Time-efficient offloading and execution of machine learning tasks between embedded systems and fog nodes

by

Darren Saguil

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Faculty of Engineering and Applied Science

Ontario Tech University

Oshawa, Ontario, Canada

December 2019

Copyright © Darren Saguil, 2019

THESIS EXAMINATION INFORMATION

Submitted by: **Darren Saguil**

Master of Applied Science in **Electrical and Computer Engineering**

Thesis title: Time-efficient offloading and execution of machine learning tasks between embedded systems and fog nodes

An oral defense of this thesis took place on December 10, 2019 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Ying Wang
Research Supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Masoud Makrehchi
Thesis Examiner	Dr. Patrick Hung

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

Abstract

As embedded systems become more prominent in society, it is important that the technologies that run on them must be used efficiently. One such technology is the Neural Network (NN). NN's, combined with the Internet of Things (IoT), can utilize the massive amounts of data produced to optimize, control, and automate embedded systems, giving them more functionality than ever before. However, the status quo of offloading all NN functionality onto external devices has many flaws. It forces the embedded system to completely rely on networks which may have high latency or connection issues. Networks may also expose them to security risks. To reduce the reliance of IoT devices on networks, we examined several solutions such as delegating some NN's to run solely on the IoT device or splitting the NN and distributing the subnetworks into different devices. It was found that, for shallow NN's, the IoT device itself could run the NN at a rate faster than offloading it to an external device, but the IoT device needed to offload its inputs once the NN's started to increase in layers and complexity. When splitting the NN, it was found that the number of messages sent between devices could be reduced by up to 97% while only reducing the accuracy of the NN by 3%.

Statement of Contributions

The contributions of this thesis are as follows:

Contribution 1: in this thesis, we discuss the effect of splitting machine learning inputs between embedded systems and fog nodes. By running low complexity models on the embedded system itself, the throughput of a mixed-dimensionality input dataset was effectively increased.

Contribution 2: in this thesis, we discuss a framework which allows users to develop, train, and test neural network models which can then be distributed among computers within the IoT hierarchy of devices

1. Time-efficient offloading for machine learning tasks between embedded systems and fog nodes (Accepted and Published in IEEE International Symposium on Real-Time Computing (IEEE ISORC 2019))

2. A Layer-Partitioning Approach for Faster Execution of Neural Network Models in Edge Networks (Manuscript in Preparation)

Acknowledgements

I would like to express gratitude to my family for their never-ending love and support. They provided everything for me to grow and learn, provided financial support, and motivated me to keep moving forwards.

I would also like to express my sincerest gratitude to Dr. Akramul Azim for his supervision and financial support during my studies. I also express thanks to my friends and colleagues in the RTEMSOFT group for their assistance in performing my experiments, writing research papers, as well as making presentations.

Thank you for all your support. None of this would have been possible without any of you.

Contents

Abstract i

Statement of Contributions ii

Acknowledgements iii

Contents iv

List of Figures vi

List of Tables vii

Abbreviations viii

1 Introduction 1
- 1.1 Current Issues with the Status Quo . . . 2
- 1.2 Thesis Objective . . . 3
- 1.3 Contributions . . . 4
  - 1.3.1 Neural Network Framework . . . 4
  - 1.3.2 Neural Network Input Splitting . . . 6
- 1.4 Experiment Overview . . . 6
- 1.5 Thesis Organization . . . 8

2 Background 9
- 2.1 Heterogeneous Computing . . . 9
  - 2.1.1 Cloud Computing . . . 10
  - 2.1.2 Fog Computing . . . 11
- 2.2 Embedded Computing . . . 12
  - 2.2.1 WCET Analysis . . . 13
- 2.3 Machine Learning . . . 13
  - 2.3.1 Classification Algorithms . . . 14
  - 2.3.2 Deep Learning . . . 15
  - 2.3.3 Federated Learning . . . 15
- 2.4 Other Consideration: Security Issues . . . 17

3 Related Work **18**
3.1 Changing Machine Learning Models . . . 19
3.2 Partial Inferencing . . . 19
3.2.1 Neurosurgeon . . . 20
3.3 Cascading Neural Network . . . 21
3.4 Distributed Deep Neural Network . . . 22
3.5 Research Extensions . . . 22

4 Proposed Approach **24**
4.1 Research Goals . . . 24
4.2 Requirements . . . 25
4.2.1 Delegation Phase Requirements . . . 25
4.2.2 Splitting Phase Requirements . . . 26
4.3 Delegation Phase Architecture . . . 26
4.3.1 Mixed Dataset Testing . . . 27
4.4 Splitting Phase Framework Architecture . . . 28
4.4.1 Layer-by-Layer Construction . . . 28
4.4.2 Chaining and Loss Aggregation . . . 30
4.4.3 Entropy Calculation . . . 32
4.4.4 Decision Making Algorithm . . . 32
4.4.5 Model Evaluation . . . 33
4.5 Workflow . . . 33
4.5.1 Delegation Phase . . . 33
4.5.2 Machine Learning Model Splitting . . . 35

5 Experiments **37**
5.1 Prerequisites . . . 37
5.1.1 Software . . . 37
5.1.2 Hardware . . . 38
5.1.3 Datasets Used . . . 39
5.2 Experimental Setup . . . 41
5.2.1 Delegation Phase Setup . . . 41
5.2.2 Splitting Phase Setup . . . 43
5.2.3 Datasets . . . 44
5.3 Results and Analysis . . . 45
5.3.1 Delegation Phase Results . . . 45
5.3.2 Splitting Phase Results . . . 48
5.3.3 Splitting Phase . . . 51

6 Conclusions **56**
6.1 Future Work . . . 58

Bibliography **60**

List of Figures

1.1 Model of the proposed system 5

2.1 Architecture of an IoT network 12
2.2 Example architecture of an ANN 14
2.3 Diagram of an MITM attack 16

3.1 Example of how a MITM attack can be stopped by partial inferences 20

4.1 Setup of the task distribution for the delegation phase 34
4.2 Setup of the task distribution for splitting the ML model 36

5.1 Example Images from the Fashion-MNIST data set 41
5.2 Example Images from the CIFAR-10 data set 42
5.3 Structure of the NN models used in this experiment. The highlighted layers indicate that they have an early exit 44
5.4 Offloading implementations compared for the single datasets 47
5.5 Offloading Implementations compared for a Mixed data set 48
5.6 Layer time results for the non-image datasets on Model 0 49
5.7 Layer time results for the image datasets on Model 0 50
5.8 Layer time results of the Wall-Turn datasets on Model 1 51
5.9 Layer time results of the image datasets on Model 2 52
5.10 Layer time results of MNIST on Model 3 52
5.11 Layer time results of CIFAR-10 on Model 3 53
5.12 Effect of changing the entropy threshold on the accuracy of the split model 54
5.13 Effects of increasing the entropy threshold on the messages sent between devices 54

List of Tables

5.1 Prediction accuracy for every dataset on every model they were executed on 45
5.2 WCET measurements (in seconds) for each model for all 4 trials, as well as the network latency 46
5.3 Total execution times of each model with every dataset 48

Abbreviations

IoT Internet of Things
ML Machine Learning
MLP Multi-Layer Perceptron
NN Neural Network
DNN Deep Neural Network
DDNN Distributed Deep Neural Network
WCET Worst Case Execution Time
MITM Man in the Middle
CV Computer Vision

Chapter 1

Introduction

Embedded systems are on track to become one of the most pervasive technologies on the market. Their power, flexibility, and low price allows them to become the solution to problems in many industries, such as manufacturing and retail. Billions of these devices are estimated to be in use by 2020 [8]. Alongside these developments, the area of Neural Network (NN) algorithms has also grown to new heights. Like embedded systems, their flexibility lets them be applied to many areas, such as the automotive industry or even the fine arts. Intuitively, combining the two technologies would open many new avenues for innovation.

One such advancement is the Internet of Things (IoT). By implementing embedded systems in many objects in our lives, such as vehicles, healthcare devices, and entertainment electronics, these devices can collect data and automate some of their functionality [12]. They do this by using any attached sensors to collect data for further analysis. Also, as embedded systems are capable of distributing their processing power over networks, they are capable of implementing NN's. Although these algorithms are computationally expensive, such that they may drain the battery of the device or potentially overheat it if used independently [16], their usage can be

hosted on remote devices such that the embedded systems themselves do not need to process the data.

To process the data, embedded systems may use fog networks or cloud computing to host their NN models. Fog networks are preferred to host this data over cloud computing due to their predictable networks latency and close proximity to a majority of end devices [4], whereas cloud data centers are usually too far away to provide a reliable network latency for cyber-physical functionalities [5]. Fog networks, however, provide the same functionality to a lesser extent using nearby fog nodes [13], also known as cloudlets [22]. Although this reduces the variance in network latency, there is still much to be improved in these systems.

1.1 Current Issues with the Status Quo

Not all things in IoT have the storage space or processing power to allow for the computationally expensive algorithms of NN's [20]. Heterogeneity in storage space and processing power between devices is a characteristic of embedded systems; each device is limited by its own set of resources as determined by their user's requirements. There is also a wide variety of algorithms to choose from, each one with different resource requirements [8]. Therefore, it is not guaranteed that every NN will work on every embedded system in IoT, and it is a difficult and time-consuming process to determine which one will.

It is also not cost-effective to provide everything with the resources to run NN algorithms on their own; the cost may need to go towards other needs such as cameras, batteries, and servos [19]. It is much more cost effective to have the devices collect the data, then have it processed on another device with the necessary computational resources. After processing, the other device can determine the action to take and

send it back, whereas the thing can act upon the result.

This introduces another problem: the things that rely on these algorithms are now constrained to edge networks. They would lose some, or all, of their functionality if anything were to disrupt their connection [20]. Although the connection to a network is a main tenet of IoT, it is not guaranteed that the things will always be connected. Even when connected, it is possible for the network to lose connection quality, thus making the data throughput of the device unpredictable.

1.2 Thesis Objective

These problems provide us with a dilemma in efficiency: how can we leverage the power of fog computing with embedded devices while maintaining the accuracy of the system? The solution presented in this paper is to reduce the reliance of embedded devices on distributed computing by splitting up the functionality of the NN onto different devices. The next step is determining which input tasks should be executed locally on the device and which tasks must be offloaded to a fog node. By performing tasks locally, they will have a predictable throughput, but it might be slower than offloading it to a more powerful device and cause the device to miss deadlines, which is a major problem in real-time systems. However, as having a predictable execution time is preferred as well; offloading to an external device should be done when it is absolutely needed to reliably make deadlines. Therefore, we are left with the following question: *what are the conditions that a task and its environment must meet to initiate the offloading process?*

One answer includes determining if a model should be ran locally or externally based on its dataset. Some datasets can be ran on shallow neural networks without sacrificing accuracy; it is possible for shallow neural networks while maintaining an

accuracy of over 90% while running at rates faster than transmitting to an external device as seen in this research.

Another solution includes determining if an offload should occur based on the entropy of the output. The model can then be split and distributed amongst the devices within the IoT hierarchy, and devices on the lower end of the hierarchy can run a shallow version of their respective model. If they are not confident with their output, they can offload their result to another device for further analysis. Not only does this cut down on messages sent between devices, it may also save time and add an extra layer of security in devices.

1.3 Contributions

There were two main contributions for this research. The first was the development of a framework which could create, split, and distribute NN's among different hardware [11]. The second main contribution were the experiments done using this framework; not only were the split models tested, entire datasets were split among different as well.

1.3.1 Neural Network Framework

Solving these problems required building a framework which can create and distribute neural networks. TensorFlow was used as the backbone for this framework as it provided high-level functions to build, train, and test neural networks. However, additional function were written in python which provided the extra functions that were needed.

A wrapper for the TensorFlow functions was provided as a class called the PartialModel. This class provided all NN functionality; it could create, train, and test

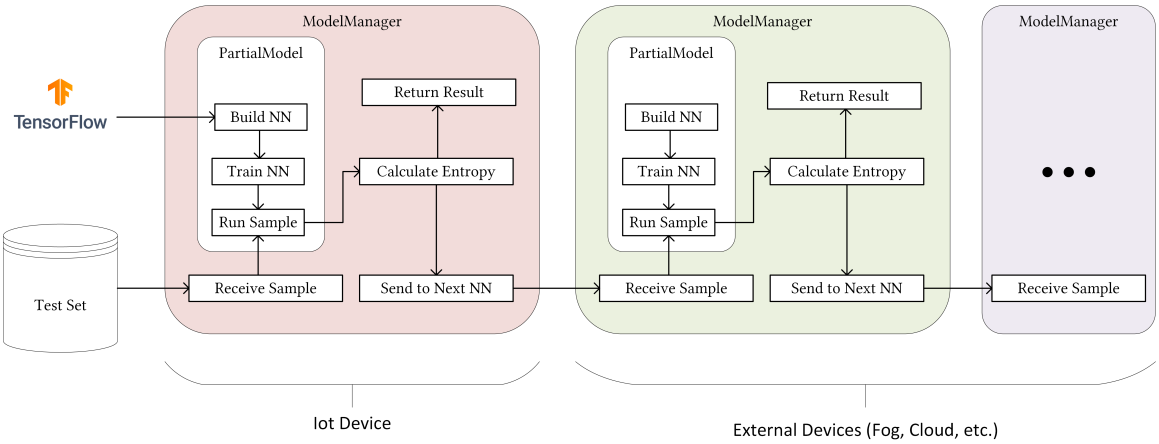

Figure 1.1: Model of the proposed system

models as needed. It also provided the ability to gauge the performance of each layer inside of the model in order to determine its throughput. The most important functionality it provided was that it gave 2 outputs; the predicted class from the input as well as the output of the layer before the prediction class. The penultimate layer output is crucial to determining the confidence of the model.

When used, any number of `PartialModel`s can then be placed inside of a `ModelManager`. This class handled the inputs and outputs to the `PartialModel`s contained within it. It is the class which also connects to other `ModelManager`s across the IoT hierarchy, allowing for a distributed neural network among devices. Figure 1.1 showcases how this system can be set up: the `ModelManager` receives the input and sends it to the `PartialModel`, then the `PartialModel` returns both its prediction and penultimate output, then the `ModelManager` determines if the entropy is high enough to be offloaded. If it is too high, the `ModelManager` will send the penultimate output to the next `ModelManager`.

1.3.2 Neural Network Input Splitting

Another contribution was that we determined an efficient load distribution of NN tasks for embedded systems to send to fog nodes. This was done by determining a runtime threshold for each machine learning input and applying this threshold during runtime. Any input which exceeds the threshold was offloaded, allowing for an increase in throughput in a mixed-input dataset environment. The contributions of this part of the research are:

- Determined an efficient load distribution of NN tasks for embedded systems to send to Fog Nodes.
- Determined the effectiveness of ML tasks when executed on an embedded system and compared it to the status quo (purely sending all NN tasks to a fog node)
- Determined how the characteristics (i.e. input dimensionality and model complexity) of the NN task may affect its throughput on an embedded system, or when sending it to a fog node.
- Developed a test bed for these data sets to determine their experimental throughput.

1.4 Experiment Overview

To examine these possible improvements, we set up a simulated IoT environment. An embedded device that served as the IoT device and a computer as a fog node was prepared and connected via Wi-Fi. We then executed a long queue of singular to mixed complexity tasks which consisted of one or many NN models running multiple data sets. Each data set had varying dimensionalities ranging from 2 dimensions

to full colour images. The Worst-Case Execution Time (WCET) for each task was then measured on the local machine (embedded device) and the fog node. The upper bound of the latency to exchange the data over Wi-Fi was also measured. From this, we determined which tasks were small enough to run more efficiently locally. We then prepared offloading algorithms which delegate these small tasks to the local machine and offloads any other tasks.

In the following phase, we split each NN model and split them into sub-models. We then took each of the sub-models from the split and distributed them amongst the IoT device and the Fog Node. An early exit mechanism was implemented so that the inference could stop after each sub-model if the device was confident in that sub-model's output. By utilizing this, the IoT device only needs to send messages only when they were not confident in their result, rather than every time, thus increasing time efficiency.

For this experiment, only classification algorithms were looked at, namely the NN. This algorithm is used in many deep learning and computer vision applications, so it is an important algorithm to examine. Also, only fog computing solutions were experimented with as opposed to cloud computing. Although cloud computing can greatly extend the resources of embedded systems over fog computing, its distance from the IoT device often leads to high latency, which makes it undesirable for IoT solutions [21]. The data sets used in this experiment varied between numerical to image data to simulate the various data types that IoT systems may encounter.

As for the experiment itself, the hardware was limited to one embedded system and fog node, and it is only using WiFi as opposed to other connection types (e.g. Ethernet cable, Bluetooth). This is to control as many variables as possible in the hardware as the main focus of this experiment was the effect of changing the models and datasets on the system. However, this limitation will only provide experimental

results for this specific hardware, as other devices might have differing processing power. Nonetheless, the main goal of this experimental setup was to provide a proof of concept when comparing our system to the status quo.

1.5 Thesis Organization

This thesis has been organized into five chapters. Chapter 2 deals with the technologies which this research was based off of, namely IoT and Machine Learning algorithms. It will deal with how these technologies can be enabled, their applications, and some of their future developments. It also deals with related works similar to the research done here. These works vary from local embedded system machine learning to splitting neural network functionality among devices. Chapter 4 details how this research was done and the composition of the framework. It also deal with the theories in which this framework is based off of. Chapter 5 shows the materials, data, and processes to perform an experiment on the framework. The results of these experiments will also be shown in this chapter. Chapter 6 sums up the entire thesis and concludes it. It also provides possible avenues into future works.

Chapter 2

Background

In this chapter, we discuss the underlying technologies which make up this research. We will discuss their origins, how they have been enabled, as well as their applications. Theories and possible avenues of research for these technologies will also be looked at as well.

The technologies examined in this chapter are the Heterogenous Computing, Machine Learning, and Real-Time Systems. Broad overviews of these topics will be given, and the area of focus this research deals with will also be discussed. These areas are fog computing, deep learning, and worst-case execution time analysis respectively.

2.1 Heterogeneous Computing

One aspect of this research is the usage of heterogeneous networks. An example of a hetergeneous network is IoT. IoT can be described as the seamless integration between classical networks and networked objects [25]. By classical networks, it is meant that physical things that are connected using some sort of wireless network protocol. This offers enhancements to the object's current features and may also create paths

to the development of new features. The Harvard Business Bureau describes the main capabilities of these networked things as monitoring, control, optimization, and autonomy [12].

Monitoring allows connected things to gather usage statistics, as well as information about the environment. They are built with sensors to collect all this data, then send them over the cloud to be stored. Control features allow the thing to be configured to the user's preference. For example, the radio stations inside a car's multimedia system can be adjusted to automatically jump to favorite stations. Optimization allows the thing to be able to adapt and change itself to the user's preferred preferences. It makes use of feedback loops from the environment to determine how to operate at maximum efficiency. The self-diagnosis and learning allow for improved performance, preventative maintenance and repair, and more [12]. Autonomy amalgamates all other features to allow this thing to operate independently without any human input.

Unfortunately, networked things cannot be the sole component of the IoT. These devices are generally simple and lack the processing power to perform tasks outside their basic functions. This means that the IoT must exploit the concept of distributed computing. By offloading resource-heavy tasks to an external and more powerful device, the potential functionality of these objects are almost endless. The main frameworks to offload strenuous tasks are fog computing and cloud computing. However, fog computing is much more desirable due to its predictable throughput [21].

2.1.1 Cloud Computing

The ever increasing need for data processing and storage has lead embedded systems to use external devices to use these extra functionalities. Since external devices capable of providing these services were not readily available at the time, cloud com-

puting became prevalent to provide this assistance [26]. Cloud computing gives many resource-constrained networks a centralized data center to store and process data. There are many ways of implementing cloud computing into IoT systems. However, the two most common ways are centralized and peer-to-peer (P2P) using a gateway service.

2.1.2 Fog Computing

Fog computing can be described as edge devices which extend the resources of nearby devices, as opposed to using a remote cloud [5]. It is a "grounded" take on the cloud computing concept, allowing for lower latency and closer proximity between things and their remote resources. This low and predictable latency of the fog computing architecture allows for better response times for real-time applications, such as health care. Figure 2.1 displays how fog, edge, and cloud computing solutions can interact with each other and are not entirely independent. However, the edge and fog devices are within the same proximity. The main difference between the fog and edge networks is that fog nodes (also referred to as "cloudlets") are similar to cloud servers, but they are on the edge of the network, while edge devices can be any sort of device, including smart mobile devices, laptops, or embedded systems.

By placing the fog nodes between the edge and the cloud, the large increase in numbers of mobile devices and sensors can be accommodated without bogging down mobile networks and cloud data centers. Also, as innovation in software applications is quickly outpacing network advances and deployment [5], fog nodes may provide better quality of service than cloud computers as they can potentially use short-ranged wireless transfers for quicker response times. These advances in software may include machine learning, which can use a lot of resources, both from the network and end devices.

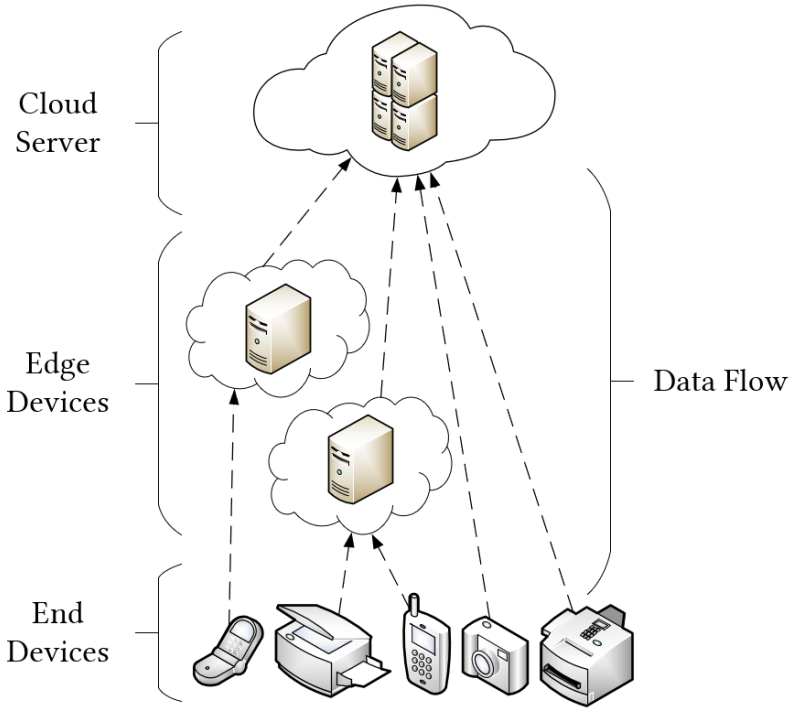

Figure 2.1: Architecture of an IoT network

2.2 Embedded Computing

The end devices found in IoT are utilizing the concept of embedded computing. These computers operate at power levels exponentially lower than general use computers [10], but they offer more specialized functions determined by the device they are embedded in. These devices can range from home appliances to vehicles, like airplanes and self-driving cars. This means that some of these devices have safety-critical functionality, such as industrial plant monitoring and alarm systems [6]. Therefore, embedded computing has timing requirements, which makes timing analysis needed for both safety and quality of service.

2.2.1 WCET Analysis

Worst Case Execution Time (WCET) analysis allows developers to determine the longest time a program will execute on specific hardware [17]. This is useful in real-time systems as it allows developers to determine if a certain task is schedulable or not, then fine-tune the system based on this analysis. The WCET of any program can be measured on runtime, or it could be estimated statically using software like the OTAWA framework or the aiT WCET Analyzer from AbsInt.

At its most basic form, the WCET can be calculated by simply finding the longest path in the program. After finding this path, calculating the WCET is as simple as adding the time to execute the header, T_H, to the time to execute the longest path, T_P, multiplied by the number of loops in which the longest path executes, LB. This is shown in Equation 2.1.

$$WCET = T_H + T_P * LB \tag{2.1}$$

2.3 Machine Learning

Machine Learning (ML) provides IoT devices with more features than ever thought possible. It is the process of taking large amounts of unlabelled or seemingly unrelated data, and finding features and relationships between them. In short, ML can be described as "finding the function" in a program, rather than performing a function. In this section, we only look at the classification part of machine learning, instead of the clustering or reinforcement methods.

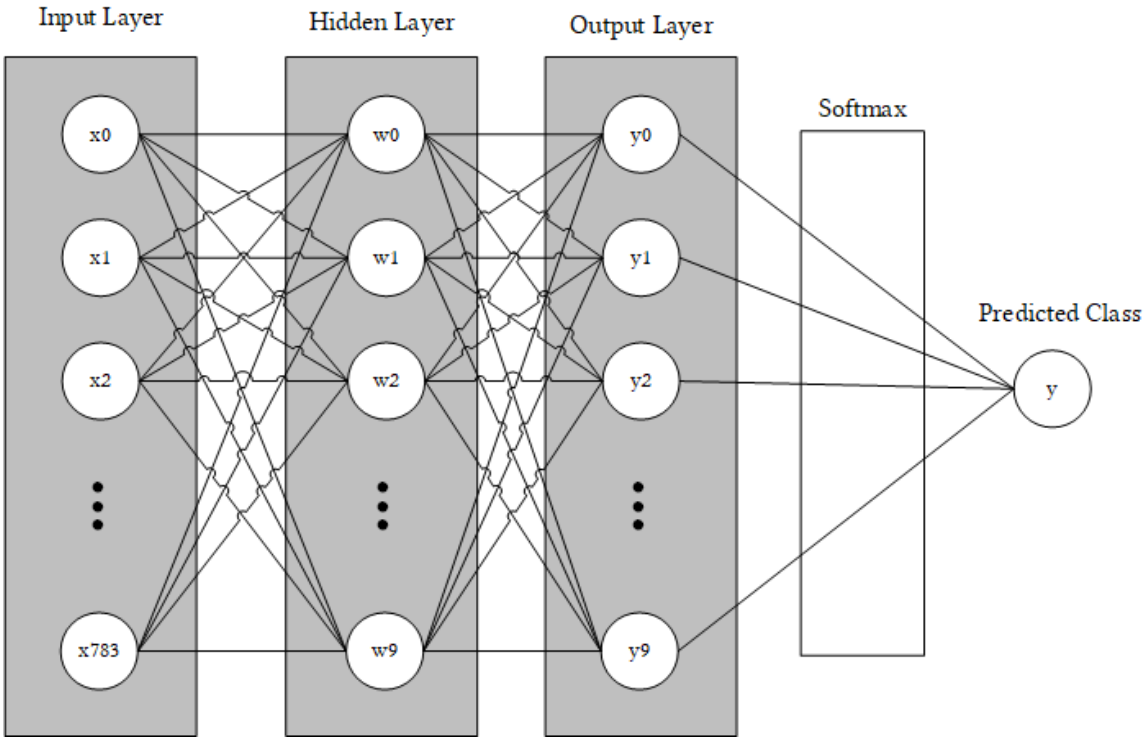

Figure 2.2: Example architecture of an ANN

2.3.1 Classification Algorithms

Classification is a subset of ML Algorithms which was used in this experiment. Classification is a supervised learning method that requires some sort of labeled data to train the model, such that the model can label unlabeled data accordingly. This can be useful in a parking lot; image data from cameras at the entrances and exits of the lot can detect vehicles that are passing through, then classify them as to whether they are entering or exiting. Another application can be the sound data from a smart home system; a classification system can classify sounds to determine the current living system. Glass breaking in the middle of the night can set off an alert signal from the smart home system to inform the user of a possible break-in. This type of ML may include many different variants, such as support vector machines, decision trees, ANN's, and deep learning. The algorithm used in this research was the ANN.

ANN's were one of the earlier implementations of machine learning in the field of embedded systems. ANN's have been around since the 1950s and have proved to be useful through to today. Today, they can provide classifications and insight for an oncoming stream of data from sensors in IoT. Also, if the features of the data are clearly defined and extractable, ANN's can update and optimize their systems to provide reinforced safety standards in safety-critical systems. These algorithms are comprised of an input layer, an output layer, and a varying amount of hidden layers in between, as seen in 2.2. Unfortunately, due to the black-box nature of ANN's (i.e., the lack of any sort of traceability in the training process and difficulty in creating predictable outcomes while training), as well as their stochastic nature due to certain layers (e.g. dropout layers), their purpose has mostly been assigned to an advisory role rather than providing any functional safety [20].

2.3.2 Deep Learning

Deep Learning, or Deep Neural Networks (DNN), is a subset of machine learning that utilizes NN's with numerous hidden layers between the inputs and the output. The number of hidden layers can range from 5 layers in *LeNet*, to over a hundred layers in *ResNet* [24]. By having a large number of hidden layers, more analysis can be done on the data, allowing for higher prediction accuracy. The most commonly used DNN in a server environment is the *AlexNet* [18], a machine learning model used for image processing and classification.

2.3.3 Federated Learning

Federated learning is a privacy-preserving distributed method of training deep neural networks [23]. This type of learning prevents the storage of confidential information

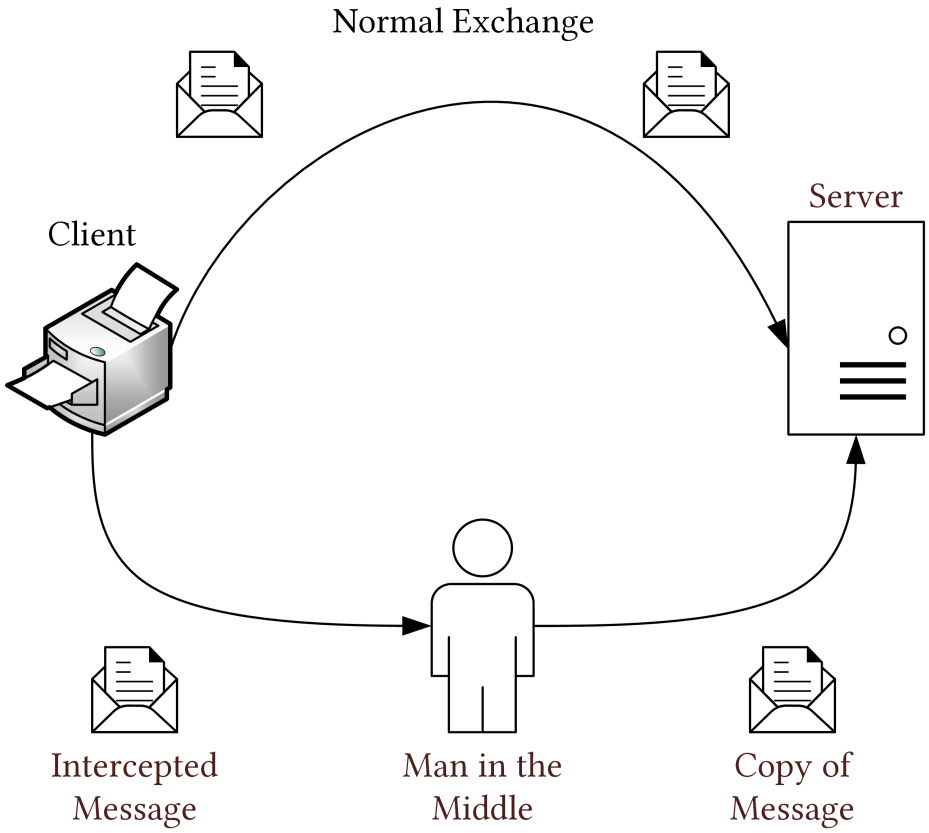

Figure 2.3: Diagram of an MITM attack

from being stored on a centralized server. It does this by providing every end device with its own master model, and then each of the end devices train the model and send it back to the centralized server. All the models are then aggregated into the centralized server. This way, no raw information is passed to the server [29]. Unfortunately, this requires massive amounts of communication costs [31], so it will not be explored in this experiment. However, it shows that NN's are susceptible to security concerns, but can be addressed in intuitive ways.

2.4 Other Consideration: Security Issues

While IoT allows for connectivity to external devices, the openness of wireless connectivity may lead to security issues as well. For example, in July 2015, the US Federal Drugs Administration discovered that the Hospira LifeCare PCA3 and PCA5 computerized infusion pumps could be remotely accessed by anyone through a hospital's network [2]. These sorts of attacks are prevalent in IoT systems as many of these systems lack encryption due to developer oversight or lack of processing power [3].

One such attack which is prevalent in IoT systems is the Man-In-The-Middle (MITM) attack. It consists of two devices exchanging messages, and an attacker intercepting each message and sending copies of each message to their recipients. An example of this can be seen in Figure 2.3. Although the experiment done in this research does not directly address this issue, it might indirectly assist in maintaining the privacy of the system, as seen in Chapter 3.

Chapter 3

Related Work

This chapter deals with works related to this research. Their research provided the groundwork which guided the process and experiments expressed in this paper. Each work discussed in this section will have their contributions, approach, and results examined. Then, a comparison to the work done in this thesis will be performed to signify the contributions of this paper. The works described in this section vary from embedded machine learning, such as computer vision systems, to Distributed Deep Neural Networks.

In this section, we examine related works regarding running ML algorithms on embedded systems. These types of improvements benefit all ML algorithms, not just NN. However, they provide insight into which algorithms work best, how they are best used, and possible improvements to the offloading of machine learning inputs. We also consider extensions on the principle of partial inferences. As it is a novel concept, there has been multiple studies and branches of research done on this topic. All three of the following related works implement the concept of partial inferences in a different way, from automatically splitting the neural network to implementing convolutional layers in their NN.

3.1 Changing Machine Learning Models

Performing computationally expensive tasks in resource constrained systems has been discussed in literature. One such study was performed by Zidek et al. in [32]. The experiments done in their research consisted of utilizing an embedded system with an attached camera to create a machine vision system. They tested several machine learning system tasks, such as Support Vector Systems (SVM), K-Nearest Neighbors (KNN), and a multilayer perceptron (MLP). Their goal was to test if a resource-constrained machine vision system could detect faults in metal plating. Their experiments showed that this is certainly possible, but to varying degrees of efficiency based on the machine learning algorithm used. In terms of accuracy, the most reliable algorithms were the Gradient Boosted Threes (GBT) and the MLP, reaching an accuracy of up to 99.9%. In terms of speed, it was found that KNN was the fastest with an accuracy near that of the MLP. However, this research was focused purely on execution and accuracy of ML algorithms on the embedded system, whereas offloading may have assisted in some of their implementations.

3.2 Partial Inferencing

Another possible improvement was examined by Jeong et al. in [16]. Instead of sending the data itself, they enabled the device to be able to send its entire execution state over a web app called the *snapshot*. They did this by implementing the NN as an in-browser web-app. Then, whenever the user wants to inference and image, the entire app (i.e. the image, functions, and restoration code for the app), will be sent instead of just the image. This way, they can send their NN inference requests to generic servers, rather than specialized ones. However, they also made another important contribution: they implemented partial inferences to the execution.

Since they are implementing a solution to be sent to generic servers, the confidentiality of the information to be sent must be secured. In order to secure the privacy of the user, they implemented partial inferences; they divided the neural network into front an rear parts. The front part was to be executed by the client, while the rear was to be executed by the client. By doing this, they do not send the original image itself but instead they send the feature data. As this feature data is difficult to read by humans, it can be considered secure; the only way to read it is to have a copy of the NN and perform hill-climbing techniques [16]. This form of pseudo-encryption is shown by Figure 3.1. It is assumed that this form of security is already performed by splitting the neural network, but is not fully tested in the experiment.

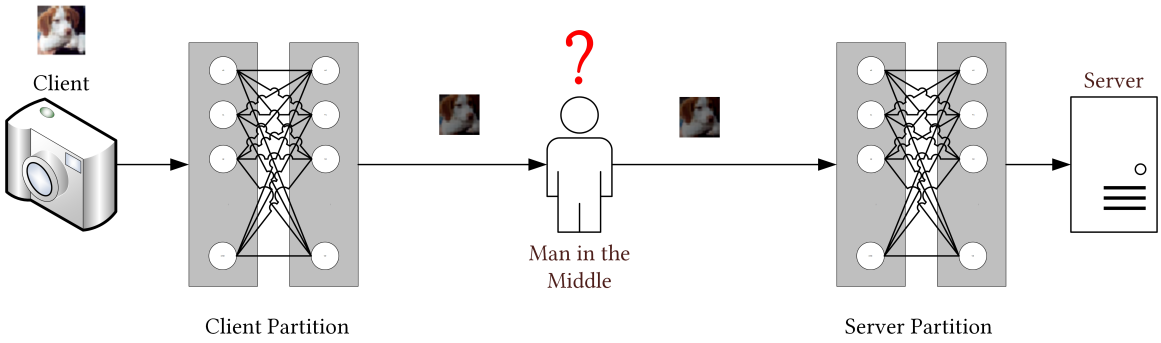

Figure 3.1: Example of how a MITM attack can be stopped by partial inferences

3.2.1 Neurosurgeon

Partial inferences involves distributing the many layers of a DNN amongst numerous devices, where some layers may execute locally, and other layers may execute remotely. After all layers have executed, the output is returned to the device that requested it. An example of this was studied by Kang et al. in [18], where they developed a program which automatically finds the layers to be remotely executed (i.e. finding a layer in the DNN as the offload point). One important aspect of this research was

that, for some DNN's, the size of the data to be transmitted is reduced without any compression, as some layers ahead of the input layer may be a pooling layer or have a low number of output neurons. This means that, by using a partial inference, data transmissions can be performed faster.

They named this program Neurosurgeon, as it can automatically split NN's without any human input. It does this by analysing the properties of every layer in a NN determining which layer is the best splitting point for either least latency or energy consumption. Neurosurgeon then divides the NN into two parts from that layer, where the front half executes client side and the back half executes on the server. THis is done over Remote Procedure Calls (RPC). Using this method, they achieve a 4% increase in throughput on wired exchanges, but a 43% to 136% increase in throughput using wireless networks (i.e. LTE and 3G networks).

3.3 Cascading Neural Network

Studies into MLP's on resource-constrained devices and how they can be offloaded have been performed by Leroux et al. in [20]. They proposed a new type of network referred to as the "Cascading Neural Network". This type of neural network provides devices with an early stopping mechanism during forward propagation. When a prediction is stopped early, it immediately provides an output even though the input has not fully been propagated throughout the network, thus trading off accuracy for speed. They also extended this concept into offloading to various external devices, from embedded GPU's to the cloud. The layers of the cascading neural network were distributed over several devices, only offloading when it is absolutely necessary.

This type of NN was implemented for three different datasets: the MNIST handwritten dataset, the CIFAR-10 dataset, and the ImageNet dataset. Each of these

datasets was outfitted with their own unique NN model to inference data, except every layer in their models was provided with a possible exit layer, including the input layer. After the execution of each layer, a confidence value was calculated. If this confidence value bypassed a certain threshold, the device could stop at that layer entirely. Using this system, they were able to reduce the average runtime by up to 40% depending on the network characteristics.

3.4 Distributed Deep Neural Network

Another similar study which uses early exits was done Kung et al. in [24]. If the DNN is confident with the result, then the classification can be made much earlier without needing to execute the following layers, thus saving execution time. This concept can be taken even further by distributing the layers among the IoT hierarchy [30], where the later layers are executed on higher devices on the hierarchy. This concept is labeled as the Distributed Deep Neural Network, or DDNN [24]. Using early exits in the DDNN, transmission between devices can be foregone entirely, thus shortening the execution time and reducing the reliance on external devices. In fact, using their early exit scheme, Kung et al managed to reduce the communication costs of an edge network by a factor of 20 times.

3.5 Research Extensions

These experiments provide insight into the capabilities and offloading methods of resource constrained systems. We look to extend this research by implementing WCET analyses of machine learning algorithms in order to preemptively determine if local execution on a resource-constrained device is feasible to make the required deadlines,

or if it is better to offload the entire task to a fog node. While it is possible for complex ML algorithms to be executed entirely on embedded systems, the time it takes to execute them depends greatly on the implementation. For example, as shown by [32], the average execution time for prediction using ML algorithms can vary from 5 to 500 milliseconds. Also, within the same implementation, different inputs may cause the machine learning algorithm to run longer, thus justifying the use of cascading neural networks proposed by [20]. This means that it can be difficult to determine if machine learning algorithms should be offloaded or run locally at all. What we propose in this research is to find an efficient distribution of machine learning tasks that can be executed locally on an embedded system before getting offloaded to a nearby fog node, in order to maintain schedulability while reducing dependence on offloading. We also wish to determine the effect in which the model's architecture has on the runtime of split NN's; the NN's done in the literature review looked at pre-built NN's, or NN's with a single type of layer (i.e. dense or convolutional).

Chapter 4

Proposed Approach

In this chapter, we discuss our proposed approach to solve our thesis objectives. We describe the methods, tools, and theories we developed and used in order to approach our objectives. The layout of this section is as follows: we first define the research goals we wished to solve. Next, we describe the frameworks, equations, and algorithms we developed in order to reach these goals.

4.1 Research Goals

As communication tasks are the most costly tasks in an edge network, the main goal of this research was to reduce the communication cost of performing NN inferences in an edge network by distributing the machine learning tasks amongst different hardware. However, there are many other factors which can be affected when attempting to reduce the communication costs. Lowering the communication costs while maintaining or improving the performance of the status quo can be considered as sub-goals. Explicitly, the goals can be stated as:

- Increase the performance of NN inferences in an edge network

- Determine the impact of delegating inputs of machine learning tasks between embedded devices and fog nodes
- Determine the impact on WCET of inferences using partial NN execution on embedded devices
- Maintain the accuracy and reliability of NN's in edge networks

4.2 Requirements

In order to reach these goals, we developed a NN framework which can create models and distribute or split it amongst devices. In order to verify if the NN is functioning properly, we set the following requirements where the framework must fulfill the requirements of two different phases of the research: the Delegation Phase and the Splitting Phase.

4.2.1 Delegation Phase Requirements

In the delegation phase, we chose whether to send entire inputs from the embedded devices to the fog nodes. This decision was based on a WCET threshold which was determined from a validation set and prior executions. The purpose of this phase was to determine if some ML models should be hosted on external devices at all; embedded systems may run these models faster than the upper bounded transmission time meaning hosting on an external device may slow it down. Overall, the requirements for this framework is that it must:

- Build entire NN models
- Determine the WCET of running each dataset on their respective models on various implementations in an edge network

- Use the same model on the embedded system and fog node
- Get the execution time of every layer in each model

4.2.2 Splitting Phase Requirements

In this phase the effects of splitting a NN and distributing it amongst the embedded system and fog node. In this phase, the embedded system will always make inferences on the input. If the embedded system is not confident in its results, it can send its current result to the fog node for further processing. This reduced the execution time of inferences on all models by eliminating unnecessary computations. To do this, the framework must:

- Build NN models, layer by layer
- Chain multiple NN models together by using the output of one model as the input of the next
- Supply each model with early exit layers
- Allow each model to calculate if they should exit early by calculating entropy on each output
- Allow users to train and test each model individually

4.3 Delegation Phase Architecture

The architecture for this phase was simple; using Tensorflow's model building API, it was possible to simply build the models for each dataset and transfer it to both the embedded systems and fog nodes with minimal alterations needed. The only

additions were callbacks to measure the execution time of every layer, and the ability to match input data to their respective models.

4.3.1 Mixed Dataset Testing

The ML algorithms used in this part of the experiment were implemented using Keras, which is a high-level neural network API built on top of TensorFlow, CNTK, or Theano. Scikit-learn was used for their capabilities of creating training and testing sets, and numpy was both used to manipulate and save large arrays of data. Using all of these tools, and the data sets collected in Section 5.2.3, five ML models used to make predictions were trained and saved. Throughout the entire experiment, the same training sets, test sets, and ML models were used. All five models were kept shallow to keep a feasible runtime on embedded systems. Four of the models used a linear sequential model (MLP), consisting of an input layer which activates using relu, dropout layer (0.1 dropout rate), and an output layer which activates with softmax. It required that the input was flattened into a 1D array. They were trained on the navigation data set with 2 inputs, 4 inputs, 24 inputs, the Fashion-MNIST data set, and the CIFAR-10 data set and were named WT2, WT4, WT24, FL, and CL respectively. The last two models were Convolutional Neural Networks (CNN) which consists of 2 convolutional layers and a single fully connected layer. It required that the input was shaped as a 3D matrix, which includes image width, height, and colour channels. They were used on the Fashion-MNIST and CIFAR-10 datasets, and were named FC and CC respectively. The accuracy and epoch training time can be seen in Table 5.1. Every model was trained with a batch size of 256, over 100 epochs.

After the data was received, offloaded, and used as input in the model, the time it took to perform each task was recorded and compared. The timing of each prediction was conducted over four different trials. The first two trials removed the offloader

from the task distribution and delegated each input to a single model. In the first trial, every part of the task distribution was done on the embedded system. The second trial consisted of the embedded system offloading every input to the fog node instead. The network latency and prediction were timed separately in this trial. The third trial kept the full task distribution seen in Figure 4.1. The WCET of every task was measured from the previous trials, allowing for the offloading of every task which bypassed the threshold in Equation 4.5 to the fog node, while every other task was performed locally. In the final trial, every input was randomly offloaded to the fog node, with an offloading chance of 50%.

4.4 Splitting Phase Framework Architecture

The splitting phase architecture fulfilled its requirements in a way similar to the delegation phase, but takes multiple models and chains them together as if they were a single model. Every sub-model within the model has an exit layer where they can finish the inference in the case of a confident result. Each of these sub-models can then be distributed amongst different devices. The full implementation is in the following subsections.

4.4.1 Layer-by-Layer Construction

To meet these requirements, an architecture was developed using TensorFlow as the back end. It allowed users to develop, train, and test consecutive models using a custom subclass of TensorFlow's Model class. The class was capable of adding 6 different types of layers common in CV with ability to customize their options. The 6 layers implemented were: Convolutional, Dense, Pooling, Activation, Flatten, and Dropout.

Convolutional

These layers use a filter with a learned kernel which convolves an image. It computes the dot-product between the filter and various parts of the image, producing a set of feature maps. These are one of the most computationally expensive layers in a CV model. Combined with Dense layers, they make up 90% of a model's computation time [18]. However, Convolutiional layers are more complex and their complexity can be seen in Equation 4.1 [15], where n_l is the number of filters, s_l is the size of the filter, and m_l is the size of the output.

$$O(n_{l-1} * {s_l}^2 * n_l * {m_l}^2) \tag{4.1}$$

Dense

These layers take all the neurons from the input or previous layer and performs a dot product with it with a kernel. This kernel contains the weights learned from training. Optionally, biases can be added to the result as well, thus creating the simple linear equation: $y = m * x + b$, where m is the kernel, and b is the bias. Although not as complex as Convolutional layers, they can accrue a large runtime due to their exponential complexity, as seen in Equation 4.2, where n is the length of the input data and m is the length of the output data.

$$O(nm) \tag{4.2}$$

Pooling

The above layers may produce too many features which may affect the accuracy and runtime of the model. Therefore, it is necessary to combine or group features using a pooling layer. These layers run a single function across all the input data, reducing

the amount of data as well as its dimensionality. They can either have max or average pooling functions.

Activation

Activation layers provide many different non-linear functions, such as rectified-linear (relu), sigmoid, softmax, and argmax. These functions are ran over all the individual neurons in the previous layer, altering the data but retaining the dimensionality of the input.

Flatten

Reduces the dimensionality of the input layer to a single dimension, though the data itself remains the same.

Dropout

Randomly makes neurons ignored during training. This is to prevent overfitting a model (i.e. making a model trained to classify specific to a single training set, rather than general purposes).

4.4.2 Chaining and Loss Aggregation

After the models were created, they were placed within a ModelManager class which takes the output of one model, and uses it as input in the next model. This essentially combines every single model into a much larger feed-forward NN, capable of being distributed amongst different hardware. In order to train this model, the backpropagation of each model was done individually, but each model also used the losses from each model before it for that specific sample, as seen in Equation 4.3. In this equation L'_k is the loss to be used by model for backpropagation, k is the index of the layer,

and w is the weight of the model. For this research, it is assumed that all models will have equal weights.

$$L'_k = \sum_{l=1}^{k} w_l L_l \tag{4.3}$$

Every model also uses this same loss to train the variables in of its exit layer. This means every model has the same exit layer shape, but different values inside of the exit layer. In order to chain the models together, the output from the exit layer is not fed into the next model, but instead the output from the layer before the output layer (penultimate layer) was given. This can only work if the penultimate layer has an output with a shape compatible with the first layer of the consecutive model so it can be used as input.

Algorithm 1 Multiple Model Training algorithm

```
procedure ModelTraining(data, label)
    x, prediction ← model_0(data)
    loss ← calculateLoss(prediction, label)
    optimize(model_0, loss)
    i ← 1
    while i < length(model) do
        x, prediction ← model_i(x)
        loss ← loss + calculateLoss(prediction, label)
        optimize(model_i, loss)
        i ← i + 1
```

The process of loss aggregation is seen in Algorithm 1. The output of the first model (i.e. the prediction and the penultimate output, x), as well as its loss are both calculated. The same happens for every other model in the entire network, except that instead of taking the data from the sample, they take the penultimate output of the previous model. The loss for the model is also added to the total loss of every previous model. After every loss calculation, the model is then optimized using the aggregated loss. This repeats until all the models have been optimized for

that specific sample. The training of the system is done completely offline and the resulting weights are then distributed to all devices within the hierarchy. Although distributed training of a NN is a possibility [7], but is not used in this system.

4.4.3 Entropy Calculation

Since every model has an exit layer, it is possible that any model could be the final output. To determine which model should be the final output, each model must perform a confidence test on their output data. The confidence test used in this research is the entropy equation as seen in Equation 4.4 [24].

$$\eta(x) = -\sum_{i=1}^{|C|} \frac{x_1 \log x_i}{\log |C|} \tag{4.4}$$

This equation outputs a value between 0 and 1, where 0 is high confidence (low entropy) and 1 is no confidence (high entropy). For example, the entropy equation will output a value of 1 if all elements of the probability vector are the same, meaning the class of the sample could equally be any one of the possible classes. The value of the entropy threshold can be determined beforehand using a validation set; the threshold which offers the best accuracy can be chosen as the threshold.

4.4.4 Decision Making Algorithm

When the model provides the ModelManager its output, the ModelManager will perform the entropy calculation on the probability vector provided. It is then compared to an entropy threshold that was determined beforehand. If this threshold is not bypassed, the model is confident and it does not inference any further. This threshold algorithm, along with the chaining algorithm can be seen in Algorithm 2.

Algorithm 2 Confidence testing algorithm

procedure EntropyCheck($data$)
  $x, prediction \leftarrow model_0(data)$
  $i \leftarrow 1$
  while $entropy(y) < T$ & $i < length(model)$ **do**
    $x, prediction \leftarrow model_i(x)$
    $i \leftarrow i + 1$

4.4.5 Model Evaluation

As the models were created using a custom subclass of TensorFlow's Model class, extra callbacks were added to the inference functions to measure the runtime of each individual layer. The timing of layers only occurs during the testing phase, as it is assumed that all training will be done on much powerful hardware, such as the cloud. In this research, we hosted all the computations on the fog node.

4.5 Workflow

4.5.1 Delegation Phase

We first investigated the status quo in the delegation phase by hosting all computations on a fog node, then comparing it to running all computations on the embedded system. To perform this, we first set up a workflow described by Figure 4.1. In this figure, we set up an embedded device which is connected to a fog node over WiFi. A simulated sensor sent data to an offloader which determines the location of its computation.

The offloader in this part of the research used was WCET-based. It simply offloaded every task whose WCET on the embedded system bypassed the threshold provided by Equation 4.5, where T_L represents the upper bound of the varying network latency, and T_F represents the time to perform the same task on the machine

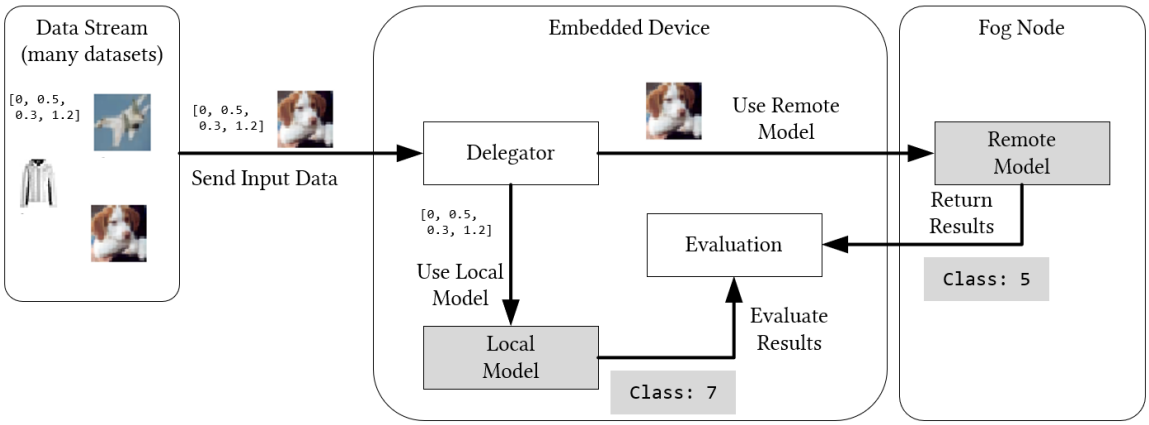

Figure 4.1: Setup of the task distribution for the delegation phase

that the data is offloaded to, in this case, the fog node. In this part of the research, the network latency also included the serialization of the data before the data exchange.

$$WCET_T = 2 * T_L + T_F \tag{4.5}$$

From this equation, we developed a simple $O(n)$ algorithm to determine if data should be offloaded to a nearby fog node, as seen in Algorithm 3. The WCET of each task, denoted by $WCET_d$, was determined beforehand. Both $WCET_d$ and $WCET_T$ were measured by running a validation set on the models, then the upper bound of the execution time for each input was considered as the WCET for their respective data sets. Any external WCET estimation software, such as OTAWA or aiT WCET Analyzers were not used in estimating the WCET for this experiment.

Algorithm 3 WCET-Based offloading algorithm

```
procedure CheckOffloadStatus(D)
    for every d ∈ D do
        if WCET_d ≥ WCET_T then
            sendToFogNode(d)
        else
            queueToLocalProcessors(d)
```

The process of Algorithm 3 is as follows. The array of data, D is given which is an array of every sample to be given to the delegator. $WCET_d$ holds the WCET of that specific data set which was measured using the validation set beforehand. If this value exceeds the threshold, $WCET_T$, then the input must be offloaded.

Using this approach, the following observations could be made about using ML in embedded systems:

- How does the dimensionality of a dataset affect its throughput during testing?
- How does the complexity of the model affect its throughput during testing?
- Which of the previously mentioned factors affects the throughput of testing the most?
- What is the WCET threshold where embedded systems must host their computations on a fog node?

4.5.2 Machine Learning Model Splitting

Using the data gleaned from the first part of the research, a time-efficient ML model partitioning scheme was produced. It followed a similar structure to the previous part of the research in which an embedded system was connected to a fog node over WiFi, but the key difference is that the embedded system is guaranteed to make computations, instead of the offloader transmitting the entire input and results between devices. The structure of this task distribution can be seen in Figure 4.2. The fog node will only make computations if the results of the embedded system inference is not confident enough.

The offloader in this part of the research uses the same algorithm as discussed in Section 4.4.4, which makes this offloader is entropy based. It calculates the entropy

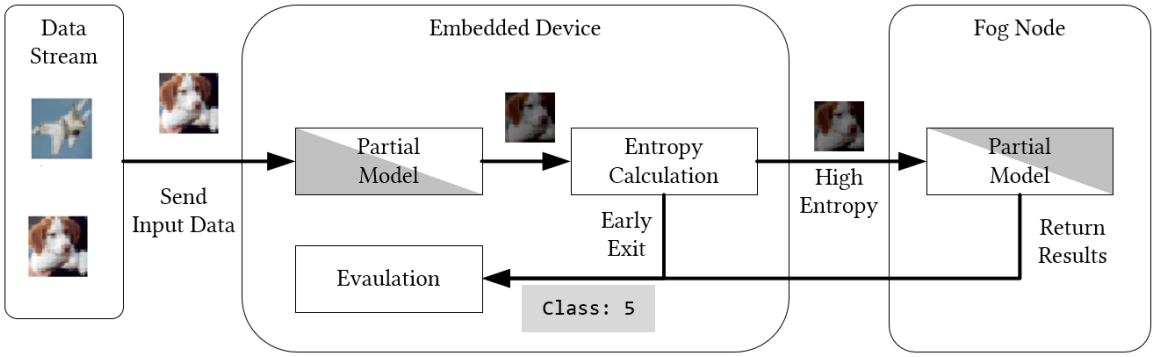

Figure 4.2: Setup of the task distribution for splitting the ML model

of every output of the exit layer in the embedded system and determines if the input should be calculated any further.

Chapter 5

Experiments

In this section, we describe how the approach seen in Chapter 4 was used in an experimental context. We describe the materials used in the experiment, how the experiment was set up, and the results of the experiment. We also discuss possible future avenues of exploration using this experiment as background.

5.1 Prerequisites

In this section, we define the software, hardware, and datasets used in this experiment.

5.1.1 Software

The main software used in this experiment was TensorFlow 2.0. TensorFlow provides developers with a Python library which can be used to create deep learning models. To develop with TensorFlow, Python version 3.7.3 was used and was written and tested using Jupyter Notebook. We describe how TensorFlow was used in our approach in the following section.

TensorFlow

TensorFlow [1] is a deep learning library which contains an interface to build graph-based models with. As the models are built on a graph, it is often visualized that the input data "flows" through the nodes of the graph, being manipulated and changed at each node. However, this research was done using TensorFlow 2.0's eager execution mode which allows for immediate evaluation of operations to Python without having to start a *Session*, the class used for interfacing with the graph. This was useful as it allowed us to get the results directly from the execution of layers. This means we could provide the location of execution of certain layers instead of TensorFlow, which determines the execution location when building the graph.

Using TensorFlow, a subclass of TensorFlow's Model class was created. This custom model, when executed, will measure the execution time of each layer and returns both the penultimate and exit layer's outputs upon completion. This way, we only needed to execute the custom model once, even if the entropy of the output bypassed the threshold.

The custom model itself was highly customizable; when initiated, it only consisted of an exit layer with the same number of neurons as output classes. It also provided functions which allowed users to add more layers before the exit layer. If the penultimate layer's output is not compatible with the final layer (i.e. the output has more than one dimension), it will automatically be flattened.

5.1.2 Hardware

The experiment was run using two different pieces of hardware, a Raspberry Pi 3 Model B acting as the embedded system, and a laptop with a 7th Generation Intel® Core™ i5 Processor acting as the fog node. It had 2 cores supporting 4 threads each,

with a processor base frequency of 2.50 GHz.

5.1.3 Datasets Used

The case studies examined were the wall-following robot navigation data set [14], and the Fashion-MNIST database [28]. The wall-following data set provides sensory inputs of a robot navigating a room. The Fashion-MNIST data set is a collection of small, gray-scale images, commonly used for benchmarking ML models. Both of these data sets provide ample data for training sets to create models, and test sets to use as simulated sensory inputs. These use-cases were chosen due to their varying data set sizes and feasibility to run on a resource-constrained device. As per Table 5.1, it can be seen that each of the chosen data sets can reach a somewhat reliable prediction accuracy with a very simple model (i.e. 2-3 hidden layers) in less than a few milliseconds. Also, the CIFAR-10 dataset was used in this experiment to showcase the effects of heavily increasing the dimensionality of the input on performance. Note that, as per Table 5.1, it does not reach a reliable prediction accuracy using the same prediction models when compared to the other use cases.

Wall-Following Navigation Data Set

The data set provided by Friere et al. in [14] provided a useful case study for this experiment. Their data consisted of ultrasound sensory data as inputs, and a direction to move (i.e. Slight-Right-Turn, Sharp-Right-Turn, Move-Forward, and Slight-Left-Turn). The goal of this data was to determine what direction the robot should take based on the ultrasound sensor inputs. The most important aspect of this data set is that it provided 3 data sets of increasing dimensionality. Using this data, the effects of increasing the dimensions on the same data set were observed to see if it affected the computation time. The data sets provided were: 2 inputs (sensors readings from

the front and left), 4 inputs (sensor readings from all cardinal directions), and 24 inputs (sensor readings from every individual sensor on the robot). All 3 data sets were split into their own training and testing sets, resulting in 3 sets of 3819 training inputs and 3 sets of 1637 testing inputs.

Fashion-MNIST Data Set

The MNIST dataset was used to benchmark NN's in many ML related experiments. It provides 28x28 pixel images of 10 different labels, and it is the task of the ML models to identify which label belongs to each image in the test set. However, the original dataset, the Handwritten Database, has become obsolete [28], so in this experiment the Fashion-MNIST dataset was used. It is a drop-in replacement, so the training, validation, and test set sizes remain the same.

This dataset was preferred provided a better challenge for ML models; the classes appear more similar to each other than the handwritten digits in the original dataset. As seen in Figure 5.1, the classes provide much finer differences to find; the classes pullover (2), coat (4), and shirt (6) seem visually similar but can be separated using ML models.

CIFAR-10

The CIFAR-10 dataset was the last dataset to be analyzed in this experiment. This dataset provided inputs of 32x32 images with 3 colour channels, making the dimensionality of this dataset much larger than the others, and thus the most resource-intensive to compute. However, unlike the Fashion-MNIST dataset, the difference in classes are much courser; the may include vegetation, mammals, and even vehicles. This can be seen in Figure 5.2 where every image is visually distinct, even those where the images are in the same class

Figure 5.1: Example Images from the Fashion-MNIST data set

5.2 Experimental Setup

In this section, we describe how the experiments were performed. We discuss how the datasets were used, and the models they were used in. We also explain the trials that were set up in order to achieve the Thesis Objective.

5.2.1 Delegation Phase Setup

To measure the effectiveness of the status quo in using fog computing with NN's, we used the task distribution portrayed by Figure 4.1. With this and the datasets collected, we developed several trials in which several machine learning models were created and tested to examine where using ML in embedded systems stands currently.

After the data was received, offloaded, and used as input in the model, the time it

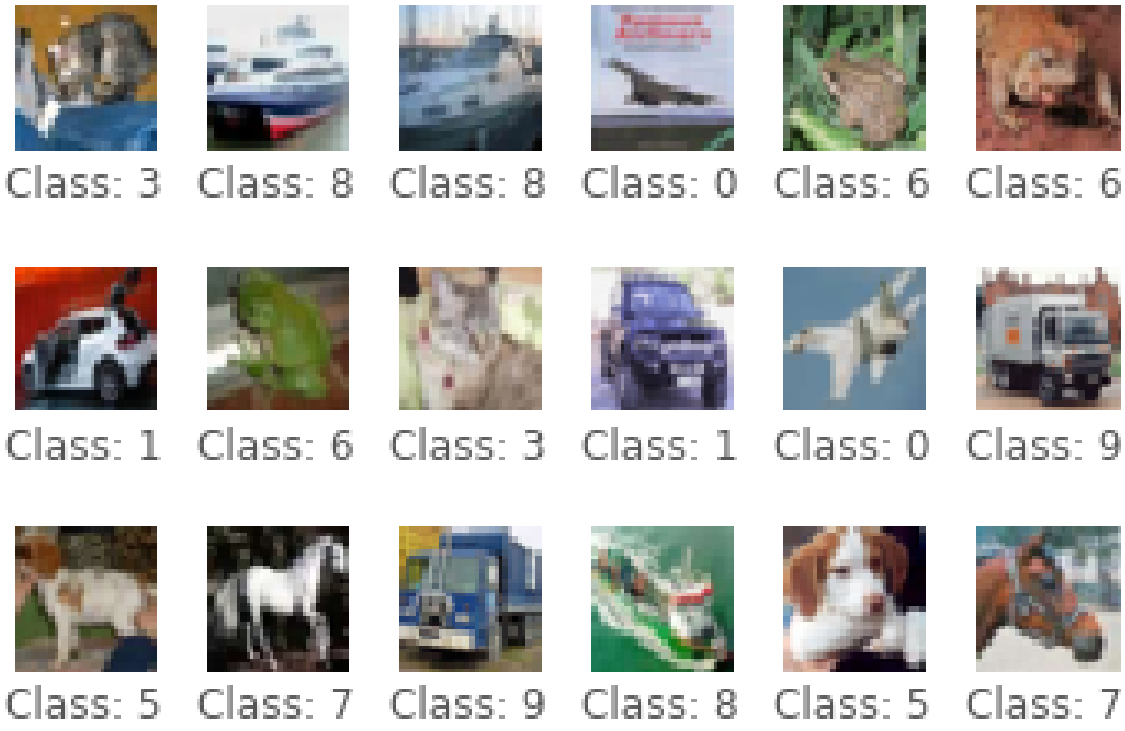

Figure 5.2: Example Images from the CIFAR-10 data set

took to perform each task was recorded and compared. The timing of each prediction was conducted over four different trials. The first two trials removed the delegator from the task distribution and delegated each input to a single model. In the first trial, every part of the task distribution was done on the embedded system. The second trial consisted of the embedded system offloading every input to the fog node instead. The network latency and prediction were timed separately in this trial. The third trial kept the full task distribution seen in Figure 4.1. The WCET of every task was measured from the previous trials, allowing for the offloading of every task which bypassed the threshold in Equation 4.5 to the fog node, while every other task was performed locally. In the final trial, every input was randomly offloaded to the fog node, with an offloading chance of 50%.

Each trial was run 10 times. The first 7 times used each data set individually, while the last 3 times ran mixed data sets. The first mixed data set contained an equal number of inputs from each individual data set in the following order: Wall-Turn-2-Inputs, Wall-Turn-4-Inputs, Wall-Turn-24-Inputs, Fashion-MNIST-MLP, Fashion-MNIST-CNN, CIFAR-10-MLP, CIFAR-10-CNN. The second mixed data set contained the same data, but the inputs were sorted by their origin (i.e., all the Wall-Turn-2 Inputs were next to each other, then the Wall-Turn 4-Inputs and so on). The third mixed data set contained the same number of inputs, but the inputs were randomly chosen from each data set. When sent to the predictor, each input was paired with its respective model. Each mixed data set had 1637 inputs.

5.2.2 Splitting Phase Setup

The models used in this part of experiment increased in complexity to handle each consecutive dataset mentioned in the previous section. There were four NN's: the simple NN with a single dense layer, a shallow NN with multiple convolutional layers and a single early exit, a deep NN with multiple dense layers of increasing size, and another deep NN with a similar structure to the previous, but with more convolutional layers. Every model has one or more exit layers; each exit layer consists of a flatten layer, a dense layer with the same number of neurons as output classes, and an activation layer that uses softmax. The structure and distribution of each of the models can be seen in Figure 5.3.

Each model was trained with a batch size of 32 over 100 epochs. The first model was run on all data sets, the second model was only used on the non-image datasets, and the third and fourth model was only run on the image datasets. The accuracy of each dataset on their training set can be seen in Table 5.1.

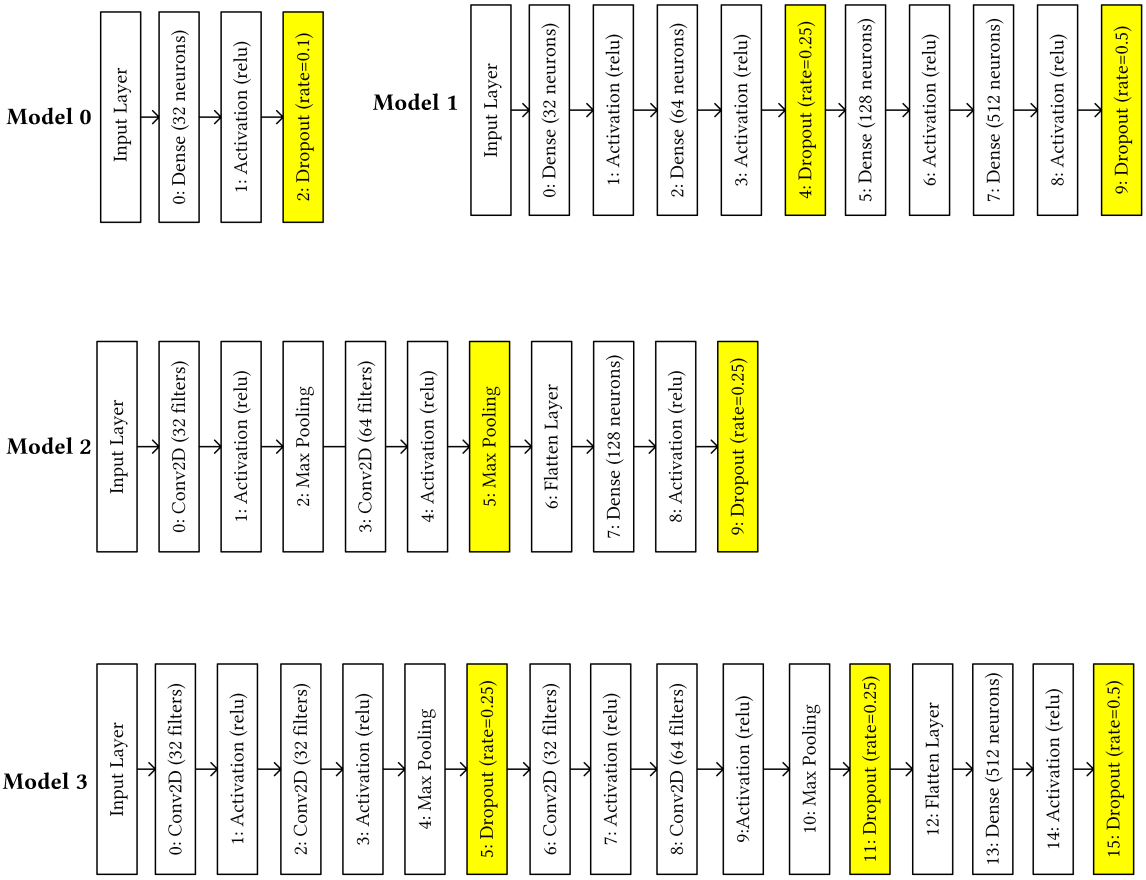

Figure 5.3: Structure of the NN models used in this experiment. The highlighted layers indicate that they have an early exit

5.2.3 Datasets

After creating the models, every model was executed completely locally and then remotely, without any split. We then compared the results to determine the feasibility of running machine learning models locally or hosting it in a remote location by using Equation 4.5. For the split models in this phase, we used an entropy threshold of 0 (always execute locally) and 1 (always execute remotely) for their respective trials.

After this, the splitting phase began. The goal was to compare the results of the previous phase to using early exits. Not only was the execution time be compared, the accuracy of the model was also examined for any impact as well. For this experiment,

Model	Classes	M0 Acc.	M1 Acc.	M2 Acc.	M3 Acc.
WT-2	4	97.48%	98.32%	-	-
WT-4	4	96.87%	97.41%	-	-
WT-24	4	97.74%	98.55%	-	-
MNIST	10	91.81%	-	99.75%	99.26%
CIFAR-10	10	76.18%	-	94.64%	91.45%

Table 5.1: Prediction accuracy for every dataset on every model they were executed on

we used the entropy thresholds of 0, 0.25, 0.5, 0.75 and 1 to find an entropy threshold that minimizes runtime and accuracy loss.

5.3 Results and Analysis

In this section, we discuss the implementation, results, and key observations of the experiments described in Section 5.2.

5.3.1 Delegation Phase Results

Over trials 1 and 2 in the experiment (i.e. running solely on the embedded system and purely offloading to the fog node, respectively), it was found that the embedded system could quickly run the MLP models independently without offloading them to the fog node. As seen in Table 5.2, the WCET of the machine learning tasks on the linear model was well below the average network latency, meaning that offloading any of these tasks at all will result in a lowered total throughput. This is further evidenced by Figure 5.4, whereas only offloading the inputs will result in a much higher WCET than simply running it on the embedded system. However, the CNN model's WCET on the embedded system exceeded the WCET threshold expressed by Equation 4.5. The average time it took to run the CNN on the embedded system was double that of the network latency and fog node computation time combined, meaning that any

Table 5.2: WCET measurements (in seconds) for each model for all 4 trials, as well as the network latency

Model	**Em. Sys.**	**Offloading**	**Efficient**	**Random**	**Latency**
WT2	0.0025	0.0054	0.0031	0.0049	0.036
WT4	0.0025	0.0058	0.0030	0.0047	0.022
WT24	0.0025	0.0058	0.0031	0.0047	0.034
FL	0.0024	0.0095	0.0030	0.0068	0.44
CL	0.0031	0.0872	0.0036	0.0072	1.13
FC	0.020	0.0075	0.0080	0.016	0.10
CC	0.025	0.0079	0.014	0.31	0.11

machine learning task using a CNN must be offloaded.

When using the mixed data sets in the trials, the third trial (i.e. efficient offloading) showed that only offloading the CNN inputs the total run time of the total trial significantly. When compared to the other implementations, the efficient offloading of the mixed dataset inputs was 33% faster than running solely on the embedded device, 43% faster than solely offloading all of the inputs to a fog node, and 50% faster than randomly offloading to the fog node. This comparison can be seen graphically for the randomized mixed task dataset in Figure 5.5, where the computation time of the efficient offloading implementation remained consistently lower than the other two implementations.

It was also noted that increasing the dimensionality of the dataset had minimal effect on the WCET of the linear implementation. As seen in Table 5.2, the time taken for every Wall-Turn dataset was almost exactly the same in every trial, and the graphs seen in Figure 5.4 show nearly the same results. The same almost could be said for the image data sets, except the high dimensionality of these data sets nearly doubled the network latency. It appears that the biggest contributing factor to the WCET of running ML algorithms on embedded systems is the algorithm used; the CNN had many high-complexity layers inside of it while the MLP's hidden layers

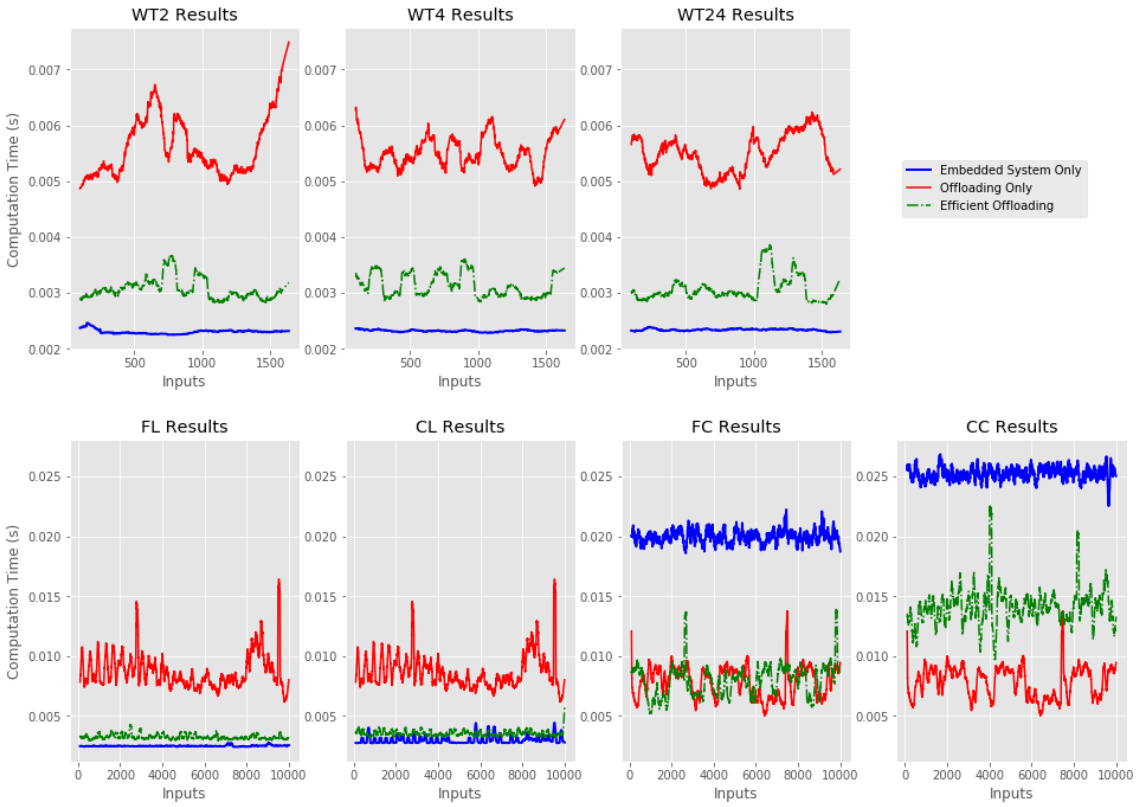

Figure 5.4: Offloading implementations compared for the single datasets

were rather simple. The dense layers used in the MLP consist of a single matrix multiplication and matrix addition. The inputs to this array were also flattened into a single 1D vector beforehand, thus making the complexity of each dense layer $O(nm)$, whereas n represents the length of the vector. The complexity of the convolutional model is much more complex. The complexity of each layer is not only affected by its own filter attributes, but also by the filter attributes of preceding convolutional layer, as well as layer's output dimensions. The resulting complexity, as calculated by Microsoft Research in [15], shows that these layers have a time complexity which has an exponentially larger growth than the dense layers in normal neural networks.

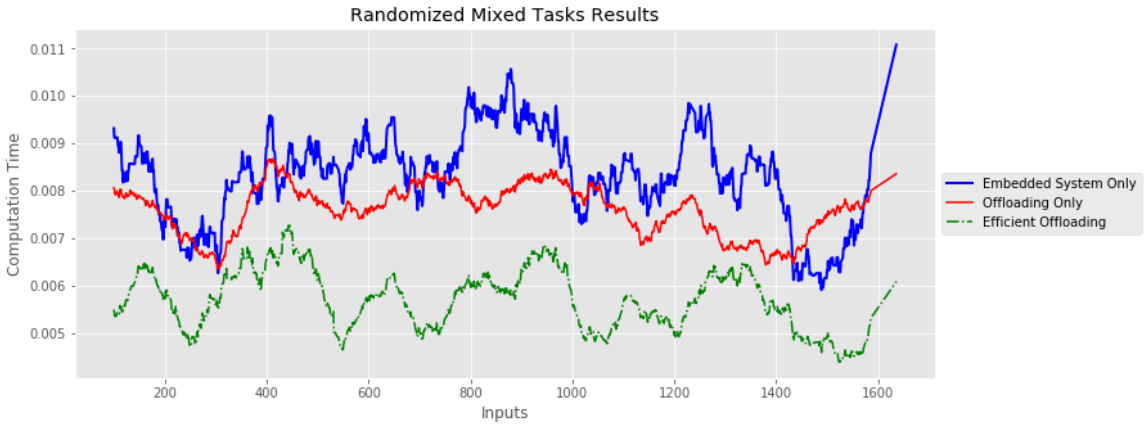

Figure 5.5: Offloading Implementations compared for a Mixed data set

Dataset	Model	IoT (s)	Fog Node (s)	Latency (s)
WT2	0	0.0067	0.016	0.014
WT2	1	0.019	0.018	0.014
WT4	0	0.0056	0.016	0.015
WT4	1	0.020	0.018	0.014
WT24	0	0.0057	0.014	0.013
WT24	1	0.019	0.018	0.014
MNIST	0	0.015	0.024	0.021
MNIST	2	0.097	0.026	0.016
MNIST	3	0.119	0.030	0.018
CIFAR-10	0	0.014	0.031	0.028
CIFAR-10	2	0.119	0.045	0.033
CIFAR-10	3	0.143	0.047	0.032

Table 5.3: Total execution times of each model with every dataset

5.3.2 Splitting Phase Results

In this phase, we ran each model by itself on the embedded system, then ran each model solely by offloading each input to the fog node. Each experiment used a static 100 inputs for every run which did not change throughout the experiment. From this phase, we determined the splitting points for each model by examining the execution time of each layer on both devices. Table 5.3 shows the total runtimes of every model with every dataset.

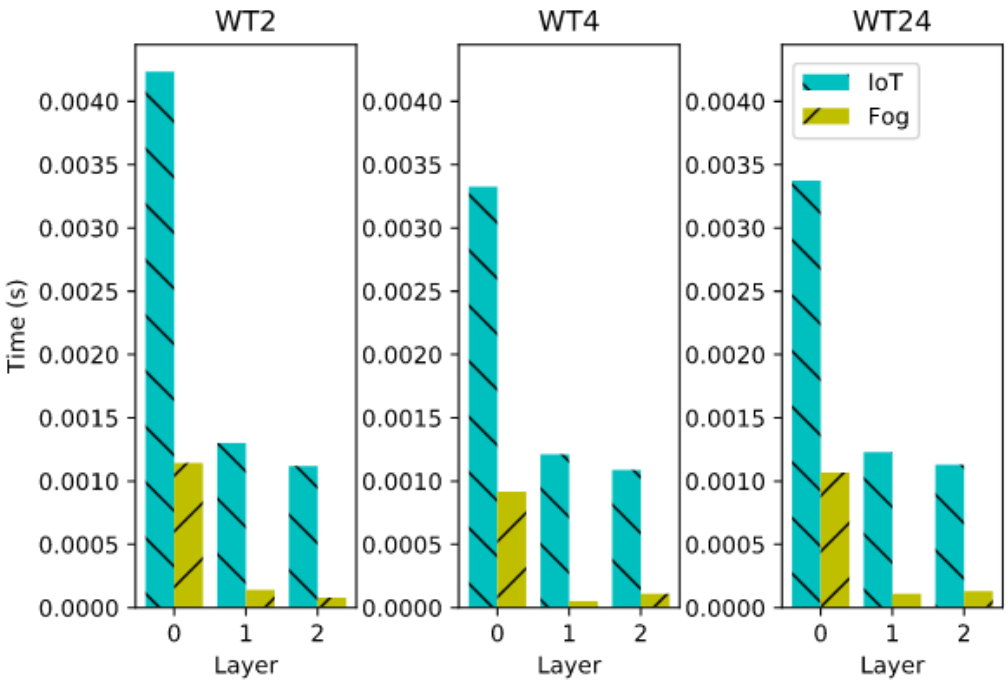

Figure 5.6: Layer time results for the non-image datasets on Model 0

For the first experiment, all the datasets were run on Model 0. It was determined that this model should not be split since the runtime of each dataset on the model did not exceed the threshold provided by Equation 4.5. For example, the upper bound of the WCET thresholds for the Model 0 trials was 0.016 seconds; none of the datasets which ran on Model 0 bypassed this in a single layer, as seen in Figure 5.6. A similar result can be seen for the image datasets in Figure 5.7; although the difference between running the model locally to running the model on the fog node increased significantly, this difference did not make up for the 0.016 seconds in network latency. Therefore, this model should be executed exclusively on the embedded system, even with the faster runtime on the fog node.

Model 1 had a different result. Some of the executions on the IoT device had an execution time which was approximately the same as the WCET threshold, as seen int Table 5.3. This means that completely offloading or splitting this model is

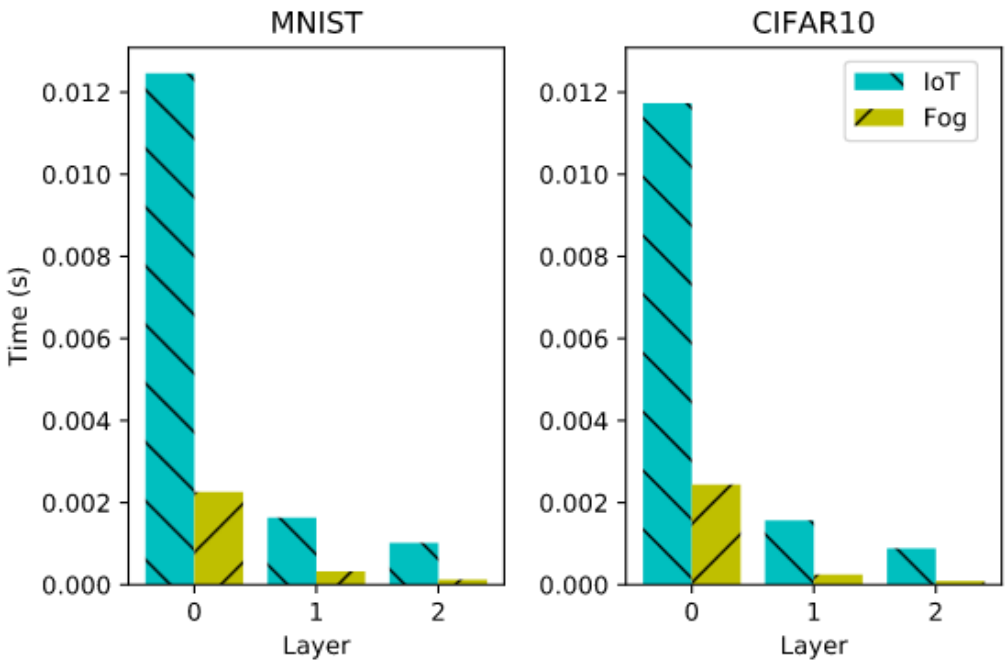

Figure 5.7: Layer time results for the image datasets on Model 0

a viable option when improving the total runtime. To find the splitting point, we look at Figure 5.8, which shows the layer-by-layer runtime analysis of Model 1 on the Wall-Turn datasets. From this, we can see that there are potential splitting points between layers 4 and 5, as well as 6 and 7. Placing the splitting point between layers 4 and 5 was preferred as layer 4 was a dropout layer and was earlier in the model.

When examining Model 2, it was found that offloading was much more effective than running exclusively on the embedded system. Table 5.3 shows that the embedded system runtime surpasses the WCET threshold for all datasets. This is because of a major bottleneck in the model. Figure 5.9 shows that the major bottleneck for this model is layer 7, a Dense layer with 128 neurons. Since layer 6 is a flatten layer, the splitting point was chosen to be layer 5 instead. Another key observation can be made from Figure 5.9; the size of the layer's output heavily affects its runtime. Although the convolutional layers are more complex, the dense layer has twice as many output

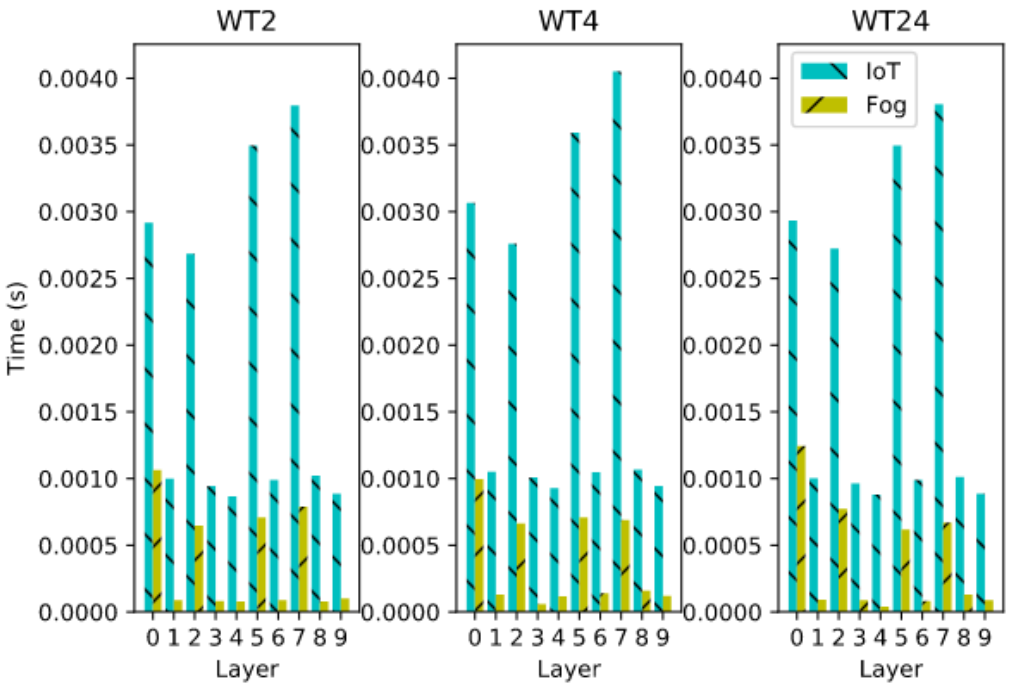

Figure 5.8: Layer time results of the Wall-Turn datasets on Model 1

neurons, making it take more than twice as long to execute.

Lastly, Model 3 displayed a similar result. The total runtime on the embedded system exceeded the WCET threshold, meaning that offloading some or all of the inputs is necessary. Figures 5.10 and 5.11 shows that the splitting points could be between layers 5 and 6 or layers 11 and 13. For this experiment, we chose layer 5 as it is earlier in the model and is also a dropout layer. A second early exit was also placed at layer 11 since its execution time was large and should be avoided to reduce runtime. Therefore, this model was partitioned into 3 sub-models, whereas the first executes on the embedded system and the last two execute on the fog node.

5.3.3 Splitting Phase

In this phase, the splitting points found in the delegation phase were implemented. To evaluate the impact of partial inference on the accuracy and runtime of the model

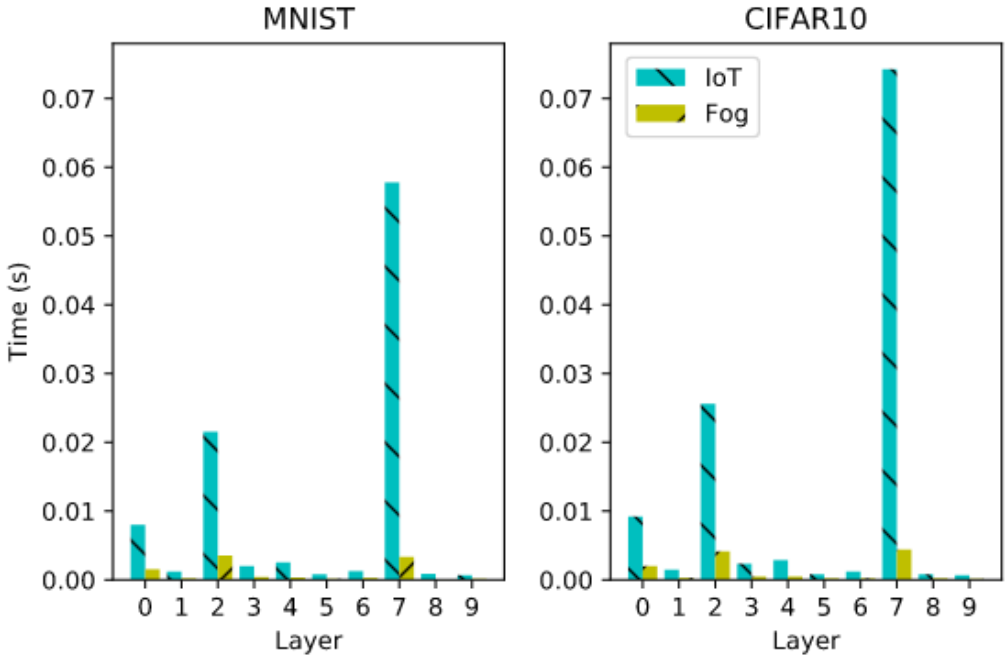

Figure 5.9: Layer time results of the image datasets on Model 2

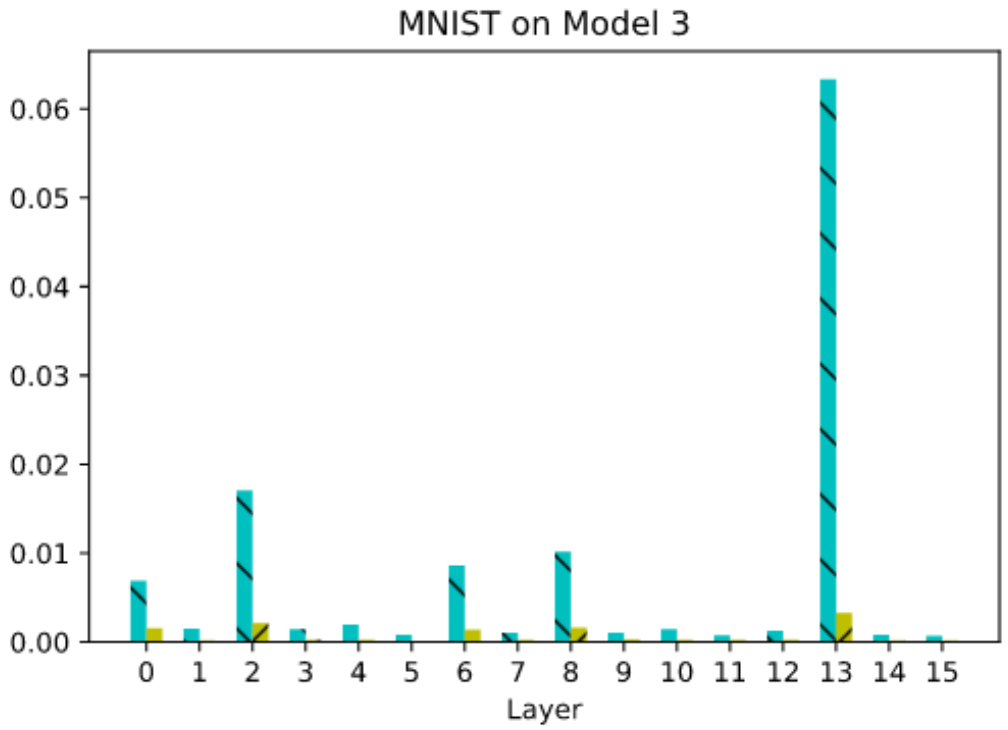

Figure 5.10: Layer time results of MNIST on Model 3

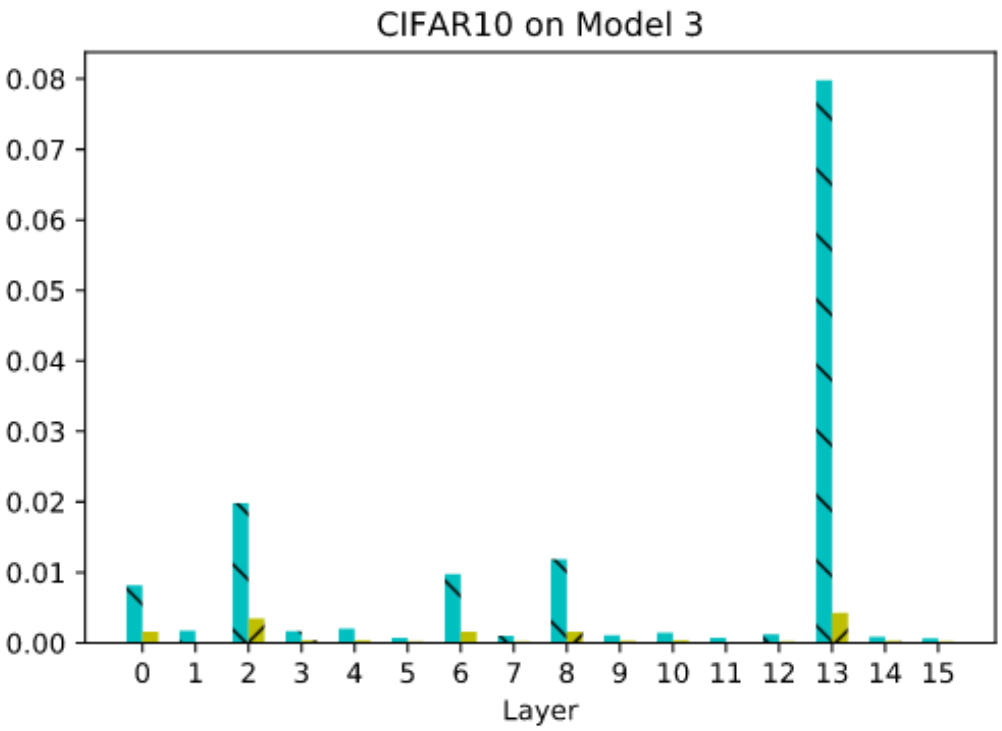

Figure 5.11: Layer time results of CIFAR-10 on Model 3

we ran each model multiple times with different datasets and entropy thresholds. Since changing the entropy thresholds alters how much of the model's layers will be executed, it affects the frequency of message transfers and accuracy of the model.

By decreasing the number of layers executed by the model, the accuracy of the model is likely to decrease as well since less analysis can be done on the input. This is seen in Figure 5.12; the higher the entropy threshold, the more likely there will be a drop in total accuracy. This is apparent when observing the datasets with lower training accuracy, namely the MNIST and CIFAR-10 datasets. The MNIST dataset on Model 3 drops from 94.21% at a threshold of 0 to 91.62% at a threshold of 0.5. Eventually, the drop in accuracy will even out, as the model may not contain enough entropy to reach the high threshold, thus reducing the number of messages being sent between devices to zero.

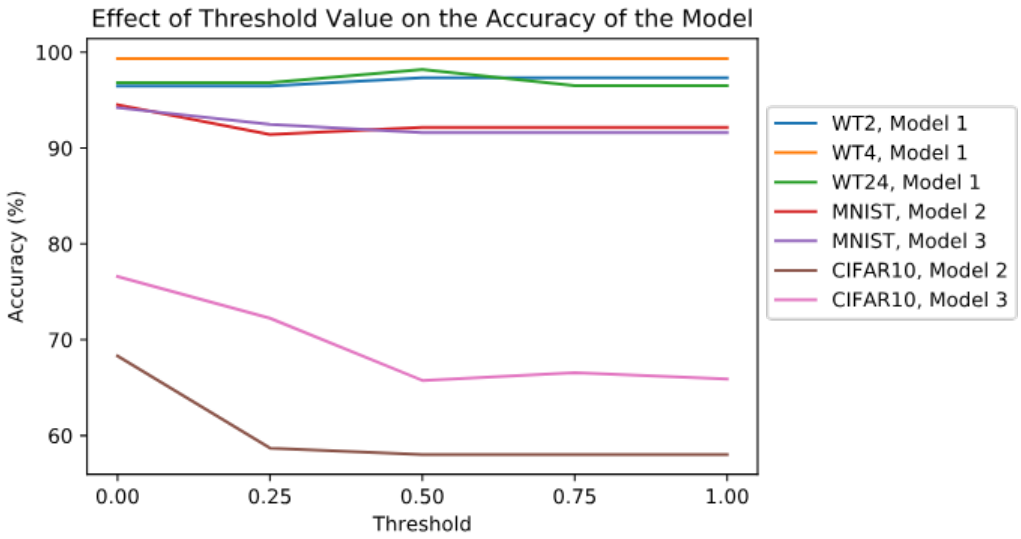

Figure 5.12: Effect of changing the entropy threshold on the accuracy of the split model

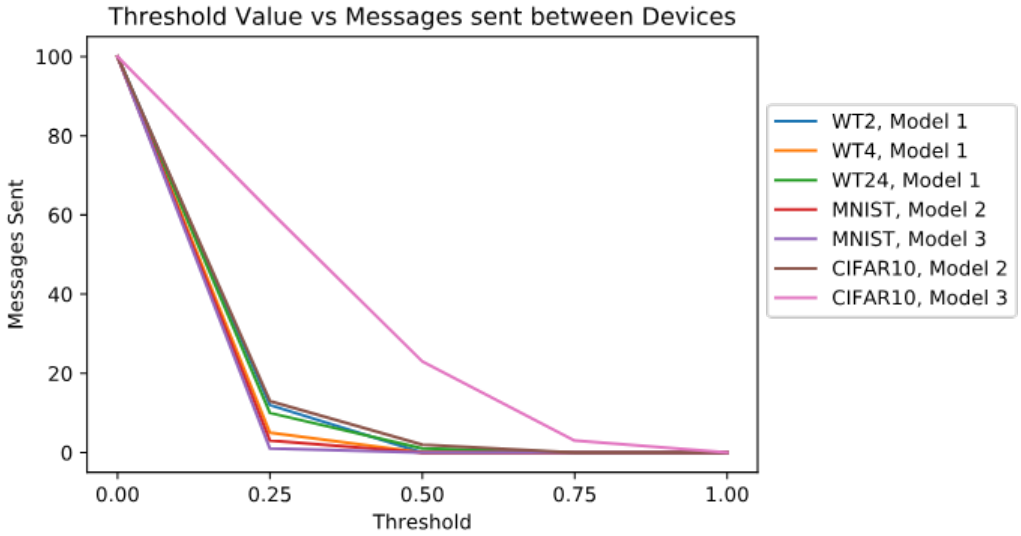

Figure 5.13: Effects of increasing the entropy threshold on the messages sent between devices

When increasing the entropy threshold of the model, we decrease the number of messages being sent between devices, thus saving time and energy. Figure 5.13 shows that there is a significant drop-off in messages sent from the device just by increasing the threshold to 0.25. For example, MNIST on model 2 only saw a significant decrease in messages sent from an entropy threshold of 0 to 0.25. The number of messages sent was reduced to 3%, saving a significant amount of time (i.e. message latency and fog node execution) at the cost of 3.09% in accuracy.

In this experiment, we also considered the CIFAR-10 dataset for comparison. This dataset showed a pronounced version of the above observations (i.e. bigger accuracy drops and gentler decline in messages sent). This can be explained by the accuracy of its training set, as seen in Table 5.1. As the models used in this experiment had a lower accuracy in the training set compared to the other datasets, more entropy was introduced into the output as a result.

Chapter 6

Conclusions

As NN's become more commonplace, it is important that when they are used in conjunction with IoT that they runs at a quick, reliable, and predictable speed. Unfortunately, there are many issues which may affect the NN's performance in edge networks, such as highly variable network latency, and possible security issues. To tackle this, we tested several solutions, ranging from the status quo (i.e. always offloading NN tasks to an external device) to splitting each NN model into sub-models and executing each sub-model on a different device in the IoT hierarchy.

To do this, a framework built off of Tensorflow 2.0 was made to build and handle multiple models and chain them together. An early exit system was also implemented; after every sub-model executed, the entropy was calculated to determine if any subsequent models should be executed. We then used this framework on a testbed utilizing several NN models and datasets, as well as an IoT device and fog node. It was found that, for models with high training accuracy, splitting the models could decrease the number of messages between devices by up to 97%, while only sacrificing 3% accuracy.

However, not all models are capable of being split. Model 0 exhibited this behaviour since its splitting point was determined to be at the very beginning. In order

to have a split model, the model itself must be segmented with low complexity layers, such as pooling or dropout layers. Therefore, more investigation should go into structuring NN models for performance on networks and heterogeneous hardware, rather than just aiming for accuracy.

Additionally, the experiment performed in this paper shows that, when in the designing embedded systems which utilize machine learning, one must consider the WCET of the ML algorithm on both the embedded system and the fog node, as well as the average network latency. When properly analyzed, it will become evident as to which tasks should be offloaded, and which tasks can be performed locally. We also proposed a WCET threshold equation in which, if a local computation time exceeds, indicates if a task should be offloaded to a nearby fog node to efficiently offload the tasks.

This is supported by the results of the delegation phase of the experiment in which several different ML tasks of varying WCETs were performed on an embedded system, fog node, and both over Wi-Fi. The embedded system showed that it was capable of running low complexity algorithms (MLP) easily on its own, but heavily slowed down when performing a complex algorithm (CNN). When offloading every ML task to the fog node, the complex algorithm was solved in a much faster time, but the added network latency to every less complex task made the entire process run slower. By only offloading the tasks that had surpassed the WCET threshold given, it was witnessed that the system sped up its computation time from 33% to 50%, depending on the original implementation.

6.1 Future Work

Unfortunately, computation time is not the sole important measure in embedded systems. Although it is arguably the most important, as real-time systems heavily depend on meeting deadlines to meet their requirements, one must consider the energy consumption and heat output as well. Many embedded systems have batteries which will drain faster if many local ML tasks are performed on it. More testing should be done on the power consumption of ML tasks on battery usage, and if an energy-saving algorithm to offload tasks can be uncovered. Similarly, if an embedded system is generating too much heat from computing ML tasks, then a heat-reducing offloading algorithm must be implemented as well.

Also, the algorithm provided by 3 only considers a network with low variance in transmission throughput. A network with a high variance in throughput may disproportionately assign machine learning tasks to the local processor. To handle varying network delay, it is possible to recalculate the WCET threshold every few inputs to gain an updated view of the current network characteristics. However, the time overhead of this action must be taken into consideration with the criticality of the prediction. It may also be possible to implement another protocol, like RT-WiFi [27], which allows for a deterministic transfer of packets over WiFi.

Cloud computing can also be taken into account as a future work. Even though it would introduce a higher latency in computations, it may also serve as a backup when the available fog computers are not available to be used. It may also be a cheaper option for the users of IoT devices as they do not need to supply themselves with fog nodes. More testing into services such as AWS and Microsoft Azure must be done in order to assess the feasibility of bringing split NN functionality to the cloud.

When implemented, the use cases in which this system can be used in can include

self-driving vehicles [9], medical uses such as monitors and alerts, as well as industrial uses such as visual inspection of materials [32]. The system provided may benefit these use cases as it provides faster and more reliable throughput to maintain utmost functionality of neural networks in real-time settings. To achieve these functions, ways to find out how to induce more deterministic results into these systems are important as well; NN's introduce randomness due to some layers such as dropout. NN's must be structured in ways to reduce randomness as much as possible. Additionally, these real-time systems will need secure communications in order to function, so further testing into how effective the NN splitting is at securing privacy must also be tested in a use case as well.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Asplund, M., and Nadjm-Tehrani, S. Attitudes and perceptions of iot security in critical societal services. *IEEE Access 4* (2016), 2130–2138.

[3] Atzori, L., Iera, A., and Morabito, G. The internet of things: A survey. *Computer Networks 54*, 15 (2010), 2787 – 2805.

[4] Auluck, N., Azim, A., and Fizza, K. Improving the schedulability of real-time tasks using fog computing. *IEEE Transactions on Services Computing* (2019).

[5] Beaty, D. L., Quirk, D., and Jaworski, J. Fog computing. *Ashrae Journal* (January 2018), 68–74.

[6] Carminati, A., Starke, R. A., and Oliveira, R. S. d. On the use of static branch prediction to reduce the worst-case execution time of real-time applications. *Real-Time Systems 54*, 3 (Jul 2018), 537–561.

[7] Chahal, K. S., Grover, M. S., Dey, K., and Shah, R. R. A hitchhiker's guide on distributed training of deep neural networks. *Journal of Parallel and Distributed Computing 137* (2020), 65 – 76.

[8] Crespo-Perez, G., and Ojeda-Castro, A. Convergence of cloud computing, internet of things, and machine learning: The future of decision support systems. *International Journal of Scientific Technology Research 6*, 7 (2017), 131–136.

[9] Daily, M., Medasani, S., Behringer, R., and Trivedi, M. Self-driving cars. *Computer 50*, 12 (December 2017), 18–23.

[10] Dally, W. J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R. C., Parikh, V., Park, J., and Sheffield, D. Efficient embedded computing. *Computer 41*, 7 (July 2008), 27–32.

[11] DSaguil. Dsaguil/splitternet. https://github.com/DSaguil/splitternet. Accessed: 2019-12-16.

[12] Early, S. Analytics, machine learning, and the internet of things. 10–13.

[13] Fizza, K., Azim, A., Auluck, N., Maruf, M., and Singh, A. Faster ota updates in smart vehicles using fog computing. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC '19 Companion)* (2019), 59–64.

[14] Freire, A. L., Barreto, G. A., Veloso, M., and Varela, A. T. Short-term memory mechanisms in neural network learning of robot navigation tasks: A case study. In *2009 6th Latin American Robotics Symposium (LARS 2009)* (Oct 2009), pp. 1–6.

[15] He, K., and Sun, J. Convolutional Neural Networks at Constrained Time Cost. *arXiv e-prints* (Dec 2014), arXiv:1412.1710.

[16] Jeong, H., Jeong, I., Lee, H., and Moon, S. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (July 2018), pp. 1492–1499.

[17] Jerabek, T., and Horauer, M. Static worst-case execution time analysis tool development for embedded systems software. pp. 7–14.

[18] Kang, Y., Hauswald, J., Gao, C., Rovinski, A., Mudge, T., Mars, J., and Tang, L. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 615–629.

[19] Kulkarni, P., and Farnham, T. Smart city wireless connectivity considerations and cost analysis: Lessons learnt from smart water case studies. *IEEE Access 4* (2016), 660–672.

[20] Leroux, S., Bohez, S., Coninck, E. D., Verbelen, T., Vankeirsbilck, B., Simoens, P., and Dhoedt, B. The cascading neural network: building the internet of smart things. *Knowledge and Information Systems 52* (2017), 791–814.

[21] Mahdavinejad, M. S., Rezvan, M., Barekatain, M., Adibi, P., Barnaghi, P., and Sheth, A. P. Machine learning for internet of things data analysis: A survey. *Digital Communications and Networks* (2017).

[22] Saguil, D., and Azim, A. Time-efficient offloading for machine learning tasks between embedded systems and fog nodes. 79–82.

[23] Sattler, F., Wiedemann, S., Müller, K., and Samek, W. Robust and communication-efficient federated learning from non-i.i.d. data. *IEEE Transactions on Neural Networks and Learning Systems* (2019), 1–14.

[24] Teerapittayanon, S., McDanel, B., and Kung, H. T. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (June 2017), pp. 328–339.

[25] Tsai, C.-W., Lai, C.-F., Chiang, M.-C., and Yang, L. T. Data mining for internet of things: A survey. *IEEE Communications, Surveys, and Tutorials 16*, 1 (2014), 77–97.

[26] Want, R., Schilit, B. N., and Jenson, S. Enabling the internet of things. *Computer 48*, 1 (Jan 2015), 28–35.

[27] Wei, Y., Leng, Q., Han, S., Mok, A. K., Zhang, W., and Tomizuka, M. Rt-wifi: Real-time high-speed communication protocol for wireless cyber-physical control applications. In *2013 IEEE 34th Real-Time Systems Symposium* (Dec 2013), pp. 140–149.

[28] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[29] Xu, G., Li, H., Liu, S., Yang, K., and Lin, X. Verifynet: Secure and verifiable federated learning. *IEEE Transactions on Information Forensics and Security 15* (2020), 911–926.

[30] Zhang, Z., Song, T., Lin, L., Hua, Y., He, X., Xue, Z., r. ma, and h. Guan. Towards ubiquitous intelligent computing: Heterogeneous distributed deep neural networks. *IEEE Transactions on Big Data* (2018), 1–1.

[31] Zhu, H., and Jin, Y. Multi-objective evolutionary federated learning. *IEEE Transactions on Neural Networks and Learning Systems* (2019), 1–13.

[32] Židek, K., Hošovský, A., and Dubjak, J. Diagnostics of surface errors by embedded vision system and its classification by machine learning algorithms. *Key Engineering Materials 669* (2016), 459–466.