture more insight from fine-grained level about what happened during training, the GPU utilization alone cannot be enough. So a profiler called TensorFlow Timeline was used alongside the GPU utilization. The profiler was configured from the TensorFlow code to start running when training started and the profiling interval was set to every 500 training steps (i.e., the training step after every 500 training steps (processing of 500 batches) was profiled and saved to each worker's local filesystem). Compared to the bash script mentioned above, the profiler captures data for the whole duration of the training step and keeps information of every *oprations* which took place during that interval.

An investigation of timeline profiling for one Parameter Server and four workers is given in Figure 5.10. The visualization was created from the timeline file of 99000th training step profiled by the chief worker. Beside containing information about the worker (who profiled the timeline), the file also contains information about the machine which hosts the *ps task*. Chrome's trace event profiling tool was used to visualize the timeline file.



(a) Training events in Parameter Server and Chief Worker.



(b) Longest GPU stall time in Chief worker.

Figure 5.10: Timeline of events during 99000th training Step from Parameter Server Architecture (one PS, four workers)

In Figure 5.10 (a), the Parameter Server is identified as "/job:worker/replica:0/task:0/device:CPU:0" and the chief worker is referred as "/job:worker/replica:0/task:0/device:GPU:0". From figure 5.10 (a) the duration of the training step can be seen as 384.031 ms (marked in blue ellipse). The last *operation* to occur in the GPU is the *RecevTensor* (marked in the red circle) and there is a significant gap before it which represents nothing happened in the GPU on that period of time. The duration of that gap is 229.431 ms, which can be seen from 5.10 (b) (marked in green ellipse). This exhibits that for this training step, 60% of the time the GPU remained inactive. Though the GPU was not doing anything, the Parameter Server was working at that time (visible from Figure 5.10 (a)). This can be a possible reason why the Parameter Server had poor GPU utilization compared to the Ring-AllReduce as the gradient aggregation happens in the GPUs

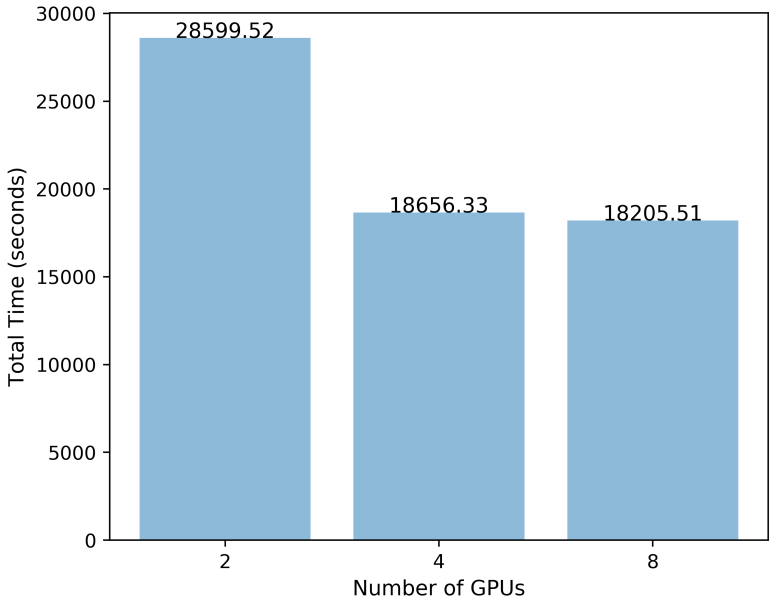


Figure 5.11: Bar graph of total completion time (seconds) - Single machine multiple GPUs

for the latter architecture, whereas in the former architecture, the workers have to wait for Parameter Server machine which receives the gradients from all workers and aggregates the gradients within itself.

5.2.4 Local experiment (Single machine multiple GPUs)

Recall from Section 4.1.2, the main purpose of the local experiment was to test the scalability of TensorFlow's *In-graph replication* method. Exclusive access to the *octopus.usask.ca* machine was granted and the experiment was executed only once. The CNN application was trained in two, four and eight GPUs. Figure 5.11 demonstrates the total completion time for these three setups.

If the result from the two GPUs is set as the base line, as the number of GPUs was increased from two to four, a 1.53x speed up was achieved, whereas increasing the GPUs to eight resulted in 1.57x improvement. Though increasing the number of GPUs reduced the total completion time, the scalability of *In-graph replication* was poor when the GPUs were increased from four to eight.

Similar behaviour was observed for the throughput. The results are given in Table 5.4. The instability of maintaining linear average throughput started to get more prominent when the number of GPUs surpassed four.

Table 5.4: Single machine multiple GPU - Statistics of the throughput (images/sec)

Number of GPUs	Mean	Standard Deviation	Variance
2	344	59	3440
4	614	158	25083
8	678	288	82986

5.2.5 Discussion

For all of the distributed experiments, Ring-AllReduce performed slightly better compared to the Parameter Server architecture. It was observed that the efficiency of Parameter Server gets affected when the number of workers are kept increasing. In contrast to that, Ring-AllReduce provided better utilization of GPUs and better throughput. As a consequence of these, Ring-AllReduce was able to finish training 3000 epochs faster than the Parameter Server.

It is too early to come to conclusion that Parameter Server is not as efficient as Ring-AllReduce. The Parameter Server was implemented based on the Standard Distributed TensorFlow and during the entire project, frequent software patches and updates were observed to get released for TensorFlow to fix bugs or improve performance. For few experiments, using the latest version of TensorFlow significantly improved performance, which led to the need to rerun the experiments many times for fair comparison. However, the core computational operations are believed to be stable as Horovod uses the same TensorFlow operations to implement the convolution or pool layers, but the communication pattern during distributed training was different. As a future work, the implementation of the Parameter Server architecture using a different framework such as PyTorch or TensorFlowOnSpark can be compared with Horovod's implementation of Ring-AllReduce. However, until that comparison Ring-AllReduce appears to have the best scalability to scale the Flower Counter application.

For the distributed experiments, the number of Parameter Server was not increased past one because it appeared to me that based on the size and number of parameters that need to synchronized (See Section 3.2 for more information), using more than one Parameter Server can decrease the performance and could be an overkill for a setup like ours. The size of trainable parameters is 1279140 bytes which is roughly 1.24 MB, so, sharding this small size of parameter sets into multiple Parameter Server machines can introduce additional overhead during training; thus only one Parameter Server was used.

6 Conclusion

6.1 Summary

This project concerned the three research questions raised in section 1.1.1. The first question was about the possibility of developing a CNN-based flower counting application which is more accurate than the previous version. Through several experimentations, it was observed that indeed it was possible to make a better alternative of the existing flower counting application by combining two other CNN models which were previously used for counting purposes in other challenging scenarios.

One caveat of using the proposed model was to prepare the ground truth density maps which required manual annotation of each individual flower visible in the collected images. Manual annotation is a lengthy tiresome task and at the initial annotating process in order to advance the remaining works of this thesis, the scalability tests were parallelly conducted by assuming the output from the previous version of flower counter as the ground truth.

Using one manually annotated camerday (1109-0704), the CNN model was trained for both 1000 and 3000 epochs to see how setting the number of epochs to large and small values can affect the accuracy of the model. Training the CNN for 3000 epochs led the model to be overfitted, whereas when trained for 1000 epochs, the model became more generalized when it was tested with dataset comprised of sampled images from the next camera-day (1109-0705) which was not used for training at all.

The second research question asks if the inference time of the newly built application is faster than the previous one. It was found that once finished training, after loading the model from the last checkpoint, the inference time of the new application was 7.5x faster than the previous solution.

The third and final research question was if the new flower counter application is capable of being sped up by the adaptation of a distributed architecture which efficiently uses resources of multiple machines each equipped with at least one GPU. To find the answer to this question, two different distributed training architectures, called Parameter Server and Ring-AllReduce were chosen.

The experiments from section 5.2 revealed that both Parameter Server and Ring-AllReduce architectures were successful to improve the training time when more workers were added. Both of the architectures were close to achieving the same throughput and completion time but Ring-AllReduce performed slightly better since the number of workers used during training surpassed four. Once set up, the Ring-AllReduce was easy to use in the distributed system, as a single line of command from the chief workers terminal initiated all the necessary connection among workers and started training. The results from experimentations prove that the

proposed flower counting application is scalable in distributed environment.

6.2 Thesis Contributions

This thesis explored potential areas to create an alternative version of the existing flower counting application which is more improved in terms of both accuracy and performance.

- The first and foremost contribution of this thesis is proposing a CNN model which combines the ideas of two other CNN previously used in counting people in dense crowd [43] and bacterial cell estimation [38]. Through several experimentations, it was observed that the proposed CNN model is well adaptable for counting flowers and it is more accurate than the previous version.

- Secondly, this thesis contributes to exploring two different distributed architectures called Parameter Server and Ring-AllReduce. Besides showing the architectural advantages, the comparison helped to better understand how a scalable solution can reduce the total completion time by efficiently utilizing the GPU resource and distribute the work among all workers. The insight from the studies can be extended and implemented in scaling other CNN-based applications as well. In addition to that, this thesis does an in-depth analysis of the total memory profiling, memory consumption and potential bottleneck of the proposed CNN model which information is crucial during syncing gradients among all workers in distributed training.

- Thirdly, this thesis contributes to finding the optimal settings to implement data parallel model using Tensorflow. It was observed that for data parallel model, *Between-graph replication* technique is more scalable than *In-graph replication*.

- Finally, the work of this thesis demonstrated the importance of better sampling techniques that can lead to predicting accurate number of flowers by being trained on sampled images which are better representatives of the entire population. This proves that, unlike the previous version, the CNN-based model may not need to include all the images of the available camera-days. The new solution is capable of achieving better accuracy, even if it gets trained on a fewer number of images when compared to the previous version.

6.3 Future Work

There were many areas which were planned to be included in this thesis but could not be completed due to their large scope. There are some areas which were also found to be worthwhile to explore as an extension of this work. After reviewing the previous works and the issues that could not get solved in this work, the following points are selected to be left for future works.

- Binomial reduction tree: The results produced from section 5.1 exhibits minor differences between Parameter Server and Ring-AllReduce architecture. In a pursuit to explore even better communication strategy in a distributed environment, other alternatives of Ring-AllReduce should be explored. One such communication pattern which looks promising to study for the future work is the binomial reduction tree-based solution which was explored in these [3],[15] and [12] literature.

- Micro-services: In an attempt to make the training process more easy to use, quite a few bash scripts were written to standardize the training setup for each machine of the distributed cluster. Transforming the distributed training into micro-services was a part of the initial outline but considering the cluster size, it was felt that using microservices can turn out to be an overkill for this project. However, the option should not be ruled out, as in near future when more machines will be added to the cluster there will be a definite need of using cluster orchestration and resource provisioning to scale the application without requiring manual intervention.

  For containerization and container orchestration, Docker and Kubernetes are selected respectively as the primary tools to transform the distributed training into micro-service. Due to Kubernetes's wide community support and high availability of resources to master its container orchestration platform, its believed Kubernetes will be the perfect resource provisioning tool for the upcoming projects.

- Better Profiling tools: Another important tool which was felt more important while experimenting with the scalability of the application was the use of a standard profiler to monitor clusters health and current status. Ganglia is the current monitoring tool which is set to measure the resource utilization of the distributed cluster of DISCUS Lab. Ganglia wasn't set up to monitor GPU utilization and for that reason, a separate script was used to measure the GPU usage during training. Considering the efforts and the steep learning curve to set up and master Ganglia, it is preferred to use a better cluster monitoring tool. After exploring many candidates, Promethues[1] was found to be a robust distributed system monitoring tool which can be well integrated with Kubernetes to check the overall health and current status of available resources of the distributed cluster.

- Data sampling technique: The binning technique applied for sampling the images worked within expectation. In future, the technique is expected to perform even better when more images from the rest of the camera-days will be added to the dataset. One caveat of using the technique is that for this project, the binning range was set manually; as more camera-days will be added after completion of manual annotation, the binning decision will need to be changed as well. A more general, automatic approach for binning setup needs to be outlined which will be addressed in the future work of this project.

- Other neural network frameworks: Besides TensorFlow, there are other neural network frameworks which can be explored to see if they are more robust to use in the distributed environment. Distributed

[1] `https://github.com/prometheus/prometheus` (accessed September 13, 2019)

PyTorch and TensorFlowOnSpark are the next frameworks that can be compared with distributed TensorFlow to study the communication strategy for gradient sharing and aggregating. At the time of this writing, PyTorch has support to build on top of different communication backends without the need for other additional frameworks. By default, PyTorch provides support for TCP, Gloo[2] and MPI backends to share gradients during distributed training.[3] TCP backend doesn't have support for communication routines on GPUs but the rest of the two backends have. Besides the architecture, PyTorch shares solid documentation of the framework and in-detail tips and tricks of production-level implementation of synchronous SGD which makes the framework a viable candidate to explore in the future projects. Also it should be noted that PyTorch has its own implementation of Ring-AllReduce. For the next projects, Horovod on top of PyTorch can be compared with the PyTorchs own implementation of Ring-AllReduce to study which one performs better.

- Generalization for other applications: The distributed parameters sharing strategies reviewed in this project can be applied to other CNN based applications to improve their scalabilities in distributed systems. Though the same performance isn't guaranteed to be achieved due to architectural differences among different CNN models, the data parallelism is assured to be useful to enhance the overall completion time to some extent. Though different CNN models can benefit from the same parallelization strategy, other neural network architecture such as Recurrent neural network may not utilize the GPU resources in the same way due to its cascade dependent structure [24].

[2] `https://github.com/facebookincubator/gloo` (accessed September 13, 2019)
[3] `https://pytorch.org/tutorials/intermediate/dist_tuto.html` (accessed September 13, 2019)

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, Savannah, GA, November 2016.

[3] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 193–205, Austin, TX, February 2017.

[4] Víctor Campos, Francesc Sastre, Maurici Yagües, Jordi Torres, and Xavier Giró-i Nieto. Scaling a convolutional neural network for classification of adjective noun pairs with tensorflow on gpu clusters. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 677–682, May 2017.

[5] Joseph Paul Cohen, Henry Z. Lo, and Yoshua Bengio. Count-ception: Counting by fully convolutional redundant counting. *CoRR*, abs/1703.08710, 2017.

[6] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 4:1–4:16, London, UK, April 2016.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, and Quoc V Le. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, Lake Tahoe, NV, December 2012.

[8] Bernard Fraenkel. For machine learning, it's all about gpus. `"https://www.forbes.com/sites/forbestechcouncil/2017/12/01/for-machine-learning-its-all-about-gpus/#48258e4b7699"`, December 2017. (Accessed on 12/01/2018).

[9] A. Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

[10] Andrew Gibiansky. Svail tech notes: Baidu open-sources fast allreduce library for deep learning. `https://www.youtube.com/watch?v=KRaVNotxmsg&t=3s`, February 2017.

[11] Javier Garcia Gonzalez. *Automatic Counting of Canola Flowers from In-Field Time-Lapse Images*. PhD thesis, University of Saskatchewan, May 2018.

[12] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[13] Krzysztof Grajek. Counting objects with faster r-cnn. `https://softwaremill.com/counting-objects-with-faster-rcnn/`, June 2017.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, Las Vegas, NV, June 2016.

[15] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. Firecaffe: Near-linear acceleration of deep neural network training on compute clusters. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2592–2600, Las Vegas, NV, June 2016.

[16] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678, Orlando, FL, November 2014.

[17] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, Lake Tahoe, NV, December 2012.

[19] Victor Lempitsky and Andrew Zisserman. Learning to count objects in images. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, NIPS'10, pages 1324–1332, Vancouver, Canada, December 2010.

[20] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 583–598, Broomfield, CO, October 2014.

[21] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: A rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 41–54, Carlsbad, CA, October 2018.

[22] Mark S. Nixon, Xin U. Liu, Cem Direkolu, and David J. Hurley. On using physical analogies for feature and shape extraction in computer vision. *The Computer Journal*, 54(1):11–25, August 2009.

[23] Daniel Oñoro-Rubio and Roberto J. López-Sastre. Towards perspective-free object counting with deep learning. In *Computer Vision – ECCV 2016*, pages 615–629, Amsterdam, The Netherlands, October 2016.

[24] Peng Ouyang, Shouyi Yin, and Shaojun Wei. A fast and power efficient architecture to parallelize lstm based rnn for cognitive intelligence applications. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, Austin, TX, June 2017.

[25] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[26] Rolf Rabenseifner. Optimization of collective reduction operations. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1–9, Kraków, Poland, 2004.

[27] D. B. Sam, S. Surya, and R. V. Babu. Switching convolutional neural network for crowd counting. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4031–4039, Honolulu, Hawaii, July 2017.

[28] Igor Saprykin. Distributed tensorflow (tensorflow dev summit 2018). `https://www.youtube.com/watch?v=-h0cWBiQ8s8`, March 2018.

[29] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[30] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, San Diego, CA, May 2015.

[31] Vishwanath A. Sindagi and Vishal M. Patel. A survey of recent advances in cnn-based single image crowd counting and density estimation. *Pattern Recognition Letters*, 107:3 – 16, 2018.

[32] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. *CoRR*, abs/1711.00489, 2017.

[33] Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, Ashish Verma, and Sameer Kumar. Efficient training of convolutional neural nets on large distributed systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 392–401, Belfast, UK, September 2018.

[34] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, Boston, MA, June 2015.

[35] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[36] Chuan Wang, Hua Zhang, Liang Yang, Si Liu, and Xiaochun Cao. Deep people counting in extremely dense crowds. In *Proceedings of the 23rd ACM International Conference on Multimedia*, pages 1299–1302, Brisbane, Australia, October 2015.

[37] Daniel Weimer, Bernd Scholz-Reiter, and Moshe Shpitalni. Design of deep convolutional neural network architectures for automated feature extraction in industrial inspection. *CIRP Annals*, 65(1):417 – 420, 2016.

[38] Weidi Xie, J. Alison Noble, and Andrew Zisserman. Microscopy cell counting and detection with fully convolutional regression networks. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization*, 6(3):283–292, 2018.

[39] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32k for imagenet training. *CoRR*, abs/1708.03888, 2017.

[40] Cong Zhang, Hongsheng Li, X. Wang, and Xiaokang Yang. Cross-scene crowd counting via deep convolutional neural networks. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 833–841, Boston, MA, June 2015.

[41] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, pages 181–193, Santa Clara, CA, July 2017.

[42] Rui Zhang. Distributed machine learning. `https://github.com/zhangruiskyline/DeepLearning/blob/master/doc/system.md#tree-based-allreduce-vs-ring-based-allreduce`, August 2017.

[43] Y. Zhang, D. Zhou, S. Chen, S. Gao, and Y. Ma. Single-image crowd counting via multi-column convolutional neural network. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 589–597, Las Vegas, NV, June 2016.

Appendix A

Get to know the Codebase

The code related to the Distributed experiment is given below. All the codes created during this thesis project are uploaded in these following version control system: Personal GitHub repo[1], USASK gitlab repo[2] and P2IRC repo[3]

A.1 Distributed Parameter Server

```
#!/usr/bin/env python
import argparse
import tensorflow as tf
from tensorflow.python.client import timeline
import time
from datetime import datetime
import subprocess
import psutil
import os

import Model.prepare as prepare
import Model.modified_MCNN as model

# Sets the threshold for what messages will be logged.
tf.logging.set_verbosity(tf.logging.INFO)

# Sets if any specific GPU needs to be selected.
os.environ["CUDA_VISIBLE_DEVICES"] = "0"

def kill(proc_pid):
    """
    This function kills all the child processes forked from the parent process sent as
        function argument.
    :param proc_pid: The process id of the parent process.
    :return:
    """
    process = psutil.Process(proc_pid)
    for proc in process.children(recursive=True):
        proc.kill()
    process.kill()


def core_model(input_image):
    """
    This function creates a object of the multi-column CNN model which is defined in
        modified_MCNN module.
    The multi-column CNN is a three column convolutional neural network architecture which
        takes a batch of images (transformed to tensors)
    as input and produces density maps which show the approximate positions of the canola
        flowers for the given images.
    :param input_image: [tensors] batch of input images.
    :return: density map: [tensors] predicted density map.
    """
    mcnn_model = model.MCNN(input_image)
    predicted_density_map = mcnn_model.final_layer_output
    return predicted_density_map
```

[1] https://github.com/rashid0531/DistributedCNN (accessed September 13, 2019)

[2] https://git.cs.usask.ca/mrc689/distributed_flower_counter_cnn (accessed September 17, 2019)

[3] https://git.cs.usask.ca/discus/p2irc-applications/tree/master/software-repo/rashid (accessed September 17, 2019)

```
def do_training(args,server,cluster_spec):
    """
    Initiates and executes the main training procedures. From loading the images from HDFS
        or NFS,
    transforming the images into tensors, defining the cost function for backpropagation and
         feeding
    the input tensors to the core model and finally executing distributed training through
        synchronous parameter updates
    etc. are the main responsibilities of this function.
    :param args: command line arguments which specify different parameters during training.
    :param cluster_spec: [dict] a dictionary which contains the cluster specification such
        as job name and ip addresses of the
    machines which will be assigned to the job. The job name is used as key and the list of
        ip addresses are used as
    values to comprise the dictionary.
    :return:
    """

    # Automatically assigns operations to the suitable device.
    with tf.device(tf.train.replica_device_setter(worker_device="/job:worker/task:%d" % int(
        args["task_index"]),
                                                  ps_device="/job:ps/cpu:0",
                                                  cluster=cluster_spec)):

        # Get the list of input images along with corresponding ground-truths which will be
            used in train, test dataset.
        train_set_image, train_set_gt, _, _ = prepare.get_train_test_DataSet(args["
            image_path"],
                                                                             args["gt_path"],
                                                                             args["
                                                                                 dataset_train_test_ratio
                                                                                 "])

        assert len(train_set_image) == len(train_set_gt), "Equal␣number␣of␣ground-truths␣and
            ␣input␣images␣are␣required."

        TRAINSET_LENGTH = len(train_set_image)

        print("Trainset␣Length␣:␣", len(train_set_image), "Ground␣Truth␣Length␣:␣", len(
            train_set_gt))

        images_input_train = tf.constant(train_set_image)
        images_gt_train = tf.constant(train_set_gt)


        dataset_train = tf.data.Dataset.from_tensor_slices((images_input_train,
            images_gt_train))
        # At time of this writing Tensorflow doesn't support a mixture of user defined
            python function
        # with tensorflow operations.So we can't use one py_func to process data using
            tenosrflow operation and
        # nontensorflow operation.

        Batched_dataset_train = dataset_train.map(
            lambda img, gt: tf.py_func(prepare.read_npy_file, [img, gt], [img.dtype, tf.
                float32]))

        Batched_dataset_train = Batched_dataset_train \
            .shuffle(buffer_size=500) \
            .map(prepare._parse_function, num_parallel_calls=args["num_parallel_threads"]) \
            .apply(tf.contrib.data.batch_and_drop_remainder(args["batch_size_per_GPU"])) \
            .prefetch(buffer_size=args["prefetch_buffer"]) \
            .repeat()

        # Create a Tensorflow iterator to iterate over the batched dataset.
        iterator = Batched_dataset_train.make_one_shot_iterator()
```

```
# Generate next item from the iterator.
mini_batch = iterator.get_next()

# If the number of GPUs is set to 1, then no splitting will be done
split_batches_imgs = tf.split(mini_batch[1], int(args["num_gpus"]))
split_batches_gt = tf.split(mini_batch[2], int(args["num_gpus"]))

predicted_density_map = core_model(split_batches_imgs[0])

# Dimension of the predicted map needs to be (Batch size per GPUx224x224x1)
assert predicted_density_map.get_shape()[0] == args["batch_size_per_GPU"], "Output␣
    batch␣size␣needs␣to␣match␣with␣input␣batch␣size"
assert predicted_density_map.get_shape()[1] == 224, "The␣length␣of␣the␣predicted␣
    density␣map␣needs␣to␣be␣224"
assert predicted_density_map.get_shape()[2] == 224, "The␣width␣of␣the␣predicted␣
    density␣map␣needs␣to␣be␣224"
assert predicted_density_map.get_shape()[3] == 1, "The␣predicted␣density␣map␣should␣
    have␣only␣one␣color␣channel"

# Definition of loss function (Pixel wise euclidean distance between ground-truth
    and predicted density map).
# is used here
cost = tf.reduce_mean(
    tf.sqrt(
        tf.reduce_sum(
            tf.square(
                tf.subtract(split_batches_gt[0], predicted_density_map)), axis=[1,
                    2, 3], keepdims=True)))

# sum over the batched data.
sum_of_gt = tf.reduce_sum(split_batches_gt[0], axis=[1, 2, 3], keepdims=True)
sum_of_predicted_density_map = tf.reduce_sum(predicted_density_map, axis=[1, 2, 3],
    keepdims=True)

# Mean squared error.
mse = tf.sqrt(tf.reduce_mean(tf.square(sum_of_gt - sum_of_predicted_density_map)))

# Mean absolute error.
mae = tf.reduce_mean(
    tf.reduce_sum(
        tf.abs(
            tf.subtract(sum_of_gt, sum_of_predicted_density_map)),
        axis=[1, 2, 3],
        keepdims=True),
    name="mae")

# Adding summary to the graph. The summary can be later visualized using Tensorboard
    .
tf.summary.scalar("Mean_Absolute_Error", mae)

# Collect summaries defined during training.
summaries = tf.get_collection(tf.GraphKeys.SUMMARIES)

# Add all the trainable variables to summary.
for var in tf.trainable_variables():
    summaries.append(tf.summary.histogram(var.op.name, var))

# Create global steps. Comes useful while tracking the last synchronized step among
    multiple worker machines
# during training.
global_step = tf.train.get_or_create_global_step()

# Initiate optimizer for updating gradients. Among many optimizers, AdamOptimizer
    will be used in this project.
optimizer = tf.train.AdamOptimizer(learning_rate=(args["learning_rate"]))

# Synchronous distributed optimizer. Only needed for distributed training. Not
    needed while training in
```

```
    # single machine.
    opt = tf.train.SyncReplicasOptimizer(optimizer,
                                         replicas_to_aggregate=len(args["worker_hosts"].
                                             split(",")),
                                         total_num_replicas=len(args["worker_hosts"].
                                             split(",")))

    training_op = opt.minimize(cost, global_step=global_step)

    # configuration to set if the device placement will be logged or not.
    config = tf.ConfigProto()
    config.log_device_placement = False
    config.allow_soft_placement = True


effective_batch_size = int(args["batch_size_per_GPU"]) * len(args["worker_hosts"].split(
    ","))

end_point = int((TRAINSET_LENGTH * int(args["number_of_epoch"])) / effective_batch_size)
end_point = 3
print("End␣Point␣:␣", end_point)

# One worker among all the worker machines takes the responsibility of chief worker. The
     chief worker periodically
# saves the model parameter in checkpoint file to ensure fault tolerance. Usually, the
    first worker in the
# worker list takes the chief role.
is_chief = (int(args["task_index"]) == 0)

sync_replicas_hook = opt.make_session_run_hook(is_chief)

# Creating profiler hook. Used for profiling CPU, GPU usage, runtime of individual
    operation.
profile_hook = tf.train.ProfilerHook(save_steps=1500, output_dir='/home/rashid/
    DistributedCNN/timeline/')

# The StopAtStepHook handles when to stop training. When the last_step is reached, the
    training is terminated.
# last_step should be equal to the end_point.
hooks = [sync_replicas_hook, tf.train.StopAtStepHook(last_step=end_point), profile_hook]

# The MonitoredTrainingSession takes care of session initialization,
# restoring from a checkpoint, saving to a checkpoint, and closing when done
# or an error occurs.

arr_examples_per_sec = []

with tf.train.MonitoredTrainingSession(master=server.target,
                                       is_chief=(int(args["task_index"]) == 0),
                                       checkpoint_dir=args["checkpoint_path"],
                                       hooks=hooks, config=config) as mon_sess:

    while not mon_sess.should_stop():
        # Run a training step asynchronously.
        # mon_sess.run handles AbortedError in case of preempted PS.

        start_time = time.time()
        _, loss_value, step = mon_sess.run((training_op, mae, global_step))
        duration = time.time() - start_time

        examples_per_sec = (args["batch_size_per_GPU"] * len(args["worker_hosts"].split(
            ","))) / duration

        arr_examples_per_sec.append(examples_per_sec)

        format_str = ('%s:␣step␣%d,␣loss␣=␣%.2f␣,␣examples/sec␣=␣␣%.1f␣')
        print(format_str % (datetime.now(), step, loss_value,
                            examples_per_sec))
```

```
    print("---␣Experiment␣Finished␣---")


def assign_tasks(args):
    """
    This function assigns jobs to the master and worker machines. From the commandline
        arguments this function
    determines which of the machines will be used as Parameter Servers and which machines
        will be used as workers.
    :param args: the command line arguments which specify different parameters during
        training.
    :return:
    """

    ps_hosts = args["ps_hosts"].split(",")
    worker_hosts = args["worker_hosts"].split(",")

    print("Number␣of␣Parameter␣Servers:␣",len(ps_hosts), "␣Number␣of␣workers:␣", len(
        worker_hosts))

    # Create a cluster from the Parameter Server and worker hosts (Tensorflow specific code)
        .
    cluster = tf.train.ClusterSpec({"ps": ps_hosts, "worker": worker_hosts})

    # Create and start a server for the local task.
    server = tf.train.Server(cluster,
                             job_name=args["job_name"],
                             task_index=int(args["task_index"]))

    if args["job_name"] == "ps":
        server.join()

    elif args["job_name"] == "worker":
        do_training(args,server,cluster)



if __name__ == "__main__":

    # The following default values will be used if command line arguments are not provided.
    DEFAULT_NUMBER_OF_GPUS = 1
    DEFAULT_EPOCH = 3000

    DEFAULT_NUMBER_OF_WORKERS = 1
    DEFAULT_NUMBER_OF_PS = 1
    DEFAULT_BATCHSIZE_PER_GPU = 8

    DEFAULT_BATCHSIZE = DEFAULT_BATCHSIZE_PER_GPU * DEFAULT_NUMBER_OF_GPUS *
        DEFAULT_NUMBER_OF_WORKERS
    DEFAULT_PARALLEL_THREADS = 8

    DEFAULT_PREFETCH_BUFFER_SIZE = 64
    DEFAULT_IMAGE_PATH = "/home/mrc689/Sampled_Dataset"
    DEFAULT_GT_PATH = "/home/mrc689/Sampled_Dataset_GT/density_map"
    DEFAULT_LOG_PATH = "/home/mrc689/tf_logs"
    DEFAULT_RATIO_TRAINTEST_DATASET = 0.7
    DEFAULT_LEARNING_RATE = 0.00001
    DEFAULT_CHECKPOINT_PATH = "/home/mrc689/tf_ckpt"
    DEFAULT_LOG_FREQUENCY = 10

    # Create arguements to parse
    ap = argparse.ArgumentParser(description="Script␣to␣train␣the␣Convolutional␣neural␣
        network␣based␣FlowerCounter␣model␣in␣distributed␣GPU␣cluster.")

    ap.add_argument("-g", "--num_gpus", required=False, help="How␣many␣GPUs␣to␣use.",default
         = DEFAULT_NUMBER_OF_GPUS)
```

```
    ap.add_argument("-e", "--number_of_epoch", required=False, help="Number␣of␣epochs",
        default = DEFAULT_EPOCH)
    ap.add_argument("-b", "--batch_size", required=False, help="Number␣of␣images␣to␣process␣
        in␣a␣minibatch",default = DEFAULT_BATCHSIZE)
    ap.add_argument("-gb", "--batch_size_per_GPU", required=False, help="Number␣of␣images␣to
        ␣process␣in␣a␣batch␣per␣GPU",default = DEFAULT_BATCHSIZE_PER_GPU)
    ap.add_argument("-i", "--image_path", required=False, help="Input␣path␣of␣the␣images",
        default = DEFAULT_IMAGE_PATH)
    ap.add_argument("-gt", "--gt_path", required=False, help="Ground␣truth␣path␣of␣input␣
        images",default = DEFAULT_GT_PATH)
    ap.add_argument("-num_threads", "--num_parallel_threads", required=False, help="Number␣
        of␣threads␣to␣use␣in␣parallel␣for␣preprocessing␣elements␣in␣input␣pipeline", default
         = DEFAULT_PARALLEL_THREADS)
    ap.add_argument("-l", "--log_path", required=False, help="Path␣to␣save␣the␣tensorflow␣
        log␣files",default=DEFAULT_LOG_PATH)
    ap.add_argument("-r", "--dataset_train_test_ratio", required=False, help="Dataset␣ratio␣
        for␣train␣and␣test␣set␣.",default = DEFAULT_RATIO_TRAINTEST_DATASET)
    ap.add_argument("-pbuff","--prefetch_buffer",required=False,help="An␣internal␣buffer␣to␣
        prefetch␣elements␣from␣the␣input␣dataset␣ahead␣of␣the␣time␣they␣are␣requested",
        default=DEFAULT_PREFETCH_BUFFER_SIZE)
    ap.add_argument("-lr", "--learning_rate", required=False, help="Default␣learning␣rate.",
        default = DEFAULT_LEARNING_RATE)
    ap.add_argument("-ckpt_path", "--checkpoint_path", required=False, help="Path␣to␣save␣
        the␣Tensorflow␣model␣as␣checkpoint␣file.",default = DEFAULT_CHECKPOINT_PATH)

    # Arguments needed for Distributed Training.
    ap.add_argument("-pshosts", "--ps_hosts", required=False, help="Comma-separated␣list␣of␣
        hostname:port␣pairs.")
    ap.add_argument("-wkhosts", "--worker_hosts", required=False, help="Comma-separated␣list
        ␣of␣hostname:port␣pairs.")
    ap.add_argument("-job", "--job_name", required=False, help="One␣of␣'ps',␣'worker'.")
    ap.add_argument("-tsk_index", "--task_index", required=False, help="Index␣of␣task␣within
        ␣the␣job.")

    ap.add_argument("-lg_freq", "--log_frequency", required=False, help="Log␣frequency.",
        default = DEFAULT_LOG_FREQUENCY)

    args = vars(ap.parse_args())


    start_time = time.time()
    tf.reset_default_graph()

    # This process initiates the GPU profiling script.
    proc = subprocess.Popen(['./gpu_profile'])
    print("start␣GPU␣profiling␣process␣with␣pid␣%s" % proc.pid)

    assign_tasks(args)
    duration = time.time() - start_time

    # Kill the GPU profiling processes.
    kill(proc.pid)

    print("Duration␣:␣", duration)
```

A.2 Distributed Ring-AllReduce

```
#!/usr/bin/env python
import argparse
import tensorflow as tf
import os
from PIL import Image, ImageFile
import numpy as np
import Model.prepare as prepare
from matplotlib import pyplot as plt
from datetime import datetime
from tensorflow.python.client import timeline
```

```
import Model.modified_MCNN as model
import os
import re
import time
from datetime import datetime
import subprocess
import psutil
import horovod.tensorflow as hvd

tf.logging.set_verbosity(tf.logging.INFO)

# This function kills all the child processes associated with the parent process sent as
    function argument.
def kill(proc_pid):
    process = psutil.Process(proc_pid)
    for proc in process.children(recursive=True):
        proc.kill()
    process.kill()


def core_model(input_image):
    mcnn_model = model.MCNN(input_image)
    predicted_density_map = mcnn_model.final_layer_output
    return predicted_density_map


def do_training(args):

    config = tf.ConfigProto()
    config.gpu_options.visible_device_list = str(hvd.local_rank())
    config.log_device_placement = False
    config.allow_soft_placement = True

    train_set_image, train_set_gt, test_set_image, test_set_gt = prepare.
        get_train_test_DataSet(args["image_path"], args["gt_path"], args["
        dataset_train_test_ratio"])

    TRAINSET_LENGTH = len(train_set_image)

    print("Trainset␣Length␣:␣", len(train_set_image) , len(train_set_gt))

    images_input_train = tf.constant(train_set_image)
    images_gt_train = tf.constant(train_set_gt)

    dataset_train = tf.data.Dataset.from_tensor_slices((images_input_train, images_gt_train)
        )
    # At time of this writing Tensorflow doesn't support a mixture of user defined python
        function with tensorflow operations.
    # So we can't use one py_func to process data using tenosrflow operation and
        nontensorflow operation.

    # Train Set
    Batched_dataset_train = dataset_train.map(
        lambda img, gt: tf.py_func(prepare.read_npy_file, [img, gt], [img.dtype, tf.float32
            ]))

    Batched_dataset_train = Batched_dataset_train \
        .shuffle(buffer_size=500) \
        .map(prepare._parse_function,num_parallel_calls= args["num_parallel_threads"]) \
        .apply(tf.contrib.data.batch_and_drop_remainder(args["batch_size_per_GPU"])) \
        .prefetch(buffer_size = args["prefetch_buffer"])\
        .repeat()


    iterator = Batched_dataset_train.make_one_shot_iterator()

    mini_batch = iterator.get_next()
```

```
image_names = mini_batch[0]

# If the number of GPUs is set to 1, then no splitting will be done
split_batches_imgs = tf.split(mini_batch[1], int(args["num_gpus"]))
split_batches_gt = tf.split(mini_batch[2], int(args["num_gpus"]))

predicted_density_map = core_model(split_batches_imgs[0])

cost = tf.reduce_mean(
        tf.sqrt(
        tf.reduce_sum(
        tf.square(
        tf.subtract(split_batches_gt[0], predicted_density_map)), axis=[1, 2, 3],
            keepdims=True)))


sum_of_gt = tf.reduce_sum(split_batches_gt[0], axis=[1, 2, 3], keepdims=True)
sum_of_predicted_density_map = tf.reduce_sum(predicted_density_map, axis=[1, 2, 3],
    keepdims=True)

#mse = tf.sqrt(tf.reduce_mean(tf.square(sum_of_gt - sum_of_predicted_density_map)))

# Changed the mean abosolute error.
mae = tf.reduce_mean(
    tf.reduce_sum(tf.abs(tf.subtract(sum_of_gt, sum_of_predicted_density_map)), axis=[1,
         2, 3], keepdims=True),name="mae")

# Adding summary to the graph.
# added a small threshold value with mae to prevent NaN to be stored in summary
    histogram.
#tf.summary.scalar("Mean Squared Error", mse)
tf.summary.scalar("Mean_Absolute_Error", mae)

# Retain the summaries from the final tower.
summaries = tf.get_collection(tf.GraphKeys.SUMMARIES)

for var in tf.trainable_variables():
    summaries.append(tf.summary.histogram(var.op.name, var))

summary_op = tf.summary.merge(summaries)

global_step = tf.train.get_or_create_global_step()

#changed for Horovod.
#optimizer = tf.train.AdamOptimizer((args["learning_rate"]) * hvd.size())

optimizer = tf.train.AdamOptimizer((args["learning_rate"]))

#train_op = optimizer.minimize(cost, global_step=global_step)

# Synchronous training.
#opt = tf.train.SyncReplicasOptimizer(optimizer, replicas_to_aggregate=len(worker_hosts)
    ,total_num_replicas=len(worker_hosts))

# Add Horovod Distributed Optimizer
opt = hvd.DistributedOptimizer(optimizer)

# Some models have startup_delays to help stabilize the model but when using
# sync_replicas training, set it to 0.

# Now you can call `minimize()` or `compute_gradients()` and
# `apply_gradients()` normally

# !!!! Doubtful
training_op = opt.minimize(cost, global_step=global_step)

#config = tf.ConfigProto(log_device_placement=False, allow_soft_placement=True)
```

```
    # The StopAtStepHook handles stopping after running given steps.
    #SHARDED_TRAINSET_LENGTH = int(TRAINSET_LENGTH/len(worker_hosts))
    effective_batch_size = int(args["batch_size_per_GPU"]) * hvd.size()

    end_point = int((TRAINSET_LENGTH * int(args["number_of_epoch"])) / effective_batch_size)
    print("My␣Rank:␣", hvd.rank())
    print("My␣local␣Rank:␣", hvd.local_rank())
    print("End␣Point␣:␣",end_point)


    # You can create the hook which handles initialization and queues.
    #sync_replicas_hook = opt.make_session_run_hook(is_chief)
    # Creating profiler hook.
    #profile_hook = tf.train.ProfilerHook(save_steps=1000, output_dir='/home/rashid/
        DistributedCNN/Model/timeline/')
    # Simple Example of logging hooks

    # last_step should be equal to the end_point
    hooks=[hvd.BroadcastGlobalVariablesHook(0), tf.train.StopAtStepHook(last_step=end_point)
        ]

    checkpoint_dir = args["checkpoint_path"] if hvd.rank() == 0 else None

    # The MonitoredTrainingSession takes care of session initialization,
    # restoring from a checkpoint, saving to a checkpoint, and closing when done
    # or an error occurs.

    arr_examples_per_sec = []

    with tf.train.MonitoredTrainingSession(checkpoint_dir= checkpoint_dir,
                                           hooks=hooks, config=config) as mon_sess:

        while not mon_sess.should_stop():
            # Run a training step asynchronously.
            # See <a href="../api_docs/python/tf/train/SyncReplicasOptimizer"><code>tf.train
                .SyncReplicasOptimizer</code></a> for additional details on how to
            # perform *synchronous* training.
            # mon_sess.run handles AbortedError in case of preempted PS.
            start_time = time.time()
            _, loss_value, step = mon_sess.run((training_op, mae,global_step))
            duration = time.time() - start_time

            examples_per_sec = (args["batch_size_per_GPU"] * hvd.size()) / duration

            arr_examples_per_sec.append(examples_per_sec)

            format_str = ('%s:␣step␣%d,␣loss␣=␣%.2f␣,␣examples/sec␣=␣␣%.1f␣')
            print(format_str % (datetime.now(), step, loss_value,
                                examples_per_sec))

            #mon_sess.run(training_op)

    # with open("exm_per_sec.txt","w") as file_object:
    #
    #     for i in range(0,len(arr_examples_per_sec)):
    #         file_object.write(str(arr_examples_per_sec[i])+"\n")

    print("---␣Experiment␣Finished␣---")



if __name__ == "__main__":

    # The following default values will be used if not provided from the command line
        arguments.
    DEFAULT_NUMBER_OF_GPUS = 1
    DEFAULT_EPOCH = 3000
```

```
DEFAULT_NUMBER_OF_WORKERS = 1
DEFAULT_NUMBER_OF_PS = 1
DEFAULT_BATCHSIZE_PER_GPU = 8

DEFAULT_BATCHSIZE = DEFAULT_BATCHSIZE_PER_GPU * DEFAULT_NUMBER_OF_GPUS *
    DEFAULT_NUMBER_OF_WORKERS
DEFAULT_PARALLEL_THREADS = 4
#DEFAULT_PREFETCH_BUFFER_SIZE = DEFAULT_BATCHSIZE * DEFAULT_NUMBER_OF_GPUS * 2

DEFAULT_PREFETCH_BUFFER_SIZE = 32
DEFAULT_IMAGE_PATH = "/home/mrc689/Sampled_Dataset"
DEFAULT_GT_PATH = "/home/mrc689/Sampled_Dataset_GT/density_map"
DEFAULT_LOG_PATH = "/home/mrc689/tf_logs"
DEFAULT_RATIO_TRAINTEST_DATASET = 0.7
DEFAULT_LEARNING_RATE = 0.00001
DEFAULT_CHECKPOINT_PATH = "/home/mrc689/tf_ckpt"
DEFAULT_LOG_FREQUENCY = 10

#DEFAULT_MAXSTEPS = (DEFAULT_TRAINSET_LENGTH * DEFAULT_EPOCH) / DEFAULT_BATCHSIZE

# Create arguements to parse
ap = argparse.ArgumentParser(description="Script␣to␣train␣the␣FlowerCounter␣model␣using␣
    multiGPUs␣in␣single␣node.")

ap.add_argument("-g", "--num_gpus", required=False, help="How␣many␣GPUs␣to␣use.",default
     = DEFAULT_NUMBER_OF_GPUS)
ap.add_argument("-e", "--number_of_epoch", required=False, help="Number␣of␣epochs",
    default = DEFAULT_EPOCH)
ap.add_argument("-b", "--batch_size", required=False, help="Number␣of␣images␣to␣process␣
    in␣a␣minibatch",default = DEFAULT_BATCHSIZE)
ap.add_argument("-gb", "--batch_size_per_GPU", required=False, help="Number␣of␣images␣to
    ␣process␣in␣a␣batch␣per␣GPU",default = DEFAULT_BATCHSIZE_PER_GPU)
ap.add_argument("-i", "--image_path", required=False, help="Input␣path␣of␣the␣images",
    default = DEFAULT_IMAGE_PATH)
ap.add_argument("-gt", "--gt_path", required=False, help="Ground␣truth␣path␣of␣input␣
    images",default = DEFAULT_GT_PATH)
ap.add_argument("-num_threads", "--num_parallel_threads", required=False, help="Number␣
    of␣threads␣to␣use␣in␣parallel␣for␣preprocessing␣elements␣in␣input␣pipeline", default
     = DEFAULT_PARALLEL_THREADS)
ap.add_argument("-l", "--log_path", required=False, help="Path␣to␣save␣the␣tensorflow␣
    log␣files",default=DEFAULT_LOG_PATH)
ap.add_argument("-r", "--dataset_train_test_ratio", required=False, help="Dataset␣ratio␣
    for␣train␣and␣test␣set␣.",default = DEFAULT_RATIO_TRAINTEST_DATASET)
ap.add_argument("-pbuff","--prefetch_buffer",required=False,help="An␣internal␣buffer␣to␣
    prefetch␣elements␣from␣the␣input␣dataset␣ahead␣of␣the␣time␣they␣are␣requested",
    default=DEFAULT_PREFETCH_BUFFER_SIZE)
ap.add_argument("-lr", "--learning_rate", required=False, help="Default␣learning␣rate.",
    default = DEFAULT_LEARNING_RATE)
ap.add_argument("-ckpt_path", "--checkpoint_path", required=False, help="Path␣to␣save␣
    the␣Tensorflow␣model␣as␣checkpoint␣file.",default = DEFAULT_CHECKPOINT_PATH)

ap.add_argument("-nw", "--number_of_workers", required=False, help="Number␣of␣Workers.",
    default = DEFAULT_NUMBER_OF_WORKERS)

ap.add_argument("-lg_freq", "--log_frequency", required=False, help="Log␣frequency.",
    default = DEFAULT_LOG_FREQUENCY)

args = vars(ap.parse_args())

# Start the application
start_time = time.time()

# Initialize Horovod
hvd.init()

tf.reset_default_graph()

# This process initiates the GPU profiling script.
```

```
#proc = subprocess.Popen(['./gpu_profile'])
#print("start GPU profiling process with pid %s" % proc.pid)

do_training(args)
duration = time.time() - start_time

#kill(proc.pid)

print("Duration␣:␣", duration)
```