On Studying Distributed Machine Learning

Simeon Eberz

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2021

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial
fulfillment of the requirements for graduation from the
Honors Program of Liberty University.

_____
Feng Wang, Ph.D.
Thesis Chair

_____
Eduard Babulak, Ph.D.
Committee Member

_____
David Schweitzer, Ph.D.
Assistant Honors Director

_____
Date

Abstract

The Internet of Things (IoT) is utilizing Deep Learning (DL) for applications such as voice or image recognition. Processing data for DL directly on IoT edge devices reduces latency and increases privacy. To overcome the resource constraints of IoT edge devices, the computation for DL inference is distributed between a cluster of several devices. This paper explores DL, IoT networks, and a novel framework for distributed processing of DL in IoT clusters. The aim is to facilitate and simplify deployment, testing, and study of a distributed DL system, even without physical devices. The contributions of this paper are a deployment of the framework to an Ubuntu virtual machine testbed and a repackaging of the framework as a Docker image for portability and fast future deployment.

On Studying Distributed Machine Learning

Introduction

Internet of Things (IoT) applications aim to increase control and understanding of physical environments by collecting data with a variety of sensors and processing and exchanging this data over the internet. As these devices produce massive quantities of complex data, processing and analyzing the data is one of the biggest hurdles for development of effective IoT solutions. Deep Learning (DL), a subset of Machine Learning (ML) and Artificial Intelligence (AI), is proving to be very useful for Internet of Things applications, because it produces some of the most accurate classifications [1]. Deep Learning could be the key to enabling applications like voice recognition in small smart speakers or image recognition in cameras.

The challenge to enabling Machine Learning on IoT edge devices is that these devices are very constrained, both in memory and computational power. To overcome these constraints, the computation is distributed between a cluster of several devices, which is a complex task. In [1], DeepThings is proposed as a framework to distribute Convolutional Neural Network (CNN) inference tasks between edge devices, enabling IoT clusters to perform DL processing that would not otherwise be possible.

The motivation of this work is twofold: to gain a better understanding of how to design an efficient distributed ML network, and to extend the usefulness of the DeepThings framework proposed in [1] by making it more portable to other devices. This work seeks to facilitate use and deployment distributed ML systems for future use and study. While the DeepThings paper only implemented and tested DeepThings on a Raspberry Pi 3 testbed, IoT networks are commonly

composed of other devices, such as STM32 microcontrollers or Arduino. Study and testing of the framework will be even simpler if physical devices are not needed. The contributions of this paper are as follows.

1) A distributed ML environment based on virtual machines, without the need for physical devices.

2) A packaging of the C implementation developed in [1] so that it can run on any Linux device without extra configuration or be run and tested in Docker right out of the box.

The rest of this paper provides background on IoT, Machine Learning, and Deep Learning, highlighting both the benefit and the complexity of processing data for Deep Learning algorithms near the edge of IoT, looks at the inner workings of the DeepThings framework and how it effectively breaks Convolutional Neural Networks into distributed tasks, and finally presents a testing of the framework in virtual machines and Docker.

Background

IoT Networks and Edge Devices

In the four decades of its lifetime, the Internet has evolved in purpose and involvement in human life. The primary purpose of the Internet has always been to share data and processing power. In past stages of the internet's evolution, this communication has been between a static web page and a researcher, or between two social media users. Today, the world is in the era of the Internet of Things. The Internet of Things, abbreviated IoT, refers to the Internet as it is connected to physical devices, often low cost, power efficient sensing devices designed to measure and increase insight into physical environments. These sensor devices are the "things" in "Internet of Things." These devices are able to exchange information directly with each other

and with servers or users. By connecting the devices to the network, data collection, storage, and analysis can be done in a separate, centralized location. This means the devices can be smaller, cheaper, and use less power, which makes the devices more accessible to consumers, or allows more devices and sensors to be used in larger deployment situations.

The general idea of the Internet of Things is captured in a basic three-layer protocol architecture [2]. The first layer is the perception layer. The perception layer includes the parts of IoT devices that sense or interact with the physical world. The second layer is the network layer, which includes connection with other devices and servers, transmission of data, and processing sensor data. The third layer is the application layer. It encompasses all specific applications that the Internet things enables, like smart homes or smart cities [2]. While this three-layer architecture gives a general idea of IoT, for modern research more complex architectures have been proposed, like a five layer architecture that adds a transport layer, a processing layer, and a business layer [2].

An important facet of IoT architecture is the distinction between cloud and fog computing system architectures. Cloud computing refers to a system architecture in which data processing is done on cloud computers, away from the sensor devices. Fog computing, on the other hand, also called edge computing, is a system architecture in which sensors and network gateways do some of the data processing [2].

To appreciate the resource constraints of IoT edge devices, consider smart assistant devices. Smart assistants are often designed for consumer use, especially in homes, like the Google Home or Amazon Echo. Both these devices feature hands-free control using a microphone and speech recognition, a speaker, and an AI voice assistant such as Google

Assistant or Alexa. These devices are used to answer questions using internet sources, stream music, or perform simple functions like setting timers. Both devices are designed to be small and inexpensive. The Google Home has measures less than 4 inches wide and 6 inches tall, and currently costs about $100 [3]. It is equipped with an ARM Cortex-A7 media processor and only 512MB of memory [3]. The Amazon echo measures just over 3 inches wide and about 10 inches tall, with a similar price point [4]. It has an Arm Cortex-A8 CPU, 256 MB RAM, and 4 GB flash memory.

The main barrier to using Deep Learning in IoT is the memory, computation, and energy requirements of Deep Learning algorithms. Deep Learning examples in IoT and mobile devices, like speech or facial recognition, are becoming increasingly prominent, but they mostly require cloud computing to execute [5]. This has two large drawbacks: first, sensitive data must be uploaded and processed off-device, exposing users to privacy dangers; and second, potentially unreliable network quality strongly influences inference execution time [5]. Privacy issues are one of the primary ethical concerns in IoT applications. Data collected in smart home applications, for example, collect data that can be used to reveal deeply personal information about its users. If this data is sent off-device, the user is forced to trust the application supplier to protect and not misuse their data. While travelling over the network, this data may be in danger of exposure to third parties. Executing Deep Learning algorithms on or nearer to IoT devices will increase individual privacy and reduce latency in the process.

Machine Learning and Neural Networks

Machine Learning is a field in which computers are equipped to learn from data without explicit programming. Rather than being programmed to complete a task, the computers perform

analysis on large amounts of input data to uncover patterns and insights, which they can then use to form a model for the task [6]. In supervised learning, the machine learns from labeled inputs and outputs to predict the correct output for new inputs. In unsupervised learning, the machine identifies patterns in and clusters input without any labeled output. The algorithms produced by Machine Learning are able to emulate human reasoning in tasks like image or speech recognition. This is why advancements in Machine Learning are considered a step towards "artificial intelligence." Unlike humans, however, computers can be better equipped to analyze huge quantities of numerical data, allowing them to sometimes detect patterns in big data that a person cannot.

Deep Learning

Deep Learning is a subset of Machine Learning. It is the state-of-the-art method of handling and extracting useful insights from complex, noisy date. Deep Learning, in particular, relies on great amounts of data for accurate results. The recent availability of powerful computer chips explains why there have been significant advances in Deep Learning only recently. Deep Learning is named such because it is based on algorithms that "learn from multiple of levels in order to provide a model that represents complex relations among data. A hierarchy of features is present such that high level features are defined in terms of lower level features" [6].
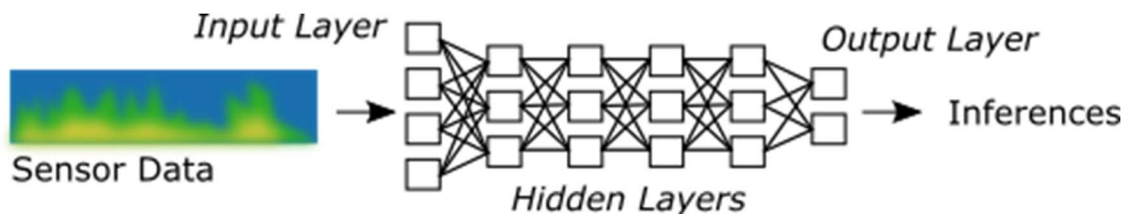

Fig. 1: fully-connected feed-forward layers in DNN

Deep Neural Networks. Neural networks are designed to mimic neurons in the brain, hence the name. Also called feed-forward neural networks, a deep neural network (DNN) contains layers of fully-connected nodes. The values of the first layer are initialized from input, then each layer transforms the output of the layer before it, and the final layer corresponds to output values, or inference classes. Deep Learning approaches can handle inference tasks that existing signal processing techniques cannot. For example, the conventional approach to solving speech recognition problems is to represent the speech signals with Gaussian Mixture Models based on hidden Markov models [6]. While still used today, these models are significantly improved when combined with Deep Learning. A deep neural network-based approach released by Microsoft in 2012 showed 30% word error rate reduction compared to the best models based on Gaussian mixtures [6].
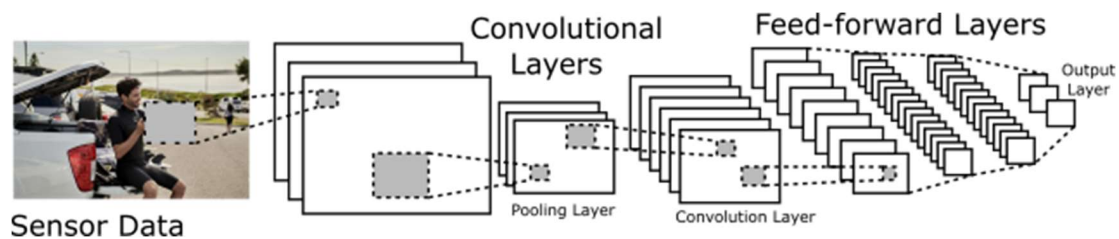

Fig. 2: convolutional and feed-forward layers in CNN

Convolutional Neural Networks. Another class of neural networks is the Convolutional Neural Network (CNN). Convolutional networks are composed of a series of convolution layers, pooling layers, and fully connected layers. Convolution layers work by convoluting a two-dimensional filter with the two-dimensional input. Pooling layers follow convolutional layers and serve to reduce the complexity of the convolutional output, which reduces the amount of computation performed, and makes the model more flexible and robust, as it is less sensitive to small variances in the input data. Feed-forward layers work like traditional DNNs [7].

"The aim of these layers is to extract simple representations at high resolution from the input data, and then converting these into more complex representations, but at much coarser resolutions within subsequent layers" [5]. This paradigm makes CNNs especially effective for computer vision, as they can extract simple features in early layers, like an eye or nose, and build these up into more complex representations in later layers, like a face. "CNNs have been extremely successful in computer vision applications, such as face recognition, object detection, powering vision in robotics, and self-driving cars" [8]. As discussed later in this paper, this work uses the You Only Live Once, version 2 (YOLOv2) algorithm to test distributed ML, which is a CNN-based computer vision algorithm.

Related Work

Large-scale distributed ML

While the simplest option for handling the data that Deep Learning algorithms process is to keep it in main memory, this is sometimes inefficient or unfeasible even for high-powered workstations. Other times, the data is naturally distributed, and moving it between systems is not permitted. Usually, this data is split "horizontally," where data instances, with all attributes, are stored in different locations, rather than being split across different attributes. At a large scale, distributed ML is motivated by overcoming algorithm complexity and memory limitations. Most distributed ML algorithms achieve distributed work by combining predictions from separately run classifiers. This allows for more flexibility in information representation or which classifier is used [9].

In contrast to the multi-core processors or interconnected workstations in larger distributed systems, IoT devices are connected by relatively unstable wireless connections,

because they are designed to communicate via small data packets [10]. Distributed ML

computation produces large data packets. A ML task may not produce the desired result if some

of these packets are late or missing [11]. Therefore, the challenge is receiving and processing

large amounts of data in a tight time frame on small, resource-constrained computing devices.

Works such as [12], [13] have looked at ways to perform distributed ML on mobile

devices. [12] discusses performing computation on local training data in mobile devices, then

using that computation result to update a global model. It presents an algorithm called Federated

Stochastic Variance Reduced Gradient (FSVRG), which is better suited for this situation because

it is "adaptive to different local data sizes, general sparsity patterns and significant differences in

patterns in data available locally, and those present in the entire data set" [12]. [13] proposes an

algorithm specifically for partitioning pre-trained DNNs onto several mobile devices, called

MoDNN. "MoDNN uses a Biased One Dimensional Partition (BODP) approach with a

MapReduce style (MR) task model and a work sharing (WSH) distribution method in which

processing tasks in each layer are first collected by a coordination device and then equally

distributed to existing edge nodes" [1].

DeepThings

The DeepThings framework is proposed in [1] to improve the performance of

Convolutional Neural Network inference on an IoT edge cluster. Three main techniques are

used: Fused Tile Partitioning (FTP), Distributed Work Stealing, and optimized work scheduling

and distribution. This section will give an overview of these techniques.

Because the convolutional chains in DNNs can be very deep, layer-based partitioning

methods result in very large memory footprints and communication overhead. FTP takes

advantage of the fact that "each output data element only depends on a local region in the input

feature maps" [1]. The convolution layers can be divided into a grid, forming smaller

independently executable tasks, hence the "tile" part of the partitioning. To reduce

communication overhead, "regions that are connected across layers can be fused into a larger

task" [1]. This means that intermediate feature maps stay on the edge assigned to process that

partition and only output feature maps are transmitted to the gateway [1].
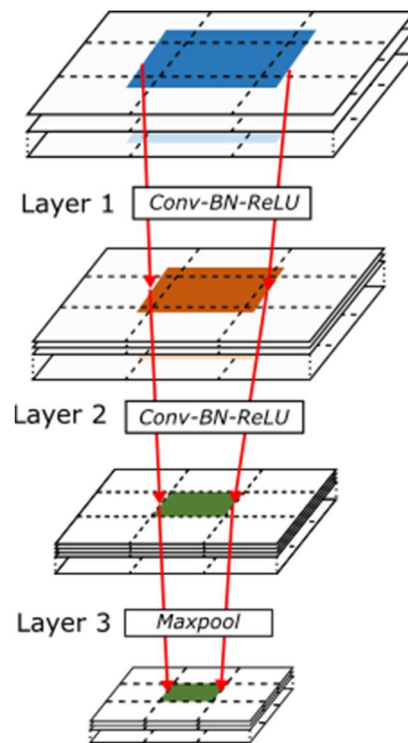
Fig. 3: Fused Tile Partitioning for CNN

Distributed Work Stealing is a technique used to deal with the dynamically shifting

workloads in IoT clusters. An edge device with items in its CNN inference task queue will

register itself with the gateway device, while an edge device with an empty queue will request

the IP of a busy device and attempt to steal work from that device. The gateway keeps the IP

addresses of busy nodes in a ring buffer and uses a round-robin approach to distribute an IP

address to a stealing request. Meanwhile, the gateway handles "collection, merging and further

processing of results from edge nodes" [1].

An improved work scheduling and distribution method is added to deal with the

overlapped data and redundant computation of the FTP scheme. As seen in Figure 3, the early

convolutional layers are divided into overlapping tiles, as the lower layer tiles depend on the

results of a wider partition. To reduce transmission and computation redundancy, the overlapping

data from one partition can be reused by an adjacent partition. However, this creates some

dependency between adjacent partitions and inhibits parallelism. To enable data reuse while

maintaining parallel execution, tasks are distributed first by even grid rows and columns and

least amount of overlap. This means that "adjacent partitions can have a higher probability to be

executed when their predecessors have finished and overlapped intermediate results are

available" [1]. The gateway collects and manages all overlapping intermediate data and serves

the appropriate overlapped data to edge nodes along with the other input data for a partition
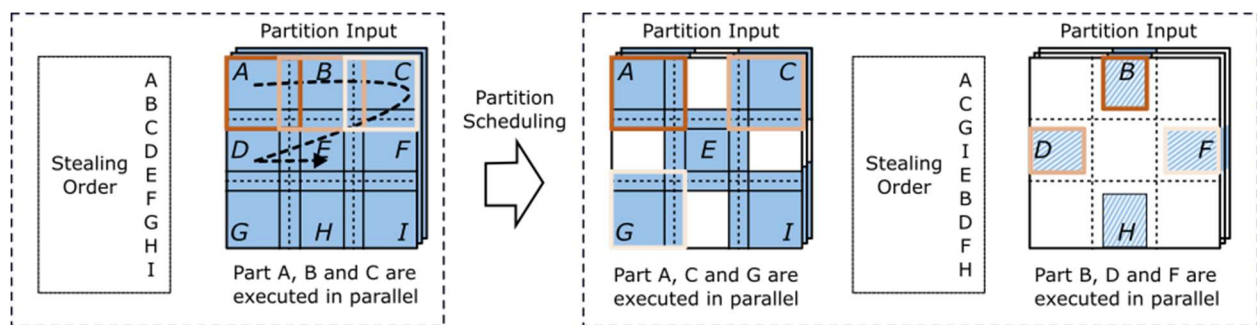
when the edge node requests a task.

Fig. 4: Improved Work Scheduling and Distribution for Overlapped Data

These three methods – FTP, Work Stealing, and improved work scheduling and

distribution – are combined in the DeepThings framework to provide "CNN inference speedups

of 1.7×–3.5× on 2–6 edge devices with less than 23 MB memory each" [1]. "Overall,

DeepThings has similar scalability but more redundant communication and computation as

compared to MoDNN" [1]. Results from the paper show that DeepThings also can reduce the

memory footprint by 68%, using Fused Tile Partitioning (FTP) on the convolutional layer. The

paper implements the DeepThings framework in the C programming language, but only uses

Raspberry Pi single board computers to test it. In its published state, the framework cannot run

directly on networks built from other devices.

The structure of the DeepThings C implementation produced by the author of [1]

includes the C files shown in Figures 5 and 6. Corresponding header files are not shown in the

Figures for brevity. The "top.c" file contains the main function of the program, and utilizes

functions from the "cmd_line_parser.c" file to parse command line arguments and pass them

down to the appropriate functions. The top level folder also includes the highest-level functions

for edges and the gateway, in "deepthings_edge.c" and "deepthings_gateway.c," and the

functions necessary for FTP. The file "ftp.c" includes the functions to control partitioning,

traversal, removing and reusing overlapped data. Functions for partitioning, merging, and queue

management are in the file "frame_partitioner.c." Both "adjacent_reuse_data_serialization.c" and

"self_reuse_data_serialization.c" contain methods for serializing data for later reuse.

The files in Figure 6 are contained in the "distiot" folder. These files handle the

distributed work stealing runtime for the cluster. As mentioned in [1], the work stealing runtime

runs as a separate thread. The files "gateway.c" and "client.c" hold the top-level functions for the

gateway and client work stealing threads, respectively. The file "network_util.c" has functions to

handle IPv4 or IPv6 connections and sending data. Helper functions for managing blob data used

by the devices are in "data_blob.c." Unsurprisingly, "thread_util.c" contains utility functions for

handling system threads, while "thread_safe_queue.c" contains functions to access a blob queue

from a work stealing thread. The gateway device uses three queues: for the results, ready tasks,

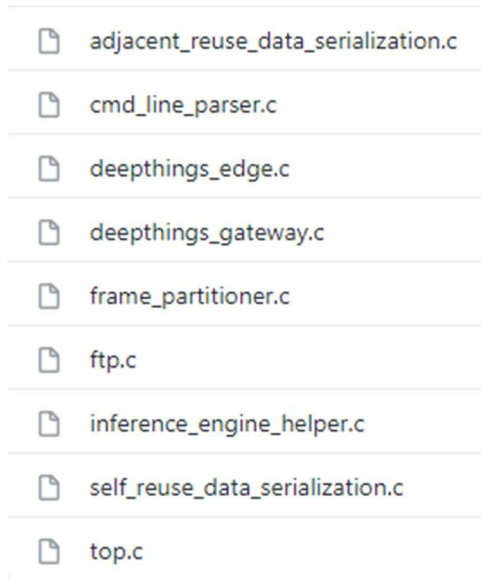and registered tasks. Each edge device uses one queue for tasks and one for results.

adjacent_reuse_data_serialization.c

cmd_line_parser.c

deepthings_edge.c

deepthings_gateway.c

frame_partitioner.c

ftp.c

inference_engine_helper.c

self_reuse_data_serialization.c

top.c

Fig. 5: Top level files in the C implementation of
DeepThings

client.c

data_blob.c

exec_ctrl.c

gateway.c

global_context.c

network_util.c

thread_safe_queue.c

thread_util.c
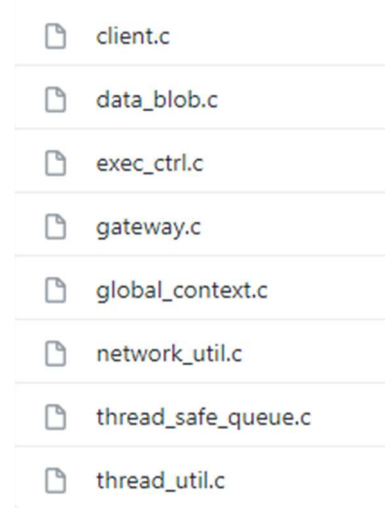
Fig. 6: C files in the /distiot folder

Docker

Docker Images

Docker is an open-source tool that aims to address the challenge of computation

reproducibility [14]. Docker makes it easier for users to run software developed on other

machines on their device by providing a binary image, which works similarly to a virtual

machine (VM) image. This Docker image contains all the dependencies and configurations

necessary to run a piece of software. Docker images are much more lightweight when run than

complete virtual machines because they share the Linux kernel with the host machine. This

feature is often useful in business applications where multiple copies of the software need to be

run on a single machine. It may be possible to run a Docker image on a device that is too

resource constrained to run a full VM. Since Docker works with all Linux distributions, it

provides a convenient way to move software to a different type of device device or multiple

devices without using a VM or spending time configuring each machine [15].

It is also worth noting that many Docker images are available in online Docker registries,

including Docker hub, a public registry with many freely available images. Once a Docker image

is developed it can be made available to other users by pushing to a registry.
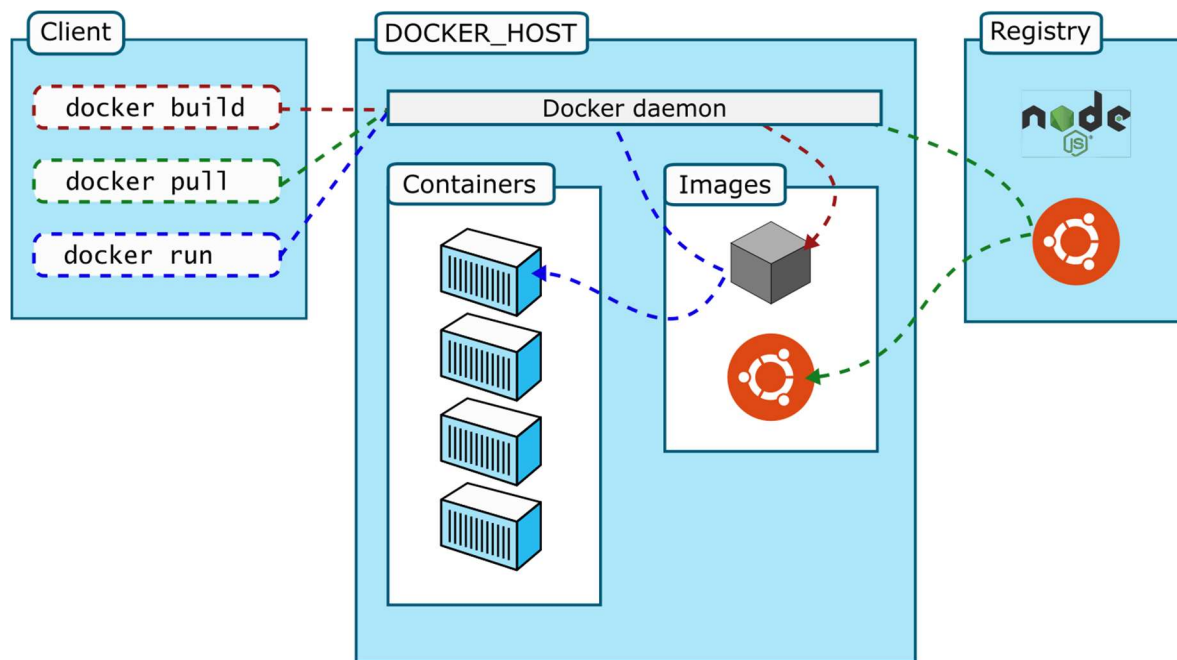
Fig. 7: Docker's Client-Server Architecture

Docker Network

Another feature of Docker that is useful to distributed Machine Learning research is

Docker network. When a Docker image is run, it becomes a container. Using Docker network,

Docker containers and services can be connected to each other or to other workloads, without

even the awareness that they are deployed on Docker [16]. Docker handles routing, packet

encapsulation, and encryption. If connected to the network outside the host machine, containers

do not use their own IP but connect through a port on the host. However, user-defined bridge networks allow multiple containers to appear as independently connected devices as they communicate with each other on the host. This bridge network setup can be used to simulate a network cluster.

Raspberry Pi Testbed

This section will discuss the original testbed setup used in [1], which is composed of Raspberry Pi 3 Model B devices (RPi3). The RPi3 is used for both the gateway device and each of the 1-6 edge devices. Each RPi3 contains a quadcore 1.2 GHz ARM Cortex-A53 processor with 1GB RAM. These devices are rather high-powered compared to many IoT edge nodes, such as the Google Home and Amazon Echo devices mentioned earlier. To create a realistic simulation of lower-powered IoT edge device clusters, each RPi3 used as an edge device was limited to a single processor core. These devices were connected over wireless local area network (WLAN) using TCP/IP protocols and socket API.

The pretrained CNN-based object detection model You Only Look Once, version 2 (YOLOv2) [16] was used to evaluate the C implementation of DeepThings on this testbed, because it is lightweight and widely used in embedded devices. The first 16 layers of YOLOv2 were partitioned and distributed. These layers make up nearly half of the inference computation and most of the memory footprint. The C-based Darknet neural network library is used as the external CNN inference engine and NNPACK is used as the backend acceleration kernel.

When compared with the approaches used in MoDNN, the methods in DeepThings require extra memory, slightly better communication overhead with a higher numbers of devices, a 41% latency reduction and an 80% throughput improvement. Overall, "scalable CNN inference

performance is significantly improved compared to existing distributed inference methods under varying static or dynamic application scenarios" [1].

VirtualBox Testing

To gain a better understanding of how to use and run DeepThings, as well as prepare dependencies and configuration for creating a Docker image, the author first setup and ran DeepThings framework on a set of virtual machines (VM). Using fresh VMs with clean Linux installations also makes it much easier to track and reproduce the dependencies and work necessary for the Docker image, as will be seen in the next section.

Oracle VM VirtualBox was used to create and manage the VMs. VirtualBox is free and open-source, and contains the VM and network configuration tools needed to setup a testbed. It also makes cloning VMs simple. An initial VM was created and conFigured in VMware Workstation 16 Player, but the free version of this software does not contain the necessary network configuration utilities, so this VM was imported into VirtualBox and then cloned to create the other devices.

Three types of devices must contain a similarly conFigured DeepThings executable: a host device, a gateway device, and one or more edge devices. These devices are distinguished when the executable file is run on each individual machine, but the creation of executable from the C implementation is the same for each machine. Like in [1], this system will also use YOLOv2 as the CNN model for evaluating the framework.

Downloading and Configuring DeepThings Implementation

Ubuntu 20.10 server, a Linux distribution based on Debian, is used for the operating system on the VMs. The VMs are limited to a single processor and 1 GB RAM. The DeepThings

implementation is publicly available on GitHub at [17], and is pulled recursively to the VM, so

that the darknet submodule is included. A few dependencies are not pre-installed in Ubuntu

server, like GCC and make, and are installed using apt-get. The implementation is set to not use

NNPACK or ARM Neon by disabling options in the Makefile, since the VM is not running as an

ARM device.

Next, a "yolo.cfg" configuration file for Darknet is included in the code files on GitHub,

but the weights for YOLOv2 must be downloaded and included in the same "models/" folder of

the project. These weights are taken from [18]. The final step before building the DeepThings

executable is configuring the gateway and edge device IP addresses. These are specified in a

configuration header file, "conFigure.h". This cluster uses the 10.10.10.x subnet. Since VMs can

be resource intensive to run, the number of edge devices in this test is limited to two.

The configuration header file also contains settable parameters for the width and height

of FTP partitions, the maximum number of fused layers, number of threads used, and whether

overlapped data is reused. The maximum number of edges and the frame number is set in this

file as well. The final set of options in "conFigure.h" pertain to debugging, with parameters for

which part of the framework outputs to debugging, the debug timing, and debug communication

size.

Once this configuration is complete, use "make" to build the DeepThings executable.

This same executable will run on each the host, gateway, and edge machines.

Configuring Network Cluster

Setting the VMs up to communicate with each other over TCP/IP requires configuration

inside and outside the VMs. Up to this point, the setup for each device in the testbed is the same

and can be done in a single VM which is then cloned to create the separate devices. VirtualBox

utilities are used to clone the original VM. Using the VirtualBox menus, each device's network

adapter is conFigured to attach to the same internal network. Each device uses the Intel

PRO/1000 MT Server adapter type. Each device is conFigured internally using the Linux

command line utility Netplan, which allows specification of a configuration file for each device

setting the static IP of their ethernet to the appropriate IP specified in DeepThings configuration.

Once these configurations are applied, the VMs should be ready to run DeepThings.
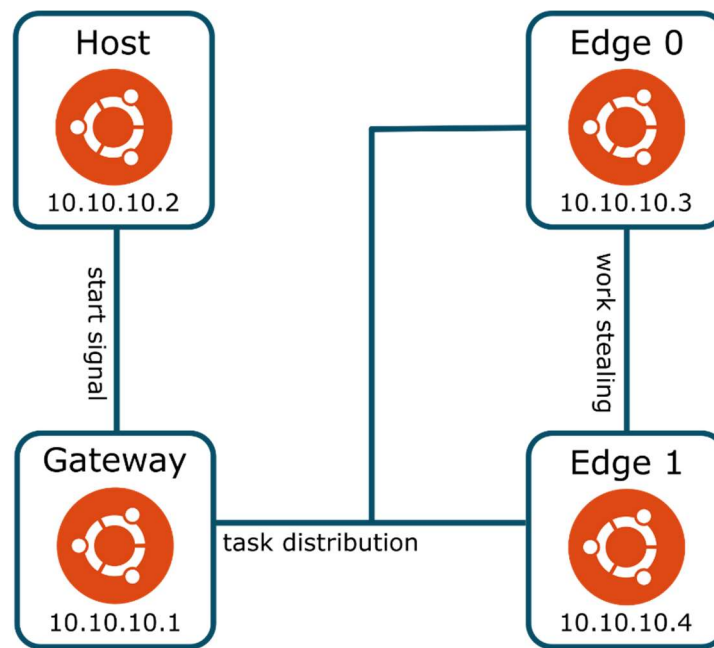
Fig. 8: VM network cluster configuration

Running DeepThings

The executable should be run on each edge device and the gateway first, setting the

parameters for total number of edges, edge id, number of fused layers, and FTP dimensions.

Once each device has loaded the YOLOv2 model configuration and weights, it waits for a signal

from the host machine to begin. The data source device has four images for processing, and the

total execution time for all four across the gateway and two edge devices is about 4 minutes.

Figures 9-12 show the output on each device during execution. As described in the background section, the edge devices begin execution with even partitions with the least number of adjacent partitions to increase reuse of overlapping intermediate results while maintaining parallel execution.

Fig. 9: Host device triggers start of execution

Fig. 10 : Gateway device distributing tasks

Fig. 11: Tasks executed by edge 0
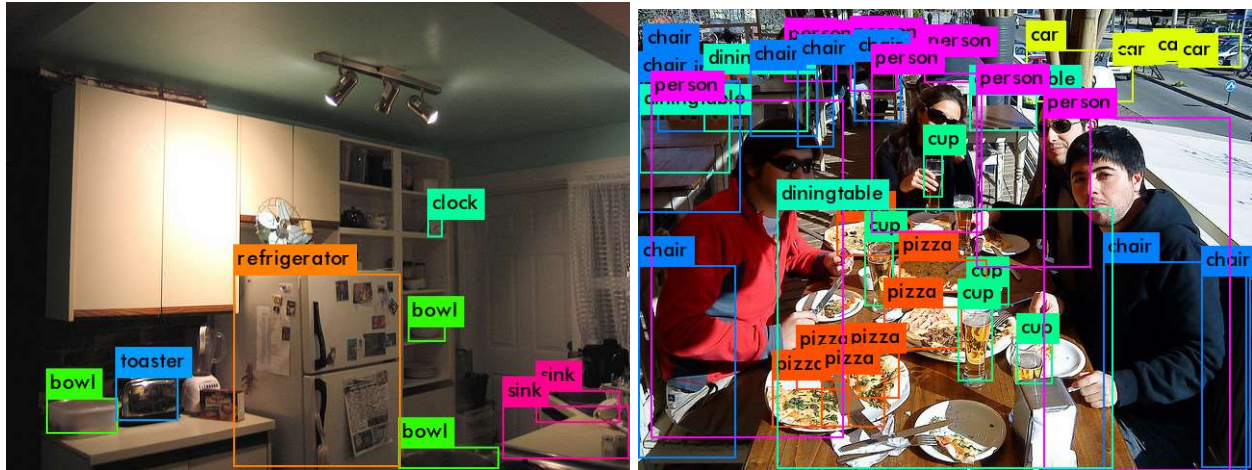

Fig. 12 : Tasks executed by edge 1

Fig. 13: Output images saved on gateway

The final results are output on the gateway device, and the images processed by the YOLOv2 algorithm are saved in the base project folder on the gateway. These images are shown in Figure 13.
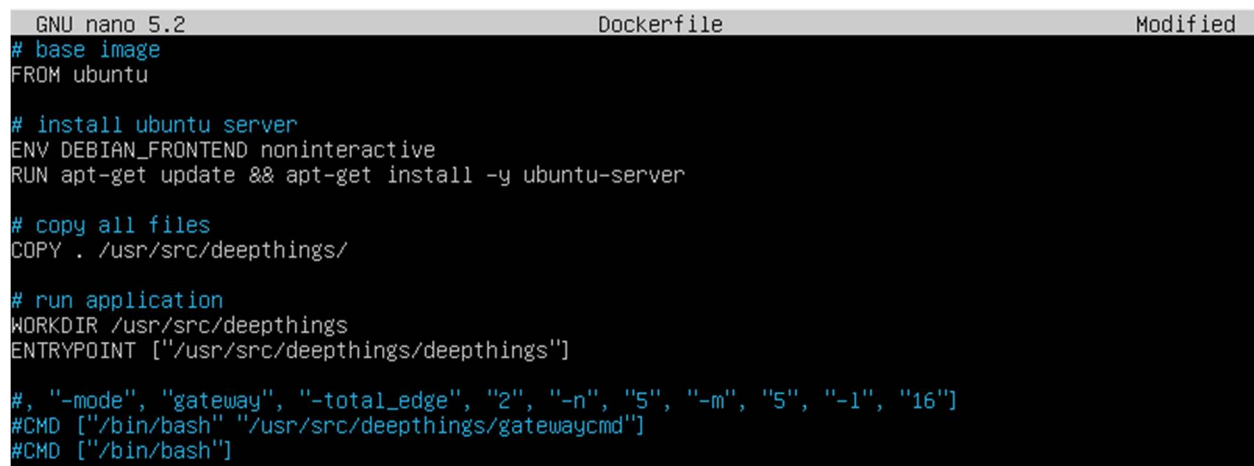
Porting to Docker

To package the DeepThings framework in an easy-to-run Docker image, start with a VM conFigured to run the framework. Packaging DeepThings with this current method requires the gateway and edge addresses and model used (in this case, YOLOv2) to be predetermined and conFigured. The recommended way to build an image from an existing project is to use a Dockerfile. A Dockerfile is just a text file that lists the commands necessary to build an image, which Docker uses to automatically build that image [19]. Using clean VMs to test the framework first makes writing the Dockerfile much easier, because it is easy to keep track of all the configuration and dependencies installed before DeepThings could be run.

Defining Dockerfile and Creating Docker Image

In this case, the image is defined to build on top of the Ubuntu image. The Ubuntu image is available on the Docker Hub, a public registry, and will be pulled from there automatically

when the image is built. While the dependencies installed during VM testing, such as GCC and

Make, are not necessary once the DeepThings executable is built, a few tools are needed from

Ubuntu-server that are not included in the base Ubuntu Docker image. Ubuntu-server contains

some utilities that are not needed to run DeepThings, but due to time constraints, it is specified

that the Docker image should install Ubuntu-server on top of the base Ubuntu Docker image.

The project folder for DeepThings is copied from the VM to the image, and the image

work directory is set to that folder on the image. Finally, Docker images require specification of

a command to run on start, or an "entrypoint" defining the executable to run when the container

starts [19]. In the future, the DeepThings executable should likely be defined as the entrypoint,

but due to time constraints, the start command is set to open a bash session in the container,

where the user can start the DeepThings executable manually for each device. The complete

Dockerfile is shown in Figure 14. Finally, Docker is used to build and tag the image from this

Dockerfile.

```
  GNU nano 5.2                    Dockerfile                              Modified
# base image
FROM ubuntu

# install ubuntu server
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update && apt-get install -y ubuntu-server

# copy all files
COPY . /usr/src/deepthings/

# run application
WORKDIR /usr/src/deepthings
ENTRYPOINT ["/usr/src/deepthings/deepthings"]

#, "-mode", "gateway", "-total_edge", "2", "-n", "5", "-m", "5", "-l", "16"]
#CMD ["/bin/bash" "/usr/src/deepthings/gatewaycmd"]
#CMD ["/bin/bash"]
```

Fig. 14: Final Dockerfile

Running the Docker Containers

Before running the containers for the IoT devices, a network is created using Docker. The network is a bridge type network, used for standalone containers that communicate with each other. The same image is used to run containers for the gateway device, each edge node, and a host. Each container must be started individually. In the run command both the name of the network created earlier and the IP address for the device being run are specified. The IP addresses chosen should match those defined for the gateway device and edges in the configuration file before the DeepThings executable was created. From inside each container, the DeepThings executable is run, using the same commands as when the executable was run on the VMs.

The created Docker image can be shared or uploaded to Docker hub, a site for sharing Docker images, and run on other computers without any extra files. Once this image is copied to a device with Docker, the only steps necessary to run a DeepThings IoT cluster is to create the Docker network and run the individual containers. Contrast this with starting from the C source code for DeepThings, available at [17]. To run DeepThings, the user must install dependencies, download and add weights, conFigure IP addresses, and make the executable. If attempting to run DeepThings on a testbed, they must also conFigure the network of several VMs or computers.

This Docker image should also facilitate setting up several real IoT devices with DeepThings. If starting from the C source code, configuration and installation must be done separately for each device. Using Docker, the configuration can be done once, the Docker image would then be shared between devices, and each device runs it. The Linux distribution used on

each IoT node does not need to be considered when creating the image. Because Docker runs containers on the same Linux kernel as the host, this virtualization should also be lightweight enough to run on constrained IoT devices.

Key Takeaways and Future Work

The author gained more insight into Deep Learning in general, and Convolutional Neural Networks in specific. How CNN layers function and the purpose they serve has been discussed and illustrated.

Through research into the DeepThings framework and reproduction of a distributed IoT network in virtual machines, the author learned about state-of-the-art methods for distributing a CNN inference task over several constrained devices. The methods of Fused Tile Partitioning, Work Stealing, and the modified work distribution to support efficient FTP both strengthened understanding of CNNs, and how to break Deep Learning work into smaller parallel tasks.

The author learned how to use the tool Docker, which has wide applications in program development, especially for web services and pages. The author learned how to use Docker to package a program into a portable image, how to use Docker network to connect devices running on the same computer, and how to simulate an IoT cluster using either VMs or Docker.

Future work includes more detailed testing and data collection on running the DeepThings Docker containers, streamlining the image, and testing the container on other IoT hardware. Further testing using the DeepThings Docker image and Docker network to determine the size, memory, and power use of an edge or gateway device running on Docker would offer more insight into which IoT devices would be capable of running the DeepThings implementation from a Docker container.

The author expects that the size of the Docker image and the runtime efficiency of the container it produces could be greatly improved by determining all Ubuntu dependences the DeepThings framework requires to run. As mentioned before, the Docker image created in this work installs Ubuntu-server, which contains software packages in excess of 1GB. Including only necessary packages and avoiding a full installation of Ubuntu-server would increase the pool of IoT devices capable of running this image.

Finally, while this work produced a portable Docker image for running DeepThings, it did not test the implementation on physical IoT devices. It is expected that this image can run on most devices capable of running Linux, but this should be attempted in a real IoT cluster in future work.

Conclusion

This paper introduced Internet of Things networks, Machine Learning and neural networks, highlighting both the benefit and the complexity of processing data for Deep Learning algorithms near the edge of IoT, and explored attempts to process neural network inference on IoT devices. It examined the framework developed in [1] more closely, discussed the main techniques it employs to effectively break Convolutional Neural Networks into distributed tasks, and reproduced the deployment of the framework to a testbed of virtual machines, to reinforce understanding of this process with hand-on execution of the algorithms. Finally, this work packaged the C implementation in Docker and tested it in a Docker network. The purpose of this work was to increase understanding of ML, CNNs, and distributed ML in IoT clusters.

References

[1]   Z. Zhao, K. M. Barijough and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters", *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst*., vol. 37, no. 11, pp. 2348-2359, Nov. 2018.

[2]   P. Sethi and S. R. Sarangi, "Internet of Things: Architectures protocols and applications", J. *Elect. Comput. Eng.*, vol. 2017, pp. 1-25, Jan. 2017.

[3]   iFixit, "Google Home Teardown," iFixit, 7 November 2016. [Online]. Available: https://www.ifixit.com/Teardown/Google+Home+Teardown/72684. [Accessed 9 February 2021].

[4]   iFixit, "Amazon Echo Teardown," iFixit, 16 December 2014. [Online]. Available: https://www.ifixit.com/Teardown/Amazon+Echo+Teardown/33953. [Accessed 9 February 2021].

[5]   N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi and F. Kawsar, "An early resource characterization of deep learning on wearables smartphones and Internet-of-Things devices", *Proc. ACM Int. Workshop Internet Things Towards Appl.,* pp. 7-12, 2015.

[6]   A. B. Nassif, I. Shahin, I. Attili, M. Azzeh and K. Shaalan, "Speech recognition using deep neural networks: A systematic review", *IEEE Access*, vol. 7, pp. 19143-19165, 2019.

[7]   S. Albawi, T. A. Mohammed and S. Al-Zawi, "Understanding of a convolutional neural network", *Proc. Int. Conf. Eng. Technol. (ICET),* pp. 1-6, Aug. 2017.

[8]   A. Voulodimos, N. Doulamis, A. Doulamis and E. Protopapadakis, "Deep Learning for computer vision: A brief review", *Comput. Intell. Neurosci.*, vol. 2018, 2018.

[9]   D. Peteiro-Barral and B. Guijarro-Berdiñas, "A Survey of Methods for Distributed Machine Learning", *Progress in Artificial Intelligence,* vol. 2, no. 1, pp. 1-11, 2013.

[10] L. Markus, "8 - Traffic models for machine-to-machine (M2M) communications: types and applications", *Machine-to-machine (M2M) Communications*, pp. 133-154, 2015.

[11] M. Abadi et al., "TensorFlow: A system for large-scale Machine Learning", *Proc. USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, vol. 16, pp. 265-283, 2016.

[12] J. Konečný, H. B. McMahan, D. Ramage and P. Richtárik, "Federated optimization: Distributed Machine Learning for on-device intelligence", *CoRR*, 2016, [online] Available: http://arxiv.org/abs/1610.02527.

[13] J. Mao, X. Chen, K. W. Nixon, C. Krieger and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network", *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, pp. 1396-1401, 2017.

[14] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review 49.1*, pp. 71-79, 2015.

[15] D. Merkel, "Docker: Lightweight Linux Containers For Consistent Development And Deployment," Houston, TX, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[16] J. Redmon and A. Farhadi, "YOLO9000: Better faster stronger", *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 6517-6525, 2017.

[17] Z. Zhao, "DeepThings," https://github.com/SLAM-Lab/DeepThings, 2018.

[18] https://pjreddie.com/media/files/yolo.weights

[19] "Best practices for writing Dockerfiles," Docker Inc., [Online]. Available:

https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. [Accessed 9

February 2021].